AEDS III - Trabalho Prático 2 - Busca Em Grafos Sem Pesos

LUCAS DOGO DE SOUZA PEZZUTO

MARIA EDUARDA FABIANO PIRES

MATHEUS MALVÃO BARBOSA

RYAN RODRIGUES

TIAGO COSTA SOARES

ALFENAS

2024

LUCAS DOGO DE SOUZA PEZZUTO
MARIA EDUARDA FABIANO PIRES
MATHEUS MALVÃO BARBOSA
RYAN RODRIGUES
TIAGO COSTA SOARES

Busca Em Grafos Sem Pesos

Trabalho apresentado à disciplina de Algoritmos e Estruturas de Dados 3 (Aeds lll), ministrada pelo professor Dr. Iago Augusto de Carvalho, no curso de Ciência da Computação na Universidade Federal de Alfenas, Campus Santa Clara.

ALFENAS
2024

SUMÁRIO

1. INTRODUÇÃO

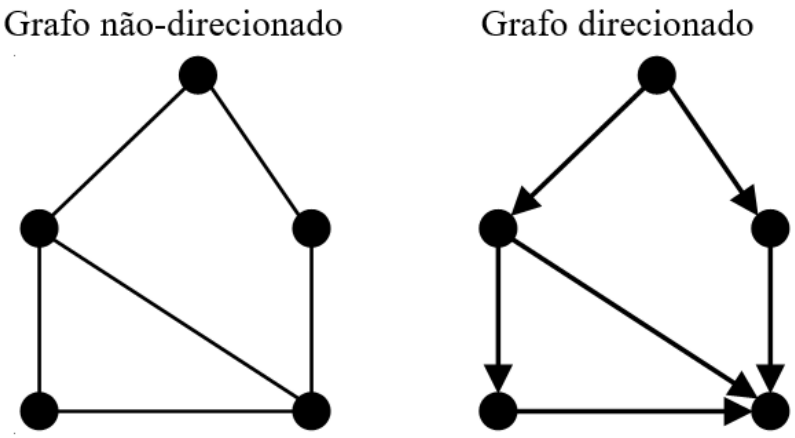
O objetivo do trabalho é compreender e implementar diferentes algoritmos de busca em grafos sem pesos. Além disso, também objetiva-se praticar a modelagem de grafos como listas de adjacência ou matriz de adjacência.

A Teoria dos Grafos é um ramo fundamental da matemática discreta que estuda as propriedades e relações entre objetos abstratos chamados grafos. Um grafo é composto por um conjunto de vértices (ou nós) e um conjunto de arestas (ou conexões/arcos) que representam as relações entre esses vértices.

Os grafos são amplamente utilizados em diversos campos, incluindo Ciência da Computação, Matemática Aplicada, Biologia, Engenharia e Redes Sociais. Eles fornecem um modelo poderoso para representar e analisar uma variedade de problemas e sistemas complexos, desde a representação de redes de computadores até a modelagem de interações entre moléculas em sistemas biológicos.

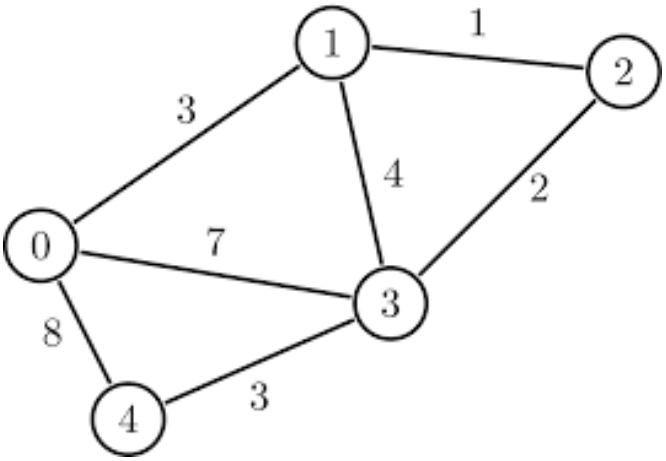
Dentro da Teoria dos Grafos, há diferentes características para a criação e formação de um grafo, aqui estamos utilizando um grafo sem pesos e não direcionado, ou seja, todas as arestas têm o mesmo custo de operação e podem ser percorridas em qualquer direção. A aresta que liga A e B pode ser percorrida de A para B quanto de B para A.

Figura 1: Direcionamento de grafos



Na Figura 1 podemos analisar a diferença entre os dois tipos de direcionamento de um grafo, o não direcionado apresenta apenas as arestas. O direcionado recebe uma seta em cada aresta, seta essa que indica o vértice destino. Ambos os grafos são sem peso, uma vez que não há um “preço” definido para a operação.

Figura 2: Grafo com pesos



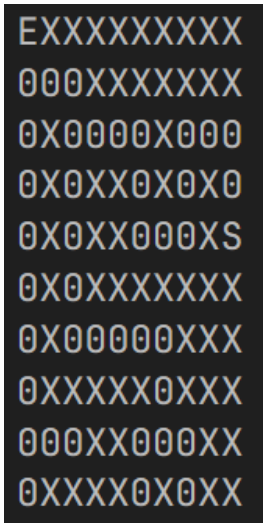
Para a exemplificação temos na Figura 2 um grafo com pesos, o peso é representado pelo número ao lado da aresta.

2. LABIRINTOS

O problema apresentado no trabalho consiste em transformar um labirinto em um grafo para realizar as operações de busca, a fim de encontrar o caminho entre a entrada e a saída.

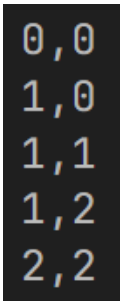
Os labirintos possuem um tamanho total de 10 linhas e 10 colunas, sendo representados com 4 símbolos, S-E-X-0 (assim como representado na Figura 3), onde S representa a saída, E a entrada, X as paredes do labirinto e 0 as passagens.

Figura 3: Labirinto



A saída desejada após a execução dos algoritmos consiste em um passo a passo com os vértices percorridos, sendo impressos em duas colunas separadas por vírgula, onde o primeiro elemento representa a linha do vértice no arquivo original, e o segundo elemento representa a coluna.

Figura 4: Exemplo de saída



3. ESTRUTURAS

3.1 NO E ARESTA

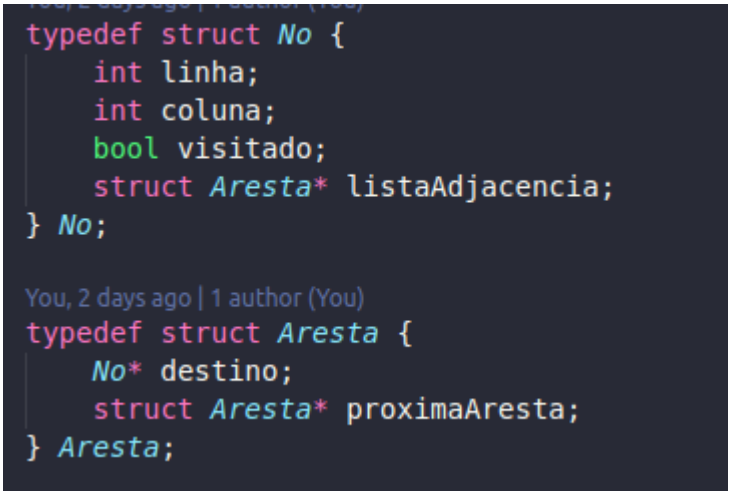
Para representar os vértices do grafo em questão foi utilizada a estrutura ‘No’ (Figura 5) que possui 4 atributos, sendo eles:

- **linha**: Um atributo inteiro que representa a linha da posição original do vértice no arquivo de entrada.
- **coluna**: Um atributo inteiro que representa a coluna da posição original do vértice no arquivo de entrada.
- **visitado**: Um atributo booleano que indica aos algoritmos de busca implementados se o vértice já foi percorrido durante a execução, impedindo que o algoritmo revisite vértices e entre em um possível loop.
- **listaAdjacencia**: Um ponteiro para a estrutura de lista de adjacência do vértice. A lista de adjacência armazena todas as arestas que partem do vértice atual, representando suas conexões com outros vértices no grafo. Cada elemento na lista de adjacência é uma aresta que liga o vértice atual a outro vértice no grafo.

Já para a definição das Arestas foi utilizada a estrutura ‘Aresta’ (Figura 5) que possui os seguintes atributos:

- **destino**: Um ponteiro para a estrutura No, que representa o vértice de destino da aresta. Esta aresta conecta o vértice atual a outro vértice no grafo, e o atributo destino armazena uma referência ao vértice de destino.
- **proximaAresta**: Um ponteiro para a próxima aresta na lista de adjacência do vértice atual. Esta estrutura de dados permite a construção da lista de adjacência, onde todas as arestas que partem do mesmo vértice são encadeadas em uma lista, facilitando o acesso às conexões do vértice.

Figura 5: Estruturas No e Aresta



3.2 PILHA

A estrutura ‘Pilha’ (Figura 6) foi implementada para suportar operações de empilhamento e desempilhamento de itens do tipo No. Possui os seguintes atributos e funções associadas:

- **itens:** Um array de ponteiros para No, com tamanho definido como LINHAS * COLUNAS, que representa os elementos armazenados na pilha. Esta estrutura permite o armazenamento de um número máximo de elementos na pilha, determinado pelo produto do número de linhas pelo número de colunas do grafo.
- **topo:** Um inteiro que representa o índice do topo da pilha, ou seja, a posição do último elemento empilhado. Inicialmente, o topo é definido como -1 para indicar que a pilha está vazia.

Funções:

- **inicializarPilha(Pilha* pilha):** Função responsável por inicializar a pilha, atribuindo o valor inicial de -1 ao topo, indicando que a pilha está vazia.
- **pilhaVazia(Pilha* pilha):** Função que verifica se a pilha está vazia. Retorna 1 se

a pilha estiver vazia e 0 caso contrário.

- **empilhar(Pilha* pilha, No* item):** Função para empilhar um item do tipo No na pilha. Incrementa o topo e atribui o item à posição correspondente na pilha.
- **desempilhar(Pilha* pilha):** Função para desempilhar um item da pilha. Retorna o item do tipo No que estava no topo da pilha e decrementa o topo.
- **topoDaPilha(Pilha* pilha):** Função que retorna o item do tipo No que está no topo da pilha, sem removê-lo. Retorna NULL se a pilha estiver vazia.

Figura 6: Pilha

```
typedef struct Pilha {
    No* itens[LINHAS * COLUNAS];
    int topo;
} Pilha;

void inicializarPilha(Pilha* pilha);
int pilhaVazia(Pilha* pilha);
void empilhar(Pilha* pilha, No* item);
No* desempilhar(Pilha* pilha);
No* topoDaPilha(Pilha* pilha);
```

3.3 FILA

A estrutura ‘Fila’ (Figura 7) foi implementada para suportar operações de enfileiramento e desenfileiramento de itens do tipo No. Possui os seguintes atributos e funções associadas:

- **itens:** Um array de estruturas, cada uma contendo dois ponteiros para No: um representando o filho e o outro representando o pai. O tamanho do array é definido como LINHAS * COLUNAS, permitindo o armazenamento de um número máximo de elementos na fila, determinado pelo produto do número de linhas pelo

número de colunas do grafo.

- **frente:** Um inteiro que representa a posição do primeiro elemento na fila. Inicialmente, é definido como 0.
- **fim:** Um inteiro que representa a posição após o último elemento na fila. Inicialmente, é definido como -1.

Funções:

- **inicializarFila(Fila* fila):** Função responsável por inicializar a fila, atribuindo o valor inicial de 0 para frente e -1 para fim.
- **filaVazia(Fila* fila):** Função que verifica se a fila está vazia. Retorna 1 se a fila estiver vazia e 0 caso contrário.
- **enfileirar(Fila* fila, No* pai, No* atual):** Função para enfileirar um item do tipo No na fila, junto com seu respectivo pai. Incrementa o valor de fim e adiciona o item à posição correspondente na fila.
- **desenfileirar(Fila* fila):** Função para desenfileirar um item da fila. Retorna o item do tipo No que estava na frente da fila e atualiza o valor de frente para indicar o próximo item na fila.

Figura 7: Fila

```
typedef struct Fila {
    You, 1 second ago | 1 author (You)
    struct {
        No* filho;
        No* pai;
    } itens[LINHAS * COLUNAS];
    int frente;
    int fim;
} Fila;


void inicializarFila(Fila* fila);
int filaVazia(Fila* fila);
void enfileirar(Fila* fila, No* pai, No*atual);
No* desenfileirar(Fila* fila);
```

3.4 GRAFO

Apesar de o código não possuir uma estrutura ‘Grafo’ propriamente dita, esse tópico aborda a implementação das funções (Figuras 8 e 9) que manipulam os vértices do nosso grafo que é representado pela matriz No* nos[LINHAS][COLUNAS]. Onde cada elemento do array é um ponteiro para um nó (No), permitindo a criação e manipulação eficiente da estrutura de dados do grafo.

- **adicionarAresta**: Esta função é responsável por adicionar uma aresta à lista de adjacência de um vértice. Ela recebe um ponteiro para um ponteiro de aresta (Aresta** listaAdjacencia) e o vértice de destino (No* destino). Uma nova aresta é alocada dinamicamente e inicializada com o vértice de destino. Em seguida, essa nova aresta é adicionada no início da lista de adjacência do vértice, atualizando o ponteiro listaAdjacencia para apontar para a nova aresta.
- **criarGrafo**: Esta função é responsável por criar o grafo com base em um labirinto fornecido. Ela recebe como entrada uma matriz representando o labirinto (char labirinto[LINHAS][COLUNAS]) e uma matriz de ponteiros para vértices (No* nos[LINHAS][COLUNAS]). A função itera sobre cada célula do labirinto e, se a célula não for uma parede, cria um vértice correspondente na matriz de vértices. Para cada vértice criado, são adicionadas arestas para as células adjacentes válidas, representando as conexões entre os vértices no grafo.

Figura 8: Função adicionarAresta

```
void adicionarAresta(Aresta** listaAdjacencia, No* destino) {
    Aresta* novaAresta = (Aresta*)malloc(sizeof(Aresta));
    novaAresta->destino = destino;
    novaAresta->proximaAresta = *listaAdjacencia;
    *listaAdjacencia = novaAresta;
}
```

Figura 9: Função criarGrafo

```
void criarGrafo(char labirinto[LINHAS][COLUNAS], No* nos[LINHAS][COLUNAS]) {
    for (int i = 0; i < LINHAS; i++) {
        for (int j = 0; j < COLUNAS; j++) {
            if (labirinto[i][j] != PAREDE) {
                No* atual = nos[i][j];
                atual->linha = i;
                atual->coluna = j;
                atual->visitado = false;
                atual->listaAdjacencia = NULL;

                // Adicionar arestas para células adjacentes válidas
                if (i > 0 && labirinto[i - 1][j] != PAREDE) {
                    adicionarAresta(&(atual->listaAdjacencia), nos[i - 1][j]);
                }
                if (i < LINHAS - 1 && labirinto[i + 1][j] != PAREDE) {
                    adicionarAresta(&(atual->listaAdjacencia), nos[i + 1][j]);
                }
                if (j > 0 && labirinto[i][j - 1] != PAREDE) {
                    adicionarAresta(&(atual->listaAdjacencia), nos[i][j - 1]);
                }
                if (j < COLUNAS - 1 && labirinto[i][j + 1] != PAREDE) {
                    adicionarAresta(&(atual->listaAdjacencia), nos[i][j + 1]);
                }
            }
        }
    }
}
```

char labirinto[10][10]

3.5 LABIRINTO

Esta seção aborda as funções desenvolvidas para manipular e utilizar labirintos em conjunto com os algoritmos de busca em grafos.

As funções (Figuras 10 e 11) são responsáveis por ler o labirinto a partir de um arquivo e construir o grafo corresponde com base na estrutura fornecida.

- **lerLabirintoEConstruirGrafo**: Esta função lê o labirinto do arquivo especificado e, em seguida, constrói o grafo correspondente, representado pela matriz de vértices 'nos'.
- **buscarLabirinto**: Função responsável por executar a busca no labirinto utilizando um dos algoritmos disponíveis: busca em profundidade ou busca em largura.

- **entradaSaida**: Esta função percorre o labirinto e identifica os vértices de entrada e saída, atribuindo-os aos ponteiros 'entrada' e 'saída', respectivamente.

Figura 10: Funções entradaSaida e lerLabirintoEConstruirGrafo

```c
void entradaSaida(No** entrada,
            No** saida,
            char labirinto[LINHAS][COLUNAS],
            No* nos[LINHAS][COLUNAS]) {
    for(int i = 0; i < LINHAS; i++) {
            for(int j = 0; j < COLUNAS; j++) {
                if(labirinto[i][j] == ENTRADA) {
                    *entrada = nos[i][j];
                } else if(labirinto[i][j] == SAIDA) {
                    *saida = nos[i][j];
                }
            }
        }
        if (*entrada == NULL || *saida == NULL) {
            printf("Entrada ou saída não encontrada no labirinto.\n");
        return;
        }
}

void lerLabirintoEConstruirGrafo(char *nomeArquivo, char labirinto[LINHAS][COLUNAS], No* nos[LINHAS][COLUNAS]) {
    lerLabirinto(nomeArquivo, labirinto);
    criarGrafo(labirinto, nos);
}
```

Figura 11: Funções entradaSaida e lerLabirintoEConstruirGrafo

```c
void buscarLabirinto(char labirinto[LINHAS][COLUNAS], No* nos[LINHAS][COLUNAS], char *extensao, char *nomeArquivo, TipoBusca tipo)
    for(int i = 1; i <= 5; i++) {
        char *nomeArquivoCompleto = construirNomeArquivoCompleto(nomeArquivo, extensao, i);
        lerLabirintoEConstruirGrafo(nomeArquivoCompleto, labirinto, nos);

        No *entrada = NULL;
        No *saida = NULL;
        entradaSaida(&entrada, &saida, labirinto, nos);

        if (tipo == PROFUNDIDADE) {
            Pilha pilha;
            inicializarPilha(&pilha);
            empilhar(&pilha, entrada);
            entrada->visitado = false;

            buscaPorProfundidade(nos, entrada, saida, &pilha);
            escreverOutput(outputFileName, pilha);
        } else if (tipo == LARGURA) {
            Fila fila;
            inicializarFila(&fila);
            enfileirar(&fila, entrada, NULL);
            entrada->visitado = true;

            No* paiSaida = buscaPorLargura(nos, *entrada, *saida, &fila);
            escreverOutputFila(outputFileName, &fila, paiSaida);
        }

        free(nomeArquivoCompleto);
    }
}
```

4. ALGORITMOS

Para resolver o problema de encontrar o caminho entre a entrada e a saída do labirinto, foram empregados os algoritmos clássicos de busca em grafos: Busca Por Profundidade e Busca Por Largura. Ambos os algoritmos exploram o grafo de maneira sistemática, mas com diferentes estratégias, proporcionando diferentes resultados e desempenhos em determinados contextos.

Para ambos os algoritmos a busca é inicializada com a entrada já empilhada (no caso da busca por profundidade) ou enfileirada (no caso da busca por largura).

4.1 BUSCA POR PROFUNDIDADE

O código (Figura 12) implementa o algoritmo de busca em profundidade para encontrar um caminho do vértice de entrada ao vértice de saída em um grafo

representado pela matriz de vértices 'nos'. Ele utiliza uma abordagem recursiva, onde cada chamada da função explora um vértice não visitado e seus vizinhos. Se o vértice atual for o destino, a busca termina.

Caso contrário, o algoritmo continua explorando os vizinhos não visitados até encontrar o destino ou não houver mais vizinhos a serem explorados naquela ramificação. Se nenhum caminho for encontrado a partir do vértice atual, ele é removido da pilha para retroceder e explorar outras ramificações. O algoritmo termina quando todas as ramificações possíveis são exploradas ou quando o destino é alcançado.

Figura 12: Função para o algoritmo de Busca por Profundidade

```
void buscaPorProfundidade(No* nos[LINHAS][COLUNAS], No* entrada, No* saida, Pilha* pilha) {

    if(topoDaPilha(pilha) == saida) {
        return;
    } else {
        No* atual = topoDaPilha(pilha);
        Aresta* listaAdjacencia = atual->listaAdjacencia;
        while(listaAdjacencia != NULL) {
            if(listaAdjacencia->destino->visitado == 0) {
                listaAdjacencia->destino->visitado = 1;
                empilhar(pilha, listaAdjacencia->destino);
                buscaPorProfundidade(nos, entrada, saida, pilha);
            }
            listaAdjacencia = listaAdjacencia->proximaAresta;
        }
        if(topoDaPilha(pilha) != saida) {
            desempilhar(pilha);
        }
    }
}
```

Podemos analisar a complexidade desse algoritmo sob a visão de dois aspectos diferentes:

Complexidade de tempo: A complexidade de tempo é dada pelo número total de operações realizadas pelo algoritmo em relação ao tamanho da entrada. Nesse caso, a entrada seria o grafo representado pelos nós e arestas.

O loop while percorre as arestas adjacentes a cada nó visitado, portanto, em cada iteração, ele examina todas as arestas de um nó.

Se V é o número de vértices (nós) no grafo e E é o número de arestas, então o tempo de execução é O(V+E).

Complexidade de espaço: A complexidade de espaço refere-se à quantidade de memória necessária para executar o algoritmo.

O espaço necessário é principalmente devido à pilha de execução, que armazena os nós visitados durante o processo de busca. No pior caso, onde todos os nós do grafo são visitados, a pilha pode conter todos os vértices, resultando em uma complexidade de espaço de O(V).

4.2 BUSCA POR LARGURA

O código (Figura 13) implementa o algoritmo de busca em largura para encontrar um caminho do vértice de entrada ao vértice de saída em um grafo representado pela matriz de vértices 'nos'. Ele utiliza uma abordagem iterativa, onde cada vértice é explorado em camadas, começando pelo vértice de entrada e avançando para os vizinhos mais próximos antes de explorar os vizinhos mais distantes. O algoritmo continua até encontrar o destino ou até que não haja mais vértices a serem explorados na fila. Se o destino não for alcançado, o algoritmo retorna NULL, indicando que não foi possível encontrar um caminho até a saída.

Figura 13: Função para o algoritmo de Busca por Largura

```
No* buscaPorLargura(No* nos[LINHAS][COLUNAS], No entrada, No saida, Fila* fila) {
    while (!filaVazia(fila)) {
        No* atual = desenfileirar(fila); // Obtemos o próximo nó da fila

        // Verifica se o nó atual é a saída
        if (atual->linha == saida.linha && atual->coluna == saida.coluna) {
            return atual; // Achou o nó de saída
        }

        // Marca o nó atual como visitado
        atual->visitado = true;

        // Itera sobre os nós adjacentes
        Aresta* listaAdjacencia = atual->listaAdjacencia;
        while (listaAdjacencia != NULL) {
            No* adjacente = listaAdjacencia->destino;
            // Verifica se o nó adjacente não foi visitado e não é uma parede
            if (!adjacente->visitado) {
                // Marca o nó adjacente como visitado
                adjacente->visitado = true;
                // Enfileira o nó adjacente e seu nó pai
                enfileirar(fila, adjacente, atual);
            }
            listaAdjacencia = listaAdjacencia->proximaAresta;
        }
    }
    return NULL; // Não encontrou o caminho até a saída
}
```

Podemos analisar a complexidade desse algoritmo sob a visão de dois aspectos diferentes:

Complexidade de tempo: A complexidade de tempo é dada pelo número total de operações realizadas pelo algoritmo em relação ao tamanho da entrada, que é o grafo representado pelos nós e arestas.

No pior caso, onde todos os nós do grafo são visitados, cada nó e cada aresta são examinados uma vez. Portanto, a complexidade de tempo é O(V+E), onde V é o número de vértices (nós) no grafo e E é o número de arestas.

Complexidade de espaço: A complexidade de espaço refere-se à quantidade de memória necessária para executar o algoritmo.

Assim como na busca em profundidade, a maior parte do espaço é consumida pela fila, que armazena os nós visitados durante o processo de busca. No pior caso, onde todos os vértices são enfileirados, a complexidade de espaço é O(V).

Portanto, a complexidade do algoritmo de busca em largura é O(V+E) em termos de tempo e O(V) em termos de espaço, onde V é o número de vértices (nós) no grafo e E é o número de arestas.

5. CONCLUSÃO

O trabalho abordou a implementação e aplicação de algoritmos de busca em grafos para resolver o desafio de encontrar caminhos em labirintos. Inicialmente, foram exploradas as estruturas de dados necessárias para representar grafos, como listas de adjacência e matrizes. Em seguida, os algoritmos de busca por profundidade e por largura foram implementados, cada um com suas características únicas de exploração do espaço de busca.

A aplicação dos algoritmos foi realizada com sucesso, permitindo encontrar caminhos entre a entrada e a saída dos labirintos fornecidos. A busca por profundidade proporciona uma exploração em profundidade do labirinto, seguindo uma abordagem mais intensiva em determinadas áreas. Por outro lado, a busca por largura garante uma exploração mais abrangente do espaço, garantindo uma cobertura completa do labirinto.

Após analisar a complexidade de ambos os algoritmos de busca implementados foi possível notar que apresentam a mesma complexidade tanto para tempo quanto para espaço.

6. REFERÊNCIAS

Carvalho, Iago. Slides de Aula sobre Busca em Profundidade e Busca em Largura. Disponível em: https://github.com/iagoac/dce529/blob/main/slides/aula_13.pdf. Acesso em: 27 mar 2024.

Fischer, Ricardo. "Causalidade e Teoria dos Grafos: Uma Breve Reflexão." LinkedIn, Disponível em: https://www.linkedin.com/pulse/causalidade-e-teoria-dos-grafos-uma-brev%C3%ADssima-ricardo-fischer/?originalSubdomain=pt. Acesso em: 31 mar. 2024.

Tab News. "Uma Breve Introdução à Teoria dos Grafos." Disponível em: https://www.tabnews.com.br/UlisssesRosa/uma-breve-introducao-a-teoria-dos-grafos. Acesso em: 31 mar 2024.