

UNIVERSIDADE FEDERAL DE ALFENAS

**CAIO FERNANDO DIAS
FELIPE DE GODOI CORREA
MATHEUS REIS DE LIMA**

**TÍTULO: IMPLEMENTAÇÃO DE DIFERENTES ALGORITMOS DE BUSCA EM
GRAFOS SEM PESOS**

ALFENAS/MG

2024

**CAIO FERNANDO DIAS
FELIPE DE GODOI CORREA
MATHEUS REIS DE LIMA**

**TÍTULO: IMPLEMENTAÇÃO DE DIFERENTES ALGORITMOS DE BUSCA EM
GRAFOS SEM PESOS**

Trabalho apresentado à disciplina Algoritmos e Estruturas de Dados 3, do curso de Ciência da Computação, da Universidade Federal de Alfenas. Área de concentração: Ciências exatas.

ALFENAS/MG

2024

1. Introdução

Este trabalho prático possui como objetivo compreender e implementar diferentes algoritmos de busca em grafos sem pesos, além de praticar a modelagem de grafos como listas de adjacência ou matriz de adjacência, para encontrar o caminho entre a entrada e saída de um labirinto, desenvolvendo a atividade em C ou C++.

Dois algoritmos diferentes serão implementados para navegar por um labirinto de tamanho 10x10, em que o símbolo “E” representa a entrada, “S” a saída e “X” as paredes e “O” o caminho disponível no labirinto. As saídas serão padronizadas, impressas em duas colunas separadas por vírgula, indicando cada ponto do caminho entre a entrada e a saída.

```
XXXXX
000XX
0X00S
X00XX
X0XXX
```

Figura 1: exemplo de labirinto 5x5.

```
0,4
0,3
1,3
2,3
2,2
3,2
4,2
```

Figura 2: exemplo de como deve ser a saída para o labirinto.

2. Estrutura de dados

A estrutura de dados “Fila” foi utilizada para implementar o primeiro algoritmo (BFS (breadth-first search)) de busca em grafos sem peso e resolver o problema do labirinto, utilizando alocação dinâmica de memória em C++.

Fila é uma estrutura de dados dinâmica do tipo FIFO (First-In, First-Out), ou seja, o primeiro elemento a ser inserido será o primeiro a ser retirado. Cada nó (elemento) da fila contém um ponteiro para o próximo nó e um espaço para armazenar os dados.

A imagem a seguir expõe como essa estrutura de dados foi utilizada no algoritmo:

```

void FFVazia(Fila *f) {
    f->prim = (BlockB*) malloc (sizeof(BlockB));
    f->ult = f->prim;
    f->prim->prox = NULL;
}

void Enfileira(Fila *f, ItemB d) {
    f->ult->prox = (BlockB*) malloc (sizeof(BlockB));
    f->ult = f->ult->prox;
    f->ult->data = d;
    f->ult->prox = NULL;
}

void Desenfileira(Fila *f, ItemB *d) {
    if(f->prim == f->ult || f == NULL || f->prim->prox == NULL) {
        cout << "Erro: Fila vazia!" << endl;
        return;
    }
    BlockB *aux = f->prim->prox;
    f->prim->prox = aux->prox;
    if (f->prim->prox == NULL) {
        f->ult = f->prim;
    }
    *d = aux->data;
    free(aux);
}

bool VerificaFilaVazia(Fila *f) {
    return (f->prim == f->ult || f == NULL || f->prim->prox == NULL);
}

```

Figura 3: implementação da Fila no algoritmo.

Para implementar o segundo algoritmo (DFS (depth-first search)) de busca em grafos sem peso e resolver o problema do labirinto, foi utilizada a estrutura de dados “Pilha”.

Pilha é uma estrutura de dados dinâmica, assim como a fila, porém, com um princípio de remoção diferente, utiliza o tipo LIFO (Last-In-First-Out), logo, o primeiro objeto a ser inserido será o último a ser removido.

A imagem a seguir expõe como essa estrutura de dados foi utilizada no algoritmo:

```

void FPVazia(Pilha *p){
    p->base = new Block;
    p->top = p->base;
    p->base->prox = nullptr;
}

void Push(Pilha *p, Posicao d){
    Block *aux = new Block;
    aux->data = d;
    aux->prox = p->top;
    p->top = aux;
}

void Pop(Pilha *p, Posicao *d){
    Block *aux;
    if(p->base == p->top || p == nullptr){
        cout << "PILHA VAZIA!\n";
        return;
    }
    aux = p->top;
    p->top = aux->prox;
    *d = aux->data;
    delete aux;
}

bool PilhaVazia(Pilha *p){
    return (p->top == p->base);
}

```

Figura 4: implementação da Pilha no algoritmo.

3. Algoritmos

A Busca em Largura ou BFS (Breadth-First Search) foi o primeiro algoritmo utilizado para encontrar a saída do labirinto. O algoritmo começa por um vértice especificado pelo usuário, depois de visitá-lo, vai ao encontro de seus vizinhos, em seguida de todos os vizinhos dos vizinhos, até o fim.

Os vértices são numerados pelo algoritmo, pela ordem em que são visitados pela primeira vez, para tal, é utilizado uma fila de vértices. No início de cada iteração, a fila compreende vértices que já foram numerados com vizinhos que ainda faltam numerar e, no início da primeira iteração, a fila contém somente o primeiro vértice, com número 0.

A complexidade do algoritmo BFS é $O(|V| + |E|)$, em que V é o número de vértices, ou casas do labirinto, e E é o número de arestas.

A imagem a seguir expõe como o algoritmo foi utilizado:

```
// Função para realizar a busca em largura no labirinto
void BFS(Labirinto *lab) {
    Fila fila;
    FFVazia(&fila);

    int inicio_i, inicio_j;
    EncontrarEntrada(lab, &inicio_i, &inicio_j);

    ItemB inicio;
    inicio.val = inicio_i * MAXTAM + inicio_j;
    Enfileira(&fila, inicio);

    int direcoes[4][2] = {{-1, 0}, {0, -1}, {0, 1}, {1, 0}}; // cima, esquerda, direita, baixo

    // Vetor para armazenar o caminho percorrido
    vector<pair<int, int>> caminho;

    while (!VerificaFilaVazia(&fila)) {
        ItemB atual;
        Desenfileira(&fila, &atual);
        inicio_i = atual.val / MAXTAM;
        inicio_j = atual.val % MAXTAM;

        for (int d = 0; d < 4; d++) {
            int prox_i = inicio_i + direcoes[d][0];
            int prox_j = inicio_j + direcoes[d][1];

            if (prox_i >= 0 && prox_i < lab->tam && prox_j >= 0 && prox_j < lab->tam &&
                lab->matriz[prox_i][prox_j] != 'X') {
                if (lab->matriz[prox_i][prox_j] == 'S') {
                    // Encontrou a saída, armazene o caminho e retorne
                    caminho.push_back({prox_i, prox_j});
                    cout << "Saída encontrada!" << endl;

                    // Imprime o caminho encontrado
                    cout << "Caminho percorrido:" << endl;
                    for (const auto& p : caminho) {
                        cout << p.first << ", " << p.second << endl;
                    }
                    return;
                }
                lab->matriz[prox_i][prox_j] = 'X'; // Marcar como visitado
                ItemB prox;
                prox.val = prox_i * MAXTAM + prox_j;
                Enfileira(&fila, prox);
                caminho.push_back({prox_i, prox_j}); // Armazena o caminho
            }
        }
    }

    cout << "Saída não encontrada!" << endl;
}
```

Figura 5: função de Busca em Largura.

A imagem a seguir expõe saída do algoritmo para um labirinto proposto:

```
g++ -Wall -Wextra -Werror -Iinclude/ -o build/objects/src/Fila.o -c src/Fila.cpp
g++ -Wall -Wextra -Werror -Iinclude/ -o build/objects/src/main.o -c src/main.cpp
g++ -Wall -Wextra -Werror -Iinclude/ -lstdc++ -lm -o ./build/app ./build/objects/src/Fila.o ./bu
caiodias@caiodias-Lenovo-IdeaPad-S145-15IWL:~/Área de Trabalho/BFS$ make run
././build/app
X X 0 0 0 0 0 X X E
0 0 0 X X X X X X 0
0 X 0 0 0 0 X 0 0 0
0 X 0 X X 0 X 0 X 0
0 X 0 X X 0 0 0 X 0
0 X 0 X X 0 0 0 X 0
0 X 0 X X X X X X X
0 X 0 0 0 0 0 X X X
0 X X X X 0 X X X
0 0 0 X X 0 0 0 X X
S X X X X 0 X 0 X X

Saída encontrada!
Caminho percorrido:
0,9
2,9
2,8
3,0
2,7
4,9
3,7
4,7
4,6
4,5
3,5
2,5
2,4
2,3
2,2
1,2
3,2
0,2
1,1
4,2
0,3
1,0
5,2
0,4
2,0
6,2
0,5
3,0
6,3
0,6
4,0
6,4
5,0
6,5
6,0
6,6
7,0
7,6
8,0
8,6
8,1
9,0
caiodias@caiodias-Lenovo-IdeaPad-S145-15IWL:~/Área de Trabalho/BFS$
```

Figura 6: Saída BFS do labirinto 10x10

A Busca em Profundidade ou DFS (Depth-First Search) foi o segundo algoritmo utilizado para encontrar a saída do labirinto, é útil em problemas que precisam percorrer todas as soluções possíveis, seguindo uma direção até o fim antes de retroceder e explorar outros caminhos.

Para seu funcionamento é necessário que um vértice inicial seja escolhido para começar a exploração, marcado como visitado e colocado na pilha. Enquanto houver vértices na pilha, um vértice é retirado, que será o atual, todos os vértices adjacentes não visitados ao atual são explorados, cada vértice não visitado é marcado como o atual e colocado na pilha, virando o vértice atual. Se não houver outros vértices adjacentes não

visitados, ocorre o retrocesso para o último vértice visitado da pilha e a repetição do processo até que a pilha esteja vazia.

A complexidade do algoritmo DFS é o mesmo do BFS, ou seja, $O(|V| + |E|)$, em que V é o número de vértices, ou casas do labirinto, e E é o número de arestas.

A imagem a seguir expõe como o algoritmo foi utilizado:

```
bool DFS(Labirinto *lab, int x, int y, bool visited[][MAXTAM], Pilha *p){
    static int row[] = {-1, 0, 0, 1};
    static int col[] = {0, -1, 1, 0};

    Push(p, {x, y});
    visited[x][y] = true;

    while (!PilhaVazia(p)) {
        Posicao curr;
        Pop(p, &curr);

        cout << "(" << curr.x << ", " << curr.y << ")" << endl; // Processa o vértice visitado

        if (lab->matriz[curr.x][curr.y] == 'S') {
            return true; // Chegou ao destino
        }

        for (int i = 0; i < 4; ++i) {
            int newRow = curr.x + row[i];
            int newCol = curr.y + col[i];

            if (newRow >= 0 && newRow < lab->tam && newCol >= 0 && newCol < lab->tam &&
                lab->matriz[newRow][newCol] != 'X' && !visited[newRow][newCol]) {
                Push(p, {newRow, newCol});
                visited[newRow][newCol] = true;
            }
        }
    }

    return false; // Não encontrou o destino
}
```

Figura 7: função de Busca em Profundidade.

A imagem a seguir expõe saída do algoritmo para um labirinto proposto:

```
catodias@catodias-Lenovo-IdeaPad-S145-15IML:~/Área de Trabalho/BFS$ make clean
removido './build/objects/src/DFS.o'
removido './build/objects/src/main.o'
foi removido o diretório: './build/objects/src'
removido './build/app'
foi removido o diretório: './build/objects'
catodias@catodias-Lenovo-IdeaPad-S145-15IML:~/Área de Trabalho/BFS$ make
g++ -Wall -Wextra -Werror -Iinclude/ -o build/objects/src/DFS.o -c src/DFS.cpp
g++ -Wall -Wextra -Werror -Iinclude/ -o build/objects/src/main.o -c src/main.cpp
g++ -Wall -Wextra -Werror -Iinclude/ -lstdc++ -lm -o ./build/app ./build/objects/src/DFS.o ./build/objects/src/main.o
catodias@catodias-Lenovo-IdeaPad-S145-15IML:~/Área de Trabalho/BFS$ make run
././build/app
X X 0 0 0 0 0 X X E
0 0 0 X X X X X 0
0 X 0 0 0 0 X 0 0 0
0 X 0 X X 0 X 0 X 0
0 X 0 X X 0 0 0 X 0
0 X 0 X X X X X X
0 X 0 0 0 0 0 X X X
0 X X X X 0 X X X
0 0 0 X X 0 0 0 X X
5 X X X 0 X 0 X X

(0, 9)
(1, 9)
(2, 9)
(3, 9)
(4, 9)
(2, 8)
(2, 7)
(3, 7)
(4, 7)
(4, 6)
(4, 5)
(3, 5)
(2, 5)
(2, 4)
(2, 3)
(2, 2)
(3, 2)
(4, 2)
(5, 2)
(6, 2)
(6, 3)
(6, 4)
(6, 5)
(6, 6)
(7, 6)
(8, 6)
(8, 7)
(9, 7)
(8, 5)
(9, 5)
(1, 2)
(1, 1)
(1, 0)
(2, 0)
(3, 0)
(4, 0)
(5, 0)
(6, 0)
(7, 0)
(8, 0)
(9, 0)
Chegou ao destino!
catodias@catodias-Lenovo-IdeaPad-S145-15IML:~/Área de Trabalho/BFS$
```

Figura 8: Saída DFS do labirinto 10x10

Referências

<https://www.ime.usp.br/~pf/algoritmos/aulas/fila.html>

<https://www.ime.usp.br/~pf/algoritmos/aulas/pilha.html>

https://www.ime.usp.br/~pf/algoritmos_para_grafos/aulas/bfs.html

https://www.ime.usp.br/~pf/algoritmos_para_grafos/aulas/dfs.html