Trabalho Prático I - Saída do labirinto

Estudantes Ana Flávia Freiria Rodrigues, Lucas Carrijo Ferrari, Raissa Nunes Peret
Professor Iago Augusto de Carvalho

1 Introdução

O problema do labirinto é um desafio clássico em computação que envolve encontrar um caminho de um ponto de entrada a um ponto de saída em um ambiente labiríntico representado por uma matriz bidimensional. Neste contexto, o labirinto é composto por células que podem ser caminhos livres ou obstáculos (paredes). O objetivo é desenvolver um algoritmo que percorra o labirinto a partir da entrada (representada pelo caractere 'E') e encontre uma rota até a saída (representada pelo caractere 'S'), evitando paredes (representadas por 'X') e sem revisitar posições já exploradas.

Este problema é relevante pois envolve conceitos fundamentais de estruturas de dados e algoritmos, como pilhas, filas, listas e técnicas de busca, além de trabalhar com manipulação de matrizes e ponteiros. Resolver o labirinto de forma eficiente requer uma compreensão profunda das estruturas de dados adequadas e das estratégias de busca apropriadas para navegar pelo espaço de estados possíveis.

2 Estrutura de dados

Para a implementação do algoritmo de resolução do labirinto, foram utilizadas as seguintes estruturas de dados:

2.1 Estrutura `Posição`

```c
typedef struct Posicao {
    int linha;
    int coluna;
    struct Posicao *proxima;
} Posicao;
```

A estrutura `Posicao` representa uma posição individual no labirinto, identificada pelas coordenadas `linha` e `coluna`. O ponteiro `proxima` permite o encadeamento de múltiplas posições, formando assim uma lista ligada. Esta estrutura é fundamental para a implementação da pilha utilizada no algoritmo de busca.

2.2 Estrutura `Pilha`

```c
typedef struct Pilha {
    Posicao *topo;
} Pilha;
```

A estrutura `Pilha` é uma representação clássica de uma pilha baseada em lista ligada. O ponteiro `topo` referencia o elemento no topo da pilha, permitindo operações de empilhamento e desempilhamento de forma eficiente.

2.3 Matriz do Labirinto

```c
char labirinto[TAMANHO_LABIRINTO][TAMANHO_LABIRINTO];
```

A matriz bidimensional `labirinto` armazena a representação do labirinto, onde cada célula pode conter:

- `E`: Ponto de entrada.
- `S`: Ponto de saída.
- `0`: Caminho livre.
- `X`: Parede ou obstáculo.

2.4 Matriz de Visitados

```c
int visitado[TAMANHO_LABIRINTO][TAMANHO_LABIRINTO];
```

A matriz `visitado` é uma matriz bidimensional de inteiros que mantém o controle das posições já exploradas pelo algoritmo. Inicialmente, todas as células são definidas como não visitadas (`0`). Ao visitar uma posição, o algoritmo marca a célula correspondente como visitada (`1`), evitando assim revisitar posições e potencialmente entrar em loops infinitos.

3 Algoritmos

3.1 Visão Geral

A ideia central é começar na posição de entrada e explorar os caminhos disponíveis, movendo-se para posições adjacentes que sejam válidas (ou seja, dentro dos limites, não sejam paredes e não tenham sido visitadas). Se não houver mais movimentos possíveis a partir da posição atual, o algoritmo retrocede para a posição anterior, permitindo que caminhos alternativos sejam explorados. Esse processo continua até que a saída seja encontrada ou que todos os caminhos possíveis tenham sido examinados.

3.2 Passos do Algoritmo

3.2.1 Inicialização

- Identificação: Percorre o labirinto para localizar as coordenadas da entrada ('E') e da saída ('S').
- Configuração: Inicializa a pilha e uma matriz de controle para marcar posições visitadas.
- Empilhamento Inicial: Empilha a posição de entrada na pilha e a marca como visitada

3.2.2 Exploração do Labirinto

- Loop Principal: Enquanto a pilha não estiver vazia:
  - Posição Atual: Obtém a posição no topo da pilha, que é a posição atual.
  - Verificação de Saída: Se a posição atual for a saída, o algoritmo termina com sucesso.
  - Movimentação:
    - Tenta mover-se para cada uma das quatro direções possíveis: cima, baixo, esquerda e direita.
    - Para cada direção, calcula as novas coordenadas.
    - Validação do Movimento:
      - Verifica se a nova posição está dentro dos limites do labirinto.
      - Confirma que a posição não é uma parede ('X') e não foi visitada anteriormente.
    - Avanço:
      - Se a posição for válida, empilha a nova posição na pilha e a marca como visitada.
      - Interrompe a busca de movimentos para continuar a exploração a partir da nova posição.
  - Retrocesso
    - Se não houver movimentos válidos a partir da posição atual, desempilha a posição (retrocede) para explorar caminhos alternativos.

3.2.3 Construção e Impressão do Caminho

- Caminho Encontrado: Se a saída for encontrada, as posições na pilha representam o caminho seguido.
- Registro do Caminho:
  - Percorre a pilha do topo até a base para armazenar as posições visitadas em ordem.
- Impressão:
  - Imprime as coordenadas do caminho encontrado, ajustando-as para que (0,0) seja o canto inferior esquerdo do labirinto.

3.3 Funções Principais

3.3.1 movimentoValido

Esta função verifica se uma posição adjacente é válida para ser explorada.

```c
int movimentoValido(char labirinto[][TAMANHO_LABIRINTO], int visitado[][TAMANHO_LABIRINTO], int linha, int coluna) {
    // Verifica se esta dentro dos limites do labirinto
    if (linha < 0 || linha >= TAMANHO_LABIRINTO || coluna < 0 || coluna >= TAMANHO_LABIRINTO) {
        return 0;
    }
    // Verifica se a posicao nao e uma parede e nao foi visitada
    if ((labirinto[linha][coluna] == '0' || labirinto[linha][coluna] == 'S') && !visitado[linha][coluna]) {
        return 1;
    }
    return 0;
}
```

3.3.2 Operações com a Pilha

- Inicialização da Pilha: Prepara a pilha para uso.

```c
void inicializarPilha(Pilha *pilha) {
    pilha->topo = NULL;
}

```

- Empilhar (empilhar): Adiciona uma nova posição ao topo da pilha.

```c
void empilhar(Pilha *pilha, int linha, int coluna) {
    Posicao *novaPosicao = (Posicao *)malloc(sizeof(Posicao));
    novaPosicao->linha = linha;
    novaPosicao->coluna = coluna;
    novaPosicao->proxima = pilha->topo;
    pilha->topo = novaPosicao;
}

```

- Desempilhar (desempilhar): Remove a posição do topo da pilha, permitindo o retrocesso.

```c
void desempilhar(Pilha *pilha) {
    if (estaVazia(pilha)) {
        return;
    }
    Posicao *temp = pilha->topo;
    pilha->topo = pilha->topo->proxima;
    free(temp);
}

```

- Verificar se a Pilha está Vazia

```c
int estaVazia(Pilha *pilha) {
    return pilha->topo == NULL;
}

```

- Obter o Topo da Pilha

```c
Posicao *topo(Pilha *pilha) {
    if (estaVazia(pilha)) {
        return NULL;
    }
    return pilha->topo;
}

```

3.3.3 imprimirCaminho

Esta função imprime o caminho encontrado, ajustando as coordenadas conforme necessário.

```c
void imprimirCaminho(Pilha *pilha) {
    // Array para armazenar as posicoes do caminho
    Posicao *caminho[TAMANHO_LABIRINTO * TAMANHO_LABIRINTO];
    int contador = 0;

    // Transferir as posicoes da pilha para o array
    Posicao *atual = pilha->topo;
    while (atual != NULL) {
        caminho[contador++] = atual;
        atual = atual->proxima;
    }

    // Imprimir as posicoes em ordem reversa (do inicio ao fim)
    for (int i = contador - 1; i >= 0; i--) {
        // Ajustar coordenadas: x = coluna, y = linha invertida
        int x = caminho[i]->coluna;
        int y = TAMANHO_LABIRINTO - 1 - caminho[i]->linha;
        printf("%d,%d\n", x, y);
    }
}
```

3.4 Análise de Complexidade

Em relação ao tempo:

A complexidade de tempo depende diretamente do número de posições do labirinto e da forma como o algoritmo explora as células. No pior caso, o algoritmo visita cada célula uma vez, resultando em uma complexidade de tempo de $O(\text{TAMANHO\_LABIRINTO}^2)$, ou seja, $O(n^2)$ para um labirinto de dimensões $n \times n$. Cada operação de empilhar ou desempilhar na pilha ocorre em tempo constante $O(1)$, mas como essas operações são realizadas para cada célula visitada, o tempo total também está limitado por $O(T^2)$, onde T é o tamanho do labirinto.

Em relação a espaço:

A complexidade de espaço depende de duas estruturas principais:

- A pilha, que no pior caso pode armazenar todas as posições do labirinto, ocupando $O(n^2)$ de espaço.
- O array visitado, que também tem tamanho $O(T^2)$.

Portanto, a complexidade de espaço é $O(\text{TAMANHO\_LABIRINTO}^2)$, ou seja, $O(T^2)$ para armazenar tanto o labirinto quanto as posições visitadas e a pilha.

3.4.1 Considerações sobre o Algoritmo

1. **Inicialização e Estruturas de Dados**
   - O código define uma estrutura de pilha baseada em uma lista encadeada, onde cada nó da pilha armazena uma posição no labirinto (coordenadas de linha e coluna).
   - A função `inicializarPilha` tem complexidade $O(1)$, pois apenas define o topo da pilha como `NULL`.
2. **Procura pelas posições inicial ('E') e final ('S')**
   - O labirinto é percorrido para encontrar as posições de início ('E') e fim ('S'). Isso ocorre em um laço aninhado de duas dimensões, iterando sobre todas as células do labirinto.
   - Como o tamanho do labirinto é $T \times T$, neste caso, $T = 10$, a complexidade dessa parte é $O(T^2)$ ou $O(100)$ no pior caso, pois o labirinto é completamente percorrido.
3. **Empilhar, Desempilhar e Verificar o Topo da Pilha**
   - As funções `empilhar`, `desempilhar` e `topo` são todas operações que atuam em uma lista encadeada.
     - `empilhar`: A função aloca dinamicamente uma nova posição e a coloca no topo da pilha. A alocação e inserção ocorrem em tempo constante, então a complexidade é $O(1)$.
     - `desempilhar`: Remove o topo da pilha e libera a memória, operação feita em tempo constante, logo, a complexidade é $O(1)$.

- `topo`: Retorna a posição no topo da pilha sem removê-la, outra operação que ocorre em tempo constante, com complexidade $O(1)$.

4. **Busca em Profundidade (DFS)**

   - A parte central do código implementa uma busca em profundidade com o uso da pilha, percorrendo o labirinto para encontrar um caminho da posição 'E' até 'S'.
   - A cada passo, o algoritmo verifica se a célula atual é a saída ('S'), caso contrário, tenta se mover em uma das quatro direções possíveis (cima, baixo, esquerda, direita).

5. **Movimentos Válidos**

   - A função `movimentoValido` verifica se o movimento proposto é válido:
     (a) Checa se a nova posição está dentro dos limites do labirinto.
     (b) Verifica se a célula não foi visitada e não é uma parede ('X').
   - Cada chamada de `movimentoValido` tem complexidade $O(1)$, pois verifica apenas um valor em uma matriz 10×10.

6. **Complexidade da DFS**

   - No pior caso, o algoritmo pode precisar visitar todas as células acessíveis do labirinto.
   - Para cada célula acessível, a DFS empilha ou desempilha posições e verifica as direções possíveis, então cada célula pode ser visitada e empilhada/desempilhada no máximo uma vez.
   - Se o labirinto for de tamanho $T \times T$, a complexidade da busca em profundidade é $O(T^2)$, porque todas as células podem ser visitadas no pior caso.

7. **Impressão do Caminho**

   - Após encontrar a saída, o código armazena todas as posições da pilha em um array auxiliar e imprime o caminho na ordem inversa.
   - Copiar as posições da pilha para o array tem complexidade proporcional ao número de posições no caminho, que pode ser até $O(T^2)$ no pior caso.
   - A impressão do caminho, por sua vez, tem complexidade linear em relação ao número de posições armazenadas, então também é $O(T^2)$ no pior caso.

8. **Conclusão da Complexidade**

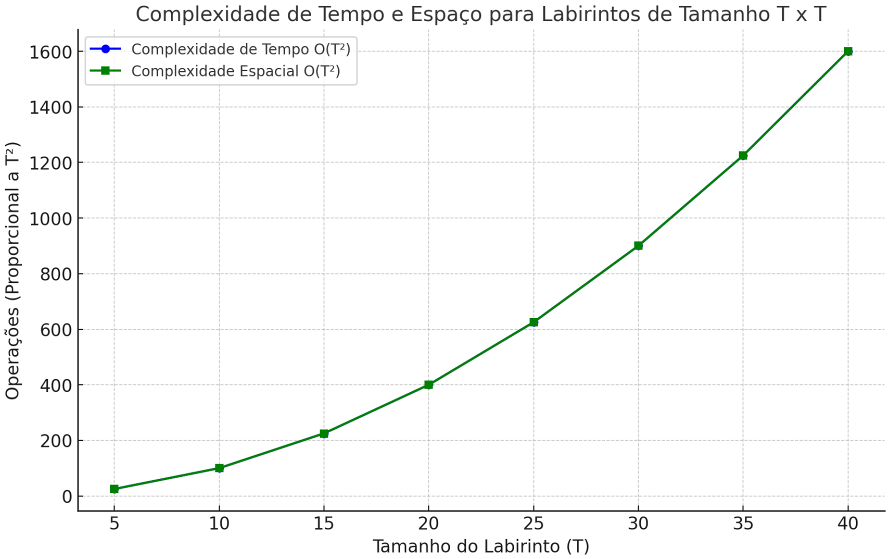
   - A maior parte do custo computacional está na execução da busca em profundidade e na verificação das células do labirinto.
   - A complexidade total do algoritmo é dominada pela DFS, e portanto a complexidade de tempo é $O(T^2)$, onde T é o tamanho do labirinto.
   - Para o labirinto de tamanho 10×10, o valor de T é fixo em 10, então a complexidade seria $O(100)$.

9. **Complexidade Espacial**

   - A complexidade espacial é dominada pelo uso da pilha e da matriz `visitado`:
     - A pilha pode armazenar até $O(T^2)$ posições no pior caso (se o caminho ocupar todo o labirinto).
     - A matriz `visitado` tem tamanho $O(T^2)$.
   - Portanto, a complexidade espacial também é $O(T^2)$.

Resumo:

- Complexidade de Tempo: $O(T^2)$, devido à busca em profundidade no labirinto.
- Complexidade Espacial: $O(T^2)$, principalmente pela matriz de visitados e a pilha.



Em resumo, a complexidade de tempo e espaço do algoritmo é $O(n^2)$, sendo n o tamanho de um dos lados do labirinto. Esta complexidade está associada ao fato de que, no pior cenário, o algoritmo pode explorar todas as células antes de encontrar a saída ou determinar que não há solução.

4 Descrição MakeFile