Introdução
O problema de encontrar a saída de um labirinto é uma questão envolvendo a exploração de um espaço delimitado por paredes e caminhos livres. O objetivo é traçar uma rota da entrada até a saída de forma eficiente, lidando com o desafio de evitar becos sem saída e descobrir o trajeto correto. Um labirinto é representado por uma matriz de 10x10, onde cada célula pode conter um obstáculo ("X"), um caminho livre ("0"), a entrada ("E") ou a saída ("S"). O desafio é percorrer essa matriz de forma sistemática, encontrando um caminho que conecte a entrada à saída e imprimindo esse caminho em um formato padronizado, especificando as coordenadas das casas visitadas.

Estruturas de dados utilizadas

Para a resolução do problema foram utilizadas as estruturas de dados pilha duplamente encadeada para armazenar o caminho correto e uma lista duplamente encadeada para a exibição dos caminhos,onde cada elemento na estrutura possui os ponteiros “ant” e “prox” que são utilizados para referenciar a posição enquanto ocorre a execução do algoritmo.

Sobre o algoritmo

Este algoritmo implementa a busca por um caminho em um labirinto de 10x10 representado por uma matriz. Ele utiliza uma pilha duplamente encadeada para armazenar e gerenciar os movimentos realizados durante a exploração do labirinto, permitindo voltar a movimentos anteriores quando necessário. A exploração do labirinto é feita testando as possíveis jogadas (direções) a partir da posição atual, até encontrar a saída ("S") ou esgotar as opções.

Estrutura do labirinto:

`"E"` representa a entrada.

`"S"` representa a saída.

`"X"` representa uma parede, que impede o movimento.

`"0"` representa um caminho livre por onde o algoritmo pode se mover.

Etapas do Algoritmo

1. **Inicialização**: O algoritmo começa inserindo na pilha a posição da entrada (representada por `"E"` no labirinto). Esta posição é o ponto de partida da exploração.
2. **Movimentação**: A cada passo, o algoritmo verifica as possíveis direções de movimento a partir da posição atual, que são:
    - **Direita** (movimento [0,1])
    - **Baixo** (movimento [1,0])
    - **Esquerda** (movimento [0,-1])
    - **Cima** (movimento [-1,0])

    Essas jogadas são testadas em ordem “circular”, e o algoritmo seleciona a primeira direção válida (que não ultrapasse os limites do labirinto e não colida com uma parede `"X"`).

3. **Função de Inserção na Pilha(mov *inserir_pilha)**: Se o movimento é válido, o algoritmo insere essa nova posição na pilha, armazenando também a direção da jogada para que, caso precise voltar a um estado anterior, possa continuar testando as direções restantes.
4. **Função de remover da Pilha(mov *remover_pilha)**: simplesmente desempilha. Utilizado para os casos de que o caminho até então empilhado é incorreto.
5. **Função de encontrar encontrar caminho(void encontrar_caminho)**:algoritmo principal onde as funções de inserção e remoção da pilha são aplicadas e toda a lógica para buscar o caminho é usada,onde a execução do algoritmo se mantém por mais tempo(princípio da localidade),usando a técnica de "retrocesso"**.**A partir disso, todas as direções possíveis a partir da posição atual serão testadas e, caso não tenha sucesso, o algoritmo remove o topo da pilha, retornando à posição anterior para tentar outros caminhos. Esse processo de "backtracking" continua até encontrar a saída ou até que todas as possibilidades sejam esgotadas.
6. **Finalização**: Se o algoritmo encontra a saída `"S"`, ele percorre a pilha (que contém o caminho da entrada até a saída) e imprime as coordenadas das posições visitadas no formato especificado. Caso contrário, ele imprime uma mensagem indicando que o labirinto não possui uma saída.

Complexidade do Algoritmo

A complexidade deste algoritmo pode ser analisada em termos de tempo e espaço:

Complexidade de Tempo:

O algoritmo explora cada casa do labirinto uma vez, então a complexidade de tempo no pior caso é proporcional ao número total de células, ou seja, O(linha x coluna).

Complexidade de Espaço:

A pilha guarda os movimentos, e no pior cenário, pode conter até todas as casas do labirinto, o que resulta em uma complexidade de espaço O(linha x coluna), já que cada movimento é armazenado como um nó da pilha.

SOBRE O MAKEFILE

PROJ_NAME=labirinto: Define o nome do arquivo executável.

C_SOURCE=$(wildcard *.c): armazena todos os "arquivos.c" presentes no diretório e colocados uma variável chamada C_SOURCE.

H_SOURCE=$(wildcard *.h): armazena todos os "arquivos.h" presentes no diretório e colocadps em variável chamada H_SOURCE.

OBJ=$(C_SOURCE:.c=.o): armazena todos os "arquivos.o" que são convertidos a partir dos "arquivos.c" pelo processo de compilação.

CC=gcc: especifica o compilador que está sendo utilizado.

CC_FLAGS: ajusta as opções do compilador:

"-c":compila os aquivos fonte sem utilizar a linkagem.

“02”: otimiza o código.

“Wall”: avisos para possíveis problemas no código.

“wextra”: avisos adicionais para garantir a qualidade do código.

all: $(PROJ_NAME) clean: Esse é o alvo principal do Makefile. Ele especifica que, ao rodar o comando `make`, os arquivos serão compilados, o executável será gerado e, em seguida, o comando clean será executado para limpar os arquivos temporários (arquivos `.o`).

$(PROJ_NAME): $(OBJ): alvo para criar o arquivo executável final,ao qual depende dos “arquivos .o”. O comando “`$(CC) -o $@ $^”` compila e cria o executável.

`$@`: nome do alvo atual, que é `labirinto` (`PROJ_NAME).`

`$^`: todos os “arquivos.o” necessários (`OBJ`).

%.o: %.c %.h: alvo com os arquivos `.c` e `.h` que serão compilados em “arquivos`.o”`. O comando “`$(CC) -o $@ $< $(CC_FLAGS)”` compila:

`$@`: Nome do arquivo de saída (arquivo `.o`).

`$<`: Nome do primeiro arquivo de dependência (arquivo `.c`).

main.o: main.c $(H_SOURCE): Compila o arquivo “`main.c”`, considerando todos os “arquivos`.h”` listados em `H_SOURCE` como suas respectivas dependências.

clean: comando para limpar todos os “arquivos.o”.

Como usar o MakeFile:

-Para compilar o programa, digitar: make all
-Para executar: ./labirinto x (onde x é o número do labirinto escolhido). Para fazer a limpeza: make clean

Conclusão:O algoritmo desenvolvido para resolver o labirinto funcionou bem no cenário de um labirinto 10x10. O código é flexível graças ao uso de alocação dinâmica e tratamento de arquivos, além de ser dividido em várias funções para facilitar sua manutenção e expansão.

Em resumo, o projeto atingiu o objetivo de encontrar a saída do labirinto, aplicando conceitos importantes de estruturas de dados e algoritmos.