UNIVERSIDADE FEDERAL DE ALFENAS

RELATÓRIO: "DESENVOLVIMENTO DE ALGORITMOS PARA A SOLUÇÃO DE LABIRINTOS"

Alunos:

Lucas Lopes Baroni (2024.1.08.017)

João Antônio Siqueira Pascuini (2024.1.08.028)

Alfenas, 2024

1. Introdução

Esse trabalho acadêmico busca construir um algoritmo utilizando a linguagem de programação C para encontrar, caso seja possível, um caminho que leva à saída de um labirinto de tamanho 10x10.

A aplicação construída para esse fim, recebe como entrada um arquivo texto (.txt) com um conjunto de 100 caracteres (cada caractere representa uma posição do labirinto). Foi previamente definido no escopo do problema que o caractere ‘E’ representa a entrada, ‘0’ representa o caminho, ‘X’ denota uma parede e ‘S’ indica a saída. Se o labirinto for processado com êxito pelo programa, o caminho necessário para solucioná-lo é disponibilizado no terminal, no formato (coluna, linha) para cada posição que deve ser percorrida.

2. Estrutura de dados

Para solucionar o problema descrito acima, utilizou-se algumas estruturas de dados, dentre elas: pilha encadeada e vetores bidimensionais. A pilha foi utilizada para gravar o caminho percorrido da entrada até a saída, por sua vez, os vetores bidimensionais foram usados para armazenar o labirinto presente no arquivo. Confira abaixo mais detalhes acerca dessas estruturas.

- **Pilha encadeada**: é uma estrutura de dados no qual o último elemento a ser adicionado é o primeiro que deve ser removido. A manipulação dessa estrutura é similar a uma pilha de pratos, uma vez que, para acessar o primeiro prato é necessário remover todos os outros que estão acima. A pilha escolhida foi alocada dinamicamente, visto que teoricamente, podem existir caminhos compostos por muitas ou poucas posições, dessa maneira, torna-se impreciso definir um tamanho máximo para essa estrutura.

- **Vetores bidimensionais**: é uma estrutura sequencialmente alocada na memória que possui a finalidade de guardar todas as posições do labirinto e seus respectivos caracteres na memória principal. Já que existem linhas e colunas no labirinto, torna-se necessário construir um vetor bidimensional para guardar essas informações. O

tamanho máximo dessa estrutura é igual a $\eta \cdot \eta$, $\eta \in \mathbb{N}$ (η é o número de linhas e colunas).

É importante ressaltar que cada elemento da pilha corresponde a uma estrutura do tipo *pathEl,* que possui os campos *line, column* e *next.* Por sua vez, cada posição do vetor bidimensional corresponde a uma variável do tipo *char*.

3. Algoritmos

Para desenvolver esse software, utilizamos muitos algoritmos distintos, cada um especializado em uma função específica. A seguir, explicaremos o objetivo e o funcionamento de cada método implementado na aplicação.

- **Código labyrinthLib.c:**
    - **labyrinth readLabyrinthFile(char fileName[])**: é um método que possui a finalidade de ler um arquivo do tipo texto (.txt) e gravar o labirinto presente nele em um vetor bidimensional presente na estrutura *labyrinth*. Essa função chama um método interno da biblioteca chamado *checkLabyrinthFile(char fileName[])* que verifica se o arquivo está nos padrões exigidos pelo sistema. Caso a verificação e a leitura sejam concluídos com êxito, a função retorna um labirinto alocado na memória principal.
        - **Análise de complexidade da função:** esse método depende diretamente da quantidade de posições presentes no labirinto, visto que, cada caracter presente no arquivo será inserido no vetor bidimensional. Dessa maneira, conclui-se que a complexidade dessa função é $O(N^2)$, sendo N a quantidade de caracteres no labirinto.
    - **void findLabEnter(labyrinth lab, int* line, int* column)**: é um método que percorre a matriz correspondente ao labirinto e procura o caractere 'E', que simboliza seu início. Ao encontrá-lo, os índices $i, j \in \mathbb{N} \mid i < 10, j < 10$ são gravados respectivamente nas variáveis *line* e *column*.

- **Análise de complexidade da função:** esse método depende diretamente da quantidade de posições presentes no labirinto, visto que, no pior caso, essa função percorrerá todas as posições do labirinto para encontrar a entrada, denotada pelo caracter 'E'. Dessa maneira, conclui-se que a complexidade dessa função é $O(N^2)$, sendo N a quantidade de posições no labirinto.

- **void showLabyrinth(labyrinth lab):** é um método que percorre um labirinto e mostra o caractere presente em cada posição no terminal.
  - **Análise de complexidade da função:** esse método depende diretamente da quantidade de posições presentes no labirinto, visto que, cada caractere da estrutura será exibido no terminal. Dessa maneira, conclui-se que a complexidade dessa função é $O(N^2)$, sendo N a quantidade de caracteres no labirinto.

- **stack* findLabPath(labyrinth lab)**: Esse é o principal método do sistema, responsável por encontrar o caminho até a saída e armazená-lo em uma pilha. Considere $P_{i,j}$ cada posição existente no labirinto. As equações presentes a seguir determina a próxima posição a ser acessada se e somente se $i, j \in \mathbb{N} \mid i < 10, j < 10$:

$$i = i_\theta + k_\alpha; \qquad k_\alpha = (\sum_{i=1}^{\alpha} 1) - 1; \qquad \alpha = \{0, 1, 2\}$$

$$j = j_\theta + l_\beta; \qquad l_\beta = (\sum_{i=1}^{\beta} 1) - 1; \qquad \beta = \{0, 1, 2\}$$

i_θ, j_θ são os valores iniciais dos índices i e j respectivamente, que em um primeiro caso, recebe os índices da entrada do labirinto, posteriormente, essas variáveis recebem os índices da posição atual.

A lógica da função calcula i e j até encontrar uma posição $P_{i,j}$ que possui o caractere '0', que simboliza um caminho válido. Se for encontrada uma posição que satisfaz a premissa anterior e ela não estiver presente na

pilha, a posição encontrada é adicionada na estrutura, os valores de i_θ, j_θ são atualizados e o ciclo de *loop* se repete.

Caso contrário, se para todo $\alpha = \{0, 1, 2\}$ e $\beta = \{0, 1, 2\}$ existir uma posição $P_{i,j}$ com caractere '0' que está presente na pilha, isso significa que a partir da posição atual não existe uma outra opção além de voltar para a casa anterior, simbolizando, dessa forma, que naquele instante o algoritmo está em um "beco sem saída". Quando isso ocorre, a posição atual recebe o caractere '1', simbolizando um caminho inválido e é removida da pilha. Além disso, as variáveis i_θ, j_θ são atualizadas com os índices da posição anterior e o ciclo de *loop* se repete.

Dessa maneira, esse sistema funcionará até que a posição $P_{i,j}$ tenha o caractere 'S' que simboliza a saída.

- **Análise de complexidade da função:** a complexidade desse método é um pouco mais complexa de ser analisada, por isso, dividiremos a análise dela em partes.

  Primeiramente, a função *findLabEnter(labyrinth lab, int* line, int* column)* é chamada dentro desse método, dessa maneira, a complexidade no pior caso inicia-se com $O(N^2)$, sendo N a quantidade de caracteres no labirinto.

  Durante o processo de encontrar o caminho, no pior dos casos (figura 1), o algoritmo precisará acessar $N^2 - 3$ posições para concluir sua operação. A fim de analisar a complexidade, podemos arredondar $N^2 - 3$ para N^2 , assim a complexidade para encontrar o caminho é $O(N^2)$.

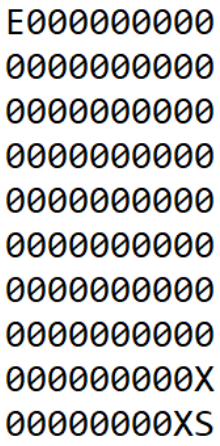

Figura 1 – Pior caso de labirinto possível.

Fonte: Elaborado pelo autor.

Dessa forma, a complexidade total desse método seria equivalente a $O(N^2) + O(N^2) = 2.O(N^2)$. Constantes podem ser desprezadas durante a análise de algoritmos, por isso a complexidade final da função é próxima de $O(N^2)$.

- **Código stackLib.c:**

  - **stack* createStack(void):** é um método que aloca e configura uma estrutura de dados do tipo pilha e retorna seu endereço na memória.
    - **Análise de complexidade da função:** a complexidade desse método não varia ao longo do tempo, sendo uma função constante, $O(1)$.

  - **int addEl(stack* list, pathEl* element):** adiciona, caso seja possível, um elemento na pilha e retorna o status da operação.
    - **Análise de complexidade da função:** a complexidade desse método não varia ao longo do tempo, sendo uma função constante, $O(1)$.

  - **pathEl* removeEl(stack* list):** remove, caso seja possível, um elemento na pilha e retorna o endereço dele.
    - **Análise de complexidade da função:** esse método depende diretamente da quantidade de elementos presentes na pilha, de maneira mais precisa, ele percorre N - 1, sendo N a quantidade de elementos na

pilha. Por isso, pode-se concluir que a complexidade desse algoritmo é próxima de $O(N)$.

- **pathEl* searchEl(stack* list, int line, int column):** esse método possui a finalidade de buscar determinado elemento na pilha. Caso ele seja encontrado, o endereço dele é retornado.
  - **Análise de complexidade da função:** esse método depende, no pior dos casos, diretamente da quantidade de elementos presentes na pilha, visto que, a finalidade dele é buscar por um elemento. A função percorre N, sendo N a quantidade de elementos na pilha. Por isso, pode-se concluir que a complexidade desse algoritmo é de $O(N)$.

- **void showStack(stack list):** é uma função que percorre a pilha e mostra seus elementos no terminal.
  - **Análise de complexidade da função:** esse método depende diretamente da quantidade de elementos presentes na pilha, visto que, a finalidade dele é exibir os elementos no terminal. A função percorre N, sendo N a quantidade de elementos na pilha. Por isso, pode-se concluir que a complexidade desse algoritmo é de $O(N)$.

- **void deleteStack(stack *list):** é uma função que percorre a pilha e libera a memória ocupada por seus elementos.
  - **Análise de complexidade da função:** esse método depende diretamente da quantidade de elementos presentes na pilha, visto que, a finalidade dele é removê-los da memória principal. A função percorre N, sendo N a quantidade de elementos na pilha. Por isso, pode-se concluir que a complexidade desse algoritmo é de $O(N)$.

4. Descrição do Makefile

O Makefile foi projetado para automatizar o processo de compilação do jogo do labirinto em C. Ele realiza a geração de arquivos objeto e a criação do executável final, além de possibilitar a limpeza dos arquivos gerados durante o processo de compilação. A seguir está uma análise detalhada de cada parte do arquivo:

- **Variáveis:**

    - **PROJ_NAME**: Define o nome do projeto, que será utilizado para gerar o executável final. No caso, *"labyrinthGame"*.

    - **C_SOURCE**: Utiliza o comando *wildcard* para capturar todos os arquivos *.c* localizados na pasta "*./source/"*. Ou seja, todos os arquivos de código-fonte em C usados no projeto.

    - **H_SOURCE**: Semelhante a *C_SOURCE*, esta variável captura todos os arquivos de cabeçalho (.h) na mesma pasta.

    - **OBJ**: Esta variável transforma os arquivos "*.c"* em seus respectivos arquivos objeto (.o) e altera o caminho de origem de *"./source/"* para *"./objects/"*, onde os arquivos objeto serão armazenados.

    - **CC**: Define o compilador que será usado, no caso, o *GCC (GNU Compiler Collection).*

    - **CC_FLAGS**: Contém os parâmetros que serão passados ao compilador:
        - *-c:* Compila o código sem vinculá-lo (gera arquivos objeto).
        - *-W, -Wall*: Habilita a exibição de warnings.
        - *-ansi*: Enforce a conformidade com o padrão ANSI C.
        - *-pedantic*: Habilita avisos sobre construções de linguagem que não são estritamente padrão.

    - **RM**: Define o comando para remover arquivos (usado na regra *clean*).

- **Regras:**

    - **all:** Esta é a regra padrão que será executada ao chamar *make*. Ela invoca a criação da pasta *objects* (regra *objFolder*) e compila o executável do projeto.

- **$(PROJ_NAME):** Compila todos os arquivos objeto para gerar o binário final (o executável). O "*$(CC)"* é utilizado para vincular os arquivos e o resultado final será o binário "labyrinthGame". O uso de '@' antes dos comandos faz com que eles não sejam impressos no terminal, a menos que ocorra um erro.

- **./objects/%.o:** Esta regra define como os arquivos "*.c"* serão compilados em arquivos objeto *.o.* Cada arquivo "*.c"* e seu respectivo "*.h"* são compilados separadamente.

- **./objects/main.o**: Regra específica para o arquivo *main.c*, que depende de todos os arquivos de cabeçalho. Esta regra segue um processo semelhante à anterior, mas é tratada separadamente devido à sua importância na execução do programa.

- **objFolder:** Cria o diretório "*objects/"* se ele ainda não existir, para armazenar os arquivos objeto.

- **clean:** Remove todos os arquivos "*.o"* e o binário gerado, limpando o diretório do projeto.

- **Explicação do funcionamento:**

Ao executar o comando *"make"*, o Makefile segue as dependências definidas. Primeiro, ele verifica se a pasta *"objects/"* existe. Se não existir, a cria. Em seguida, compila os arquivos *".c"* em seus respectivos arquivos objeto *".o"* e, por fim, vincula os arquivos objeto para criar o binário *"labyrinthGame"*.

O comando *"make clean"* pode ser usado para remover os arquivos gerados durante a compilação, garantindo uma "limpeza" no diretório.