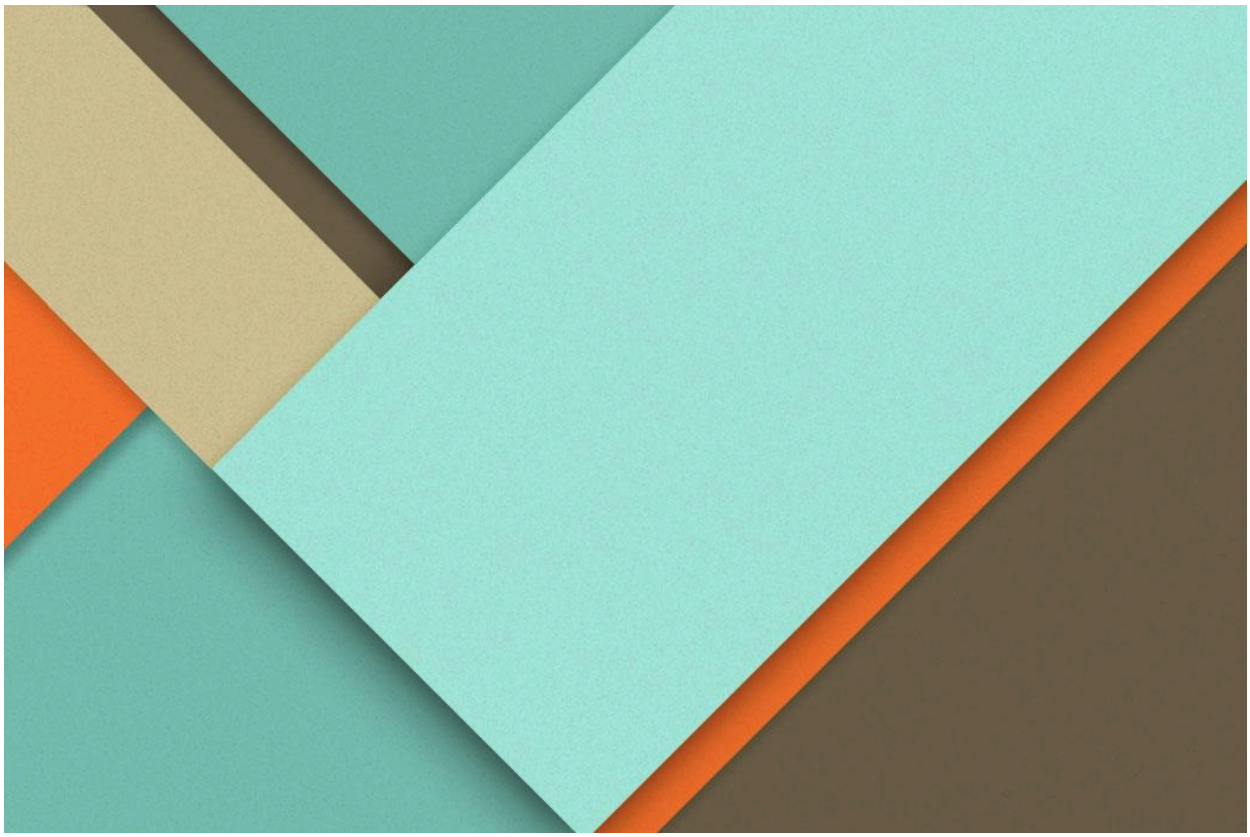

Trabalho de AEDS2_Prática

Universidade Federal de Alfenas - UNIFAL

20/09/2024

Discentes:

Dalton D’Angelis Sacramento 2024.1.08.005

Enzo Maranezi 2024.1.08.008

Docente:

Iago Augusto Carvalho

Introdução

O objetivo do nosso trabalho é implementar um algoritmo para explorar um labirinto de tamanho 10x10, que possui entrada **"E"**,saída **"S"** e paredes **"X"**.

Estruturas de Dados

As estruturas de dados utilizadas foram a pilha encadeada e a busca em profundidade. A primeira terá como objetivo armazenar as coordenadas de cada posição do labirinto ao decorrer da execução do código, enquanto a busca em profundidade objetiva encontrar a saída, sendo caracterizada por explorar o caminho até o fim antes de voltar e tentar outras alternativas.

Constantes

Para começar a resolver o problema, é necessário primeiro pensar em como resolvê-lo em partes, portanto, para começar, inicializamos duas variáveis (N e M) como 10, que serão, respectivamente, as linhas e as colunas do labirinto, que vai ser lido como uma matriz.

```
// Tamanho das linhas e colunas da matriz
const int N=10;
const int M=10;
```

Structs

A estrutura **"no"**, que possui dois valores inteiros (x,y) e um struct **"no"** com um ponteiro para o **"prox"**, ou seja, uma estrutura de alocação dinâmica que será utilizada para representar uma pilha, já a estrutura **"pos"** representa uma posição do labirinto, e guarda nela um caracter que indica o que há naquela posição, como por exemplo a entrada **"E"**.

```
// Estrutura para a topoP
typedef struct no{
    int x;
    int y;
    struct no *prox;
}no;

typedef struct pos{
    char argumento;
}pos;
```

Funções

A função **"escritaPilha"**, como o nome já diz, escreve as coordenadas da estrutura começando pelo topo utilizando um loop. Enquanto o nó auxiliar for diferente de **NULL**, quer dizer que o nó está dentro da pilha, printando os elementos correspondentes incrementando o nó auxiliar (aux = aux->prox) para percorrer toda a estrutura dinâmica.

```
// Função para imprimir a topoP
void escritaPilha(no *topoP) {
    no *aux=topoP;
    while (aux!=NULL) {
        printf("(%d, %d)\n", (*aux).x, (*aux).y);
        aux=(*aux).prox;
    }
}
```

A função **"insertionPilha"**, adiciona um novo nó no topo da pilha, atualizando o ponteiro do topo para o novo nó.

```
// Função para adicionar um nó no final da topoP
void insertionPilha(no **topo, no *novo) {
    (*novo).prox=(*topo);
    (*topo)=novo;
}
```

A função **"transferenciadepilha"**, por meio de um loop, transfere as coordenadas dos nós de uma pilha para a outra. O loop confere se o nó auxiliar que foi criado para percorrer a pilha é **NULL**, enquanto o nó auxiliar for diferente de **NULL**, ele é incrementado (aux = aux->prox ou aux = *aux.prox), para percorrer toda a estrutura.

```
void transferenciadepilha(no **topoP2, no *topoP){
    no *aux=topoP;
    while (aux!=NULL){
        no *aux2=malloc(sizeof(no));
        (*aux2).x=(*aux).x;
        (*aux2).y=(*aux).y;
        insertionPilha(topoP2,aux2);
        aux=(*aux).prox;
    }
}
```

A função **"removePIlha"**, remove o nó do topo da pilha, atualizando o ponteiro topo além de fazer a verificação do topo ser diferente de nulo, caso o **"if"** for falso quer dizer que a pilha está vazia.

```
// Função para remover o último nó da topoP
void removePilha(no **topo) {
    if((*topo)!=NULL){
        (*topo)=(*(*topo)).prox;
    }
}
```

A função **"encontrarSaida"**, faz a lógica de buscar a resolução do labirinto, utilizando a pilha para armazenar as coordenadas do labirinto durante a busca, ao passar pelas coordenadas marca um **"X"** para simbolizar que a posição já foi visitada evitando loops desnecessários.

```
// Função para encontrar a saída do labirinto
void encontrarSaida(no **topoP, pos lab[N][M], int *retorno) {
    //Criação da coordenada auxiliar
    no *aux=*topoP;
    int x=(*aux).x;
    int y=(*aux).y;

    if (lab[x][y].argumento=='S') {
        no *novoS=malloc(sizeof(no));
        (*novoS).x=x+1;
        (*novoS).y=y;
        insertionPilha(topoP,novoS);
        *retorno=2;
    }

    lab[x][y].argumento='X'; // Marcar como visitado

    // Movimentos possíveis: baixo, direita, esquerda, cima
```

A função tenta se mover nas quatro direções possíveis (esquerda, direita, baixo, cima) verificando se a posição está dentro do limite da matriz e se o caminho é um **"0"** ou um **"S"**.

```
//Baixo
if (x+1<N && (lab[x+1][y].argumento!='X' || lab[x+1][y].argumento=='S')) {
    no *novoB=malloc(sizeof(no));
    (*novoB).x=x+1;
    (*novoB).y=y;
    insertionPilha(topoP,novoB);
    if(*retorno==2||topoP==NULL){
        return;
    }
    encontrarSaida(topoP,lab,retorno);
    return;
}
//Direita
if (y+1<M && (lab[x][y+1].argumento=='0' || lab[x][y+1].argumento=='S')) {
    no *novoD=malloc(sizeof(no));
    (*novoD).x=x;
    (*novoD).y=y+1;
    insertionPilha(topoP,novoD);
    if(*retorno==2||topoP==NULL){
        return;
    }
    encontrarSaida(topoP,lab,retorno);
    return;
```

```
//Esquerda
if (y-1>=0 && (lab[x][y-1].argumento=='0' || lab[x][y-1].argumento=='S')) {
    no *novoE=malloc(sizeof(no));
    (*novoE).x=x;
    (*novoE).y=y-1;
    insertionPilha(topoP,novoE);
    if(*retorno==2||topoP==NULL){
        return;
    }
    encontrarSaida(topoP,lab,retorno);
    return;
}
//Cima
if (x-1>=0 && (lab[x-1][y].argumento=='0' || lab[x-1][y].argumento=='S')) {
    no *novoC=malloc(sizeof(no));
    (*novoC).x=x-1;
    (*novoC).y=y;
    insertionPilha(topoP,novoC);
    if(*retorno==2||topoP==NULL){
        return;
    }
    encontrarSaida(topoP,lab,retorno);
    return;
```

```
    //Remove um elemento da pilha se nao conseguir avançar
    removePilha(topoP);
    if(*retorno==2||topoP==NULL){
            return;
    }
    else{
        encontrarSaida(topoP,lab,retorno);
    }
}
```

Ao passar pelas coordenadas e marcando-as como visitadas, verifica se o local pode ser a saída **"S"**, mudando o retorno para 2 caso positivo. A função **"leituramatriz"** lê o arquivo disponibilizado por meio de um loop e guarda as informações em uma matriz.

```
void leituramatriz(pos lab[N][M], FILE *pontarq) {
    // Leitura da matriz
    for (int i = 0; i < 10; i++) {
        for (int j = 0; j < 10; j++) {
            lab[i][j].argumento = fgetc(pontarq);
        }
        fgetc(pontarq);
    }
```

A função **"entrada"** procura a posição no labirinto onde está o carácter correspondente é o **"E"**,que representa a entrada. O loop com os dois **"for"** percorre a matriz toda do labirinto, **"i"** para as suas 10 linhas e **"j"** para as 10 colunas.

```
/*Função para encontrar as coordenadas da entrada*/
void entrada(pos lab[N][M], no *novo){
    for (int i=0; i<N; i++) {
        for (int j=0; j<M; j++) {
            if (lab[i][j].argumento == 'E') {
                (*novo).x=i;
                (*novo).y=j;
            }
        }
    }
}
```

Complexidade do Algoritmo

Considerando o pior caso possível, onde a função vai visitar todas as coordenadas do labirinto, considerando que o labirinto tem 10 linhas e 10 colunas, o algoritmo é de complexidade **O(n)**, onde **"N"** é o número de linhas e **"M"** é o número de colunas.

Makefile

O Makefile é um arquivo de texto para ajudar na compilação de um conjunto de arquivos de código fonte, contendo instruções de compilação do código, pode automatizar tarefas, como a limpeza de arquivos temporários e organizar o código desenvolvido.

Para executar o makefile, basta escrever no terminal: **"make all"**, assim, o utilitário make vai executar o target all.

No começo do makefile, definimos o nome do projeto como **"labirinto"**.

```
# Nome do projeto
PROJ_NAME = labirinto
```

O comando **C_SOURCE** captura os arquivos **.c** da pasta source.

```
# Arquivos .c
C_SOURCE = $(wildcard ./source/*.c)
```

Definir o comando de remoção.

```
# Comando utilizado como target do clean
RM = rm -rf
```

Linkagem do executável, linka os arquivos objeto para criar o executável.

```
# Linka os arquivos objeto em um executável
$(PROJ_NAME): $(OBJ)
    @ echo 'Construindo o binário usando o linker GCC: $@'
    $(CC) $^ -o $@
    @ echo 'Binário pronto!: $@'
    @ echo ' '
```

Compilação dos arquivos fonte, dizendo que um arquivo .o na pasta **"objects"** depende do seu correspondente **.c** na pasta source.

```
# Regra para compilar cada arquivo .c em um .o
./objects/%.o: ./source/%.c
    @ echo 'Compilando o target usando o GCC: $<'
    $(CC) $(CC_FLAGS) $< -o $@
    @ echo ' '
```

Compilar o arquivo **main.o**.

```
# Compila main.o
./objects/main.o: ./source/main.c $(H_SOURCE)
    @ echo 'Compilando o target usando o GCC: $<'
    $(CC) $(CC_FLAGS) $< -o $@
    @ echo ' '
```

Criar a pasta **"objetcs"**.

```
# Cria a pasta objects se não existir
objFolder:
    @ mkdir -p objects
```

Conclusão

Nosso algoritmo utiliza de uma estrutura dinâmica (pilha) para armazenar as coordenadas da matriz do labirinto, assim, por meio da função de encontrarSaida, o código percorre todas as coordenadas possíveis colocando um **"X"** onde foi visitada e armazenando sua coordenada na pilha. Portanto, o código testa todas as possibilidades possíveis até encontrar a saída **"S"** desejada, e printa a pilha que armazena as coordenadas do labirinto, fazendo assim, a resposta do problema.