Universidade Federal de Ouro Preto - UFOP
Instituto de Ciências Exatas e Biológicas - ICEB
Departamento de Computação - DECOM
Ciência da Computação

Trabalho Prático II

BCC202 - Estrutura de Dados I

Paulo Eduardo Costalonga Armani, Henrique Lauar
Professor: Pedro Henrique Lopes Silva

Ouro Preto
10 de julho de 2023

Sumário

Lista de Tabelas

Lista de Códigos Fonte

1 Introdução

Para este trabalho é necessário entregar o código em C e um relatório referente ao que foi desenvolvido. O algoritmo a ser desenvolvido é de um labirinto com um rato dentro, e devemos ajudá-lo a sair. Ele é baseado em um labirinto qualquer e possíveis caminhos diferentes, porém é necessário achar o menor caminho de saída. Embora pareça simples, as configurações resultantes podem ser surpreendentemente complexas e imprevisíveis.

O labirinto é uma mistura de vários caminhos com possíveis saídas ou não. Criado com objetivo de que independente do que estiver dentro dele, será de certa forma díficil sair.

A codificação deve ser feita em C, usando somente a biblioteca padrão da GNU, sem o uso de bibliotecas adicionais. Além disso, deve-se usar um dos padrões: ANSI C 89 ou ANSI C 99.

1.1 Especificações do problema

É necessário processar adequadamente um arquivo de entrada que contém a dimensão de uma matriz na sua primeira linha, um caracter na segunda, e os caracteres que formam o labirinto correspondentes nas linhas seguintes. Com base nos caracteres de entrada, que formam o labirinto, é necessário achar a menor saída para o rato dentro do labirinto. Além disso, é necesário utilizar o conceito de lista encadeada, filas e pilhas na construção do código.

1.2 Ferramentas utilizadas

Algumas ferramentas foram utilizadas durante a criação deste projeto:

- Ambiente de desenvolvimento do código fonte: Visual Studio Code. [1]
- Linguagem utilizada: C.
- Ambiente de desenvolvimento da documentação: Overleaf LaTeX. [2]

1.3 Ferramentas adicionais

Algumas ferramentas foram utilizadas para auxiliar no desenvolvimento, como:

- *Live Share*: ferramenta usada para *pair programming* à distância.
- *Valgrind*: ferramentas de análise dinâmica do código.

1.4 Especificações da máquina

A máquina onde o desenvolvimento e os testes foram realizados possui a seguinte configuração:

- Processador: AMD® Ryzen 5 5500u with radeon graphics × 12.
- Memória RAM: 8GB.
- Sistema Operacional: Ubuntu 22.04.2 LTS.

1.5 Instruções de compilação e execução

Para a compilação do projeto, basta digitar o seguinte comando para rodar o arquivo *Makefile* disponível:

Compilando o projeto

```
make
```

Usou-se para a compilação as seguintes opções:

[1] VScode está disponível em https://code.visualstudio.com/
[2] Disponível em https://www.overleaf.com/

- *-o*: para definir o arquivo de saída.

- *-g*: para compilar com informação de depuração e ser usado pelo Valgrind.

- *-Wall*: para mostrar todos os possível *warnings* do código.

- *-c*: para compilação do código e geração dos arquivos objetos.

Para a execução do programa basta digitar :

```
./exe < arquivo.in
```

Onde o arquivo de entrada contém o tamanho da matriz, um caracter, e os caracteres que representam o labirinto.

2 Implementação

O desenvolvimento foi realizado utilizando da técnica de pair programming, onde a dupla programou e participou ativamente do código ao mesmo tempo. O uso das ferramentas Live Share para o compartilhamento de código e do Google Meet para a comunicação em equipe foi o que viabilizou o uso da técnica citada.

2.1 Codificação do Labirinto

Para a codificação do labirinto foram criados dois arquivos: labirinto.h, responsável por guardar a prototipagem usada nas funções e labirinto.c, contendo a implementação das funções em si.

Começando pelo labirinto.h, temos o seguinte código:

```
#ifndef LABIRINTO_H
#define LABIRINTO_H

typedef struct {
    int x;
    int y;
    bool visitado;
} Posicao;

typedef struct {
    Posicao *posicoes;
    int comprimento;
} Percurso;

typedef struct {
    char valor;
    bool visitado;
} Celula;

typedef struct {
    int linhas;
    int colunas;
    Celula** maze;
    Posicao origem;
    Posicao saida;
    Percurso percurso;
} Labirinto;

Labirinto* alocarLabirinto();
void desalocarLabirinto(Labirinto* labirinto);
Labirinto* leLabirinto();
bool acharSaida(Posicao origem, Posicao destino, Labirinto *labirinto);
bool ehValido(Posicao pos, Labirinto* labirinto);
Posicao achaOrigem(Labirinto* labirinto);
Posicao achaDestino(Labirinto* labirinto);
void imprimePercursoNoLabirinto(Labirinto* labirinto);
void imprimeCoordenadas(Labirinto* labirinto);
bool dfs(Posicao origem, Posicao destino, Labirinto* labirinto, int linhas,
    int colunas);

#endif // LABIRINTO_H
```

Código 1: Headers do labirinto.

- A estrutura **Posicao** armazena as coordenadas x e y, e também bool visitado, para saber se a posição foi visitado ou não.

- A estrutura **Percurso** armazena um vetor de posições do tipo Posicao, e também o comprimento que será usado para saber a quantidade de passos até sair do labirinto.
- A estrutura **Celula** armazena um char valor que serve para saber qual é o caractere inserido (parede, espaço vazio, ou 'M'), e o bool visitado para saber se a posição já foi visitada.
- A estrutura **Labirinto** armazena a dimensão do labirinto, a matriz maze do tipo Celula, a posição de origem (no caso 'M'), a posição de saída que é o destino final, e o percurso.
- A função **alocarLabirinto()** aloca dinamicante o labirinto com as dimensões passadas no terminal.
- A função **desalocarLabirinto(Labirinto* labirinto)** desaloca o labirinto. Esse passo é importante para evitar vazamentos de memória e problemas de performance.
- A função **leLabirinto(int linhas, int colunas)** lê os valores da matriz a partir de uma entrada no terminal e armazena-os na estrutura labirinto. Dessa forma, é possível salvar o labirinto para posteriormente o rato sair dele.
- A função **acharSaida(Posicao origem, Posicao destino, Labirinto *labirinto)** recebe a posição do rato, a posição de destino, usa recursividade para achar o menor caminho até o final do labirinto e retorna true se tiver achado, e false se não. Essa é a parte importante da implementação onde ocorre o caminhar do rato de acordo com os caminhos possíveis.
- A função **ehValido(Posicao pos, Labirinto *labirinto)** verifica se a posição atual do rato é válida ou não, caso seja uma parede retorna falso, caso seja espaço retorna true.
- A função **achaOrigem(Labirinto *labirinto)** procura o rato no labirinto e retorna a posição dele como sendo a origem.
- A função **achaDestino(Labirinto *labirinto)** retorna o destino final já que será sempre no mesmo lugar.
- A função **imprimePercursoNoLabirinto(Labirinto *labirinto)** imprime o percurso do rato até a saída do labirinto, sendo o menor percurso possível.
- A função **imprimeCoordenadas(Labirinto *labirinto)** imprime as coordenadas de cada passo do rato até a saída do labirinto.

No arquivo labirinto.c temos a implementação das funções acima.

A seguir, o código da função **ehValido**:

```
1  bool ehValido(Posicao pos, Labirinto *labirinto)
2  {
3      //verifica se a posicao esta dentro do labirinto
4      if (pos.x < 0 || pos.x >= labirinto->linhas || pos.y < 0 || pos.y >=
           labirinto->colunas)
5      {
6          return false;
7      }
8
9      //verifica se a posicao eh uma parede
10     if (labirinto->maze[pos.x][pos.y].valor == '#' || labirinto->maze[pos.x][
           pos.y].valor == '*')
11     {
12         return false;
13     }
14     return true;
15 }
```

Código 2: Função ehValido.

Esta função recebe dois parâmetros: pos, que é uma estrutura Posicao contendo as coordenadas x e y de uma posição no labirinto, e labirinto, que é um ponteiro para a estrutura Labirinto.

Essa condição verifica se as coordenadas x e y da posição estão dentro dos limites do labirinto. Se a posição estiver fora dos limites, ou seja, se pos.x for menor que zero, ou pos.x for maior ou igual ao número de linhas do labirinto, ou pos.y for menor que zero, ou pos.y for maior ou igual ao número de colunas do labirinto, a função retorna false. Isso indica que a posição não é válida dentro do labirinto.

Esta condição verifica se o valor da célula do labirinto na posição pos.x e pos.y é igual a '#' ou '*'. Se for o caso, a função também retorna false. Isso indica que a posição contém uma parede ('#') ('*'), portanto não é uma posição válida para o rato se mover.

Se nenhuma das condições acima for atendida, isso significa que a posição é válida dentro do labirinto, e a função retorna true.

Em resumo, a função ehValido() verifica se uma posição no labirinto é válida, verificando se as coordenadas estão dentro dos limites do labirinto e se a célula correspondente não é uma parede ou já foi visitada. Ela retorna true se a posição for válida e false caso contrário.

A seguir está a função **acharSaida**:

```
 bool acharSaida(Posicao origem, Posicao destino, Labirinto *labirinto)
{
    if (!ehValido(origem, labirinto) || !ehValido(destino, labirinto) ||
        origem.visitado)
    {
        printf("Origem: %d %d\n", origem.x, origem.y);
        return false;
    }


    if (origem.x == destino.x && origem.y == destino.y)
    {
        return true;
    }

    labirinto->maze[origem.x][origem.y].visitado = true;
    origem.visitado = true;

    int dx[] = {-1, 1, 0, 0};
    int dy[] = {0, 0, -1, 1};

    for (int i = 0; i < 4; i++)
    {
        Posicao newPos;
        newPos.x = origem.x + dx[i];
        newPos.y = origem.y + dy[i];
        newPos.visitado = false;

        if (ehValido(newPos, labirinto) && !labirinto->maze[newPos.x][newPos.y
            ].visitado)
        {
            if (acharSaida(newPos, destino, labirinto))
            {

                labirinto->percurso.posicoes[labirinto->percurso.comprimento]
                    = newPos;
                labirinto->percurso.comprimento++;
                labirinto->maze[newPos.x][newPos.y].valor = '.';
                return true;
            }
        }
```

```
39      }
40
41      return false;
42  }
```

Código 3: Função acharSaida.

Esta função é uma função recursiva que recebe três parâmetros: origem, que é uma estrutura Posicao representando a posição de partida do rato no labirinto, destino, que é uma estrutura Posicao representando a posição de saída desejada, e labirinto, que é um ponteiro para a estrutura Labirinto.

A primeira condição verifica se a origem e o destino são posições válidas e se a origem já foi visitada. Se alguma dessas condições for verdadeira, significa que a busca não pode continuar. Nesse caso, a função imprime as coordenadas da origem e retorna false, indicando que não foi possível encontrar a saída.

A segund condição verifica se a origem coincide com o destino. Se isso for verdadeiro, significa que a posição atual é a saída desejada. Nesse caso, a função retorna true, indicando que a saída foi encontrada.

Essas linhas (15, 16) marcam a posição atual (origem) como visitada tanto na estrutura Labirinto quanto na própria estrutura Posicao. Isso evita que o rato passe pela mesma posição várias vezes.

Esses vetores dx e dy representam as direções nas quais o rato pode se mover no labirinto. Cada vetor possui 4 elementos, correspondendo às direções: cima, baixo, esquerda e direita.

Este laço for percorre as 4 direções possíveis e cria uma nova posição newPos com as coordenadas atualizadas. Essa posição é inicializada como não visitada.

Dentro do loop, primeiro verifica-se se a nova posição newPos é válida e não foi visitada. Se isso for verdadeiro, a função acharSaida() é chamada recursivamente com a nova posição como origem. Se essa chamada recursiva retornar true, significa que a saída foi encontrada a partir da nova posição. Nesse caso, a posição é adicionada ao percurso (labirinto-¿percurso.posicoes), o valor da célula no labirinto é atualizado para '.', e a função retorna true.

Se nenhuma das condições anteriores for atendida dentro do loop, isso significa que nenhuma das direções possíveis leva à saída. Nesse caso, a função retorna false.

Em resumo, a função acharSaida() é uma função recursiva que realiza uma busca em profundidade no labirinto para encontrar a saída. Ela tenta explorar as posições adjacentes a partir da posição de origem, verificando se cada posição é válida, não foi visitada e, em seguida, chamando a função recursivamente. Se a saída for encontrada, a função retorna true e atualiza o percurso do labirinto. Caso contrário, retorna false.

A seguir está a função **imprimeCoordenadas**:

```
1   void imprimeCoordenadas(Labirinto *labirinto)
2   {
3       // imprime as coordenadas do percurso do rato
4       for (int i = labirinto->percurso.comprimento - 1; i >= 0; i--)
5       {
6           printf("%d, %d\n", labirinto->percurso.posicoes[i].x, labirinto->
                percurso.posicoes[i].y);
7       }
8   }
```

Código 4: Função imprimeCoordenadas.

Essa função imprime as coordenadas do rato, desde a origem até a saída. Começa o laço pela última posição do vetor labirinto-¿percurso.comprimento, e vai até a primeira, porém como na recursividade os valores ficaram armazenados do último ao primeiro, a primeira coordenada do rato está na última posição do vetor.

A função **achaOrigem** percorre a matriz do labirinto procurando pelo caracter 'M', quando encontrado atribui os valores de i e j à origem.x e origem.y.

A função **achaDestino** atribui à variável saida.x o valor de linhas - 2, e atribui a saida.x o valor de colunas - 1. Já que o destino final do labirinto será sempre o mesmo.

A função **imprimePercursoNoLabirinto** percorre a matriz do labirinto e imprime posição por posição com o menor percurso do rato no labirinto.

2.2 Implementação de filas

Começando com o fila.h, com o seguinte código:

```
#ifndef FILA_H
#define FILA_H
#include "labirinto.h"

typedef struct {
    Posicao *posicoes;
    int inicio;
    int fim;
} Fila;

bool bfs(Posicao origem, Posicao destino, Labirinto* labirinto, int linhas,
    int colunas);

#endif // FILA_H
```

Código 5: TAD Fila.

Nesse código, temos a struct Fila, onde foi criado um vetor posicoes do tipo Posicao, um inteiro para armazenar o inicio e outro inteiro para armazenar o fim. Em seguida, é definida a função que vai ser implementada no arquivo fila.c.

Arquivo fila.c, com o seguinte código:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include "fila.h"

bool bfs(Posicao origem, Posicao destino, Labirinto *labirinto, int linhas,
    int colunas)
{
     // Verifica se a origem e o destino ãso ávlidos e se a origem áj foi
        visitada
     if (!ehValido(origem, labirinto) || !ehValido(destino, labirinto) ||
        origem.visitado)
     {
          printf("Origem: %d %d\n", origem.x, origem.y);
          return false;
     }

     // Vetores que representam as çõdirees que o rato pode ir
     int dx[] = {0, 1, 0, -1};
     int dy[] = {1, 0, -1, 0};

     // çãInicializao da fila para o BFS
     Fila *fila = malloc(sizeof(Fila));
     fila->posicoes = malloc(sizeof(Posicao) * linhas * colunas);
     fila->inicio = 0;
     fila->fim = 0;
```

```
24
25      // Marca a çãposio atual (origem) como visitada
26      labirinto->maze[origem.x][origem.y].visitado = true;
27
28      // Adiciona a origem à fila
29      fila->posicoes[fila->fim] = origem;
30      fila->fim++;
31
32      while (fila->inicio != fila->fim)
33      {
34           Posicao atual = fila->posicoes[fila->inicio];
35           fila->inicio++;
36
37           // Verifica se a çãposio atual é o destino
38           if (atual.x == destino.x && atual.y == destino.y - 1)
39           {
40                labirinto->maze[destino.x][destino.y].visitado = true;
41                labirinto->percurso.posicoes[labirinto->percurso.comprimento] =
                       destino; // Adiciona a çãposio ao percurso
42                labirinto->maze[destino.x][destino.y].valor = 'o';
43                labirinto->percurso.comprimento++;
44
45                free(fila->posicoes);
46                free(fila);
47                return true;
48           }
49
50           // Explora todas as çõdirees ípossveis
51           for (int i = 0; i < 4; i++)
52           {
53                Posicao novaPos;
54                novaPos.x = atual.x + dx[i];
55                novaPos.y = atual.y + dy[i];
56                novaPos.visitado = false; // Nova çãposio inicialmente ãno
                       visitada
57
58                // Verifica se a nova çãposio éá vlida e se ãno foi visitada
59                if (ehValido(novaPos, labirinto) && !labirinto->maze[novaPos.x
                       ][novaPos.y].visitado)
60                {
61                     // Marca a nova çãposio como visitada
62                     labirinto->maze[novaPos.x][novaPos.y].visitado = true;
63                     labirinto->percurso.posicoes[labirinto->percurso.
                           comprimento] = novaPos; // Adiciona a çãposio ao
                           percurso
64                     labirinto->maze[novaPos.x][novaPos.y].valor = 'o';
65                     labirinto->percurso.comprimento++;
66                     // Enfileira a nova çãposio
67                     fila->posicoes[fila->fim] = novaPos;
68                     fila->fim++;
69                }
70           }
71      }
72      // Se a fila ficar vazia e o destino ãno for çalcanado, ãno áh ísada
73      free(fila->posicoes);
74      free(fila);
75      return false;
76 }
```

Código 6: Função bfs.

A função inicia com a verificação da origem do labirinto e inicializando uma fila para o algoritmo BFS. A fila é implementada usando uma estrutura de dados chamada "Fila", que contém um array de posições (posicoes), um índice de início (inicio) e um índice de fim (fim). Adiciona a origem à fila, colocando-a no array de posições e incrementando o índice de fim.

Inicia um loop enquanto a fila não estiver vazia (enquanto o índice de início for diferente do índice de fim).

Dentro do loop, obtém a posição atual da fila, removendo-a da fila (incrementando o índice de início).

Verifica se a posição atual é o destino. Se for, marca a célula correspondente no labirinto como visitada, adiciona a posição ao percurso do labirinto, atualiza o valor da célula para 'o' e retorna verdadeiro, indicando que um caminho válido foi encontrado.

Explora todas as direções possíveis a partir da posição atual. Para cada direção, calcula a nova posição somando os valores de dx[i] e dy[i] às coordenadas da posição atual.

Verifica se a nova posição é válida (dentro dos limites do labirinto) e se ainda não foi visitada. Se essas condições forem verdadeiras, marca a nova posição como visitada, adiciona a posição ao percurso do labirinto, atualiza o valor da célula para 'o' e adiciona a nova posição à fila (colocando-a no array de posições e incrementando o índice de fim).

Se o loop terminar e a fila estiver vazia, isso significa que não foi possível encontrar um caminho do ponto de origem ao ponto de destino. Nesse caso, a função libera a memória alocada para a fila e retorna falso.

2.3 Implementação de pilhas

Começando com o pilha.h, com o seguinte código:

```
#ifndef PILHA_H
#define PILHA_H
#include "labirinto.h"

typedef struct {
     Posicao *posicoes;
     int cabeca;
} Pilha;

bool dfs(Posicao origem, Posicao destino, Labirinto *labirinto, int linhas,
    int colunas);

#endif // PILHA_H
```

Código 7: TAD Pilha.

Nesse código, temos a struct Pilha, onde foi criado um vetor posicoes do tipo Posicao e um inteiro para a cabeça. Em seguida, é definida a função que vai ser implementada no arquivo fila.c.

Arquivo pilha.c, com o seguinte código:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include "pilha.h"

bool dfs(Posicao origem, Posicao destino, Labirinto *labirinto, int linhas,
    int colunas)
{
```

```
    // Verifica se a origem e o destino ãso ávlidos e se a origem áj foi visitada
    if (!ehValido(origem, labirinto) || !ehValido(destino, labirinto) || origem.visitado)
    {
        printf("Origem: %d %d\n", origem.x, origem.y);
        return false;
    }

    // Vetores que representam as çõdirees que o rato pode ir
    int dx[] = {0, 1, 0, -1};
    int dy[] = {1, 0, -1, 0};


    // Marca a çãposio atual (origem) como visitada
    labirinto->maze[origem.x][origem.y].visitado = true;

    // Empilha a origem
    Pilha *pilha = malloc(sizeof(Pilha));
    pilha->posicoes = malloc(sizeof(Posicao) * linhas * colunas);
    pilha->cabeca = 0;
    pilha->posicoes[pilha->cabeca] = origem;

    while (pilha->cabeca != -1)
    {
        Posicao atual = pilha->posicoes[pilha->cabeca--];

        // Verifica se a çãposio atual e o destino
        if (atual.x == destino.x && atual.y == destino.y)
        {
            return true;
        }

        // Explora todas as çõdirees ípossveis
        for (int i = 0; i < 4; i++)
        {
            Posicao novaPos;
            novaPos.x = atual.x + dx[i];
            novaPos.y = atual.y + dy[i];
            novaPos.visitado = false; // Nova çãposio inicialmente ãno visitada

            // Verifica se a nova çãposio éá vlida e se ãno foi visitada
            if (ehValido(novaPos, labirinto) && !labirinto->maze[novaPos.x][novaPos.y].visitado)
            {
                // Marca a nova çãposio como visitada
                labirinto->maze[novaPos.x][novaPos.y].visitado = true;
                labirinto->percurso.posicoes[labirinto->percurso.comprimento] = novaPos; // Adiciona a çãposio ao percurso
                labirinto->percurso.comprimento++;
                labirinto->maze[novaPos.x][novaPos.y].valor = 'o';

                // Empilha a nova çãposio
                pilha->posicoes[++pilha->cabeca] = novaPos;
            }
        }
    }

    // Se a cabeca ficar vazia e o destino ãno for çalcanado, ãno áh ísada
    return false;
```

```
64 }
```

Código 8: Função dfs.

Este código implementa a função dfs que realiza uma busca em profundidade (DFS) . O objetivo é encontrar um caminho válido do ponto de origem até o ponto de destino no labirinto.

Verifica se a origem e o destino são válidos e se a origem já foi visitada. Se alguma dessas condições for verdadeira, a função imprime as coordenadas da origem.

Inicializa dois vetores, dx e dy, que representam as direções que o "rato"pode se mover no labirinto. O vetor dx representa as variações na coordenada x (horizontal), e o vetor dy representa as variações na coordenada y (vertical).

Marca a posição atual (origem) como visitada, definindo o campo visitado da célula correspondente no labirinto como verdadeiro.

Inicializa uma pilha para o algoritmo DFS. A pilha é implementada usando uma estrutura de dados chamada "Pilha", que contém um array de posições (posicoes), um índice de cabeça (cabeca) e um tamanho máximo.

Empilha a origem, colocando-a no array de posições e decrementando o índice de cabeça.

Inicia um loop enquanto a cabeça da pilha não estiver vazia (enquanto o índice de cabeça for diferente de -1).

Dentro do loop, obtém a posição atual do topo da pilha e a remove da pilha (decrementando o índice de cabeça).

Verifica se a posição atual é o destino. Se for, retorna verdadeiro, indicando que um caminho válido foi encontrado.

Explora todas as direções possíveis a partir da posição atual. Para cada direção, calcula a nova posição somando os valores de dx[i] e dy[i] às coordenadas da posição atual.

Verifica se a nova posição é válida (dentro dos limites do labirinto) e se ainda não foi visitada. Se essas condições forem verdadeiras, marca a nova posição como visitada, adiciona a posição ao percurso do labirinto, atualiza o valor da célula para 'o' e empilha a nova posição (incrementando o índice de cabeça e colocando a posição no array de posições).

Se o loop terminar e a pilha estiver vazia (a cabeça for igual a -1), isso significa que não foi possível encontrar um caminho do ponto de origem ao ponto de destino. Nesse caso, a função retorna falso.

2.4 Implementação do Labirinto

Seguido pelo main.c, o arquivo main:

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include "labirinto.h"
#include "fila.h"
#include "pilha.h"

int main()
{
    int linhas, colunas;
    char tipo;
    scanf("%d %d ", &linhas, &colunas);
    scanf("%c ", &tipo);

    Labirinto *labirinto = leLabirinto(linhas, colunas);

    labirinto->origem = achaOrigem(labirinto);
    labirinto->saida = achaDestino(labirinto);
```

```
19
20      switch (tipo)
21      {
22      case 'r':
23           acharSaida(labirinto->origem, labirinto->saida, labirinto);
24           break;
25      case 'f':
26           bfs(labirinto->origem, labirinto->saida, labirinto, linhas, colunas)
                 ;
27           break;
28      case 'p':
29           dfs(labirinto->origem, labirinto->saida, labirinto, linhas, colunas)
                 ;
30           break;
31      }
32
33      printf("%d \n", labirinto->percurso.comprimento); // imprime nro de
            passos do percurso
34      imprimePercursoNoLabirinto(labirinto);
35
36      desalocarLabirinto(labirinto);
37
38      return 0;
39  }
```

Código 9: Função main.

Primeiro são declaradas e inicializadas as variáveis linhas, colunas e tipo.

Lê a quantidade de linhas e colunas do labirinto e armazena nas variáveis linhas e colunas, respectivamente.

Lê o tipo de busca a ser realizada (caractere 'r' para recursividade, 'f' para busca em largura ou 'p' para busca em profundidade) e armazena na variável tipo.

Chama a função leLabirinto(linhas, colunas) para ler o labirinto da entrada padrão e armazená-lo em uma estrutura de dados do tipo Labirinto. O ponteiro para o labirinto é atribuído à variável labirinto.

Usa a função achaOrigem(labirinto) para encontrar a posição de origem (ponto de partida) no labirinto e atribui essa posição ao campo origem do labirinto.

Usa a função achaDestino(labirinto) para encontrar a posição de destino (saída) no labirinto e atribui essa posição ao campo saida do labirinto.

Com base no tipo de busca selecionado, executa a busca apropriada no labirinto. Dependendo do valor de tipo, executa uma das seguintes ações:

Se tipo for 'r', chama a função acharSaida passando a posição de origem, a posição de destino e o labirinto. Se tipo for 'f', chama a função bfs passando a posição de origem, a posição de destino, o labirinto, o número de linhas e o número de colunas. Se tipo for 'p', chama a função dfs passando a posição de origem, a posição de destino, o labirinto, o número de linhas e o número de colunas. Imprime o comprimento do percurso encontrado, acessando o campo comprimento da estrutura percurso do labirinto.

Imprime o labirinto com o percurso marcado, usando a função imprimePercursoNoLabirinto(labirinto).

Desaloca a memória alocada para o labirinto, chamando a função desalocarLabirinto(labirinto).

2.5 Implementação do Labirinto

Seguido pelo main.c, o arquivo main:

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include "labirinto.h"
#include "fila.h"
#include "pilha.h"

int main()
{
     int linhas, colunas;
     char tipo;
     scanf("%d %d ", &linhas, &colunas);
     scanf("%c ", &tipo);

     Labirinto *labirinto = leLabirinto(linhas, colunas);

     labirinto->origem = achaOrigem(labirinto);
     labirinto->saida = achaDestino(labirinto);

     switch (tipo)
     {
     case 'r':
          acharSaida(labirinto->origem, labirinto->saida, labirinto);
          break;
     case 'f':
          bfs(labirinto->origem, labirinto->saida, labirinto, linhas, colunas)
              ;
          break;
     case 'p':
          dfs(labirinto->origem, labirinto->saida, labirinto, linhas, colunas)
              ;
          break;
     }

     printf("%d \n", labirinto->percurso.comprimento); // imprime nro de
         passos do percurso
     imprimePercursoNoLabirinto(labirinto);

     desalocarLabirinto(labirinto);

     return 0;
}
```

Código 10: Função main.

Primeiro são declaradas e inicializadas as variáveis linhas, colunas e tipo.

Lê a quantidade de linhas e colunas do labirinto e armazena nas variáveis linhas e colunas, respectivamente.

Lê o tipo de busca a ser realizada (caractere 'r' para recursividade, 'f' para busca em largura ou 'p' para busca em profundidade) e armazena na variável tipo.

Chama a função leLabirinto(linhas, colunas) para ler o labirinto da entrada padrão e armazená-lo em uma estrutura de dados do tipo Labirinto. O ponteiro para o labirinto é atribuído à variável labirinto.

Usa a função achaOrigem(labirinto) para encontrar a posição de origem (ponto de partida) no labirinto e atribui essa posição ao campo origem do labirinto.

Usa a função achaDestino(labirinto) para encontrar a posição de destino (saída) no labirinto e atribui essa posição ao campo saida do labirinto.

Com base no tipo de busca selecionado, executa a busca apropriada no labirinto. Dependendo do valor de tipo, executa uma das seguintes ações:

Se tipo for 'r', chama a função acharSaida passando a posição de origem, a posição de destino e o labirinto. Se tipo for 'f', chama a função bfs passando a posição de origem, a posição de destino, o labirinto, o número de linhas e o número de colunas. Se tipo for 'p', chama a função dfs passando a posição de origem, a posição de destino, o labirinto, o número de linhas e o número de colunas. Imprime o comprimento do percurso encontrado, acessando o campo comprimento da estrutura percurso do labirinto.

Imprime o labirinto com o percurso marcado, usando a função imprimePercursoNoLabirinto(labirinto).

Desaloca a memória alocada para o labirinto, chamando a função desalocarLabirinto(labirinto).

3 Testes

Para os experimentos foram usados duas dimensões do labirinto diferentes de entrada seguindo o modelo já explicado anteriormente. Os testes foram realizados no hardware especificado anteriormente.

Além do mais, para saber o consumo de memória foi utilizado o *valgrind* do linux. Cada entrada foi executada três vezes.

3.1 Entrada 1

O primeiro teste recebeu como entrada os seguintes dados:

```
7 11
f
***********
*         *
********* *
*M*   * * *
* *** * * *
*       *
***********
```

Código 11: Entrada 1.

A saída obtida foi:

```
14
***********
*         *
********* *
*M*ooo*o* *
*o***o*o* *
*ooooooo*
***********
```

Código 12: Saída 1.

- O consumo de memória foi de 10,322 bytes.

3.2 Entrada 2

O segundo teste recebeu como entrada os seguintes dados:

```
7 11
p
***********
*         *
******** *
*M*   * * *
* *** * * *
*       *
***********
```

Código 13: Entrada 2.

A saída obtida foi:

```
14
***********
*         *
******** *
*M*ooo*o* *
*o***o*o* *
*ooooooo*
***********
```

Código 14: Saída 2.

- O consumo de memória foi de 10,322 bytes

4 Análise

A seguir está um resumo dos resultados obtidos a partir de cada entrada, bem como os pontos fortes e fracos do código-fonte do trabalho.

4.1 Resumo dos experimentos

Entrada 1:

Teste	Consumo de memória (em bytes)
Teste 1	10,322

Tabela 1: Resumo dos testes da entrada 1.

Entrada 2:

Teste	Consumo de memória (em bytes)
Teste 2	10,322

Tabela 2: Resumo dos testes da entrada 2.

4.2 Pontos fortes

- O código é bem estruturado e fácil de ler, com comentários úteis explicando o que cada função faz.
- A alocação dinâmica de memória para armazenar a matriz do labirinto é uma boa prática, pois permite que o tamanho do labirinto seja definido dinamicamente e evita problemas de buffer overflow.
- As funções dfs e bfs fizeram com que o código fique mais organizado e dá mais de uma opção para solução do labirinto, sendo possível analisar e ver qual metódo compensaria mais de ser usado

4.3 Pontos fracos

- Em caso de um labirinto com dimensões grandes demais, o programa pode ficar lento.

5 Conclusão

Inicialmente, nós analisamos o que foi pedido dentro do pdf do trabalho prático, buscamos compreender como adaptar o código para que fosse possível implementar o labirinto utilizando filas e pilhas. Foi necessário estudar sobre Busca em Largura e Busca em Profundidade para saber exatamente o que estávamos implementando e a melhor forma de colocar em prática.

Em seguida, começamos com o desenvolvimento das funções de busca em largura e busca em profundidade. Conseguimos achar bastante materiais desses assuntos na internet, o que facilitou o nosso estudo. Como as funções básicas do labirinto já estavam prontas, a maior dificuldade era entender como inserir aquelas funções já prontas em um novo código.

Por fim, utilizamos o valgrind que foi muito útil para localizar os vazamentos de memória, assim que retiramos todos os bugs, finalizamos e enviamos.