Departamento de Ciências Da Computação
UFMG

Trabalho Prático 1

Trabalho referente a discplina de Algoritmos e Estrutura de Dados II cursada no primeiro semestre do ano de 2016.

Rodrigo Lima de Araujo, 2015116383

Belo Horizonte
2016

1. Introdução

Teseu, o filho do Rei de Atenas se encontra em perigo. Ele deve desvendar um labirinto, matar o minotauro, e voltar são e salvo. Mas antes disso ele deve se armar e encontrar a espada que está posicionada em um determinado lugar do labirinto.

Teseu não está completamente no escuro. Ele sabe o tamanho do labirinto e a posição em que a espada se encontra, além de possuir um mapa (matriz binária onde 0(zero) significa que naquela posição há uma passagem e 1(um) significa que naquela posição há uma parede). Agora ele precisa explorar o labirinto até encontrar um caminho para a espada, ou descobrir que ela não pode ser alcançada.

Para a resolução desse problema, diversas medidas podem ser tomadas, mas já que o nome Teseu significa "o homem forte por excelência"[1] o método para resolução utilizado foi a força bruta. Isto é, todos os possíveis caminhos até a espada foram testados, encontrando dois possíveis resultados, ou existe pelo menos um caminho, ou não há caminho até a espada. Esse metódo pode ainda ser abordado de duas formas, Iterativa ou Recursiva, nesse trabalho as duas formas foram exploradas, então, duas formas para a resolução do problema são expostas. Outras exigências são: não ocorrer nenhum *leak* de memória e a geração de um arquivo txt, contendo uma matriz binária em que os números 1(um) representam um caminho correto com todas as outras posições marcadas com 0(zero) até a espada, se este existir, ou contendo apenas um 0(zero) caso não exista caminho.

2. Implementação

2.1 TADs

Duas TADs (*Tipo Abstrato de dados*) foram implementadas para a conclusão desse trabalho, a TADPilha e a TADLabirinto.

2.1.1 TADPilha

Algumas estruturas de dados são necessárias para a manipulação de pilhas:

A

```
4 typedef struct Celula *Apontador;//Estrutura para apontar o próximo item da pilha
5
6 typedef struct //Define tipo item
7 {
8     int x; //Coordenada x do item
9     int y; //Coordenada y do item
10 }ponto;//Estrutura Item da Pilha
11
12 typedef struct Celula //Define tipo célula
13 {
14     ponto coord; //item da célula
15     Apontador Prox; //Apontador para a próxima célula
16 }Celula;//Estrutura Célula da Pilha
17
18
19 typedef struct //Define o tipo Pilha
20 {
21     Apontador Fundo, Topo; //Apontadores para o fundo e o Topo da Pilha
22     int Tamanho; //Inteiro contendo o tamanho da pilha
23 }TipoPilha; //Estrutura da Pilha
24
```

Figura 1: Estruturas de Dados TADPilha

struct *Apontador* é utilizada para facilitar o entendimento da utilização de ponteiros na manipulação com pilhas. Ela indica um ponteiro para uma célula. A estrutura de dados *ponto* trata-se do item que será empilhado, ela contem dois inteiros dentro de si, a linha

[1]Brandão, Junito, Mitologia Grega, Petrópolis; Ed. Vozes, vol III, p. 149

x e a coluna *y* da coordenada. O tipo *Celula* indica uma posição da pilha, ou seja, cada posição da pilha vai possuir uma célula que, por sua vez, possui um item e um *Apontador* para a próxima posição (célula). E por último, o *TipoPilha* que contém apontadores para o Topo e o Fundo da pilha, e ainda contém um inteiro que indica o tamanho da pilha.

Além dessas estruturas de dados, na TADPilha foram implementadas também diversas funções para a manipulação de pilhas. A seguir encontram-se suas descrições:

2.1.1.1 void FPVazia(TipoPilha *Pilha);

```
26 void FPVazia(TipoPilha *Pilha);
27 /*--------------------------------------------------------------
28 Prototipo : void FPVazia(TipoPilha *Pilha);
29 Funcao : Cria uma pilha vazia.
30 Entrada : Recebe o endereço de memória no qual a pilha será criada
31 Saida:
32 --------------------------------------------------------------*/
```

Figura 2: Função FPVazia

Essa função recebe um endereço com o tamanho necessário para uma *struct* do *TipoPilha.* Primeiramente ela aloca memória dinamicamente do tamanho de uma *Celula* para o Topo da pilha. Depois, ela iguala o *Apontador* para o topo com o *Apontador* para o fundo, já que a pilha está vazia, depois faz com que esses *Apontadores* apontem para *NULL* (fim da pilha). Por último faz com que o tamanho da pilha seja 0(zero).

2.1.1.2 int Vazia(TipoPilha *Pilha);

```
35 int Vazia(TipoPilha *Pilha);
36 /*--------------------------------------------------------------
37 Prototipo : int Vazia(TipoPilha *Pilha);
38 Funcao : Checa se a pilha está vazia
39 Entrada : Ponteiro para a pilha a ser avaliada
40 Saida: Inteiro contendo o resultado da comparação, 1 para Vazia, 0 para não Vazia
41 --------------------------------------------------------------*/
```

Figura 3: Função Vazia

Essa função retorna 1(um) caso a pilha estiver vazia e 0(zero) se não estiver. Seu procedimento é bem simples. Ela retorna uma comparação entre os *Apontadores* Topo e Fundo, se eles forem iguais, o resultado da comparação será 1(um) e significará que a pilha está vazia.

2.1.1.3 void Empilha(ponto ponto1, TipoPilha *Pilha);

```
44 void Empilha(ponto ponto1, TipoPilha *Pilha);
45 /*--------------------------------------------------------------
46 Prototipo : void Empilha(ponto ponto1, TipoPilha *Pilha)
47 Funcao : Empilha um item na pilha.
48 Entrada : Ponto a ser empilhado, Ponteiro para a Pilha em que ele será empilhado
49 Saida:
50 --------------------------------------------------------------*/
```

Figura 4: Função Empilha

Essa função empilha o item *ponto1,* ou seja, coloca *ponto1* no topo da pilha. Primeiramente, um *Apontador* é declarado e alocado dinamicamente para guardar uma nova *Celula.* O item do Topo recebe o valor de *ponto1,* o *Apontador* Prox da *Celula* recém-alocada recebe o antigo primeiro item da pilha e então o Topo da pilha recebe a *Celula* empilhada. Por último, o Tamanho da pilha é incrementado.

2.1.1.4 ponto Desempilha(TipoPilha *Pilha);

```
53 ponto Desempilha(TipoPilha *Pilha);
54 /*-------------------------------------------------------------
55 Prototipo : ponto Desempilha(TipoPilha *Pilha);
56 Funcao : Retira o último item empilhado da pilha e retorna o seu
   valor.
57 Entrada : Ponteiro para a Pilha da qual o valor será desempilhado
58 Saida: Item Desempilhado.
59 -------------------------------------------------------------*/
60
```

Figura 5: Função Desempilha

Essa função é utilizada para Desempilhar uma *Celula* da pilha. Primeiramente, ela declara um ponteiro para *Celula* auxiliar. Depois, checa se a pilha está vazia. Se estiver, o usuário é informado do erro, libera a memória alocada para a pilha e então encerra a execução. Se a pilha não estiver vazia, o *Apontador* auxiliar recebe a Célula Cabeça, atual Topo da Pilha que será desempilhada. O *Apontador* Topo passa a apontar então para a antiga segunda *Celula*. A memória em que a *Celula* desempilhada estava alocada é então liberada. O tamanho é decrementado e por fim, retorna o item que estava contido na *Celula* desempilhada (Agora Célula Cabeça).

2.1.1.5 int Tamanho(TipoPilha *Pilha);

```
62 int Tamanho(TipoPilha *Pilha);
63 /*-------------------------------------------------------------
64 Prototipo : int Tamanho(TipoPilha *Pilha);
65 Funcao : Retorna o tamanho da pilha avaliada
66 Entrada : Ponteiro para a pilha que será avaliada.
67 -------------------------------------------------------------*/
```

Figura 6: Função Tamanho

A função Tamanho retorna o tamanho atual da pilha. Seu procedimento é muito simples, toda vez que é chamada, a função simplesmente retorna a variável Tamanho contida na *struct TipoPilha*.

2.1.2 TADLabirinto

O arquivo txt contendo o labirinto que Teseu recebe tem o formato:

```
6 1 0 4 4
1 1 1 1 1 1
0 0 1 1 1 1
1 0 0 1 0 1
1 1 0 1 1 1
1 0 0 0 0 1
1 1 1 1 1 1
```

Figura 7: Formato do arquivo txt recebido

onde, na primeira linha encontram-se os parâmetros do labirinto: o primeiro número representa as dimensões do labirinto, o segundo e terceiro representam, respectivamente, a linha e a coluna em que Teseu inicia sua jornada enquanto os dois últimos números representam as coordenadas do ponto em que a espada espera por nosso herói.

Para a leitura desses dados a seguinte estrutura de dados foi implementada:

```
4 typedef struct
5 {
6     int N;//Dimensão do Labirinto
7     int x, y;//Ponto de Entrada no Labirinto
8     int sx, sy;//Destino, Ponto onde está a espada
9     int **mapa;//Matriz, Mapa do Labirinto
10 }Labirinto;//Estrutura de Dados para o Labirinto
11
```

Figura 8: Estrutura de Dados Labirinto

Além dessa estrutura de dados, na TADLabirinto foram implementadas também diversas funções para a leitura, exploração, manipulação em geral do labirinto. A seguir encontram-se suas descrições:

2.1.2.1 Labirinto* LeLabirinto(char *entrada);

```
13 Labirinto* LeLabirinto(char *entrada);
14 /*-------------------------------------------------------------
15 Prototipo : Labirinto* LeLabirinto(char *entrada);
16 Funcao : Abre o txt contendo os dados sobre o labirinto, aloca a estrutura do Labirinto
17 e Grava os dados do arquivo em uma variavel do tipo Labirinto (struct criada)
18 Entrada : String contendo o nome do arquivo em que o labirinto se encontra
19 Saida: Ponteiro para uma variavel do tipo Labirinto contendo os dados lidos do arquivo txt
20 -------------------------------------------------------------*/
21
```

Figura 9: Função LeLabirinto

Essa função abre um arquivo txt, cujo nome é passado por referência na string entrada. Inicialmente, o arquivo com os dados do Labirinto é aberto, utilizando a um ponteiro do tipo FILE, utilizando o comando *fopen* com argumento *r* para a leitura de valores. Depois, é alocada dinamicamente a memória necessária para o Labirinto por meio do comando *malloc*, e então, após a confirmação de que a memória foi alocada e o arquivo foi aberto corretamente, os parâmetros contidos na linha inicial do arquivo são lidos e salvos nas respectivas variáveis internas à estrutura Labirinto. Então, é alocada a memória para a matriz que vai conter o mapa do labirinto. Também utilizando o comando *malloc* para alocar as linhas, dessa vez é necessário utilizar também um *loop* para que as colunas sejam alocadas. A alocação é confirmada e então o mapa é lido de fato, tendo como recurso dois *fors* encadeados, um para varrer as linhas e outros para varrer as colunas da recém-alocada matriz mapa. Os dados do arquivo txt são lidos e posicionados corretamente na matriz mapa, finalmente, o arquivo fonte é fechado e um ponteiro para a variável do tipo Labirinto é retornado.

2.1.2.2 int CaminhaLabirintoRecursivo(Labirinto *lab, int x, int y, int **sol);

```
22
23 int CaminhaLabirintoRecursivo(Labirinto *lab, int x, int y, int **sol);
24 /*-------------------------------------------------------------
25 Prototipo : int CaminhaLabirintoRecursivo(Labirinto *lab, int x, int y, int **sol);
26 Funcao : Percorre, de forma recursiva, todos os possíveis caminhos do Labirinto até encontrar, ou não, o ponto em
27 Entrada : Variável do tipo Labirinto contendo os dados do Labirinto, coordenada x e y em que a execução
28 se encontra, matriz solução.
29 Saida: Inteiro que informa se existe ou não caminho para o ponto em que a espada se encontra.
30 -------------------------------------------------------------*/
31
```

Figura 10: Função CaminhaLabirintoRecursivo

Com essa função Teseu caminha pelo labirinto. Existem condições iniciais para a garantia de que o labirinto lido é válido, como se o ponto em que Teseu entra é uma parede e se o ponto em que a espada se encontra é uma parede. Depois, vem a condição de parada da recursividade, que é: se a posição chamada for a posição em que a espada se encontra. Caso seja, o ponto em que a espada se encontra é marcado como parte do caminho correto e então, retorna-se 1(um) indicando que existe ao menos uma solução para este labirinto. Depois vem as chamadas recursivas realmente. Caso a condição de parada ainda não tenha sido atingida, duas coisas são checadas, primeiro se a posição na qual a função foi chamada não é parede. Se a posição não for parede, o próximo passo é checar se a função já foi chamada para essa posição. Caso o ponto atual não seja parede **e** a função ainda não tenha sido chamada para aquele ponto, marca-se na matriz *sol* como ponto em que Teseu já esteve. Depois vem as condições para chamada da função recursivamente. Existe um *if* para checar se a posição em que a função será chamada pertence ao labirinto. Se ela pertencer, a função é chamada recursivamente para cada vizinho da posição atual dentro de um *if*. A função é chamada recursivamente até que o ponto de chamada seja o ponto da espada. Quando isso acontecer, será retornado 1(um), o que terá uma reação em cascata nas chamadas anteriores da árvore de execução, confirmando a condição de cada um dos *ifs* que chamaram a função para o caminho correto. Por outro lado, para os caminhos que não chegam à espada, eventualmente chegam à uma parede, nesse caso, o primeiro *if* (que checa se o ponto em que a função chamada é parede **e** se a função já foi chamada para aquele ponto) não é satisfeito, e 0(zero) é retornado, esse resultado é então propagado até que toda a árvore de execução retorne 0(zero) denotando a impossibilidade de um caminho até a espada para aquele labirinto.

2.1.2.3 int** AlocaSolucao(int n);

```
32
33 int** AlocaSolucao(int n);
34 /*-------------------------------------------------------------
35 Prototipo : int** AlocaSolucao(int n);
36 Funcao : Aloca uma matriz n por n e retorna um ponteiro para ela.
37 Entrada : inteiro contendo as dimensões da matriz
38 Saida: Ponteiro para a matriz alocada
39 -------------------------------------------------------------*/
```

Figura 11: Função AlocaSolucao

Essa função aloca uma matriz solução para conter o o caminho correto. Inicialmente, é declarado um ponteiro para ponteiros é são alocadas n(parâmetro de entrada) linhas, dessa vez utilizando *calloc*, simplesmente porque além de alocar a memória, essa função zera as posições de memória alocadas. Após a confirmação de que a memória foi alocada, há um laço para que as colunas sejam alocadas. No laço há ainda a confirmação de que a memória foi alocada. Após isso, se nenhum erro foi encontrado, a função retorna o ponteiro para ponteiros criado.

2.1.2.4 int CaminhaLabirintoIterativo(Labirinto *lab, int **sol);

```
43 int CaminhaLabirintoIterativo(Labirinto *lab, int **sol);
44 /*-------------------------------------------------------------
45 Prototipo : int CaminhaLabirintoIterativo(Labirinto *lab, int **sol);
46 Funcao : Percorre, de forma iterativa todos os possíveis caminhos do Labirinto até encontrar, ou não, o ponto em que está a espada.
47 Grava na matriz Sol um caminho correto até a espada.
48 Entrada : Variável do tipo Labirinto contendo os dados do Labirinto, matriz solução.
49 Saida: Inteiro que informa se existe ou não caminho para o ponto em que a espada se encontra.
50 -------------------------------------------------------------*/
```

Figura 12: Função CaminhaLabirintoIterativo

Essa função percorre todos os caminhos possíveis, checa se há um caminho até a espada e gera a matriz solução, se um caminho for encontrado. Inicialmente é declarado um ponteiro do TipoPilha e a memória necessária para uma pilha é alocada dinamicamente utilizando o comando *malloc*. São criados um ponto que vai percorrer o mapa, e um contador. Uma pilha vazia é criada no endereço alocado anteriormente. As mesmas condições iniciais da função recursiva são checadas: se a espada se encontra em uma parede e se a entrada é uma parede. Caso alguma dessas condições seja verdadeira, a memória alocada para a pilha é liberada e retorna-se 0(zero) informando que é impossível alcançar o ponto da espada para o labirinto dado. Se a entrada não for uma parede, o ponto de entrada na matriz solução é marcado como 1(um), porque ele, necessariamente, fará parte de qualquer solução encontrada. Depois é iniciado um laço de repetição que executa o bloco enquanto o ponto que percorre o mapa não alcance o ponto em que a espada se encontra. Dentro do laço as estão as mesmas condições da função recursiva: se o vizinho para a qual será feita a comparação se encontra dentro do mapa e então se a posição vizinha da vez (direita, cima, esquerda ou baixo) é parede ou passagem. Caso ela seja passagem, marca-se no mapa que Teseu já passou por ali, a posição atual de Teseu é atualizada e o ponto atual é empilhado na pilha, afinal, ele pode fazer parte da solução. Após essas condições para que Teseu ande, existe uma outra condição que checa se Teseu está em um beco sem saída. Caso esteja, o ponto em que ele se encontra é marcado no mapa. Depois, há uma condição para checar se a pilha está vazia. Caso ela não esteja, quer dizer que Teseu se encontra no interior do labirinto, porém está em um beco sem saída. Então ele volta para sua posição anterior, utilizando como artífício a pilha, desempilhando a posição em que ele se encontrava antes de chegar ao beco sem saída. Depois que Teseu voltou uma posição há uma condição para checar se ele ainda se encontra em um beco sem saída. Se ele ainda estiver, a execução continua, mas se Teseu ainda tiver possíveis caminhos partindo do ponto em que ele se encontra, sua posição é reempilhada, afinal aquele ponto pode fazer parte da solução final. Entretanto, se a pilha estiver vazia no momento em que Teseu chegou a um beco sem saída (Vale ressaltar que para todos os efeitos, Teseu considera os pontos em que já esteve como “paredes”) isso quer dizer que ele voltou à primeira posição empilhada, e não encontrou nenhum caminho. Logo não existem caminhos possíveis. Nesse caso a memória alocada para a pilha é liberada e 0(zero) é retornado. Caso o *loop* chegue ao fim isso quer dizer que o caminho foi encontrado e melhor ainda, como todos os pontos que faziam parte de algum caminho errado foram desempilhados, na pilha encontra-se um caminho correto! Então, após o fim do laço, existe um segundo *while*. Ele repete o bloco seguinte enquanto a pilha não estiver vazia. No bloco do *while*, o ponto p recebe as coordenadas desempilhadas da pilha e marca 1(um) na posição referente da matriz solução (toda zerada até o momento) formando um caminho possível da entrada de Teseu até a espada. Após a execução desse *loop*, a memória alocada para a pilha é liberada e 1(um) é retornado, denotando que existe pelo menos um caminho até a espada.

2.1.2.5 void GeraSaida(int **sol, int n, char *entrada, int DeuBom);

```
54 void GeraSaida(int **sol, int n, char *entrada, int DeuBom);
55 /*------------------------------------------------------------
56 Prototipo : void GeraSaida(int **sol, int n, char *entrada, int DeuBom);
57 Funcao : Gera um arquivo txt contendo um caminho até a espada, se existir, senão gera um arquivo com um '0'.
58 Entrada : Matriz solução, contendo o caminho (se existir), inteiro contendo as dimensões da matriz, string com
59 o nome do arquivo a ser gerado, inteiro que informa se há caminho ou não.
60 Saida:
61 ------------------------------------------------------------*/
```

Figura 13: Função GeraSaida

Essa função gera um arquivo txt de saída, informando o resultado da aventura de Teseu. Inicialmente é criado um ponteiro do tipo FILE para lidar com o arquivo. O arquivo cujo nome está guardado na *string* entrada (passada como parâmetro) é criado ou sobrescrito comando *fopen*, argumento *w*. Há então um *if* que avalia o resultado da execução. No início, a função recebeu um inteiro como parâmetro. Nele se encontra o resultado da execução, se um caminho tiver sido encontrado, a variável *DeuBom* terá valor 1(um), caso contrário, seu conteúdo será 0(zero). Então se a execução tiver encontrado um possível caminho, dois *loops* são percorridos, varrendo a matriz *sol* e gravando seu conteúdo no arquivo txt criado. Caso o resultado da exploração tenha sido negativo, apenas um número 0(zero) é gravado no txt de saída. Depois o arquivo é fechado.

2.1.2.6 void LiberaMemoria(Labirinto *lab, int **sol);

```
63 void LiberaMemoria(Labirinto *lab, int **sol);
64 /*------------------------------------------------------------
65 Prototipo : void LiberaMemoria(Labirinto *lab, int **sol)
66 Funcao : Libera a memória alocada para a variável Labirinto e a matriz Solução
67 Entrada : Variável do tipo Labirinto contendo os dados do Labirinto, matriz solução.
68 Saida:
69 ------------------------------------------------------------*/
```

Figura 14: Função LiberaMemoria

Esse procedimento é utilizado para liberar a memória alocada dinamicamente durante todo o processo. Ele recebe o ponteiro para o labirinto e o ponteiro para ponteiros da solução. Primeiro existe um *for* para desalocar as colunas da matriz solução, depois as linhas são desalocadas. Após isso, as colunas da matriz mapa do labirinto são desalocadas e então suas linhas. Por fim todo o labirinto é desalocado.

2.2 Função Principal

A função principal do código final une as TADs fazendo que todas as funções implementadas trabalhem com o propósito de desvendar o labirinto. A seguir a função é esmiuçada:

```c
#include "TADLabirinto.h" //
#include "TADPilha.h"     //Inclusão das TADs

int main(int argc, char *argv[])//Assinatura da função principal com os argumentos como parâmetros
{
    Labirinto *lab = LeLabirinto(argv[1]);//Declara um ponteiro para uma variável do tipo Labirinto e le os dados de um txt
    int DeuBom,i,j;//inteiros para loop e resposta da situação final do código
    int **sol = AlocaSolucao(lab->N);//Declara um ponteiro para ponteiros e aloca a matriz
    printf("Labirinto Lido com Sucesso:\n");
    printf("Entrada de Teseu: [%d] [%d].\nPosicao da espada: [%d] [%d]\n\n",lab->x, lab->y, lab->sx, lab->sy);
    if(argv[3][0] == '0')//O terceiro argumento é 0 (Execução deve ser Recursiva)?
    {//sim
        printf("Utilizando funcao Recursiva\n");
        DeuBom = CaminhaLabirintoRecursivo(lab, lab->x, lab->y, sol);//Chama função recursiva
    }
    else//não
    {
        printf("Utilizando funcao Iterativa\n");
        DeuBom = CaminhaLabirintoIterativo(lab, sol);//Chama função iterativa
    }
    if(DeuBom == 1)//O caminho foi encontrado?
        printf("\n\nDeu Bom! :D\nUm caminho até a espada foi encontrado:\n");//Sim
    else//Não
        printf("\n\nDeu Ruim! :/\nNao existe caminho até a espada.\n");

    GeraSaida(sol, lab->N, argv[2], DeuBom);//Chama a função que gera a saída
    printf("\nArquivo %s gerado com Sucesso.\n", argv[2]);
    LiberaMemoria(lab, sol);//Chama a função que Libera a memória
    printf("Memória Liberada, no Leak ;)\n");
    return 0;
    //Fim da execução
}
```

Figura 15: Função main

A função main chama as funções implementadas anteriormente organizando a ordem de execução e informando ao usuário o *status* de execução momentâneo.

3. Resultados

Após a implementação do programa diversos Labirintos foram testados para a validação do programa. Os resultados são expostos e discutidos a seguir.

3.1 Análise de Complexidade

3.1.1 Em função do Espaço

Para a resolução do Matrizes problema certo espaço de memória é necessário. Tomando como parâmetro significativo as matrizes temos:

Requisitos de Memória

Matrizes	lab→mapa	sol
Tamanho	n^2	n^2

Para a leitura do labirinto e escrita do caminho correto, é necessário salvar todas as posições de uma matriz nxn. Multiplicando o número de linhas pelo número de colunas e tendo em mente que as matrizes são do tipo *int*, obtemos que:

$$E(n) = 2(n^2 \cdot 4 \text{ bytes}) \quad (1)$$

De (1) obtemos que E(n) é dominado assintóticamente por n^2, ou seja é $O(n^2)$. Isso implica que se o tamanho dos labirintos de entrada aumentarem, o espaço necessário para a correta execução do programa aumentará exponencialmente.

3.1.2 Em função do Tempo

3.1.2.1 Pior Caso

3.1.2.1.1 Funções de Complexidade Linear

As funções em que não há nenhum loop ou recursão tem complexidade linear. Ou seja, os seus comandos serão executados apenas uma vez independente do tamanho do labirinto. Essas funções somam uma constante 1 na função de complexidade final. As funções cuja complexidade é linear são:

- FPVazia;
- Empilha;
- Desempilha;
- Tamanho.

3.1.2.1.2 Funções de Complexidade Polinomial (Iterativas)

Nas funções *GeraSaida, LeLabirinto, AlocaSolucao, CaminhaLabirintoIterativo,* a complexidade não é linear. Nela possuímos loops. Isso nos dá que a complexidade de cada uma dessas funções é:

$$(2)\ T(n) = 1 + \sum_{i=0}^{n} a$$

Onde o número 1 representa as operações de complexidade linear executadas fora do loop, a é dado pelo número de operações relevantes no loop de cada uma das funções dependendo do número de loops, outros somatórios podem ser adicionados à função de complexidade no lugar apropriado. Desenvolvendo (2) obtemos:

$$(3)\ T(n) = an(bn) + 1 = an^2 + 1$$

Para *GeraSaida,* como apenas um loop é utilizado e considerando a operação de imprimir em um txt como operação relevante, temos que a = 1 assim sendo, obtemos:

$$T(n)_{GeraSaida} = n^2 + 1$$

Para *LeLabirinto*, temos um loop para alocação da matriz mapa de 0 até n e dois loops encadeados que também vão de 0 até n. Assim sendo (2) muda para:

$$(4)\ T(n) = 1 + \sum_{i=0}^{n} a + \sum_{j=0}^{n} b \sum_{k=0}^{n} c$$

E, como a = b = c = 1 temos que :

$$T(n)_{LeLabirinto} = 1 + n + [n(n)] = n^2 + n + 1$$

Para *AlocaSolucao,* temos um loop de 0 até n para alocar as colunas da matriz solução. Além disso, a = 2, (2) volta a valer, então temos que:

$$T(n)_{AlocaSolucao} = 2n^2 + 1$$

Para *CaminhaLabirintoIterativo*, temos um loop que, no pior caso, vai de 0 até n^2 e um que possui um caminho correto, que, novamente, para o pior caso vai de 0 até n^2. Então sua função de complexidade é dada por:

$$T(n)_{CamLabIter} = 1 + \sum_{i=0}^{n^2} a + \sum_{j=0}^{n^2} b$$

Desenvolvendo $T(n)_{CamLabIter}$, considerando comparações como operações significativas e sabendo que a é o número de operações dentro do primeiro loop, a = 13 e considerando atribuições como operações significativas para o segundo loop, b o número de operações dentro do segundo loop e b = 2, obtemos:

$$T(n)_{CamLabIter} = 1 + 13n^2 + 2n^2 = 15n^2 + 1$$

3.1.2.1.2 Função Recursiva (Equação de recorrência)

A complexidade de uma função Recursiva é dada pela Equação de Recorrência, dada por:

$$\begin{cases} T(n) = aT(b) + c \\ T(1) = 1 \end{cases}$$

Onde a é o número de chamadas recursivas, b é o tamanho do problema, na próxima chama e c é o número de operações significativas executadas. Para o código implementado:

- a = 4, pois a função é chamada recursivamente 4 vezes, uma dentro de cada if;
- b = n – (i-1), onde i é incrementado a cada chamada da recursão, como n^2 tem de ser percorrido por completo, conclui-se que a cada chamada da recursão um passo foi dado, ou seja, um passo a menos é necessário.
- c = 2, pois a cada chamada duas operações são executadas (atribuição na matriz sol e o retorno).

Dessa forma, a Equação de Recorrência fica:

$$T(n) = \cancel{4T(n-1)} + 2$$
$$\cancel{T(n-1)} = \cancel{8T(n-2)} + 2$$
$$\cancel{T(n-2)} = \cancel{16T(n-3)} + 2$$
$$\cancel{T(n-i)} = 4^i T(n-(i-1)) + 2$$

$$T(1) = T(n-(i-1)) \Leftrightarrow 1 = n - (i - 1) \Leftrightarrow n = i$$

$$T(n)_{CamLabRec} = \sum_{i=0}^{n} 4^i = \frac{4(4^n - 1)}{3}$$

Para a execução do programa temos duas opções, a execução de forma recursiva e a de forma interativa. Portanto possuímos também duas Funções de Complexidade. A Função de Complexidade é dada pela soma da função de complexidade dos blocos executados. Portanto, a Função de Complexidade quando o programa é executado de forma iterativa é dada por:

$$T(n) = T(n)_{AlocSol} + T(n)_{LeLab} + T(n)_{GerSaid} + T(n)_{CamLabIter}$$

Logo,

$$T(n) = (15n^2 + 1) + (2n^2 + 1) + (n^2 + n + 1) + (n^2 + 1) = 19n^2 + n + 5$$

Por outro lado, considerando a função recursiva:

$$T(n) = T(n)_{AlocSol} + T(n)_{LeLab} + T(n)_{GerSaid} + T(n)_{CamLabRec}$$

Logo,

$$T(n) = (2n^2+1) + (n^2+n+1) + (n^2 + 1) + \frac{4(4^n - 1)}{3}$$

3.1.2.2 Melhor Caso

Para o melhor caso, é considerado quando a espada está localizada logo na entrada do labirinto, nesse caso, as complexidades de *CaminhaLabirintoRecursivo* e *CaminhaLabirintoIterativo* tornam-se lineares. Nesse caso, o programa passa a ter uma única função de complexidade, uma vez que $T(n)_{CamLabiter} = T(n)_{CamLabRec}$, dessa forma T(n) é dado por:

$$T(n) = T(n)_{AlocSol} + T(n)_{LeLab} + T(n)_{GerSaid} + 1$$

$$T(n) = (2n^2 + 1) + (n^2+n+1) + (n^2 +1) + 1 = 4n^2 + n + 4$$

3.1.2.3 Caso Médio

Para um grande número de entradas aleatórias é coerente imaginar que o número de passos que Teseu precisará caminhar estará próximo a $n^2/2$ porque, apesar de ocorrer um desvio desse valor, estatisticamente é provável que esse desvio se anule, tanto acima quanto abaixo do valor de $n^2/2$. Nesse caso, a função de complexidade de *CaminhaLabirintoIterativo* muda para $T(n)_{CamLabIter} = 1 + (15n^2/2)$, e a função do programa passa a ser $T(n) = 11{,}5n^2 + n + 5$. No entanto esses resultados não puderam ser comprovados, por falta de ferramentas físicas e didáticas.

2.2 Fase de Testes e Discussão dos Resultados

Foram feitos diversos testes com o programa, foram utilizados os labirintos disponibilizados no fórum, além de inúmeras variações dos mesmos. Foram feitos testes com o melhor caso e ficou estabelecido que o tempo de execução também cresce exponencialmente com n, no entanto, o tempo de execução continuava significativamente menor do que o pior caso, por exemplo, para um labirinto com n = 10000, o tempo de execução do pior caso foi t=15,552s e para o melhor caso foi t=14,528. Por conta da disponibilidade de tempo, a análise dos resultados foi focada no pior caso. Utilizando o comando *time* o tempo de execução para diversos tamanhos do pior caso a seguinte tabela foi montada:

	Tempo de Execução em s	
N	Iterativo	Recursivo
100	0,008	0,006
200	0,026	0,022
300	0,051	0,043
400	0,083	0,075
500	0,117	0,103

Foram plotados gráficos a partir dessa tabela, um para a função Recursiva e um para a Iterativa. Eles se encontram a seguir:

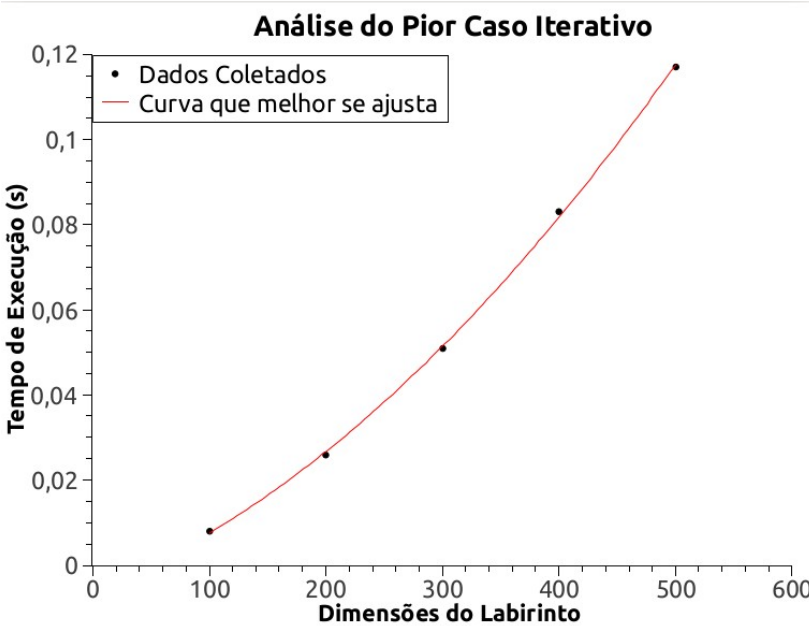

Figura 16: Gráfico Pior Caso Iterativo

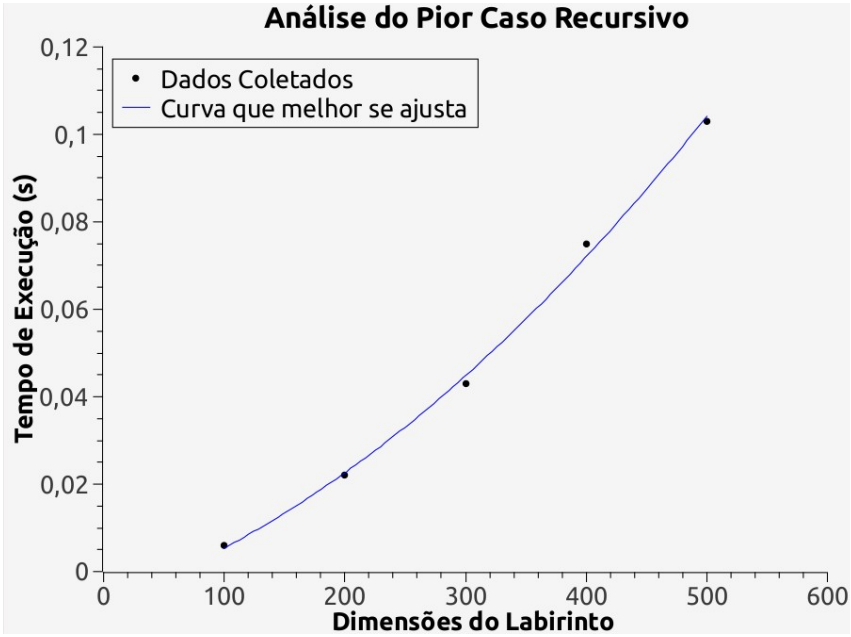

Figura 17: Gráfico Pior Caso Recursivo

Por meio da aproximação da curva que mais se ajusta aos pontos é possível concluir que os gráficos aproximam de sua dominação assintótica. Foi possível perceber também que para valores de n baixos, como os mostrados no gráfico, o algoritmo recursivo era viável. No entanto quando os valores aumentam, a complexidade do

recursivo aumenta muito rápido, tornando o algoritmo inviável. Para comprovar essa afimação, foi executado um teste com um labirinto 1000x1000 e que, devido ao posicionamento da espada, o programa varreria as n^2 posições. Os resultados foram t = 15,552s para a função Iterativa e a função Recursiva não conseguiu encontrar resultados. A função de complexidade encontrada foi polinomial, e ficou claro que a melhor maneira de se resolver o problema foi a Iterativa, pois ela é válida para qualquer n, enquanto a Recursiva não resolve o problema a partir de n = 500 (Valor econtrado empíricamente).

4. Conclusão

As principais dificuldades encontradas foram com a assimilação e manipulação correta de conceitos novos, tanto na hora de implementar o código, quanto na hora da análise de complexidade. No entanto, após estudo e pesquisa, o código e a análise ficaram mais fáceis. Então considero que o objetivo do TP foi concluído, uma vez que após sua execução me sinto um melhor programador, capaz de lidar com diversos tipos de problemas. Gostaria de ressaltar que foi muito divertido fazer esse trabalho. O tempo de desenvolvimento de código foi gasto com prazer e nesses momentos, eu tinha a sensação de estar jogando um jogo. Espero que os próximos TPs sejam tão divertidos quanto este.