

SERVIÇO PÚBLICO FEDERAL · MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE VIÇOSA · UFV
CAMPUS FLORESTAL

Trabalho 1

Projeto e Análise de Algoritmos

Labirinto Backtracking

Aymê Faustino dos Santos - 4704
Matheus Nascimento Peixoto - 4662
Matheus Nogueira Moreira - 4668

Sumário

| | |
|--------------------------------------------------|-----------|
| 1) Introdução | 3 |
| 2) Organização do Projeto | 3 |
| 3) Desenvolvimento | 4 |
| 3.1) Leitura dos arquivos e criação do Labirinto | 4 |
| 3.2) Arquivo “CaminhoRamificado” | 4 |
| 3.3) Funcionamento do Backtracking | 5 |
| 3.4) Funcionalidades “extras” | 8 |
| 4) Compilação e Execução | 10 |
| 5) Resultados | 10 |
| 6) Conclusão | 12 |
| 7)Referências | 12 |

1) Introdução

Neste documento, será apresentada uma análise detalhada do projeto prático desenvolvido na linguagem de programação C, como parte da matéria Projeto e Análise de Algoritmos (CCF330). O trabalho envolve a criação de uma solução para um problema inserido em um jogo inspirado nas aventuras de Indiana Jones, no qual o objetivo é encontrar um caminho para que o personagem alcance o tesouro. Nesse contexto, é obrigatório o uso da técnica de backtracking.

O backtracking é uma técnica utilizada para encontrar soluções para problemas computacionais. Ela trabalha de forma sistemática, explorando todas as possíveis soluções para um problema até encontrar a correta. Esta documentação detalha o processo de criação desse algoritmo.

2) Organização do Projeto

Como é possível observar na **Figura 1**, o projeto foi dividido entre as pastas “**Headers**”, onde se encontram as definições dos tipos (TAD’s), juntamente com os cabeçalhos das funções, “**src**”, onde se encontram as implementações das funções que fazem parte do projeto e a pasta “**ArquivosTeste**”, pasta na qual estão contidos os arquivos do tipo “.txt” que servirão para testes.

Além dessas pastas existe o arquivo **main.c** e o arquivo **makefile**.

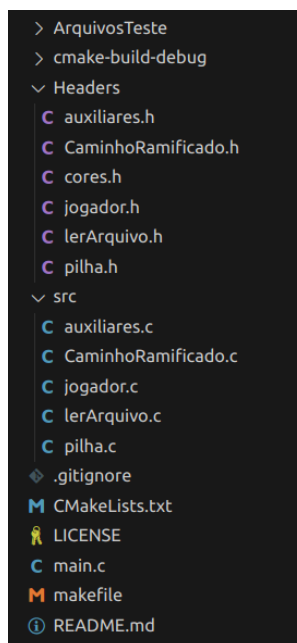


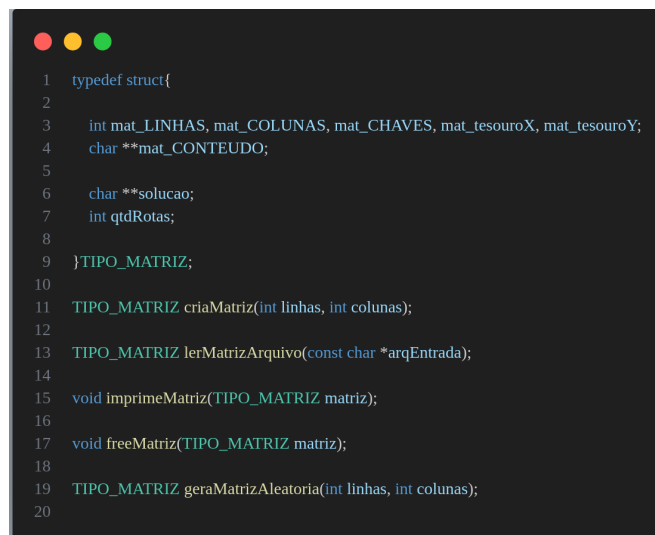
Figura 1: Organização do projeto

3) Desenvolvimento

Neste capítulo serão apresentados alguns dos principais pontos para o desenvolvimento do projeto para a resolução do Labirinto com Backtracking.

3.1) Leitura dos arquivos e criação do Labirinto

Para a interpretação dos arquivos e já criação da matriz, a qual representará o labirinto, existe o arquivo “**lerArquivo**”, onde uma matriz é definida de modo que todo o seu conteúdo seja armazenado, inclusive a posição exata do tesouro, a quantidade de chaves, as possíveis soluções e a quantidade de rotas já encontradas, valor esse que é incrementado a cada solução válida. Tais comentários são observáveis na **Figura 2**, a qual mostra a definição desse tipo.



```
1  typedef struct{
2
3      int mat_LINHAS, mat_COLUNAS, mat_CHAVES, mat_tesouroX, mat_tesouroY;
4      char **mat_CONTEUDO;
5
6      char **solucao;
7      int qtdRotas;
8
9  }TIPO_MATRIZ;
10
11  TIPO_MATRIZ criaMatriz(int linhas, int colunas);
12
13  TIPO_MATRIZ lerMatrizArquivo(const char *arqEntrada);
14
15  void imprimeMatriz(TIPO_MATRIZ matriz);
16
17  void freeMatriz(TIPO_MATRIZ matriz);
18
19  TIPO_MATRIZ geraMatrizAleatoria(int linhas, int colunas);
20
```

Figura 2: arquivo “lerArquivo.h”

3.2) Arquivo “CaminhoRamificado”

Neste arquivo se encontra a implementação de uma outra pilha, a qual será de grande importância para guardar as ramificações que serão encontradas durante o percurso do labirinto (percurso da matriz).

Como será visto no subcapítulo a seguir, essa pilha servirá para o jogador compreender os caminhos que já foram percorridos, guardando as ramificações, possibilitando posteriormente que estas sejam interpretadas para que não volte a percorrer o mesmo caminho que anteriormente fora percorrido.

Para ilustrar sua implementação, na **Figura 3** será possível visualizar sua definição e o cabeçalho de suas funções no arquivo “**CaminhoRamificado.h**”.

```

1  typedef struct CaminhoRamificado{
2
3      int RamificadoEsquerda,RamificadoDireita,RamificadoCima,RamificadoBaixo;
4      int Ramificado_Coluna,Ramificado_Linha;
5      struct CaminhoRamificado *prox;
6
7  }CaminhoRamificado;
8
9  typedef struct PilhaRamificado{
10
11      CaminhoRamificado *topo;
12
13  }PilhaRamificado;
14
15  void inicializarRamificacao(PilhaRamificado *ptr);
16  int estaVazioRamificacao(PilhaRamificado *ptr);
17  void pushRamificacao(PilhaRamificado *ptr,int line,int collun);
18  void popRamificacao(PilhaRamificado *ptr);
19  int EstaPresentePilhaRamificado(PilhaRamificado *ptr,int line ,int collun);
20  void ApresentarRamificacao(PilhaRamificado *ptr);
21  void LiberarRamificacao(PilhaRamificado *ptr);

```

Figura 3: Definição da pilha de ramificações

3.3) Funcionamento do Backtracking

Nos arquivos “**jogador.h**” e “**jogador.c**”, estão contidas as funções fundamentais relacionadas ao funcionamento do algoritmo de backtracking. O arquivo .h não apenas contém as definições das funções, mas também define vários tipos abstratos de dados. Estes tipos são inicializados com dados provenientes de um arquivo lido no início da execução do programa, graças à função “**inicializarJogador**”. Além das funções, o arquivo “.h” é responsável por estabelecer a estrutura dos dados, inicializando campos como uma matriz que armazena os valores lidos no arquivo de texto. Essa matriz retém as informações do arquivo, incluindo suas dimensões, com “**TamanhoJogadorX**” representando o número de linhas e “**TamanhoJogadorY**” o número de colunas. Além disso, o campo “**chavesColetadas**” mantém o registro do número de chaves já coletadas, começando com um valor inicial de 0.

Na **Figura 4**, apresentada a seguir, é possível observar como ficou tal arquivo, definindo os tipos que serão utilizados.

```

1  #include "../Headers/pilha.h"
2
3  typedef struct ConteudoJogador{
4
5      char Valor;
6
7  }ConteudoJogador;
8
9  typedef struct MatrizJogador{
10
11      //A matriz Jogador
12      ConteudoJogador **ConteudoJogador;
13      }MatrizJogador;
14
15  typedef struct Jogador{
16
17      //Chaves coletadas, inicia com 0
18      //Dimensões da matriz Jogador
19      int ChavesColetadas,TamanhoJogadorX,TamanhoJogadorY;
20      //Interação com a struct MatrizJogador
21      MatrizJogador *PtrJ;
22
23      int contadorBacktracking;
24  }Jogador;
25
26  void inicializarJogador(Jogador *Ptr, TIPO_MATRIZ Var_TipoMatriz);
27  void Movimentar(Jogador *Ptr, TIPO_MATRIZ *Var_TipoMatriz, PilhaCoordenadas *pilha, PilhaCoordenadas *pilhaChaves, PilhaRamificado *pilhaRamificacao, FilaPilhas *Fila);
28  int backtracking(Jogador *Ptr, TIPO_MATRIZ *Var_TipoMatriz, int *I, int *J, PilhaCoordenadas *pilhaChaves, PilhaCoordenadas *pilha, int *I_Ramificacao, int *J_Ramificacao, int *control, PilhaRamificado *pilhaRamificacao, FilaPilhas *Fila);
29  int ExisteRamificacao(Jogador *Ptr, int I, int J);
30  int EstaPresentePilhaChaves(Jogador *Ptr, PilhaCoordenadas *pilhaChaves, int line, int collun);
31  int LimparMatriz(Jogador *Ptr, PilhaCoordenadas *pilha, PilhaCoordenadas *pilhaChaves, int *I_Ramificacao, int *J_Ramificacao);
32  void Printar(Jogador *Ptr);
33
34  int ExisteDireita(Jogador *Ptr, int I, int J);
35  int ExisteEsquerda(Jogador *Ptr, int I, int J);
36  int ExisteBaixo(Jogador *Ptr, int I, int J);
37  int ExisteCima(Jogador *Ptr, int I, int J);

```

Figura 4: Arquivo jogador.h

Após a inicialização, a função “**Movimentar**” é chamada, juntamente com um conjunto de parâmetros essenciais para a execução do backtracking. Todos esses parâmetros são inicializados no escopo da função. Entre eles, estão incluídas três pilhas que são inicializadas com suas funções correspondentes. Por fim, a função 'backtracking' é chamada para dar continuidade ao processo. É possível observar isto na **Figura 5**, apresentada a seguir.

```
1 void Movimentar(Jogador *Ptr, TIPO_MATRIZ *Var_TipoMatriz, PilhaCoordenadas *pilha, PilhaCoordenadas *pilhaChaves, PilhaRamificado *pilhaRamificacao, FilaPilhas *Fila){
2     int X,Y;
3     int a,b;
4     int control = 0;
5     a = Var_TipoMatriz->mat_LINHAS;
6     b = Var_TipoMatriz->mat_COLUNAS;
7     X = Y = 0;
8     initialize(pilha);
9     initialize(pilhaChaves);
10    inicializarRamificacao(pilhaRamificacao);
11    initializeFilaPilhas(Fila);
12
13    backtracking(Ptr, Var_TipoMatriz, &X, &Y, pilhaChaves, pilha, &a, &b, &control, pilhaRamificacao, Fila);
14
15 }
```

Figura 5: Função movimentar

Dentro do algoritmo de backtracking, várias verificações condicionais são realizadas para diferentes valores possíveis encontrados pelo jogador na matriz. Primeiro, se o jogador encontra uma chave, suas coordenadas são adicionadas à pilha “**pilhasChaves**”, e o número de chaves coletadas é incrementado. Em seguida, se o tesouro é encontrado com o número correto de chaves, suas coordenadas são inseridas em uma pilha, a qual armazena as coordenadas válidas até o tesouro. Esta pilha é, por sua vez, salva em uma fila de pilhas, criada para armazenar todos os caminhos possíveis até o tesouro.

Esta **fila de pilhas** é definida no arquivo “pilha.h”, tendo o desenvolvimento de suas funções em “pilha.c”. A seguir, na **Figura 6**, é possível observar sua definição.

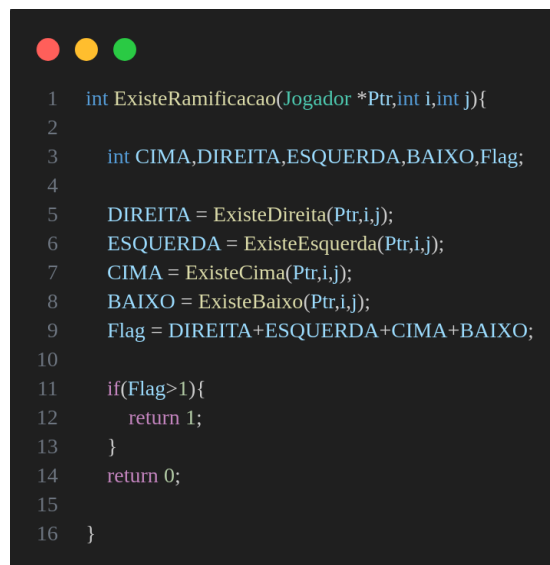
```
1 typedef struct Coordenadas {
2     int line, collun;
3     int QuantidadeElementos;
4     struct Coordenadas* next;
5 } Coordenadas;
6
7 typedef struct PilhaCoordenadas {
8     //int qtd;
9     Coordenadas* topo;
10 } PilhaCoordenadas;
11 //FILA DE Pilhas
12 typedef struct Node {
13     PilhaCoordenadas pilha;
14     struct Node* next;
15 } Node;
16
17 typedef struct FilaPilhas {
18     Node* front;
19     Node* rear;
20 } FilaPilhas;
```

Figura 6: Definições em “pilha.h”

Após o armazenamento na fila de pilhas, a posição do tesouro na pilha é retirada, e a variável de controle é modificada para 1. Estes passos têm razões específicas, que serão explicadas em detalhes mais adiante. O desvio condicional é concluído com a função retornando 1.

Por fim, no último desvio condicional, caso uma posição válida ou uma posição com chave seja encontrada, essa posição é marcada como 'V' na matriz e adicionada à pilha de coordenadas. Dentro deste bloco 'if', existe uma estrutura 'if-else' que identifica a presença de uma ramificação na posição recém-chegada. Para tal, existem cinco funções, “**ExisteDireita**”, “**ExisteCima**”, “**ExisteEsquerda**”, “**ExisteBaixo**” e “**ExisteRamificacao**”. As quatro primeiras retornam 1 se as direções direita, cima, esquerda e baixo forem válidas a partir de um ponto [i][j], e 0 caso contrário. Na função “**ExisteRamificacao**”, esses valores de retorno são somados. Se o resultado for maior que 1, significa que há uma ramificação, caso contrário, não existe uma ramificação naquele ponto.

A seguir, na **Figura 7**, é possível observar o funcionamento de tal função.



```
1  int ExisteRamificacao(Jogador *Ptr,int i,int j){
2
3      int CIMA,DIREITA,ESQUERDA,BAIXO,Flag;
4
5      DIREITA = ExisteDireita(Ptr,i,j);
6      ESQUERDA = ExisteEsquerda(Ptr,i,j);
7      CIMA = ExisteCima(Ptr,i,j);
8      BAIXO = ExisteBaixo(Ptr,i,j);
9      Flag = DIREITA+ESQUERDA+CIMA+BAIXO;
10
11     if(Flag>1){
12         return 1;
13     }
14     return 0;
15
16 }
```

Figura 7: Implementação de “ExisteRamificacao”

Se o valor de retorno da função “**ExisteRamificacao**” for 1 ou se as coordenadas [i][j] forem iguais às da última ramificação, representadas por “**I_Ramificacao**” e “**J_Ramificacao**”, o primeiro bloco 'if' é acionado. Os valores da última ramificação são atualizados, e as coordenadas são inseridas numa **pilha de ramificações**. A seguir, é verificado para qual direção a ramificação será explorada. Uma condição é imposta para cada direção: o elemento no topo da pilha de ramificação não deve ter sido explorado para aquela direção ainda (deve estar marcado como -1), e o caminho para aquela direção deve ser válido a partir deste ponto. Se essas condições não forem atendidas para uma direção específica, a ramificação no topo da pilha é marcada com 0 nessa direção. Se as condições forem satisfeitas, a ramificação no topo da pilha é marcada com 1 nessa direção. As variáveis “*i” e “*j” são atualizadas de acordo com a direção válida, e a função de backtracking é chamada recursivamente.

Quando a recursão termina, a ramificação para aquela direção específica foi explorada completamente. Alguns comandos importantes são então executados. Primeiro, se a variável de controle retornou como 1, indicando que o tesouro foi coletado, o número de caminhos válidos é incrementado. Em seguida, é chamada uma função chamada “**LimparMatriz**”, que, resumidamente, restaura a matriz com '0' nas posições válidas e 'C' nas posições de chave, utilizando as pilhas de coordenadas e chaves, até a posição da última ramificação. Após isso, a variável de controle retorna para 0 e $[i][j]$ é restaurado para a posição da última ramificação.

Quando todas as direções da ramificação específica são testadas, a função “**LimparMatriz**” é chamada, e o elemento do topo da **pilha de ramificações** é removido. Os caminhos a partir daquela posição foram explorados completamente. “**I_Ramificacao**” e “**J_Ramificacao**” são atualizados para as coordenadas do novo elemento no topo da **pilha de ramificações**, e a matriz é limpa novamente, desta vez até as novas coordenadas. Finalmente, $[i][j]$ é atualizado com os valores de “**I_Ramificacao**” e “**J_Ramificacao**”, e a função de backtracking é chamada novamente para continuar a exploração de novos caminhos.

O bloco 'else' é acionado quando a posição $[i][j]$ não é uma ramificação, tendo apenas uma direção válida. Para entrar neste bloco, a variável de controle deve ser igual a 0, para evitar que a progressão continue a partir do ponto em que o tesouro foi descoberto. Além disso, deve haver um caminho válido naquela direção e as coordenadas atuais $[i][j]$ não devem ser iguais às coordenadas da ramificação atual. Neste caso, a função retorna 0 e termina quando a recursão é encerrada.

3.4) Funcionalidades “extras”

Dentre as possibilidades de “extras” exemplificadas na documentação, o grupo criou uma função capaz de gerar labirintos aleatórios, a função `geraMatrizAleatoria`, a qual recebe as dimensões do labirinto, define que a quantidade total de chaves não pode ser maior do que 5% da quantidade total de “casas” desse labirinto e define as posições aleatoriamente, com base na “`rand()`” e em um “switch-case”. Caso o número de chaves já tenha atingido o máximo, no lugar são adicionados zeros, a fim de que a possibilidade de caminhos seja maximizada.

Tal funcionalidade pode ser observada na **Figura 8**, demonstrada a seguir.


```

1 TIPO_MATRIZ geraMatrizAleatoria(int linhas, int colunas)
2 {
3     TIPO_MATRIZ mat = criaMatriz(linhas, colunas);
4
5     int totalElementos = linhas * colunas;
6     // as chaves não serão mais do que 5% + 1 do total de elementos
7     int totalChaves = (int)(totalElementos * 0.05) + 1;
8     int contaChaves = 0;
9
10    /*
11     define aleatoriamente a linha e coluna para X,
12     não podendo ser 0 para não coincidir com a posição inicial
13    */
14    int Tesouro_Linha = rand()%linhas;
15    int Tesouro_Coluna = rand()%colunas;
16
17    int numAleatorio;
18
19    //preencher matriz
20    for(int i = 0; i < linhas; i++){
21        for(int j = 0; j < colunas; j++){
22            numAleatorio = rand()%3;
23
24            switch (numAleatorio) {
25                case 0:
26                    mat.mat_CONTEUDO[i][j] = '0';
27                    break;
28                case 1:
29                    mat.mat_CONTEUDO[i][j] = '1';
30                    break;
31                case 2:
32                    if((contaChaves < totalChaves) && (i != 0 && j != 0)){
33                        mat.mat_CONTEUDO[i][j] = 'C';
34                        contaChaves++;
35                    }else{
36                        mat.mat_CONTEUDO[i][j] = '0';
37                    }
38                    break;
39                default:
40                    mat.mat_CONTEUDO[i][j] = '1';
41                    break;
42            }
43        }
44    }
45
46    //define a posição inicial como válida
47    mat.mat_CONTEUDO[0][0] = '0';
48
49    //modo de evitar que a posição do tesouro seja a mesma da posição inicial
50    if(Tesouro_Linha == 0 && Tesouro_Coluna == 0){
51        Tesouro_Linha += 1;
52        Tesouro_Coluna += 1;
53    }
54
55    mat.mat_CONTEUDO[Tesouro_Linha][Tesouro_Coluna] = 'X';
56
57    mat.mat_tesouroX = Tesouro_Linha;
58    mat.mat_tesouroY = Tesouro_Coluna;
59    mat.mat_CHAVES = contaChaves;
60
61    return mat;
62 }

```

Figura 8: Criação de uma labirinto aleatório

Além dessa, existe a possibilidade de apresentação de todos os caminhos que foram encontrados ao final da execução do programa. Isto se dá graças a **fila de pilhas**, anteriormente comentada. Sua visualização será possível na **Figura 11**, que se encontra no capítulo **Resultados**.

Para a impressão dos valores, possibilitando uma melhor visualização, existe o arquivo “**cores.h**”, que possibilita que as impressões realizadas apareçam coloridas, possibilitando uma melhor distinção daquilo que é exibido, além de deixar o terminal mais harmonioso. Sua visualização pode ser observada nas **Figuras de 9 a 12**.

4) Compilação e Execução

Com a criação do arquivo “**makefile**” realizada pelo grupo, para compilar todas os arquivos necessários para a execução do projeto, basta digitar, no terminal, o seguinte comando:

‘make all’

Após isso, para a execução, caso esteja em uma máquina com Linux como sistema operacional, basta digitar

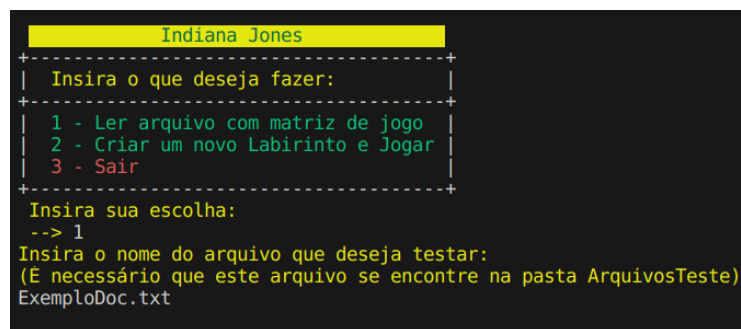
‘make Linux’

Mas, em caso de ser uma máquina com Windows, digite

‘make Windows’

Em todos os casos, para a leitura de arquivos de teste, é preciso que estes se encontrem na pasta “**ArquivosTeste**”, visto que a função que faz a leitura desses arquivos foi criada de modo que já interpreta que os arquivos se encontram nesta pasta, não exigindo assim que o usuário tenha que digitar o caminho completo até seu arquivo e simultaneamente contribui para a organização dos arquivos.

Tal observação também é realizada durante a execução do programa, como pode ser observado na **Figura 9**.



```
Indiana Jones
+-----+
| Insira o que deseja fazer: |
+-----+
| 1 - Ler arquivo com matriz de jogo |
| 2 - Criar um novo Labirinto e Jogar |
| 3 - Sair |
+-----+
| Insira sua escolha: |
|--> 1 |
| Insira o nome do arquivo que deseja testar: |
| (É necessário que este arquivo se encontre na pasta ArquivosTeste) |
| ExemploDoc.txt |
```

Figura 9: Menu inicial

5) Resultados

Como resultados, após a execução e avaliação dos caminhos através do método de **Backtracking**, são apresentados, no caso de tesouro encontrado, a quantidade de caminhos que foram obtidos e permite que o usuário escolha se deseja a exibição de todos os caminhos ou de apenas do melhor caminho encontrado, isto é, o com a menor distância.

Na **Figura 10** é possível observar os resultados encontrados para o labirinto que se encontra na especificação do Trabalho. Na **Figura 11** se encontra a exibição para todos os caminhos encontrados para o mesmo labirinto. Em ambos os testes demonstrados nas **Figuras 9, 10 e 11** foi utilizado por base o arquivo “ExemploDoc.txt”, que é o mesmo exemplo que se encontra na especificação e está presente dentre os arquivos na pasta “ArquivosTeste”. Sua impressão pode ser observada também na **Figura 10**, quando a matriz lida é impressa. Já na **Figura 12**, um labirinto 5 por 5 foi criado aleatoriamente com base na função `geraMatrizAleatoria`, citada anteriormente.

```

----- Matriz Lida -----

[0] [0] [0] [0] [0] [0] [0] [0] [0] [C]
[0] [1] [1] [1] [1] [0] [1] [1] [1] [0]
[0] [0] [0] [0] [0] [0] [0] [0] [1] [0]
[0] [1] [0] [1] [0] [1] [0] [0] [1] [0]
[0] [1] [0] [1] [0] [1] [0] [1] [1] [0]
[0] [1] [0] [1] [0] [1] [0] [0] [0] [0]
[1] [1] [0] [1] [0] [1] [0] [0] [1] [0]
[0] [0] [0] [1] [0] [1] [0] [0] [1] [0]
[C] [1] [1] [1] [0] [1] [1] [1] [1] [1]
[0] [0] [0] [0] [0] [0] [0] [0] [0] [X]

-----

+-----+
| Tesouro Encontrado! |
+-----+
| Caminhos Encontrados: 6 |
+-----+

+-----+
| Qual a saída desejada? |
| [1] Apresentar Melhor Caminho |
| [2] Apresentar Todos os Caminhos Disponiveis |
+-----+
Insira sua escolha:
--> 

```

Figura 10: Primeira parte dos resultados

```

Insira sua escolha:
--> 2

Caminho:
[0] [0] [0] [1] [0] [2] [0] [3] [0] [4] [0] [5] [0] [6] [0] [7] [0] [8] [0] [9] [1] [9] [2] [9] [3] [9] [4] [9] [5] [9] [5] [8] [5] [7] [6] [7] [7] [7] [6] [6] [6]
[5] [6] [4] [6] [3] [6] [3] [7] [2] [7] [2] [6] [2] [5] [2] [4] [2] [3] [2] [2] [3] [2] [4] [2] [5] [2] [6] [2] [7] [2] [7] [1] [7] [0] [8] [0] [9] [0] [9] [1] [9] [2] [9] [3]
[9] [4] [9] [5] [9] [6] [9] [6] [9] [7] [9] [8] [9] [9]

Caminho:
[0] [0] [0] [1] [0] [2] [0] [3] [0] [4] [0] [5] [0] [6] [0] [7] [0] [8] [0] [9] [1] [9] [2] [9] [3] [9] [4] [9] [5] [9] [5] [8] [5] [7] [6] [7] [7] [7] [6] [6] [6]
[5] [6] [4] [6] [3] [6] [3] [7] [2] [7] [2] [6] [2] [5] [2] [4] [2] [3] [2] [2] [3] [2] [4] [2] [5] [2] [6] [2] [7] [2] [7] [1] [7] [0] [8] [0] [9] [0] [9] [1] [9] [2] [9] [3]
[9] [4] [9] [5] [9] [6] [9] [7] [9] [8] [9] [9]

Caminho:
[0] [0] [0] [1] [0] [2] [0] [3] [0] [4] [0] [5] [0] [6] [0] [7] [0] [8] [0] [9] [1] [9] [2] [9] [3] [9] [4] [9] [5] [9] [5] [8] [5] [7] [6] [7] [6] [6] [5] [6] [4] [6]
[3] [6] [3] [7] [2] [7] [2] [6] [2] [5] [2] [4] [2] [3] [2] [2] [3] [2] [4] [2] [5] [2] [6] [2] [7] [2] [7] [1] [7] [0] [8] [0] [9] [0] [9] [1] [9] [2] [9] [3] [9] [4] [9] [5]
[9] [6] [9] [7] [9] [8] [9] [9]

Caminho:
[0] [0] [0] [1] [0] [2] [0] [3] [0] [4] [0] [5] [0] [6] [0] [7] [0] [8] [0] [9] [1] [9] [2] [9] [3] [9] [4] [9] [5] [9] [5] [8] [5] [7] [6] [7] [6] [6] [5] [6] [4] [6]
[3] [6] [2] [6] [2] [5] [2] [4] [2] [3] [2] [2] [3] [2] [4] [2] [5] [2] [6] [2] [7] [2] [7] [1] [7] [0] [8] [0] [9] [0] [9] [1] [9] [2] [9] [3] [9] [4] [9] [5] [9] [6] [9] [7]
[9] [8] [9] [9]

Caminho:
[0] [0] [0] [1] [0] [2] [0] [3] [0] [4] [0] [5] [0] [6] [0] [7] [0] [8] [0] [9] [1] [9] [2] [9] [3] [9] [4] [9] [5] [9] [5] [8] [5] [7] [5] [6] [4] [6] [3] [6] [3] [7]
[2] [7] [2] [6] [2] [5] [2] [4] [2] [3] [2] [2] [3] [2] [4] [2] [5] [2] [6] [2] [7] [2] [7] [1] [7] [0] [8] [0] [9] [0] [9] [1] [9] [2] [9] [3] [9] [4] [9] [5] [9] [6] [9] [7]
[9] [8] [9] [9]

Caminho:
[0] [0] [0] [1] [0] [2] [0] [3] [0] [4] [0] [5] [0] [6] [0] [7] [0] [8] [0] [9] [1] [9] [2] [9] [3] [9] [4] [9] [5] [9] [5] [8] [5] [7] [5] [6] [4] [6] [3] [6] [2] [6]
[2] [5] [2] [4] [2] [3] [2] [2] [3] [2] [4] [2] [5] [2] [6] [2] [7] [2] [7] [1] [7] [0] [8] [0] [9] [0] [9] [1] [9] [2] [9] [3] [9] [4] [9] [5] [9] [6] [9] [7] [9] [8] [9] [9]

Contador Backtracking: 1297

```

Figura 11: Exibição de todos os caminhos possíveis

Já no caso de impossibilidade de se chegar ao baú e/ou chegar neste sem as chaves, é exibido o que se pode observar na **Figura 12**.

```

+-----+
| Insira sua escolha: |
| --> 2 |
| Digite o numero de colunas e linhas: |
| 5 5 |
+-----+

----- Matriz Lida -----

[0] [0] [0] [1] [0]
[1] [X] [1] [1] [1]
[0] [C] [C] [1] [0]
[1] [1] [1] [0] [0]
[1] [0] [0] [1] [0]

-----

Indiana Jones nao consegue abrir o bau :(

Contador Backtracking: 5

```

Figura 12: Exibição no caso de impossibilidade de abertura do baú

É possível observar que em todos os casos, como na **Figura 11** como na **Figura 12** é realizada a contagem de recursões, apresentadas pelo “Contador Backtracking”. No caso do arquivo “ExemploDoc.txt” percebe-se que a recursão foi realizada 1.297 vezes, enquanto no labirinto gerado aleatoriamente apenas 5, visto sua disposição

6) Conclusão

Com a implementação deste projeto, o grupo pôde melhor compreender a técnica de backtracking, servindo bem como um complemento das aulas expositivas ministradas em sala de aula.

Com os resultados obtidos ao final da execução do Trabalho Prático 01 é possível chegar à conclusão de que o grupo atendeu bem as expectativas, obtendo resultados precisos para os mais diversos tipos de testes realizados.

Desse modo, é compreensível que o grupo conseguiu aprender e aplicar os conhecimentos na técnica mencionada, bem como perceber sua importância para a resolução de problemas de mesmo tipo.

7)Referências

IDE's utilizadas:

- CLion: <https://www.jetbrains.com/pt-br/clion/>
- Code With Me: <https://www.jetbrains.com/pt-br/code-with-me/>
- Visual Studio Code: <https://code.visualstudio.com/>

Informações e Dicas:

- StackOverflow: <https://stackoverflow.com/>