

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

Trabalho Prático 02

Projeto e Análise de Algoritmos (CCF 330)

Saída da Caverna com Programação Dinâmica

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

Sumário

1)Introdução.....	3
2)Desenvolvimento do Algoritmo.....	3
2.1) Organização dos Arquivos.....	3
2.2) Funcionamento do Algoritmo ‘Oficial’.....	4
2.2.1) Leitura dos arquivos e Arquivo “LeituraArquivo”	4
2.2.2) Arquivo “Pilha”	4
2.2.3) Arquivo “PD_Estudante” e o funcionamento da Programação Dinâmica.....	5
2.3) Recebimento do arquivo a ser lido.....	10
3)Extras.....	10
3.1) Versão “Heurística”	10
3.2) “Como seria possível permitir todas as movimentações com PD?”	11
3.3) Geração Aleatória de Mapas.....	11
4)Compilação e Execução.....	14
5)Resultados.....	14
6)Conclusão.....	15
7)Referências.....	15

1) Introdução

Após a saída do Labirinto do Trabalho Prático 1, o personagem se viu em uma caverna, a qual poderia estar repleta de Monstros, os quais lhe causariam danos caso passasse por eles, mas também algumas Poções, as quais são capazes de aumentar seus pontos de vida.

Visando o melhor aprendizado e fixação dos conteúdos abordados durante as aulas expositivas de Projeto e Análise de Algoritmos (CCF 330), foi desenvolvido um algoritmo seguindo o método de Programação Dinâmica (o qual será referenciado por PD em alguns momentos desse documento).

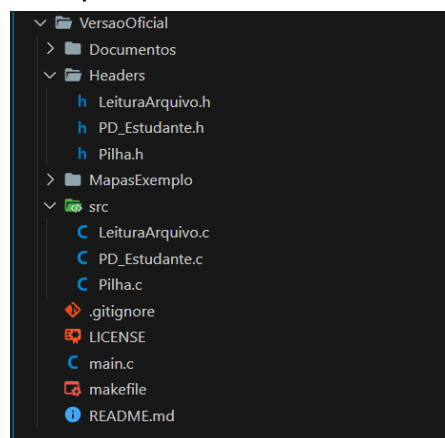
Além deste, como extras, que serão melhor explicados no capítulo específico sobre, foi desenvolvido uma outra versão, utilizando Pilhas e Filas, como uma heurística para, junto à PD, alcançar os melhores resultados. Outro algoritmo criado foi o de geração aleatória de mapas, de modo a permitir testes variados nos programas.

2) Desenvolvimento do Algoritmo

Neste capítulo serão apresentados alguns dos principais pontos para o desenvolvimento do projeto para o encontro da saída da caverna com programação dinâmica.

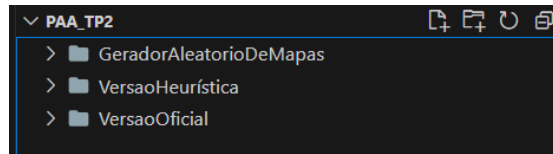
2.1) Organização dos Arquivos

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 “**MapasExemplo**”, 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**.



Figuras 1: Organização da Versão Oficial

Além da versão oficial, a pasta contendo o trabalho ainda contém um programa gerador aleatório de mapas e uma versão com heurística aplicada que serão aprofundados nos próximos tópicos.



Figuras 2: Organização dos Programas Desenvolvidos

2.2) Funcionamento do Algoritmo ‘Oficial’

2.2.1) Leitura dos arquivos e Arquivo “LeituraArquivo”

Para a interpretação dos arquivos, foi criado um arquivo denominado '**LeituraArquivo.h**', onde está definido um TAD chamado '**MatrizMapa**' que armazenará o conteúdo lido do arquivo. Isso inclui as dimensões do mapa, as coordenadas iniciais do estudante, as coordenadas da saída da caverna, a vida do estudante e uma matriz de inteiros que armazenará o conteúdo presente em cada posição da matriz lida. Além disso, no arquivo .h estão definidas as funções '**LeituraMatriz**', que é responsável por ler o arquivo de entrada e armazenar o conteúdo no novo TAD **MatrizMapa**, assim como uma função para imprimir o conteúdo armazenado por esse mesmo TAD. Tais comentários são observáveis na **Figura 3**, a qual apresenta a definição desse tipo.

```
1 typedef struct MatrizMapa{
2     int ColunasMapa,LinhasMapa;
3     int ColunaInicial, LinhaInicial, ColunaFinal, LinhaFinal;
4     int VidaJogador;
5     int **ConteudoMapa;
6 }MatrizMapa;
```


Figura 3: Tipo MatrizMapa

2.2.2) Arquivo “Pilha”

Neste arquivo encontra-se a implementação de uma pilha, a qual terá o papel de armazenar as coordenadas por onde o estudante passará com mais vida até chegar na saída do tabuleiro. A pilha armazenará dois inteiros, representando as coordenadas de linha e coluna da matriz. Além de apresentar funções de interação com um TAD pilha padrão chamado '**PilhaCoordenadas**', como uma função para inicialização, uma para inserção (que incluirá uma função chamada '**ExisteCoordenadas**' para garantir que nenhuma

coordenada repetida seja inserida), verificar se está vazia, remoção e apresentar todo o conteúdo armazenado.

Para ilustrar sua implementação, na **Figura 4** será possível visualizar sua definição e o cabeçalho de suas funções no arquivo 'Pilha.h'.



```
1  #include "../Headers/LeituraArquivo.h"
2
3  typedef struct Coordenadas {
4
5      int line, collun;
6      struct Coordenadas* next;
7
8  } Coordenadas;
9
10 typedef struct PilhaCoordenadas {
11     Coordenadas* topo;
12 } PilhaCoordenadas;
13
14 void initialize(PilhaCoordenadas* ptr);
15 int isEmpty(PilhaCoordenadas* ptr);
16 int existeCoordenadas(PilhaCoordenadas* ptr, int ptrLine, int ptrCollun);
17 void push(PilhaCoordenadas* ptr, int ptrLine, int ptrCollun);
18 void pop(PilhaCoordenadas* ptr);
19 void ApresentarCoordenadas(PilhaCoordenadas* ptr);
20 void LiberarPilha(PilhaCoordenadas* ptr);
```

Figura 4: Arquivo Pilha.h

2.2.3) Arquivo “PD_Estudante” e o funcionamento da Programação Dinâmica

Nos arquivos **PD_Estudante.h** e **PD_Estudante.c** é onde ocorre a resolução da problemática em questão. Primeiramente, é definido um TAD Estudante que armazena um campo chamado '**TabelaPD**', que servirá como uma tabela e será preenchida durante a execução do programa. Além disso, há um campo para armazenar os pontos de vida atuais do estudante chamado '**PontosVidaAtual**'. Todos esses campos serão inicializados com a função '**InicializarEstudante**' com o auxílio do TAD '**MatrizMapa**'. Essa função armazenará no campo '**PontosVidaAtual**' o valor inicial de pontos de vida dado pelo arquivo de entrada lido. O campo '**TabelaPD**' será inicializado com as mesmas dimensões da matriz lida e terá todas as posições inicializadas com zero, exceto na posição inicial do jogador, que será inicializada com os pontos de vida lidos no arquivo de entrada.

Para ilustrar sua implementação, na **Figura 5** será possível uma melhor visualização da função.

```
1 void InicializarEstudante(MatrizMapa *ptrMapa, Estudante *ptrEs)
2 {
3     //Inicia com os pontos de vida lidos no arquivo
4     ptrEs->PontosVidaAtual = ptrMapa->VidaJogador;
5
6     //Inicializa a tabela que armazenará os pontos de vida do jogador durante o percurso
7     ptrEs->TabelaPD = (int**)malloc(ptrMapa->LinhasMapa * sizeof(int*));
8     for(int i = 0; i < ptrMapa->LinhasMapa; i++){
9         ptrEs->TabelaPD[i] = (int*)malloc(ptrMapa->ColunasMapa * sizeof(int));
10    }
11
12    /*
13     * A tabela será inicializada com zero para todas as posições, exceto a inicial, inicializada
14     * com a vida inicial do estudante
15     */
16    for(int i = 0; i < ptrMapa->LinhasMapa; i++){
17        for(int j = 0; j < ptrMapa->ColunasMapa; j++){
18            if(i == ptrMapa->LinhaInicial && j == ptrMapa->ColunaInicial){
19                ptrEs->TabelaPD[i][j] = ptrMapa->VidaJogador;
20            }else{
21                ptrEs->TabelaPD[i][j] = 0;
22            }
23        }
24    }
25 }
```

Figura 5: Função InicializarEstudante

Logo em seguida, a função '**InicializarDeslocamento**' é acionada e será responsável por chamar funções de inicialização da pilha de coordenadas, através da função '**initialize**', e do TAD Estudante com a função '**InicializarEstudante**'. Esta função ainda possui um desvio condicional: caso a função '**Deslocar**' retorne 1, indicando que o estudante conseguiu encontrar a saída, o conteúdo da pilha será escrito em um arquivo de saída, registrando as coordenadas que o estudante percorreu para sair com o máximo de vida possível. Se a função '**Deslocar**' retornar 0, uma mensagem também será escrita no arquivo, informando que o estudante, infelizmente, não conseguiu sair com vida. Para ilustrar sua implementação, na **Figura 6** será possível uma melhor visualização da função.

```
1 void InicializarDeslocamento(MatrizMapa *map, Estudante *ptrEst, PilhaCoordenadas *ptrPilha){
2
3     initialize(ptrPilha);
4     InicializarEstudante(map, ptrEst);
5     if(Deslocar(map, ptrEst, ptrPilha) == 1){
6         Resultado(ptrPilha);
7     }else{
8         EscreverArquivoDeSaidaErro();
9     }
10 }
```

Figura 6: Função InicializarDeslocamento

Em seguida, a função '**Deslocar**' é acionada, com o objetivo de encontrar o caminho para que o estudante saia com vida, se existir. Primeiramente, são iniciadas variáveis auxiliares, sendo as mais importantes '**linhaAtual**' e '**colunaAtual**', que são inicializadas com a posição inicial do estudante. Em seguida, inicia-se um loop "do-while" que será executado enquanto os pontos de vida atuais do estudante forem maiores que zero.

Dentro desse laço, ocorrem duas condicionais: a primeira verifica se as variáveis '**linhaAtual**' e '**colunaAtual**' apontam para as coordenadas finais, indicando que a saída foi encontrada e o estudante ainda está com vida; a segunda verifica se as coordenadas alcançadas são [0,0], indicando que não há mais caminho a ser percorrido pelo estudante. Ambas, se verdadeiras, encerram o laço.

Após isso, ocorre um "if-else", onde o campo '**TabelaPD**' é preenchido. O if é acionado caso '**linhaAtual**' ou '**colunaAtual**' sejam iguais a zero, indicando que apenas a direção esquerda ou acima está disponível, evitando problemas relacionados a exceder as dimensões da matriz durante a execução.

Tanto no if quanto no else, o preenchimento da tabela é semelhante: se for possível, a posição ['**linhaAtual**', '**colunaAtual-1**'] da tabela é atualizada com o valor dos pontos de vida atuais mais o conteúdo nessa mesma posição lido do arquivo e armazenado em '**MatrizMapa**', caso seja possível ir para cima, o processo é semelhante, mas a posição atualizada é ['**linhaAtual-1**', '**colunaAtual**'], e a posição lida do mapa é ['**linhaAtual-1**', '**colunaAtual**']. A diferença está no else, onde esses dois valores são atualizados na tabela ao mesmo tempo. Várias condicionais são então utilizadas para determinar qual caminho seguir.

Caso o valor à esquerda seja maior que o encontrado na posição de cima, a variável '**colunaAtual**' é decrementada em uma unidade; caso o valor de cima seja maior, a variável '**linhaAtual**' que será decrementada. Se ambos os valores forem iguais, o processo difere: soma-se constantemente os valores encontrados nas direções esquerda e cima, utilizando a variável do tipo '**MatrizMapa**' e somando os valores nas posições acima e à esquerda até o momento em que essas somas diferem. Quando não são mais iguais, as variáveis '**AUXL**' e '**AUXC**', que representam, respectivamente, a soma total das posições de cima e da esquerda, são comparadas. Se a primeira for maior, o número de linhas é decrementado; caso contrário, é o número de colunas, indicando qual direção apresenta um melhor caminho.

Parte deste comentário pode ser observado a seguir, na **Figura 7**.

```

LinhaAUX = linhaAtual;
ColunaAux = colunaAtual;

while(LinhaAUX>0 || ColunaAux>0 && AUXC==AUXL){
  if(AUXL==0 && AUXC==0){
    AUXL = map->ConteudoMapa[LinhaAUX][ColunaAux];
    AUXC = map->ConteudoMapa[LinhaAUX][ColunaAux];
  }
  if(LinhaAUX>0&&ColunaAux==0){

    AUXL += map->ConteudoMapa[LinhaAUX -1][UltimaColuna];
    AUXC += map->ConteudoMapa[LinhaAUX-1][ColunaAux];

    LinhaAUX -=1;
  }else if(ColunaAux>0 && LinhaAUX==0){
    AUXL += map->ConteudoMapa[LinhaAUX][ColunaAux-1];
    AUXC += map->ConteudoMapa[UltimaLinha][ColunaAux-1];
    ColunaAux -=1;
  }else{
    AUXL += map->ConteudoMapa[LinhaAUX -1][ColunaAux];
    UltimaColuna = ColunaAux;
    AUXC+=map->ConteudoMapa[LinhaAUX][ColunaAux-1];
    UltimaLinha = LinhaAUX;

    ColunaAux -=1;
    LinhaAUX -=1;
  }
}
if(AUXL>AUXC){

  linhaAtual-=1;
}else{

  colunaAtual-=1;
}

```

Figura 7: Parte da função Deslocar

Vale destacar que os pontos de vida atuais são atualizados toda vez no final do corpo do “do-while” com o valor presente nas posições da ‘**TabelaPD**’ nas coordenadas [**linhaAtual**, **colunaAtual-1**] atualizadas.

Até este ponto, o algoritmo consegue buscar o caminho no qual o estudante tenha mais pontos de vida, mas isso não garante necessariamente que ele alcance a saída e, conseqüentemente, o melhor resultado. Quando o laço “do-while” é concluído, ainda pode haver posições na tabela com valor zero, indicando que não foram alcançadas.

Portanto, dentro da função '**Deslocar**', um loop “for” percorre toda a '**TabelaPD**', iniciando nas linhas e colunas iniciais e decrementando até zero. Esse loop busca as posições ainda zeradas. Durante esse processo, nenhuma posição é recalculada. As posições ainda zeradas na coluna inicial são sempre incrementadas pelo único valor possível que podem ter, que é o valor presente abaixo delas, ou seja, na posição [**i+1**, **j**] da tabela. Para as outras posições, o procedimento envolve comparar a soma do valor presente na posição [**i+1**, **j**] da tabela com o conteúdo do mapa na posição [**i**, **j**] e o mesmo conteúdo acrescido do valor na posição [**i**, **j+1**] da tabela. Isso decide qual é o maior valor que a posição ainda zerada pode receber. É importante destacar que para

essas comparações serem executadas simultaneamente, os valores de 'i' e 'j' devem estar dentro do limite da matriz. Caso contrário, operações semelhantes são executadas, levando em consideração o limite. Logo após isso, as posições à esquerda e acima da posição da saída final, se ambas existirem, são comparadas, decidindo qual é o maior valor de pontos de vida que o estudante terá.

Desse modo, ao final da função 'Deslocar', tem-se a 'TabelaPD' com todas as suas posições que se encontram à esquerda e acima da posição inicial preenchidas, como é possível notar na **Figura 8**, na qual foi gerado um exemplo, para fins de melhor visualização, utilizando de um exemplo criado pelo programa de geração aleatória de mapas, o qual também foi enviado e será explicado a seguir no capítulo 3.3.

```
matheuspeixoto@matheuspeixoto-550XDA:~/Documentos/GitHub/PAA_TP2/VersaoOficial$ ./main caverna6.txt
[47] [44] [44] [32] [6] [0] [0] [0]
[34] [23] [33] [24] [11] [0] [0] [0]
[34] [37] [25] [25] [25] [0] [0] [0]
[0] [0] [0] [0] [0] [0] [0] [0]
```

Figura 9: Visualização da TabelaPD após seu preenchimento. Visualização possível através da função 'ApresentarTabelaPD'

Finalmente, a função realiza uma última e simples comparação. Caso os pontos de vida atuais do jogador sejam maiores que zero, a função 'FazerCaminho', a qual pode ser vista na **Figura 9**, é chamada. Nessa etapa, uma pilha é preenchida com as coordenadas do melhor percurso que o estudante pode fazer, começando da posição final e empilhando as coordenadas das posições com maior valor à direita ou para baixo. Se os valores forem iguais, um cálculo diferente é feito, resumidamente, é armazenando a posição mais próxima à posição inicial. Ao final desse condicional, a função retorna 1, indicando que conseguiu encontrar o melhor caminho para o estudante sair do tabuleiro. Se os pontos de vida forem menores do que zero, a função retorna 0, indicando que não existe caminho possível para o estudante sair com vida.

```
1 void FazerCaminho(Estudiante *est, MatrizMapa *map, PilhaCoordenadas *PILHA){
2     int ControlI = map->LinhaFinal, ControlJ = map->ColunaFinal, FLAGI, FLAGC;
3     push(PILHA, ControlI, ControlJ);
4     while(!((ControlI==map->LinhaInicial && ControlJ==map->ColunaInicial))){
5         if((ControlI+1<=map->ColunasMapa-1 && ControlI+1<=map->LinhasMapa-1 && est->TabelaPD[ControlI+1][ControlJ]>est->TabelaPD[ControlI][ControlJ+1])){
6             ControlI++;
7             push(PILHA, ControlI, ControlJ);
8         }else if((ControlJ+1<=map->ColunasMapa-1 && ControlJ+1<=map->LinhasMapa-1 && est->TabelaPD[ControlI+1][ControlJ]<est->TabelaPD[ControlI][ControlJ+1])){
9             ControlJ++;
10            push(PILHA, ControlI, ControlJ);
11        }else if((ControlJ+1<=map->ColunasMapa-1 && ControlI+1<=map->LinhasMapa-1 && est->TabelaPD[ControlI+1][ControlJ]==est->TabelaPD[ControlI][ControlJ+1])){
12            FLAGI = ControlI - map->LinhaInicial;
13            FLAGC = ControlJ - map->ColunaInicial;
14            if(FLAGC>FLAGI){
15                ControlI++;
16            }else{
17                ControlJ++;
18            }
19            push(PILHA, ControlI, ControlJ);
20        }else if((ControlI+1<=map->ColunasMapa-1)){
21            ControlI++;
22            push(PILHA, ControlI, ControlJ);
23        }else if((ControlJ+1<=map->LinhasMapa-1)){
24            ControlJ++;
25            push(PILHA, ControlI, ControlJ);
26        }
27    }
28    push(PILHA, map->LinhaInicial, map->ColunaInicial);
29 }
30
31 }
```

Figura 9: Função FazerCaminho

2.3) Recebimento do arquivo a ser lido

De modo a possibilitar a variação do arquivo a ser lido, uma vez que o código não possui uma interface, foi desenvolvido, dentro do arquivo **main.c**, um meio pelo qual o usuário passará o nome do arquivo através do terminal, no momento de executar o programa.

Através de “**int argc, char *argv[]**” foi possível realizar tal ação, de modo que o nome inserido é concatenado com a string “MapasExemplo”, fazendo que o nome do arquivo inserido seja buscado dentro da pasta assim denominada, o que visa uma melhor organização dos arquivos presentes neste programa.

Mais adiante, no **Capítulo 4**, será apresentada, de maneira mais detalhada, como é passada essa informação do arquivo a ser utilizado e como executá-lo.

3)Extras

3.1) Versão “Heurística”

O trabalho apresenta duas versões distintas para a resolução da problemática: a abordagem 'Oficial' e a abordagem 'Heurística'. Na abordagem 'Oficial' foi adotada uma estratégia gulosa, onde a escolha do próximo movimento é feita de maneira a maximizar imediatamente o valor na tabela. Inicialmente, o algoritmo realiza avanços para a esquerda ou para cima, optando pelo caminho que resulta no maior valor acumulado. Posteriormente, um loop é utilizado para preencher os componentes restantes da tabela, garantindo que todas as possibilidades sejam consideradas, e, assim, garantindo uma solução ótima. Já na versão 'Heurística', a abordagem é exclusivamente gulosa. O avanço de posições é realizado de maneira mais seletiva, sem a necessidade de preencher completamente a tabela.

Essa estratégia visa uma eficiência computacional superior, uma vez que não são consideradas todas as possíveis combinações. Além disso, a utilização de uma fila se destaca, uma vez que armazena as posições que foram pré-processadas, otimizando ainda mais o desempenho do algoritmo. Ambas as abordagens apresentam vantagens e desvantagens, sendo a escolha entre elas dependente das especificidades do problema e dos requisitos de eficiência computacional. Enquanto a abordagem 'Oficial' garante uma solução global ótima, a abordagem 'Heurística' busca uma solução aproximada de forma mais eficiente, com a aplicação de técnicas gulosas e o uso estratégico de filas para armazenamento prévio de dados processados.

De modo a ilustrar tal implementação, na **Figura 10** é possível visualizar a implementação da função ‘**Deslocar**’ dessa versão.

```

1  int Deslocar(MatrizMapa *ptr, Aventureiro *ptrAv, Fila *Filas){
2      //Calcular a primeira Linha
3      int controlJ = 1, controlI = 1, j, NovoJ = ptr->ColunaInicial, NovoI = ptr->LinhaInicial ;
4      for(i = NovoI; controlI == 1; i++){
5          ptrAv->TabelaPD[i][NovoJ] = ptrAv->PontosVidaAtual;
6          for(j = NovoJ-1; controlJ == 1; j--){
7
8              controlJ = EsquerdaLivre(i, j, ptrAv, ptr);
9
10         }
11         j = EscolherMelhorCaminho(i, ptrAv, ptr);
12         controlI = CimaLivre(i, j, ptrAv, ptr);
13         NovoJ = j;
14         if(i-1 > 0){
15             NovoI = i-1;
16             ptrAv->PontosVidaAtual += ptr->ConteudoMapa[NovoI][NovoJ];
17         }
18         if(ptrAv->PontosVidaAtual <= 0){
19             break;
20         }
21         controlJ = 1;
22     }
23
24     if(ptrAv->TabelaPD[ptr->LinhaFinal][ptr->ColunaFinal] > 0){
25         PreencherFila(ptr, ptrAv, Filas);
26         return 1;
27     }
28     return 0;
29 }

```

Figura 10: Função Deslocar da “Versão Heurística”

3.2) “Como seria possível permitir todas as movimentações com PD?”

No caso de se permitir todos os tipos de movimentos, diferentemente do que foi observado na **Figura 9**, todas as posições, independentemente de onde o personagem começa sua jornada, seriam preenchidas.

Isso geraria um trabalho maior do que o que foi empenhado, tanto do grupo, quanto do algoritmo, o qual necessariamente apresentaria um custo maior, sendo obrigado a fazer uma quantidade significativamente maior de comparações, aumentando, conseqüentemente, o seu tempo de execução.

3.3) Geração Aleatória de Mapas

Como um dos extras desenvolvidos pelo grupo, na pasta GeradorAleatorioDeMapas se encontra um programa capaz de gerar arquivos que podem ser utilizados para testes do programa principal.

Em seu arquivo de cabeçalho foi utilizado um “*typedef struct*” igual ao utilizado no arquivo LeituraArquivo.h, anteriormente explicado.

Sua única função, a ‘**GerarCavernaAleatoria**’ possui algumas variáveis que têm por finalidade a definição de limites tanto para os monstros, quanto para as poções a serem distribuídas pelo mapa, de modo que a geração de “casas” com valor zero sejam priorizadas.

Desse modo, os monstros não podem ultrapassar 30% do valor da área total da caverna, enquanto as poções, 20%.

Na primeira linha do arquivo gerado, assim como na especificação, são inseridos os valores das dimensões da matriz, bem como a pontuação de vida do personagem, os quais são todos gerados aleatoriamente.

Para a distribuição dos valores na matriz é realizado um “switch-case” com os valores de 0 a 2 gerados aleatoriamente e armazenados em uma variável. Desse modo, caso o valor gerado seja 0, a posição da matriz recebe também um zero, caso 1, é verificado se a quantidade total de poções já inseridas não ultrapassa o limite. Caso não, um valor qualquer de 1 a 15 é inserido. Mas caso tenha ultrapassado, a posição recebe um zero.

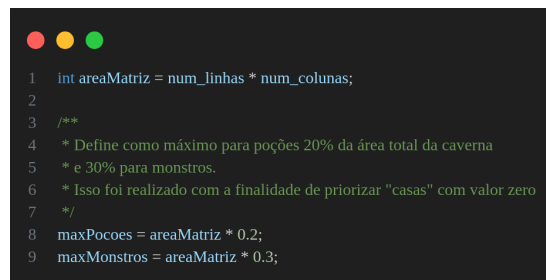
De modo análogo ocorre caso o valor gerado seja um 2, fazendo a mesma verificação para os monstros e atribuindo um valor de -1 a -15 caso o máximo não tenha sido atingido.

Após isto, são definidas as posições inicial e final, de modo que é realizada uma verificação nas dimensões da matriz, também aleatoriamente geradas, já no início do programa, de modo que a posição final ficará sempre na parte superior esquerda da matriz enquanto a inicial sempre na parte inferior direita.

Isto foi realizado devido a restrição de movimentos do personagem, que só pode se movimentar para cima e para a esquerda.

Tudo isto é escrito em um arquivo que se encontrará na pasta ‘MapasExemplo’ e será nomeado como ‘**Caverna Aleatória.txt**’.

A fins de ilustrar e facilitar a compreensão do que foi dito neste subcapítulo, estarão as **Figuras 11, 12 e 13**.



```
1 int areaMatriz = num_linhas * num_colunas;
2
3 /**
4  * Define como máximo para poções 20% da área total da caverna
5  * e 30% para monstros.
6  * Isso foi realizado com a finalidade de priorizar "casas" com valor zero
7  */
8 maxPocoas = areaMatriz * 0.2;
9 maxMonstros = areaMatriz * 0.3;
```

Figura 11: Definição dos limites para poções e monstros

```

1  for(int i = 0; i < num_linhas; i++){
2      map->ConteudoMapa[i] = (int *)malloc(num_colunas * sizeof(int));
3      for(int j = 0; j < num_colunas; j++){
4          auxAleatorio = rand() % 3;
5          switch (auxAleatorio)
6          {
7              case 0:
8                  map->ConteudoMapa[i][j] = 0;
9                  break;
10             case 1:
11                 if(contaPocoas <= maxPocoas){
12                     map->ConteudoMapa[i][j] = rand() % 15 + 1;
13                     contaPocoas++;
14                 }
15                 else{
16                     map->ConteudoMapa[i][j] = 0;
17                 }
18                 break;
19             case 2:
20                 if(contaMonstros <= maxMonstros){
21                     map->ConteudoMapa[i][j] = (rand() % 15 + 1) * (-1);
22                     contaMonstros++;
23                 }
24                 else{
25                     map->ConteudoMapa[i][j] = 0;
26                 }
27                 break;
28             default:
29                 break;
30             }
31         }
32     }
33 }

```

Figura 12: Switch-case para definição das posições da matriz

```

1  // Define a posição de F (Fim) na metade esquerda superior da matriz
2  auxLinhaFinal = rand() % (num_linhas / 2);
3  auxColunaFinal = rand() % (num_colunas / 2);
4
5  map->ConteudoMapa[auxLinhaFinal][auxColunaFinal] = 'F';
6  map->ConteudoMapa[auxLinhaInicial][auxColunaInicial] = 'I';
7
8  // Escreve o mapa no arquivo
9  for (int i = 0; i < num_linhas; i++) {
10     for (int j = 0; j < num_colunas; j++) {
11         if (i == auxLinhaInicial && j == auxColunaInicial)
12         {
13             fprintf(arq, "I ");
14             continue;
15         }
16         if (i == auxLinhaFinal && j == auxColunaFinal)
17         {
18             fprintf(arq, "F ");
19             continue;
20         }
21         fprintf(arq, "%d ", map->ConteudoMapa[i][j]);
22     }
23     fprintf(arq, "\n");
24 }

```

Figura 13: Definição das posições Inicial e Final e escrita no arquivo

4)Compilação e Execução

Neste capítulo serão dados os direcionamentos para a compilação e para a execução dos arquivos que compõem esse Trabalho Prático 02.

- Antes de qualquer outro passo a seguir, é importante que se verifique se os arquivos que serão testados, em todos os casos que serão apresentados abaixo, se encontram na pasta “MapasExemplo”, visto que o algoritmo, visando uma melhor organização dos arquivos utilizados, procura diretamente dentro desta pasta.
- ❖ Para a compilação e execução da “Versão Oficial”, é preciso que se encontre na pasta **VersaoOficial** e, no terminal, digite o seguinte:

make

Caso esteja em um computador com Sistema Operacional Windows:

main.exe nomeArquivo.txt

Caso em um computador com S.O. Linux:

./main nomeArquivo.txt

- ❖ Para a compilação e execução da “Versão Heurística”, é preciso que se encontre na pasta **VersaoHeurística** e, no terminal, digite o seguinte:

make

Caso esteja em um computador com Sistema Operacional Windows:

heuristica.exe nomeArquivo.txt

Caso em um computador com S.O. Linux:

./heuristica nomeArquivo.txt

- ❖ Para a compilação e execução do programa de Geração de Mapas Aleatórios, é preciso que se encontre na pasta GeradorAleatorioDeMapas e, no terminal, digite o seguinte:

make

Caso esteja em um computador com Sistema Operacional Windows:

random.exe nomeArquivo.txt

Caso em um computador com S.O. Linux:

./random nomeArquivo.txt

5)Resultados

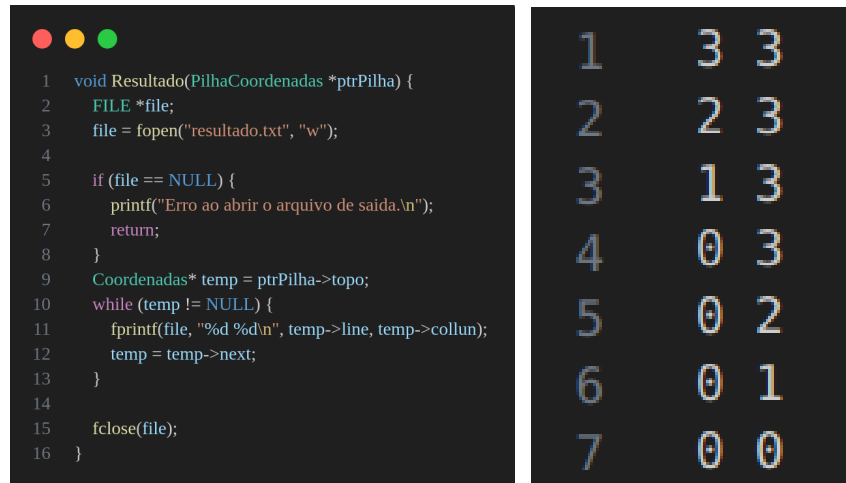
Com a utilização das funções ‘**Resultado**’ e ‘**EscreverArquivoDeSaidaErro**’ obtém-se os resultados desejados após a execução do programa.

A função ‘**Resultado**’, como é possível observar na **Figura 14**, é a responsável por imprimir no arquivo de mesmo nome a saída esperada após a execução do programa, para os casos em que um caminho válido é encontrado.

Assim como solicitado na especificação do Trabalho, são impressas as linhas e colunas, indicando as coordenadas encontradas como o melhor caminho. A

seguir, na **Figura 15**, será possível observar a saída para o exemplo criado pelo grupo nomeado de **'caverna2.txt'**.

Já a outra função citada neste capítulo é a responsável por informar ao usuário, também em um arquivo nomeado como **'resultado'**, que não foi possível sair da caverna com vida, imprimindo a mensagem "impossível", assim como solicitado.



Figuras 14 e 15: Função Resultado e saída obtida para caverna2.txt

6) Conclusão

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

Com os resultados obtidos ao final da execução do Trabalho Prático 02 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/>
- Medium.com: <https://medium.com/@arjunatlast/solving-the-maze-problem-using-d-c-c4e1a9a09279>

Vídeos que de alguma forma contribuíram para o desenvolvimento:

-  Min cost In Maze Traversal | Dynamic Programming and Greedy...