UNIVERSIDADE FEDERAL DE VIÇOSA - *CAMPUS* FLORESTAL

Fábio Trindade Ramos - mat.3869
Gabriel Vitor da Fonseca Miranda - mat.3857
Victória Caroline Silva Rodrigues - mat.3584

Trabalho Prático 2
Projeto e Análise de Algoritmos

Trabalho prático da disciplina Projeto e Análise de Algoritmos - CCF 330, do curso de Ciência da Computação da Universidade Federal de Viçosa - Campus Florestal.

Professor: Daniel Mendes

Florestal
2022

SUMÁRIO

1. INTRODUÇÃO

Este trabalho tem como objetivo encontrar um caminho mínimo em que o personagem Samus consiga percorrer o mapa até sua nave, sem que seja alcançado pela lava vulcânica do planeta Zebes. O mapa da caverna é ilustrado na figura 1, em que as áreas em amarelo são os caminhos livres, em marrom as partes bloqueadas por rochas e as partes com números indicam a presença de um inimigo. Durante seu percurso, Samus terá que ser mais rápido que o tempo de subida da lava e ainda poderá enfrentar alguns adversários que irão contabilizar no tempo de percurso.

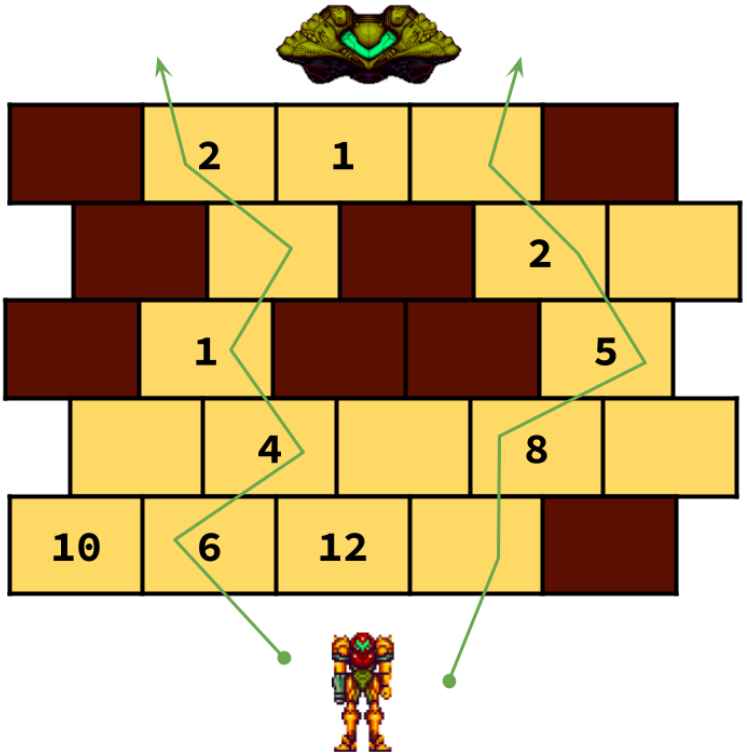

Figura 1. Mapa representando os caminhos possíveis do planeta Zebes.

Desse modo, o presente trabalho tem como desafio desenvolver um algoritmo utilizando programação dinâmica, que é uma abordagem de otimização que transforma um problema complexo em uma sequência de problemas mais simples.

2. Desenvolvimento

Para a implementação do trabalho proposto, foi utilizada a linguagem de programação C, juntamente com a ferramenta Git para gerenciar o controle das versões do código. Além disso, com intuito de manter a organização do código e facilitar o processo de análise dos

algoritmos, foram criados 3 TADs: a *Caverna,* contendo as dimensões do mapa, tempo de Samus e da lava, além do tempo total gasto pelo Samus, o *geradorArquivo,* para gerar mapas aleatórios e o *modoAnalise,* no qual serão feitas as análises de performance do algoritmo dependendo dos parâmetros previamente definidos.

3. Implementação

A seguir será apresentado e explicado detalhadamente o funcionamento de cada uma das principais funcionalidades implementadas no trabalho prático. Para melhor visualização, estes foram organizados em subtópicos.

3.1 Estruturas de dados

Neste trabalho prático foram criadas 3 estruturas de dados, para um melhor encapsulamento e resolução do problema, o primeiro tad criado foi o *caverna*, nele declaramos algumas variáveis que serão utilizadas para armazenar os dados iniciais da caverna, dados esses como coluna, linha, unidades de tempo gastas pelo Samus, o tempo da lava, o tempo total gasto, e a própria caverna. Veja abaixo a estrutura caverna criada:

```
typedef struct {
    int **caverna;
    int linha;
    int coluna;
    int unidadeTempoGasta;
    int tempoLava;
    int tempoTotal;

}Caverna;
```

Figura 2. TAD Caverna.

A segunda estrutura de dados foi o tad *geradorArquivo,* cujo o principal objetivo é criar de forma aleatória uma caverna e preenchê-la em um arquivo.txt, para este tad não foi preciso criar uma struct apenas as funções já resolviam o problema. Veja abaixo as funções usadas:

```
void gerarCaracteres(char* str);
void gerarArquivo(char* nomeArquivo,int tamLinha, int tamColuna,
int tempoSamus,int tempoLava);
```

Figura 3. TAD geradorArquivo.

A terceira estrutura de dados criada foi a *modoAnalise,* que tem como função gerar arquivos aleatórios e calcular o tempo para n cavernas de n linhas e m colunas. Veja abaixo as funções utilizadas:

```
void gerarArquivos();
void calcularTempos();
```

Figura 4. TAD modoAnalise.

3.2 Programação dinâmica

Para implementar a programação dinâmica no mapa e encontrar o caminho mais curto utilizou-se como raciocínio para resolução do problema a mesma estratégia vista em aula sobre as torres de hanoi. Desse modo, para calcular o caminho utilizou-se a função *caminhoMinimo()*, essa função é responsável por encontrar qual é a célula com menor valor no mapa e somar com a próxima célula abaixo, ou seja, esse processo ocorre de baixo para cima.

A primeira linha da matriz é inserida na matriz sem nenhuma modificação, visto que é o ponto de partida do caminho. Para as próximas linhas da matriz será utilizada a linha anterior, calculando o valor mínimo de entre as duas células possíveis de subir a caverna. Abaixo segue as etapas:

1. Selecionar as duas células da linha anterior e verificar se são valores válidos para subir na caverna.
2. Selecionar o valor mínimo entre esses dois valores.
3. Somar o valor original ao valor mínimo encontrado na etapa 2.

   As etapas desse processo são ilustradas na figura 5 abaixo.

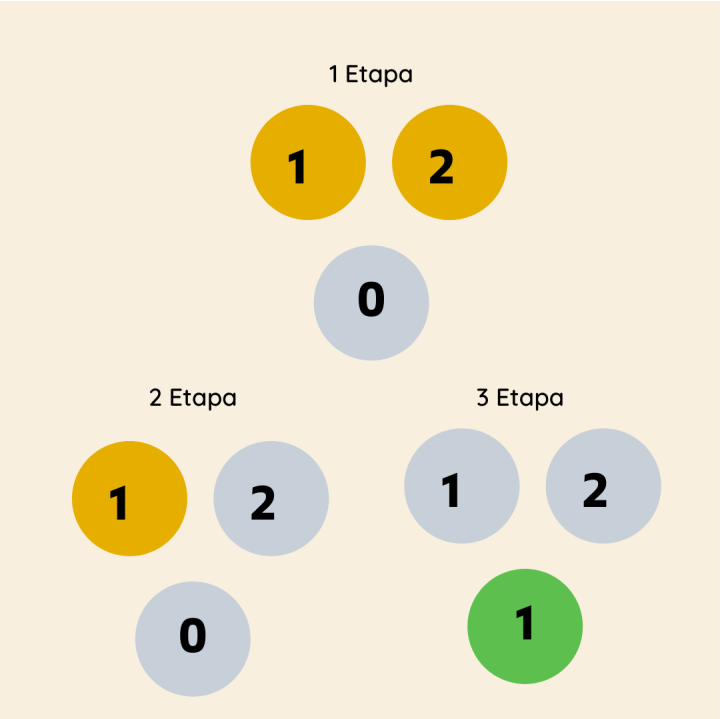

Figura 5. Representação dos passos da escolha do caminho mínimo.

Outra decisão implementada pela função do *caminhoMinimo(),* é quando tivermos duas células bloqueadas por rochas, representada na matriz com o valor -1, deve-se retornar o valor -1 para a célula de origem, ou seja, não será possível utilizar esse caminho, ilustrada na figura 6.

Figura 6. Representação dos passos para caminhos inválidos.

Além disso, para que fosse possível percorrer o mapa de maneira correta devido ao seu deslocamento, notou-se que existia um padrão para as linhas ímpares e pares. No qual, para as linhas pares era necessário checar um caminho possível nas colunas j e j+1, e para as linhas ímpares, as posições j e j-1, sendo j a coluna, como mostrada no algoritmo na figura 7 abaixo.

```
if(posicaoEhValida(caverna, i, j)){
    if(i%2!=0){
        if((posicaoEhValida(caverna,i-1,j)) && (posicaoEhValida(caverna,i-1,j+1))){
            return min(caverna->caverna[i-1][j],caverna->caverna[i-1][j+1]);
        }
        else if((!posicaoEhValida(caverna,i-1,j)) && (posicaoEhValida(caverna,i-1,j+1))){
            return caverna->caverna[i-1][j+1];
        }else if((posicaoEhValida(caverna,i-1,j)) && (!posicaoEhValida(caverna,i-1,j+1))){
            return caverna->caverna[i-1][j];
        }else{

            return -1;
        }
    }else{
        if((posicaoEhValida(caverna,i-1,j-1)) && (posicaoEhValida(caverna,i-1,j))){
            return min(caverna->caverna[i-1][j-1],caverna->caverna[i-1][j]);
        }
        else if(!(posicaoEhValida(caverna,i-1,j-1)) && (posicaoEhValida(caverna,i-1,j))){
            return caverna->caverna[i-1][j];
        }else if((posicaoEhValida(caverna,i-1,j-1)) && !(posicaoEhValida(caverna,i-1,j))){
            return caverna->caverna[i-1][j-1];
        }else{
            return -1;
        }
```

Figura 7. Algoritmo da função caminhoMinimo().

Dessa maneira, utilizando-se a função *caminhoMinimo*, podemos calcular o caminho mínimo em cada posição da matriz. Desse modo, utilizou-se a função *calcula()*, representada na figura 8, para fazer a chamada de cada linha e coluna da matriz original para que se pudesse realizar as mudanças explicadas anteriormente utilizando a função *caminhoMinimo*.

```
for( i=0; i<caverna->linha;i++){
    for( j=0; j<caverna->coluna;j++){
        if(caminhoMinimo(caverna, i, j)==-1){
            caverna->caverna[i][j] = -1;
        }else{
            caverna->caverna[i][j] = caverna->caverna[i][j] + caminhoMinimo(caverna, i, j);
        }
    }
}
```

Figura 8. Realizando as modificações na matriz original.

3.2.1 Caminho mínimo

Para encontrar o caminho mínimo do mapa, primeiramente é necessário verificar se existe uma posição válida em que o tempo total gasto por Samus para subir a caverna seja igual ou inferior ao tempo que a lava gasta para preenchê-la, desse modo, utilizou-se a função *inicio()*, que irá retornar a posição da coluna da última linha que o personagem deve escolher. Caso, não exista uma posição válida, não é possível que Samus saia a tempo da caverna por nenhum caminho do mapa.

```
int inicio(Caverna *caverna,int i,int j){
    int menor = 1000;
    int posC = -1;
    int tempoTotal =caverna->tempoLava*caverna->linha;

    for( i = 0; i <caverna->coluna;i++){
        if((caverna->caverna[caverna->linha-1][i] < menor) && (caverna->caverna[caverna->linha-1][i]!=-1)
        && (tempoTotal >=caverna->caverna[caverna->linha-1][i])){
            menor = caverna->caverna[caverna->linha-1][i];
            posC = i;
        }
    }
    return posC;
}
```

Figura 9. Função responsável por encontrar a posição do válida para iniciar o percurso de Samus até a nave.

Caso exista um caminho possível iremos utilizar, as variáveis posI e posJ, para a posição da linha e posição da coluna, respectivamente. Dessa forma, iremos utilizar a mesma estratégia da função *minimoCaminho*, no qual, para as linhas pares será necessário checar um caminho possível para próxima linha nas colunas posJ e posJ+1, e para as linhas ímpares, as posições posJ e posJ-1, sendo posJ a coluna, mostrada na figura 10. Além disso, utilizou-se a função *posicaoEhValida()*, para verificar se é uma posição válida na matriz.

```
for(i=0;i<caverna->linha-1;i++){
    posI-=1;
    if(posI%2!=0){
        if(posicaoEhValida(caverna,posI,posJ) && posicaoEhValida(caverna,posI,posJ-1)) {
            min2=minimo(caverna->caverna[posI][posJ],caverna->caverna[posI][posJ-1]);
            if (caverna->caverna[posI][posJ]!=min2)
            {
                posJ-=1;
            }
        }else if(!posicaoEhValida(caverna,posI,posJ) && posicaoEhValida(caverna,posI,posJ-1)){
            posJ-=1;
        }
    }else{

        if(posicaoEhValida(caverna,posI,posJ) && posicaoEhValida(caverna,posI,posJ+1)) {
            min2=minimo(caverna->caverna[posI][posJ],caverna->caverna[posI][posJ+1]);
            if (caverna->caverna[posI][posJ]!=min2)
            {
                posJ+=1;
            }
        }else if(!posicaoEhValida(caverna,posI,posJ) && posicaoEhValida(caverna,posI,posJ+1)){
            posJ+=1;
        }
    }
```

Figura 10. Percorrendo o caminho mínimo.

3.3 Células livres (Tarefa extra)

Para chegar em uma célula livre é necessário que a partir de alguma das células iniciais(fundo da caverna) Samus consiga alcançá-la subindo nível por nível, seja para a direita ou para a esquerda. Dito isso, Samus não conseguirá alcançar uma célula livre se as células abaixo dela a esquerda e a direita não forem um caminho possível para Samus, ou seja, se as células abaixo se categorizam como : não existente(limite da caverna), uma rocha ou uma célula que não é alcançável.

```
void calcularCelInalcancaveis(Caverna caverna){
    int i,j;
    int copia;
    for( i=caverna.linha-2; i>=0;i--){
        for( j=0; j<caverna.coluna;j++){
            if(caverna.caverna[i][j]!=-1){
            if(i%2!=0){
            if(posicaoEhValida(&caverna,i+1,j)==false && posicaoEhValida(&caverna,i+1,j+1)==false) {
                copia= caverna.caverna[i][j];
                caverna.caverna[i][j]=-1;
                printf("Célula inalcançável: %d ; Posição-> i:%d j:%d\n",copia,i,j);
            }
            }else{
             if(posicaoEhValida(&caverna,i+1,j)==false && posicaoEhValida(&caverna,i+1,j-1)==false) {
                copia= caverna.caverna[i][j];
                caverna.caverna[i][j]=-1;
                printf("Célula inalcançável: %d ; Posição> i:%d j:%d\n",copia,i,j);
            }
        }
            }

        }
    }
    printf("\n");
}
```

Figura 11. Algoritmo de células inalcançáveis.

O algoritmo da figura 11 aplica a lógica explicada acima. Os dados são percorridos da matriz a partir da penúltima linha, já que todas as células livres finais são alcançáveis por Samus. Ao percorrer os dados linha por linha é analisado se as células da direita e da esquerda abaixo são posições inválidas, caso sejam, a célula verificada se torna uma célula com posição inválida e é apresentada como uma célula inalcançável. Vale ressaltar que a posição das células abaixo de outra célula depende da linha em que esta célula está devido a representação da caverna em uma matriz..

3.4 Gráfico de desempenho (Tarefa extra)

Foi plotado um gráfico de barras para visualizar como o algoritmo se comporta com diferentes quantidades de entrada. O gráfico foi feito com os seguintes tamanhos de entrada: 10^2, 50^2, 100^2, 500^2, 1000^2, 5.000^2 e 10.000^2. Após visualizar a figura 12 é possível perceber que:

- Para entradas menores ou iguais a 5.000^2 os tempos são menores que 1 segundo.
- Para entrada igual a 10.000^2 o tempo foi aproximadamente 4 vezes maior que o tempo levado para o algoritmo executar com uma entrada igual a 5.000^2.
- Se o algoritmo utilizasse o conceito de backtracking os tempos de execução seriam muito maiores, principalmente para as entradas muito grandes.

Vale dizer que o tempo para as 5 primeiras entradas citadas foi muito pequeno, por isso o gráfico não conseguiu apresentá-los, no entanto, os tempos obtidos são respectivamente 0.0, 0.0, 0.001, 0.008 e 0.03. Além disso, os tempos foram obtidos com até 6 casas decimais, sendo assim, podemos concluir que os valores de tempo iguais à 0 são possíveis pois possuem tempo menor que $1 * 10^{-6}$.

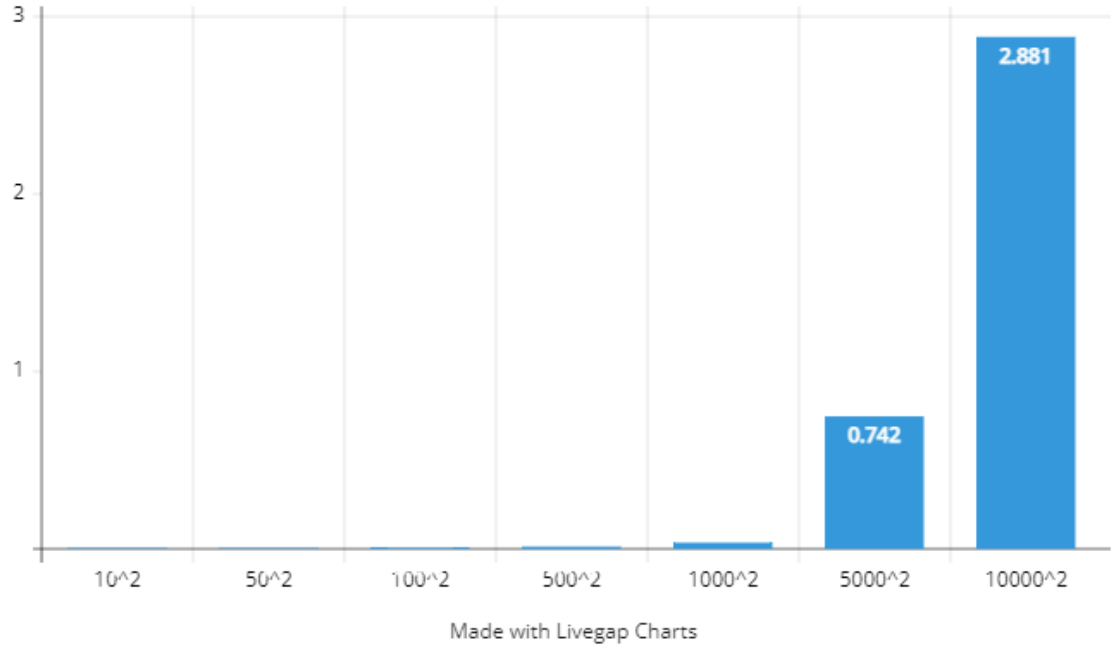

Figura 12. Gráfico Tempo x Entrada

3.5 Ilustração do melhor caminho(Tarefa extra)

Este algoritmo marca com um *X* o caminho em que Samus deve percorrer, e as demais células com o símbolo -(menos) . Para isso deve ser fornecido ao algoritmo a matriz gerada após utilizar o algoritmo de programação dinâmica. O algoritmo utiliza a mesma lógica utilizada para mostrar as posições percorridas pela Samus na função *calcula*, no entanto, a diferença é que são atribuídos valores igual a -2 nas posições em que Samus deve percorrer. Após isso, basta percorrer novamente a nova matriz gerada mostrando o caracter X onde as posições são iguais a -2 e -(menos) caso contrário. A figura 13 mostra o algoritmo desta função e a figura 14 mostra o desenho do caminho feito pela Samus na caverna fornecida na especificação do trabalho.

```c
void desenharMelhorCaminho(Caverna caverna){
    int i,j,min2,cont=0;
    int posI=caverna.linha-1;
    int posJ=inicio(&caverna,caverna.linha,0);
    caverna.caverna[posI][posJ]=-2;
    for(i=0;i<caverna.linha-1;i++){
        posI-=1;
        if(posI%2!=0){
            if(posicaoEhValida(&caverna,posI,posJ) && posicaoEhValida(&caverna,posI,posJ-1)) {
                min2=minimo(caverna.caverna[posI][posJ],caverna.caverna[posI][posJ-1]);
                if (caverna.caverna[posI][posJ]!=min2){
                    posJ-=1;
                } }else if(!posicaoEhValida(&caverna,posI,posJ) && posicaoEhValida(&caverna,posI,posJ-1)){
                posJ-=1;}
        }else{
            if(posicaoEhValida(&caverna,posI,posJ) && posicaoEhValida(&caverna,posI,posJ+1)) {
                min2=minimo(caverna.caverna[posI][posJ],caverna.caverna[posI][posJ+1]);
                if (caverna.caverna[posI][posJ]!=min2)
                {posJ+=1;}
            }else if(!posicaoEhValida(&caverna,posI,posJ) && posicaoEhValida(&caverna,posI,posJ+1)){
                posJ+=1;}}
        caverna.caverna[posI][posJ]=-2;
    }
    for(i=0;i<caverna.linha;i++){
    for(j=0;j<caverna.coluna;j++){
        if (i%2!=0 && cont==0){
            printf("  ");
            cont++;}
        if(caverna.caverna[i][j]==-2) printf("[X]");
        else printf("[-]");
        }
        cont=0;
        printf("\n");}}
```

Figura 13. Algoritmo que ilustra o melhor caminho.

```
Desenhando caminho...
[-][-][X][-][-]
  [-][X][-][-][-]
[-][X][-][-][-]
  [X][-][-][-][-]
[-][X][-][-][-]
```

Figura 14. Saída do algoritmo que ilustra o melhor caminho.

3.6 Geração de arquivos (Tarefa extra)

Para fazer a geração de arquivos foram criadas duas funções a *gerarCaracteres()* que tem como objetivo criar os caracteres que serão preenchidos no arquivo, nela foi criado um *rand() % 3*, pois se o valor 0 for retornado, é gerado *"###"* em uma determinada posição do arquivo, e se o retorno for 1 é gerado uma string do tipo "000", e se não for nenhum destes é gerado um valor de 1 a 999.

Outrossim, a segunda função é a *gerarArquivo(),* cujo o principal objetivo é passado a linha, coluna, o tempo de Samus, e o tempo da lava, ir gerando dentro do arquivo a caverna que será percorrida por Samus.

3.7 Execução

Para fazer a execução do programa no Linux, basta abrir o terminal (ctrl+c) e digitar o comando: **make**. E para executar o programa no Windows basta digitar o seguinte comando:**gcc main.c -o run sources/geradorArquivo.c sources/caverna.c sources/modoAnalise.c**, e logo em seguida digitar: **run.exe**.

4. Conclusão

A seguir serão apresentados, respectivamente, os resultados encontrados a partir da execução das funções descritas na aba Implementação e as considerações finais a respeito deste trabalho prático.

4.1 Resultados

O código em questão foi testado com o exemplo disponibilizado na especificação do trabalho prático e obteve os resultados esperados. Abaixo serão apresentados os resultados obtidos a partir da execução.

Para realização do teste é necessário realizar a leitura do arquivo, que terá na primeira a altura da caverna, a largura de cada nível da caverna, o tempo que Samus gasta para realizar cada movimento e o tempo que a lava gasta para subir cada nível da caverna. Nas linhas seguintes, será informado o mapa que ele terá que percorrer, sendo composto por 3 caracteres cada célula: caminho bloqueado (###), caminho vazio (000) e valores entre 001 e 999. Na figura 15, está representada a entrada utilizada.

```
5 5 4 7
### 002 001 000 ###
  ### 000 ### 002 000
### 001 ### ### 005
  000 004 000 008 000
010 006 012 000 ###
```

Figura 15. Representação das entradas por arquivo.

Após a leitura do arquivo, temos como saída as células livres inalcançáveis no mapa, o caminho mínimo percorrido por Samus e o desenho do caminho percorrido pelo personagem no mapa, mostrada na figura 16.

```
Calculando celulas inalcancaveis...
Celula inalcancavel: 0 ; Posicao-> i:3 j:4

Procurando melhor caminho...
Samus deve seguir as coordenadas abaixo:
4 1
3 0
2 1
1 1
0 2

Desenhando caminho...
[-][-][X][-][-]
  [-][X][-][-][-]
[-][X][-][-][-]
  [X][-][-][-][-]
[-][X][-][-][-]
```

Figura 16. Representação da saída no terminal.

Além disso, foi possível realizar o modo análise, que é responsável por verificar como é o desempenho do algoritmo variando o tamanho das entradas. Na figura 17 e 18, temos a entrada dos arquivos gerados e a análise de cada entrada, respectivamente.

```
void gerarArquivos(){
    gerarArquivo("entrada10.txt",10,10,0,1);
    gerarArquivo("entrada50.txt",50,50,0,1);
    gerarArquivo("entrada100.txt",100,100,0,1);
    gerarArquivo("entrada500.txt",500,500,0,1);
    gerarArquivo("entrada1000.txt",1000,1000,0,1);
    gerarArquivo("entrada5000.txt",5000,5000,0,1);
    gerarArquivo("entrada10000.txt",10000,10000,4,3);
}
```

Figura 17. Entradas para geração dos arquivos aleatórios para análise de desempenho.

```
Tempo de execução para entrada 10x10:0.000000
Tempo de execução para entrada 50x50:0.000000
```

```
Tempo de execução para entrada 100x100:0.001000
Tempo de execução para entrada 500x500:0.008000
Tempo de execução para entrada 1000x1000:0.030000
Tempo de execução para entrada 5000x5000:0.742000
Tempo de execução para entrada 10000x10000:2.881000
```

Figura 18. Análise de desempenho dos arquivos gerados.

Através desses dados é possível gerar um gráfico para analisar o desempenho do algoritmo descrito anteriormente na aba implementação, como visto na figura 19.

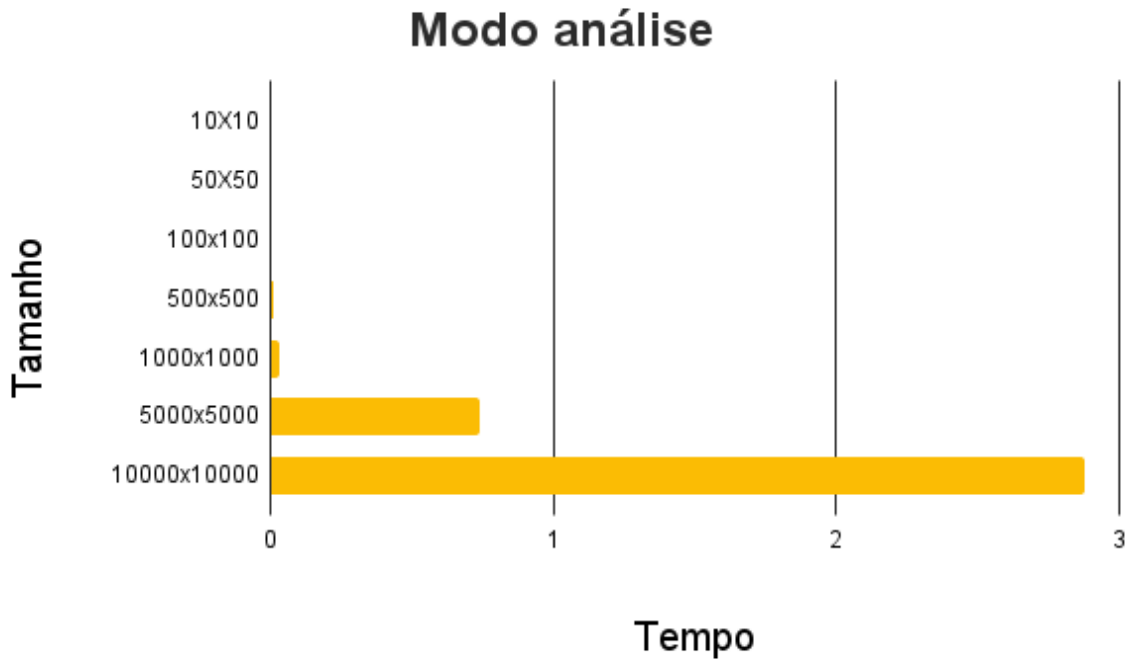

Figura 19. Gráfico de desempenho do algoritmo.

4.2 Considerações finais

Neste trabalho prático foi explorado um tema de grande importância para os estudos, principalmente na disciplina de Projeto e Análise de Algoritmos que é a programação dinâmica. Programação dinâmica é um método para a construção de algoritmos para a resolução de problemas computacionais, em especial os de otimização combinatória. Ela é aplicável a problemas nos quais a solução ótima pode ser computada a partir da solução ótima previamente calculada e memorizada - de forma a evitar recálculo - de outros subproblemas que, sobrepostos, compõem o problema original.

O que um problema de otimização deve ter para que a programação dinâmica seja aplicável são duas principais características: subestrutura ótima e superposição de subproblemas. Um problema apresenta uma subestrutura ótima quando uma solução ótima

para o problema contém em seu interior soluções ótimas para subproblemas. A superposição de subproblemas acontece quando um algoritmo recursivo reexamina o mesmo problema muitas vezes.

Outrossim, o objetivo principal deste trabalho foi criar as funções *caminhoMinimo(),* e a *calcula(),* a função *caminhoMinimo()* calcula de cima para baixo até a ultima linha de uma matriz qual o menor caminho possível para que Samus possa atravessar, já a *calcula(),* calcula um caminho possível de baixo para cima na matriz para Samus(personagem principal) poder caminhar, e se ele conseguir fazer este percurso samus consegue vencer e atravessar a caverna.

Portanto, esta abordagem mais prática da implementação do algoritmo de programação dinâmica, melhorou ainda mais os conhecimentos adquiridos em sala de aula, pois fez com que conceitos do algoritmo que foram apresentados fossem utilizados para a implementação dessa estrutura.

5. Referências

[1] Bellman, Richard. Programação dinâmica. Departamento de engenharia de produção, 2020. Disponível em: <Link>. Acesso em: 04 de março de 2022.

[2] Programação dinâmica. Wikipédia, 2021. Disponível em:<Link>. Acesso em: 06 de março de 2022.