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 1
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 consiste em desenvolver um caminho possível a partir de um labirinto, representado na figura 1, na qual, o personagem principal, denominado Ness, deve percorrer o mapa até eliminar seu inimigo, Giygas. O caminho é representado por meio dos caracteres “-”,“|”e “+”, sendo um caminho horizontal, vertical e um cruzamento, respectivamente. Os inimigos de Ness são simbolizados pelas letras U,T,S,B e G, sendo G seu alvo principal para finalizar o jogo.

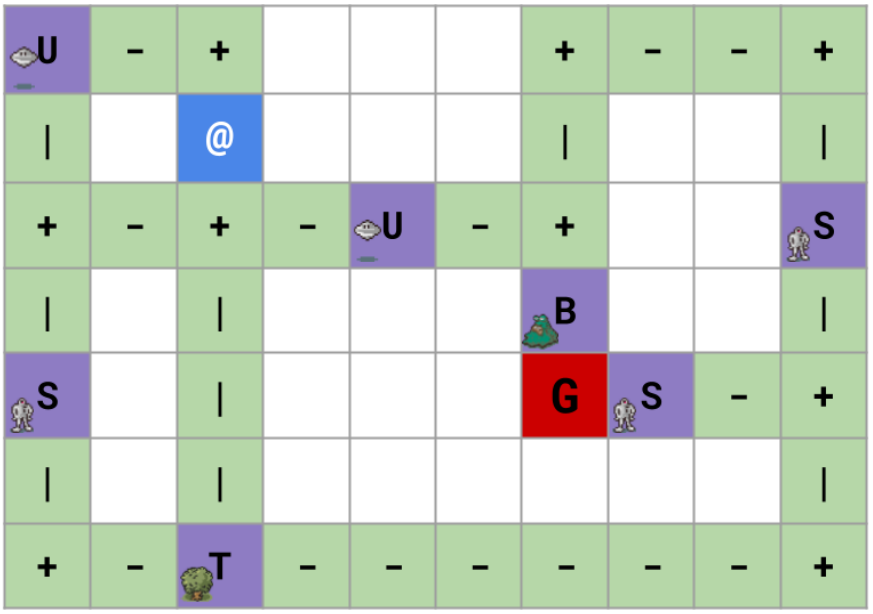

Figura 1. Mapa do jogo.

Além disso, tem-se que o caminho deve-se ser percorrido somente uma vez, exceto os cruzamentos que podem ser visitados diversas vezes. Desse modo, o presente trabalho tem como desafio desenvolver um algoritmo com backtracking, uma técnica que consiste em resolver os problemas de forma recursiva, que busca todas as combinações possíveis para resolver um problema computacional.

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 4 TADs: o *Monstro,* que terá a força e a recompensa de cada monstro; o *Heroi,* que terá a força de Ness, a quantidade de PK Flash (técnica especial) e sua

posição final e inicial no mapa; o *Mapa,* com as informações do mapa, do herói e dos monstros; *e, o ModoAnalise*, com a quantidade de chamadas recursivas que foram realizadas e o seu nível máximo de recursão.

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 foram criadas 4 estruturas de dados para uma melhor encapsulamento e e resolução do problema, o primeiro TAD criado foi o *heroi,* nele declaramos as variáveis *forca* usada para indicar a força do personagem Ness, a variável *kflash,* e as variáveis i e j, que marcam a posição inicial de Ness, veja a estrutura de dados *Heroi,* na figura 2.

```
typedef struct{
    int forca;
    int kflash;
    int i;
    int j;
}Heroi;
```

Figura 2. TAD Heroi

A segunda estrutura criada foi o TAD *monstro*, cujo o principal objetivo é representar um monstro, armazenando assim sua força e recompensa que Ness irá ganhar caso derrote um monstro qualquer, veja a estrutura de dados monstro, na figura 3.

```
typedef struct{
    int forca;
    int recompensa;
}monstro;
```

Figura 3. TAD monstro

A terceira estrutura foi o TAD *Mapa*, no qual armazenamos um ponteiro para uma matriz que irá representar o mapa a ser percorrido pelo personagem principal. Ademais, foram declaradas as variáveis para o personagem principal e os monstros que Ness terá que lutar, representada na figura 4.

```
typedef struct{
    char **mapa;
    int linha;
    int coluna;
    Heroi heroi;
    monstro U;
    monstro T;
    monstro S;
    monstro B;
    monstro G;
}Mapa;
```

Figura 4. TAD Mapa.

A quarta estrutura é o TAD *modoAnalise*, responsável por contar a quantidade de vezes que a função movimentar será chamada, no caso quantas chamadas recursivas foram feitas e qual é nível máximo da árvore de recursão, como mostrada na figura 5.

```
typedef struct{
    long long int quantidadeRecursao;
    int nivel;
}modoAnalise;
```

Figura 5. TAD modoAnalise

3.2 Backtracking

O backtracking implementado consiste principalmente dos seguintes raciocínios: marcar o mapa, lutar ao encontrar um inimigo e finalizar se for o Giygas e Ness conseguir derrotá-lo, movimentar para uma posição se possível, uma simulação para evitar loops infinitos e voltar dados ao estado original quando o caminho percorrido não for o correto.

3.2.1 Parâmetros do backtracking

```
int movimentar(Mapa *mapa, int i, int j,Pilha* pilha,char posAntiga,int
opAntiga, modoAnalise *m, int nivel,int *ptr);
```

Figura 6. Método que implementa o backtracking

A seguir é explicado o uso dos parâmetros da função ilustrada na figura 6:

- **mapa**: É o mapa fornecido pelo arquivo de texto e onde é feito as operações necessárias para que Ness tente derrotar Giygas.

- **i**: Linha atual onde se encontra o herói.
- **j**: Coluna atual onde se encontra o herói.
- **pilha**: Pilha utilizada para mostrar o caminho percorrido pelo herói caso ele consiga derrotar Giygas.
- **posAntiga**: É o símbolo anterior ao atual, ou seja, é o símbolo da posição anterior ao que o herói está.
- **opAntiga**: É a operação anterior, ou seja, a operação que levou Ness a posição atual, se ele veio de cima, de baixo, da esquerda ou da direita.
- **m**: Utilizado para utilizar o backtracking em modo análise se requisitado.
- **nivel**: Calcula o nível máximo da árvore de níveis.
- **ptr**: Conta a quantidade de chamadas recursivas.

3.2.2 Marcação do mapa

Sempre que Ness se movimenta é necessário marcar o caminho que ele passou para que ele não percorra o mesmo caminho, a marcação é feita substituindo o símbolo atual por um caractere *"X"*. No entanto, posições com o símbolo de cruzamento não são marcados, pois podem ser percorridos mais de uma vez. Vale ressaltar que quando Ness abate um inimigo o seu símbolo no mapa é transformado em um cruzamento. A implementação destas marcações podem ser vistas nas figuras 7 e 8.

```
if(aux!=('+')) mapa->mapa[i][j]='X';
```

Figura 7. Bloco de código onde o caminho do Ness é marcado.

```
            if(lutar(mapa,aux)) {
                mapa->mapa[i][j]='+';
                teveLuta=true;
                  novaForca=mapa->heroi.forca;
                  novokFlash=mapa->heroi.kflash;
                }
```

Figura 8 .Bloco de código onde ao Ness derrotar um inimigo a sua posição é marcada como um cruzamento.

3.2.3 Lutar contra um inimigo

Ao encontrar um inimigo no mapa, Ness luta contra ele. Nesta luta ocorrem 4 possibilidades:

- Ness luta contra um inimigo sem ser o Giygas e ganha, seguindo o seu caminho e aumentando sua força com a recompensa adquirida.

- Ness luta contra um inimigo sem ser o Giygas e perde, finalizando a tentativa de passar por esse caminho.
- Ness luta contra o Giygas e ganha, finalizando o backtracking e as tentativas de passar por outro caminho.
- Ness luta contra o Giygas e perde, finalizando a tentativa de passar por esse caminho.

```
if(aux=='G'|| aux=='U' || aux=='S' || aux=='T' || aux=='B'){
    if(aux=='G'){

        if(lutar(mapa,aux)){
          sprintf(string,"Linha %d, Coluna %d; P: %d, k: %d\n",i,j,forca,kflash);
          inicializarItem(&item,string);
          addItem(pilha,&item);
        return true;
        }else{
            mapa->mapa[i][j]=aux;
            return false;
        }
    }else{
       if(lutar(mapa,aux)) {
           mapa->mapa[i][j]='+';
           teveLuta=true;
             novaForca=mapa->heroi.forca;
             novokFlash=mapa->heroi.kflash;
           }
       else{
           mapa->mapa[i][j]=aux;
           return false;
       }
    }
}
```

Figura 9 .Bloco de código em que é feita a tomada de decisões sobre o resultado da luta entre Nesse e o seu inimigo.

A variável *teveLuta*, mostrada na figura 9, é utilizada para verificar se a posição em que Ness percorreu pelo caminho que derrota o Giygas teve uma luta ou não, e caso positivo é mostrado a sua força e PK Flash do Ness naquela posição. O método *lutar* retorna se o Ness venceu ou não a luta com o inimigo, sendo *true* caso vença e *false* caso contrário. É importante destacar que ao perder uma luta e finalizar o caminho, é necessário voltar a marcação ao estado original.

```
int lutar(Mapa* mapa,char monstro){
    char v[] = {'U', 'T', 'S', 'B'};
    int poder[] = {mapa->U.forca, mapa->T.forca, mapa->S.forca, mapa->B.forca};
    int recompensa[] = {mapa->U.recompensa,mapa->T.recompensa, mapa->S.recompensa,
mapa->B.recompensa};
    int k;
     if(monstro=='G'){
         if(mapa->heroi.forca >=  mapa->G.forca){
                mapa->heroi.forca += mapa->G.recompensa;
                return true;
            }else {
                return false;
            }
     }
    for( k = 0; strlen(v); k++){
        if(monstro == v[k]){
            if(mapa->heroi.forca >= poder[k]){
                mapa->heroi.forca += recompensa[k];
                return true;
            }else if(mapa->heroi.kflash > 0){
                mapa->heroi.kflash -= 1;
                mapa->heroi.forca += recompensa[k];
                return true;
            }
        }
    }

return false;
}
```

Figura 10 .Bloco de código em que ocorre a luta entre Nesse e o seu inimigo.

A figura 10 mostra como ocorre a luta entre Ness e o seu inimigo. Para que Ness ganhe uma luta, a sua força tem que ser maior ou igual a do seu inimigo, caso ela seja menor é possível utilizar o seu PK flash em inimigos que não sejam o Giygas, se possuir algum. Ao ganhar uma luta contra um inimigo ele aumenta sua força de acordo com a recompensa fornecida pelo mesmo. É diminuída uma unidade de poder especial a cada vez que Ness precise utilizá-lo.

3.2.4 Movimentar pelo mapa

Esta é a parte do backtracking onde ocorre a recursão. Ness primeiramente tenta se movimentar na seguinte ordem: cima, baixo, direita e esquerda. Antes de se movimentar é

verificado se é possível ir para a posição pretendida. Ele só pode se movimentar de acordo com as seguintes restrições:

- Ness não pode se movimentar para fora do mapa e nem onde há “.”.
- Se a sua posição atual é “-” ele só pode se mover para a direita ou esquerda, se possível.
- Se a sua posição atual é “|" ele só pode se mover para cima ou para baixo, se possível.
- Se a sua posição atual é “+” ele pode se mover para qualquer direção, desde que seja possível.

```
if(posicaoEhValida(mapa, i, j+1) &&  aux != '|'){  //para Direita
    acabou=movimentar(mapa, i, j+1,pilha,aux,2,m,nivel+1,ptr);
}
```

Figura 11 .Bloco de código em que ocorre a movimentação para a direita

A movimentação para as outras direções são análogas às da figura 11. O método *posicaoEhValida* verifica se a posição que Ness quer ir é possível, caso ela seja igual a “X”, “.” ou a esteja fora do mapa é retornado *false* e é finalizado a tentativa de se movimentar para essa posição, tentando a próxima. Caso contrário é retornado *true* e Ness se movimenta para a respectiva posição.

3.2.5 Simulação

Em alguns momentos pode ser que Ness se movimente sempre para as mesmas posições. Isso ocorre quando há dois cruzamentos seguidos, pois os cruzamentos não podem ser marcados, sendo assim é necessário identificar e mostrar para Ness que ele deve sair do loop.

```
    if(posAntiga=='+' && aux=='+'){
       if(simular(opAntiga,0)){
               mapa->heroi.forca=forca;
               mapa->heroi.kflash=kflash;
           return false;}
    }
```

Figura 12. Bloco de código em que ocorre a tomada de decisão de acordo com a resposta da simulação

É feito uma simulação utilizando a operação anterior e a operação que será realizada. Sempre que a operação anterior for pra cima e a posição a ser realizada for baixo e vice-versa, ou se a operação anterior for direita e a posição a ser realizada for esquerda e

vice-versa, haverá um loop e então é finalizada a tentativa de ir por este caminho. Como podemos ver na figura 13, é retornado *true* sempre que a simulação identifica um loop e *false* caso contrário.

```
int simular(int opAnterior, int opAtual){
    if((opAnterior==0 && opAtual==1) ||
     (opAnterior==1 && opAtual==0) ||
      ((opAnterior==2 && opAtual==3) ||
      (opAnterior==3 && opAtual==2) )){ return true;}
    else{return false;}}
```

Figura 13. Bloco de código em que ocorre a simulação

3.2.6 Caminho incorreto

Há 3 possibilidades em que um caminho é incorreto, ou seja, este não é o caminho que deve ser percorrido. São as seguintes ocasiões:

- Ness não conseguiu derrotar um inimigo.
- Ocorreu um loop.
- Em uma recursão foi testado todos os movimentos e nenhum deles foi o correto para derrotar o Giygas.

Sempre que um caminho é reconhecido como incorreto é necessário desmarcar a posição em que ele foi reconhecido como inválido e voltar a força e o poder especial do Ness para a sua força e quantidade de poder especial que ele tinha na posição anterior, como ilustrada na figura 14.

```
//Desmarcando posição atual
    mapa->mapa[i][j]=aux;
    //Voltando atributos do Ness ao que tinha antes de entrar nesta posição
    mapa->heroi.forca=forca;
    mapa->heroi.kflash=kflash;
    return false;
```

Figura 14. Bloco de código em que valores são voltados ao da posição anterior

3.2 Gerador de mapas

Neste trabalho foi implementado um gerador de mapas como atividade extra. Todas as funcionalidades deste gerador de mapas se encontram no TAD geradorMapas, nele tem-se as 3 funcionalidade principais que são: *modoFacil(), modoMédio()* e o *modoDifícil(),* em cada funcionalidade dessa é criado aleatoriamente os poderes de Ness, de Giygas e de cada monstro e suas recompensas que são dadas a Ness caso ele vença. Todos estes valores são

escritos de forma aleatória em um arquivo. Assim que o poderes de todos os personagens do jogo são inscritos no arquivo é chamada a função *geradorMapa()* que tem como parâmetro uma matriz que irá receber os caracteres, e as posições que estarão cada personagem do jogo, gerando tudo de forma aleatória, logo depois os valores que estão nesta matriz serão passados para o arquivo. Veja na tabela abaixo o nível de dificuldade de cada modo.

Nível de dificuldade	Linhas	Colunas	Monstros
modoFacil()	7	10	7
modoMedio()	10	13	10
modoDificil()	13	16	13

Tabela 1. Nível de dificuldade de cada modo.

É importante ressaltar que para cada modo os valores que são gerados, como o poder de Ness e seu PKflash, o poder de Giygas, e o poder dos outros monstros e suas recompensas, os valores que serão gerados aleatoriamente são dobrados a cada modo.

3.3 Modo análise

O principal objetivo dessa funcionalidade é verificar como está o funcionamento da função *movimentar()*, explicada anteriormente. Desse modo, implementou-se o TAD *modoAnalise*, com a quantidade de chamadas recursivas que foram realizadas e o seu nível máximo de recursão. Além disso, foi utilizado a biblioteca *Time.h* juntamente com a função *clock()*, no início e no final da função *movimentar()* para verificar o tempo de execução do algoritmo.

Para calcular o quantidade de chamadas recursivas foi passado uma variável por referência iniciando em zero e sendo acrescentando mais um a cada nova chamada da função *movimentar()*.

Para calcular o nível máximo de recursão, primeiramente deve-se saber que em uma função recursiva os valores são empilhados até que se possa chegar na condição de parada, que no caso seria quando Ness encontrar um caminho possível para eliminar seu inimigo, Giygas, testando todas as n possibilidades. Dessa forma, é criada como se fosse uma árvore de acordo com o processo de tentativa de sub-tarefas, ilustrado na figura 16, na qual as bolinhas amarelas simbolizam a chamada da função. Pode-se observar que o nível máximo desse exemplo é cinco.

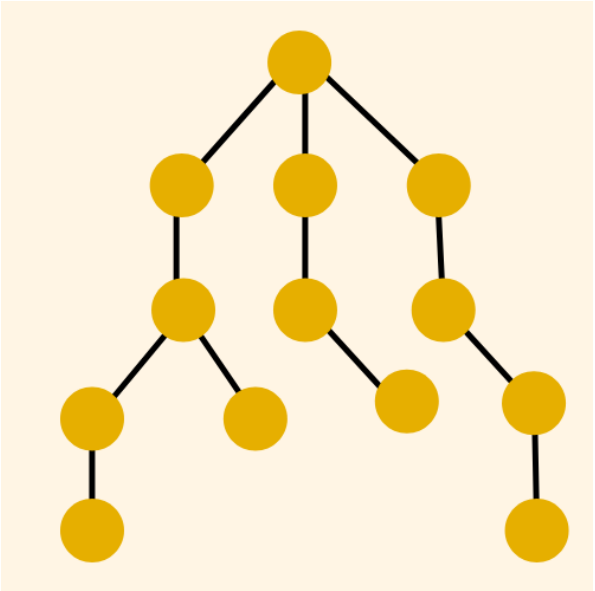

Figura 15. Ilustração representando uma árvore de recursão.

Utilizando esse raciocínio, utilizou-se uma variável inteira denominada nível na função *movimentar(),* que irá ser incrementada a cada chamada recursiva. Para que o valor não seja perdido quando a função começar o seu desempilhamento, é necessário passar por uma condição de verificação para salvar o maior valor encontrado, mostrada na figura 17.

```
if(nivel == 0){
    m->nivel = nivel;
}
if(nivel > m->nivel){
    m->nivel = nivel;
}
```

Figura 16. Armazenando o nível máximo de uma função recursiva.

O modo análise será ativado juntamente com a função *movimentar()*. Para visualizar os resultados será apresentado um menu de opções, no qual o usuário poderá decidir se gostaria de verificar os resultados da performance da recursão.

3.4 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/geradorMapa.c sources/heroi.c sources/mapa.c sources/modoAnalise.c sources/monstro.c sources/pilha.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 linha as variáveis P e K e nas próximas cinco linhas as variáveis P' e X' para cada um dos inimigos, na ordem U, T, S, B e G. Logo em seguida, será informado as dimensões do mapa, largura e altura, respectivamente. E por fim, o mapa que será utilizado no jogo, representado na figura 18.

```
20 2
10 5
25 10
30 15
40 20
80 0
7 10
U-+...+--+
|.@...|..|
+-+-U-+..S
|.|...B..|
S.|...GS-+
|.|......|
+-T------+
```

Figura 17. Representação das entradas no terminal.

Após a leitura do arquivo, temos como saída o caminho percorrido por Ness para eliminar seu rival inimigo, Giygas, representado na figura 19. Caso não seja encontrado nenhum caminho possível, será apresentado uma mensagem, ilustrada na figura 20.

```
Linha 1, Coluna 2;
Linha 0, Coluna 2;
Linha 0, Coluna 1;
Linha 0, Coluna 0; P: 25, k: 2
```

```
Linha 1, Coluna 0;
Linha 2, Coluna 0;
Linha 3, Coluna 0;
Linha 2, Coluna 0;
Linha 2, Coluna 1;
Linha 2, Coluna 2;
Linha 3, Coluna 2;
Linha 2, Coluna 2;
Linha 2, Coluna 3;
Linha 2, Coluna 4; P: 30, k: 2
Linha 2, Coluna 5;
Linha 2, Coluna 6;
Linha 3, Coluna 6; P: 50, k: 1
Linha 2, Coluna 6;
Linha 1, Coluna 6;
Linha 0, Coluna 6;
Linha 0, Coluna 7;
Linha 0, Coluna 8;
Linha 0, Coluna 9;
Linha 1, Coluna 9;
Linha 2, Coluna 9; P: 65, k: 1
Linha 3, Coluna 9;
Linha 4, Coluna 9;
Linha 5, Coluna 9;
Linha 4, Coluna 9;
Linha 4, Coluna 8;
Linha 4, Coluna 7; P: 80, k: 1
Linha 4, Coluna 6; P: 80, k: 1
```

Figura 18. Representação do caminho percorrido por Ness no terminal.

```
Apesar de todas as tentativas, Ness falha em derrotar Giygas!
```

Figura 19. Representação no terminal quando nenhum caminho foi possível.

Após a execução da função *movimentar()*, a partir do arquivo de entrada representado na figura 18, temos que a representação da análise do algoritmo. Como pode-se observar houveram 119 chamadas recursivas, o nível máximo de chamadas recursivas foram 41, sendo executada no tempo de 1 milésimo de segundo, descrito na figura 21.

```
====================MODO ANALISE=====================
Nivel maximo da chamada recursiva:41
Quantidade de chamadas recursivas que foram feitas: 119
Tempo gasto: 1.000000 ms
```

Figura 20. Representação do modo análise no terminal.

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 foi o backtracking, este algoritmo representa um refinamento da busca por força bruta, em que múltiplas soluções podem ser eliminadas sem serem explicitamente examinadas, em linhas gerais uma árvore é percorrida sistematicamente de cima para baixo e da esquerda para a direita. Quando essa pesquisa falha, ou é encontrado um nodo terminal da árvore, entra em funcionamento o mecanismo backtracking. Esse procedimento faz com que o sistema retorne pelo mesmo caminho percorrido com a finalidade de encontrar soluções alternativas.

Outrossim, o objetivo principal deste trabalho foi criar a função *movimentar()*, que utiliza os conceitos de backtracking e usa a força bruta para resolver um problema. Dessa forma, o problema a ser resolvido era um labirinto onde o herói teria que percorrer um percurso para vencer seu inimigo, este labirinto estava em forma de matriz, então passada a posição da coluna e a posição da linha da matriz na função, o herói ia percorrendo os caminhos para derrotar seu inimigo Giygas. Logo, a lógica principal era ir verificando se existia algum caminho para cima, baixo, esquerda ou direita para Ness andar na matriz, se havia, era chamada a função recursiva até Ness encontrar Giygas.

Portanto, esta abordagem mais prática da implementação do backtracking, 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]Backtracking. Wikipédia, 2020. Disponível em: Link. Acesso em: 17 de Fev. de 2022.

[2]Toffolo, Túlio. Backtracking. decom, 2022. Disponível em: Link. Acesso em: 15 de Fev. de 2022.