



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

Trabalho Prático I - Projeto e Análise de Algoritmos

Algoritmo com Backtracking para saída de labirinto

Caio Menezes Oliveira [5784]

Amanda Caroline Melo Assunção [5366]

Melissa Alanis Santos Oliveira [5384]

Leticia Cristina Almeida da Silva [5781]

Florestal - MG

2024

Sumário

1. Introdução.....	3
2. Desenvolvimento.....	3
2.1 Arquivos Labirinto.....	3
2.1.1 - alocaLabirinto.....	3
2.1.2 - processaLabirinto.....	4
2.1.3 - posicaoValida.....	5
2.1.4 - movimenta_estudante.....	6
2.1.5 - imprimeCaminho.....	8
2.1.6 - Modo Análise.....	8
2.2 Main.....	9
2.3 Extras.....	10
2.3.1 - Interface.....	10
2.3.2 - Labirintos de Teste.....	11
2.3.3 - Células Amarelas.....	14
2.3.4 - Criação de Portal.....	15
2.3.5 - Gráfico.....	15
3. Compilação e Execução.....	17
4. Resultados.....	18
5. Conclusão.....	20
6. Referências.....	21

1. Introdução

A presente documentação gira em torno do projeto e da implementação na linguagem C de um algoritmo para encontrar a saída, com o uso do **Backtracking**, de um dado labirinto que é fornecido ao programa em um arquivo de texto (txt). O objetivo deste trabalho prático é fazer com que, além de aprofundar os conhecimentos da equipe no que tange o paradigma em questão, o estudante possa cruzar o labirinto (caso possível) diante das paredes, portas e chaves que estão dispostas no caminho.

2. Desenvolvimento

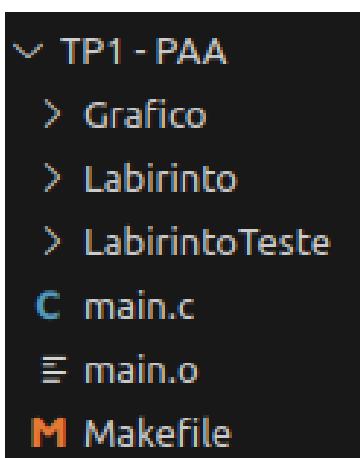


Imagem 1 - Folder do Trabalho no VS

Para o desenvolvimento deste trabalho, o programa foi estruturado de forma modular, utilizando funções independentes que, em conjunto, possibilitam com que seja procurada uma saída, se existir, em um dado labirinto.

Nesse contexto, o trabalho foi dividido em alguns arquivos principais: `labirinto.c`, `labirinto.h`, `labirintoteste.c`, `labirintoteste.h`, `grafico.py` que abrigam os métodos implementados da biblioteca e as opções extras deste trabalho prático. Além disso, também foi criado o **`main.c`**, que contém o método `main` definido para executar as operações.

A seguir, serão apresentadas as explicações acerca das funções implementadas.

2.1 Arquivos Labirinto

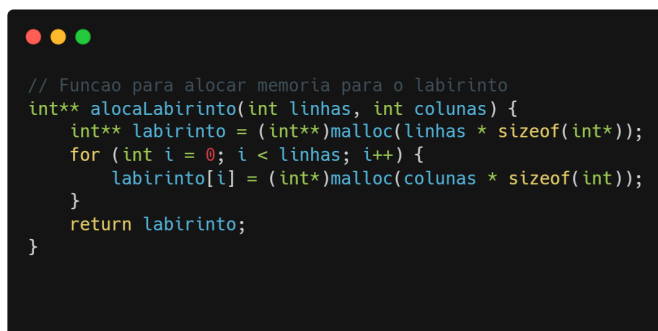
Primeiramente, para a criação de todas as funções em torno da implementação do labirinto e da fuga do estudante foi-se criado um TAD composto por “**Labirinto.h**” e “**Labirinto.c**” para a melhor organização e estruturação dessa parte do algoritmo.

2.1.1 - `alocaLabirinto`

Esta função tem seu objetivo principal relacionado a alocação dinâmica de memória para a construção do labirinto. Para isso, são passados como parâmetros o

número de linhas e de colunas e a partir da utilização de ponteiros e da função malloc é criado um array de ponteiros, e cada ponteiro deste array representará uma linha da matriz.

Após isso, um laço “for” é utilizado para iterar por todas as linhas do labirinto. Em cada iteração, o ponteiro correspondente àquela linha é configurado para apontar para um bloco de memória. Tal método garante com que cada linha tenha espaço suficiente para armazenar os elementos das colunas.

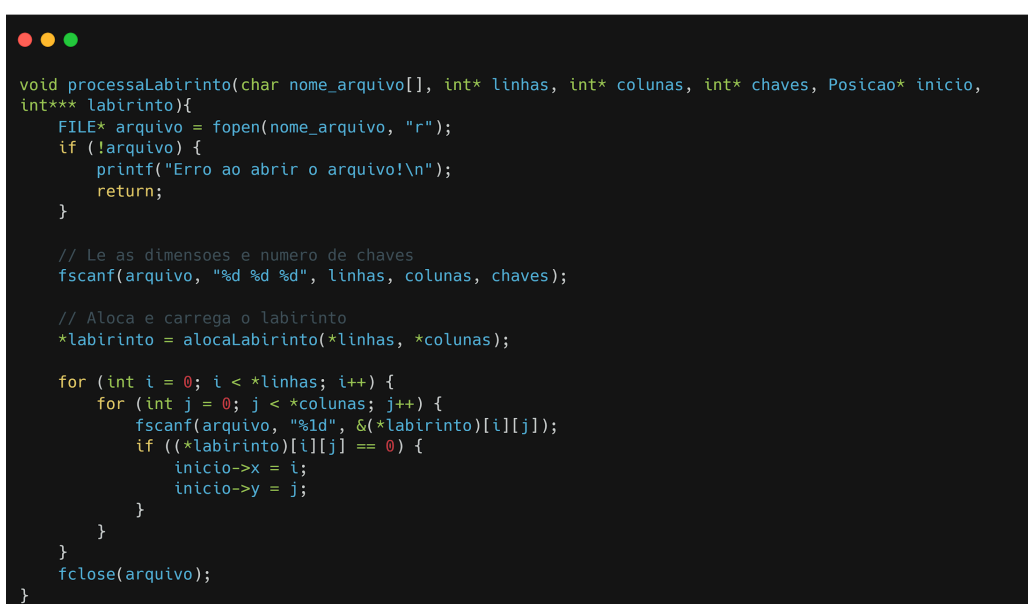


```
// Funcao para alocar memoria para o labirinto
int** alocaLabirinto(int linhas, int colunas) {
    int** labirinto = (int**)malloc(linhas * sizeof(int*));
    for (int i = 0; i < linhas; i++) {
        labirinto[i] = (int*)malloc(colunas * sizeof(int));
    }
    return labirinto;
}
```

Imagem 2 - Função alocaLabirinto()

2.1.2 - processaLabirinto

O procedimento tem como objetivo carregar os dados de um labirinto a partir de um arquivo e inicializar as estruturas necessárias para sua manipulação. Inicialmente, a função tenta abrir o arquivo especificado pelo nome passado no argumento. Em seguida, é feita a leitura das dimensões do labirinto e da quantidade de chaves que o estudante possui.



```
void processaLabirinto(char nome_arquivo[], int* linhas, int* colunas, int* chaves, Posicao* inicio,
int** labirinto){
    FILE* arquivo = fopen(nome_arquivo, "r");
    if (!arquivo) {
        printf("Erro ao abrir o arquivo!\n");
        return;
    }

    // Le as dimensoes e numero de chaves
    fscanf(arquivo, "%d %d %d", linhas, colunas, chaves);

    // Aloca e carrega o labirinto
    *labirinto = alocaLabirinto(*linhas, *colunas);

    for (int i = 0; i < *linhas; i++) {
        for (int j = 0; j < *colunas; j++) {
            fscanf(arquivo, "%ld", &(*labirinto)[i][j]);
            if ((*labirinto)[i][j] == 0) {
                inicio->x = i;
                inicio->y = j;
            }
        }
    }
    fclose(arquivo);
}
```

Imagem 3 - Função processaLabirinto()

Após isso, com o conhecimento das dimensões do labirinto é feita a alocação da matriz que o armazena, por meio da invocação do método **alocaLabirinto**.

Por fim, cada célula da matriz é percorrida enquanto os valores de cada linha do arquivo são lidos de forma a preencher a matriz. Durante esse processo, a função verifica se algum valor é igual a zero, o que indica a posição inicial do labirinto. Nesse caso, as coordenadas dessa posição são armazenadas na estrutura **inicio**, e por último o arquivo é fechado.

2.1.3 - posicaoValida

A função tem como objetivo verificar se uma posição é válida para movimentação, levando em conta as regras do labirinto e número de chaves que o estudante possui. Para isso, o procedimento começa verificando se o ponto (x,y) está dentro dos limites do labirinto (ser maior que zero e menor que **linhas** para x, maior ou igual a zero, e menor que **colunas** para y).

Em seguida, é verificado se a posição é uma célula vazia, por onde o estudante pode passar - representada pelo número 1 -, se é um espaço que possui uma chave - simbolizado pelo número 4, ou se é a saída ou entrada de um portal - 5 e 6. Se for o caso, a função retorna 1, indicando uma posição válida. Além disso, é analisado se o ponto é um local que possui uma porta - representado pelo número 3 - e se o estudante possui pelo menos uma chave, e caso seja verdadeiro indica que é possível se movimentar por esse espaço, retornando 1. Se nenhuma das condições for satisfeita, o procedimento retorna 0, assinalando que não é célula acessível.

```
// Funcao para verificar se a posicao eh valida
int posicaoValida(int** labirinto, int linhas, int colunas, int x, int y, int chaves) {
    if (x >= 0 && x < linhas && y >= 0 && y < colunas) { // Dentro dos limites
        if (labirinto[x][y] == 1 || labirinto[x][y] == 4 || labirinto[x][y] == 5 || labirinto[x][y] == 6)
        { // Celula livre ou chave
            return 1;
        } else if (labirinto[x][y] == 3 && chaves > 0) { // Porta e tem chave
            return 1;
        }
    }
    return 0;
}
```

Imagem 4 - Função posicaoValida()

2.1.4 - movimenta_estudante

A função **movimenta_estudante** é responsável por executar o processo de backtracking, testando todas as movimentações possíveis para o estudante até que ele encontre, ou não, a saída do labirinto. O fluxo do código segue as seguintes etapas:

1. Verifica-se se o estudante chegou ao **caso base** (linha 0), se sim, a função retorna 1.
2. A célula atual é marcada como visitada, e seu estado anterior é armazenado para restaurar caso seja necessário no retorno.
3. A primeira possibilidade de movimentação testada é o portal, representado pela célula 5 (seu funcionamento será detalhado no tópico 2.3.4). Caso ele consiga chegar ao caso base por meio desse caminho, é retornado 1.
4. O backtracking é aplicado às demais direções de movimento: cima, esquerda, direita e baixo, nessa ordem. Para cada possibilidade, a função **posicaoValida** é invocada, se o movimento for válido, a função realiza uma chamada recursiva com os novos valores, retornando 1 caso um dos caminhos leve até a saída.
5. Se, após testar todas as quatro direções, o estudante não alcançar a saída, a célula é restaurada ao seu estado inicial, e a função retorna 0.

Vale ressaltar que, na etapa 4, é realizado o tratamento das chaves. Dessa forma, caso o movimento seja para uma porta válida, decrementa-se a variável auxiliar **novas_chaves**. Se o movimento for inválido, a recursividade é usada para "recuperar" a chave, uma vez que a variável local passada como parâmetro na próxima chamada recursiva restaura automaticamente o valor anterior. Um tratamento semelhante é feito com as chaves das células amarelas e será explicado no decorrer do texto.

```
// Funcao recursiva para movimentar o estudante
int movimenta_estudante(int** labirinto, int linhas, int colunas, int x, int y, int chaves, int nivel,
    Posicao* caminho, unsigned int* passos, int* tamanho_caminho) {
    #if MODO_ANALISE == 1 /// Variaveis para analise
        chamadas_recursivas++;
        if (nivel > nivel_maximo_recursao) nivel_maximo_recursao = nivel;
    #endif
    // Caso base: chegou na primeira linha
    if (x == 0) {
        labirinto[x][y] = 9;
        caminho[*tamanho_caminho].x = x;
        caminho[*tamanho_caminho].y = y;
        (*tamanho_caminho)++;
        (*passos)++;
        return 1;
    }
    // Marca a posicao como visitada
    int estado_atual = labirinto[x][y];
    labirinto[x][y] = 9;
```

Imagem 5 - Função movimenta_estudante()

```

if(estado_atual == 5){
    // Busca a saída do portal (valor 6 no labirinto)
    for (int i = 0; i < linhas; i++) {
        for (int j = 0; j < colunas; j++) {
            if (labirinto[i][j] == 6) {
                // Tenta mover para a nova posicao
                if (movimenta_estudante(labirinto, linhas, colunas, i, j, chaves, nivel + 1, caminho,
passos, tamanho_caminho)) {
                    // Se conseguiu, adiciona a posicao ao caminho e atualiza seu tamanho
                    caminho[*tamanho_caminho].x = x;
                    caminho[*tamanho_caminho].y = y;

                    (*tamanho_caminho)++;
                    return 1;
                }
            }
        }
    }
}

// Movimentos possiveis (cima, esquerda, direita, baixo)
int movimentos[4][2] = {{-1, 0}, {0, -1}, {0, 1}, {1, 0}};

// Tenta mover para as 4 direcoes
for (int i = 0; i < 4; i++) {
    // Calcula a nova posicao
    int novoX = x + movimentos[i][0];
    int novoY = y + movimentos[i][1];

```

Imagem 6 - Parte 2 função movimenta_estudante()

```

if (posicaoValida(labirinto, linhas, colunas, novoX, novoY, chaves)) { // Verifica se a posicao eh
valida
    (*passos)++; // Contabiliza todas as movimentacoes do estudante
    // Se for uma porta, consome uma chave
    int novas_chaves = chaves;
    if (labirinto[novoX][novoY] == 3) {
        novas_chaves--;
    }
    // O estudante encontrou uma chave
    else if (labirinto[novoX][novoY] == 4) {
        novas_chaves++;
    }
    // Tenta mover para a nova posicao
    if (movimenta_estudante(labirinto, linhas, colunas, novoX, novoY, novas_chaves, nivel + 1,
caminho, passos, tamanho_caminho)) {
        // Se conseguiu, adiciona a posicao ao caminho e atualiza seu tamanho
        caminho[*tamanho_caminho].x = x;
        caminho[*tamanho_caminho].y = y;
        // *matriz[x][y] = 9;
        (*tamanho_caminho)++;
        return 1;
    }
}
}
// Desmarca a posicao e retorna ao estado anterior
labirinto[x][y] = estado_atual;
return 0;
}

```

Imagem 7 - Parte 3 Função movimenta_estudante()

2.1.5 - imprimeCaminho

A função **imprimeCaminho** é responsável pela exibição de cada elemento do labirinto representado por uma matriz bidimensional, utilizando cores para diferenciar os elementos de acordo com seus respectivos valores. Se o valor for 9, ele é exibido com uma cor verde sem números visíveis, desta maneira, destacando as posições visitadas pelo estudante. Para os demais valores, foram implementadas as cores de fundo que já estavam explicitadas na especificação deste trabalho prático, juntamente dos seus respectivos números.

```
// Imprime o labirinto colorido
void imprimeCaminho(int** labirinto, int linhas, int columnas){
    for (int i = 0; i < linhas; i++) {
        for (int j = 0; j < columnas; j++) {
            //printf("%d ", labirinto[i][j]);
            if(labirinto[i][j]==9){
                printf(cor_verde " " resetar_cor);
            }
            else if(labirinto[i][j]==0){
                printf(cor_verde "%d " resetar_cor, labirinto[i][j]);
            }
            else if(labirinto[i][j]==2){
                printf(cor_azul "%d " resetar_cor, labirinto[i][j]);
            }
            else if(labirinto[i][j]==3){
                printf(cor_vermelha "%d " resetar_cor, labirinto[i][j]);
            }
            else if(labirinto[i][j]==4){
                printf(cor_amarela "%d " resetar_cor, labirinto[i][j]);
            }
            else if(labirinto[i][j]==0){
                printf(cor_verde "%d " resetar_cor, labirinto[i][j]);
            }
            else if(labirinto[i][j]==1){
                printf(cor_branca "%d " resetar_cor, labirinto[i][j]);
            }
            else{
                printf("%d ", labirinto[i][j]);
            }
        }
        printf("\n");
    }
}
```

Imagem 8 - Função imprimeCaminho()

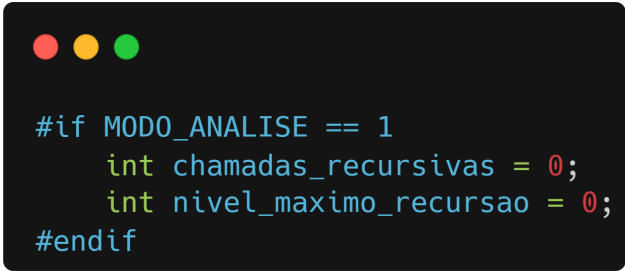
2.1.6 - Modo Análise

Para a implementação do modo análise, conforme solicitado na especificação do trabalho, o grupo optou por criar um **#define MODO_ANALISE** no arquivo **labirinto.h**. Quando o valor é definido como **1**, o modo análise é ativado, e quando definido como **0**, o programa executa normalmente, sem a contabilização de chamadas recursivas e níveis de recursão. Para ativar ou desativar o modo análise, é necessário alterar manualmente o valor atribuído ao **#define MODO_ANALISE** no arquivo **.h**.

No arquivo **labirinto.c**, foram adicionados comandos condicionais para verificar se o modo análise está ativo. Duas variáveis globais, **chamadas_rekursivas** e **nivel_maximo_rekursao**, foram declaradas para contabilizar o total de chamadas recursivas e o nível máximo de recursividade alcançado. Essas variáveis são manipuladas em função de verificações condicionais no código.

Na função **movimenta_estudante**, a lógica do modo análise incrementa a variável **chamadas_rekursivas** a cada invocação do procedimento. Além disso, uma verificação é realizada para atualizar **nivel_maximo_rekursao** caso o nível de profundidade atual da chamada recursiva seja maior que o valor anteriormente registrado. A mesma abordagem foi aplicada na função **movimenta_estudanteMatriz**, que realiza os mesmos procedimentos descritos anteriormente, juntamente com a impressão do caminho realizado pelo estudante.

No main, uma verificação condicional foi adicionada para assegurar que o modo análise esteja ativado quando necessário. Como as variáveis globais **chamadas_rekursivas** e **nivel_maximo_rekursao** foram definidas em outro arquivo, elas são declaradas como **extern** para que possam ser utilizadas corretamente. Para exibir os resultados do modo análise, uma condição analisa se **MODO_ANALISE** está ativa, e, caso esteja, os valores totais de chamadas recursivas e o nível máximo de recursividade são exibidos. Essa verificação foi incorporada nas opções 2 e 4 do menu, uma vez que são responsáveis por mostrar os resultados do labirinto.



```
#if MODO_ANALISE == 1
    int chamadas_rekursivas = 0;
    int nivel_maximo_rekursao = 0;
#endif
```

Imagem 9 - Definição do modoAnálise

2.2 Main

A função main, por sua vez, é o procedimento em que ocorre a interação com o usuário propriamente. É mostrado ao usuário, no início da execução, um menu para que este escolha qual operação deseja realizar — o que inclui carregar um novo arquivo de dados, processar e exibir resposta, gerar um labirinto de teste e sair do programa.

Além disso, existe uma flag nomeada como **resultado_labirinto**, a qual é inicializada com o valor **-1**, usada para controlar o fluxo de execução do programa. As opções 2 e 4 do menu são independentes e, quando selecionadas, se a flag estiver com o valor **-1**, a função **movimenta_estudante** é chamada. O retorno dessa função determina se há uma saída no labirinto: se o retorno for 0, significa que não há saída, e se for 1, uma saída foi encontrada. Na opção 2, se houver saída, o programa exibe as linhas e colunas pelas quais o estudante se movimentou, por meio da invocação do procedimento **imprimeCaminho**, e, ao final, apresenta uma matriz representando o caminho realizado. Já na opção 4, se uma saída for encontrada, a função **movimentaEstudanteMatriz** é chamada, e, assim, o programa detalha todo o percurso, mostrando cada movimento realizado e uma matriz que representa o caminho progressivamente. Se não houver saída, uma mensagem informando essa situação é apresentada.

Por fim, têm-se as particularidades de cada opção, como tratamento de exceções, inicializações de estruturas auxiliares, implementação da análise de desempenho, cálculo do tempo de execução para geração de gráfico, e as chamadas das respectivas funções correspondentes.

2.3 Extras

A equipe optou por realizar algumas das tarefas ditas “opcionais”, como a implementação de nova interface para exibir os resultados, geração de labirintos testes, criação de células amarelas no caminho, e elaboração de gráfico relacionando complexidade e tempo. Em seguida, serão apresentados os detalhes sobre as funcionalidades implementadas.

2.3.1 - Interface

Nesta opção extra, foi implementada uma funcionalidade que exibe o caminho percorrido pelo estudante dentro do labirinto, destacando progressivamente os movimentos efetuados, a partir de uma opção selecionada no menu. A implementação ocorre por meio da criação de uma matriz auxiliar, que é uma cópia do labirinto original. Essa estrutura invoca a função **movimentaEstudanteMatriz**, que realiza o backtracking da mesma forma que a função **movimenta_estudante**, explicada anteriormente, com alguns diferenciais importantes: a matriz é exibida continuamente durante o processo, utilizando a função **imprimeCaminho**, previamente explicada, e mostrando as coordenadas onde o estudante se encontra em cada momento. Ou seja,

nesta opção não é necessário salvar o caminho feito até a saída, pois todas as movimentações feitas pelo estudante são apresentadas imediatamente após serem efetuadas. Esses recursos aprimoram a visualização e a compreensão do trajeto percorrido no labirinto.

2.3.2 - Labirintos de Teste

O gerador de labirintos de teste, desenvolvido em um programa separado e chamado a partir de uma escolha do usuário no Main funciona da seguinte maneira: A função **geraLabirintoTeste()** é convocada juntamente com a passagem dos parâmetros que definem o tamanho do labirinto a ser criado, a quantidade de chaves que o estudante iniciará a rota, quantas chaves estarão disponíveis no caminho, as portas, o nome do arquivo, a dificuldade - em que os valores representam, respectivamente, (1) **Fácil**, (2) **Médio** e (3) **Difícil** - e, por fim, o valor 0 ou 1, que representará a existência de um portal no labirinto em questão ou não.

Aprofundando no procedimento **geraLabirintoTeste**, citado no parágrafo anterior, tem-se a criação do arquivo de saída, que é preenchido inicialmente com os valores das linhas, colunas e chaves na primeira linha, de maneira semelhante à especificação.

Ao início do código, a função **verificaLimite()** convoca o procedimento **calculaCelulasLivres()** para que possa ser realizada uma verificação acerca da possibilidade de criação de labirintos válidos. Logo, tem-se o cálculo do tamanho do labirinto - a partir da quantidade de células - e, em seguida, estima-se a quantidade de paredes que esse labirinto possuirá, pois tal previsão é feita a partir do seu nível de dificuldade. Após isso, esse valor é subtraído do total de células e é verificado se há a disponibilidade de células livres para que os demais elementos sejam aplicados ao labirinto.

```
int calculaCelulasLivres(int linhas, int colunas, int dificuldade) {
    int total_celulas = linhas * colunas;
    /*Como são criadas paredes de acordo com a dificuldade, faz se uma estimativa de quantas
    serão criadas uma vez que é aleatoriamente e não tem como prever
    A previsão é feita dividindo a quantidade total de células pela quantidade
    de probabilidade de paredes escolhidas na geração de labirintos*/
    int paredes_estimada = 0;
    if (dificuldade == 1) {
        paredes_estimada = total_celulas / 5; // 20%
    } else if (dificuldade == 2) {
        paredes_estimada = total_celulas / 4; // 25%
    } else if (dificuldade >= 3) {
        paredes_estimada = total_celulas / 3; // 33%
    }
    //Apos estimar a quantidade
    int reservadas = 1; // Para entrada
    int celulas_livres = total_celulas - paredes_estimada - reservadas;
    return celulas_livres;
}
```

Imagem 10 - Função **calculaCelulasLivres()**

```

int verificaLimites(int linhas, int colunas, int portas, int chaves_caminho, int portal, int dificuldade, int chaves) {
    // Verifica se os elementos do labirinto são lógicos
    if(linhas <= 0 || colunas <= 0 || portas < 0 || chaves_caminho < 0 || portal < 0 || dificuldade <= 0 || chaves < 0){
        return 1;
    }
    int celulas_livres = calculaCelulasLivres(linhas, colunas, dificuldade);
    // Soma a quantidade de elementos, para o portal, caso exista, é contabilizado 2 pois possui entrada e saída
    int elementos = portas * 3 + chaves_caminho + (portal > 0 ? 2 : 0);
    //Para as portas, o valor é multiplicado por 3, pois ao se criar uma porte duas paredes são adicionadas ao seu lado
    //Caso a quantidade de elementos seja maior ou igual a quantidade de celulas livres dividido por 2 não é possível criar
    um labirinto
    if (elementos >= celulas_livres / 2 ) {
        // A comparação é feita com celulas livres / 2 para seguir a proporção de labirintos pequenos, porque por mais que
        //tenha celulas livres pode não ser viável e lógico criar o labirinto
        return 1;
    }
    if(colunas <3){ // Levando em consideração a maneira que que foi implementada a criação de labirintos
        //se tivesse menos que 3 colunas seria inviável
        return 1;
    }
    return 0;
}

```

Imagem 11 - Função verificaLimites()

Em seguida, equipe optou por gerar a matriz do labirinto da seguinte maneira:

1. Inicialmente, todas as posições do labirinto são caminhos livres (**células 1's**).
2. A seguir, com o suporte do procedimento **numeroAleatorio()**, que se utiliza da biblioteca **<time.h>** e da função **rand()**, são plotadas paredes (**células 2's**) em lugares randômicos da matriz. Vale, neste momento, destacar que fora decidido pelo grupo que a dificuldade do labirinto se aplicaria nesta função da seguinte forma: No nível fácil, a cada 4 posições, uma delas tem chance de ser parede - e essa mesma lógica se aplica para os demais graus de dificuldade, em que esse intervalo entre as posições tende a diminuir e, conseqüentemente, a chance de mais paredes serem geradas tende a aumentar.
3. Após a aplicação das paredes, o local inicial do estudante (**célula 0**) é gerado aleatoriamente em uma das posições da última linha do labirinto.
4. São adicionadas de acordo com a quantidade passada pelo parâmetro e em lugares aleatórios também as portas (**células 3's**), que podem ocupar apenas células que eram anteriormente caminho livre. Uma decisão importante tomada pelo grupo neste cenário é a de que, a fim de evitar portas no meio de muitos caminhos livres (o que resultaria em

uma passagem ilógica), uma porta é gerada entre duas paredes - uma à esquerda e outra à direita.

5. Logo, são adicionadas as chaves no caminho (**células 4's**), que também são quantificadas pelo usuário. Vale destacar que adicionamos esse detalhe na geração dos labirintos devido à implementação da outra tarefa extra "**chaves amarelas**".
6. Por fim, de maneira a ajustar o labirinto de teste gerado com a tarefa extra de "**opção da equipe**" que também foi implementada, tem-se a existência de um portal, o qual permite que o estudante possa atravessar o labirinto sem muitos problemas. Com relação à implementação do portal no código, tanto a entrada (**célula 5**), quanto a saída (**célula 6**) precisam ocupar posições que eram caminhos livres anteriormente.

Para finalizar a criação do labirinto de teste, a matriz gerada a partir das particularidades retratadas acima é passada para o arquivo de saída, que resultará em um caso de teste idêntico aos fornecidos pelo professor na especificação do trabalho prático, que podem inclusive ser processados em seguida no Main.

A seguir, tem-se alguns labirintos que foram gerados a partir da implementação do extra de "**Labirintos de Teste**". Vale ressaltar que, a cada teste gerado, aumentamos o nível de dificuldade para que se tenha uma gama maior de labirintos. As imagens estão dispostas da seguinte maneira: ao lado esquerdo, tem-se a solicitação das informações necessárias para que seja gerado um labirinto; já do lado direito é exibido o labirinto final em formato "nomeArquivo".txt .

```

Bem-Vindo(a) ♥
-----
Opcoes do labirinto:
-----
(1) Carregar novo arquivo de dados.
(2) Processar e exibir resposta.
(3) Gerar um labirinto de teste (Extra).
(4) Observar caminho feito pelo estudante (Extra).
(5 ou qualquer outro caracter) Sair do programa.
-----

Digite um numero: 3

Digite a quantidade de linhas: 10
Digite a quantidade de colunas: 10
Digite a quantidade de chaves: 2
Digite a quantidade de portas: 5
Digite a quantidade de chaves no caminho: 2
Digite o nível de dificuldade (1 a 3 - fácil para difícil): 1
Digite a quantidade de portais (0 ou 1): 1
Digite o nome do arquivo que você deseja gerar. (Exemplo: 'teste.txt'): teste1.txt
Labirinto gerado e salvo no arquivo 'teste1.txt'
Pressione Enter para continuar...

```

```

TP1 - PAA > ≡ teste1.txt
1      10 10 2
2      121111112
3      2111232112
4      2211611122
5      2324111211
6      1123212111
7      2421111111
8      1111112232
9      1122111111
10     1111232511
11     1211011111

```

Imagem 12 - Teste 1

```

Bem-Vindo(a) ♥
-----
Opcoes do labirinto:
-----
(1) Carregar novo arquivo de dados.
(2) Processar e exibir resposta.
(3) Gerar um labirinto de teste (Extra).
(4) Observar caminho feito pelo estudante (Extra).
(5 ou qualquer outro caracter) Sair do programa.
-----

Digite um numero: 3

Digite a quantidade de linhas: 10
Digite a quantidade de colunas: 10
Digite a quantidade de chaves: 2
Digite a quantidade de portas: 5
Digite a quantidade de chaves no caminho: 1
Digite o nível de dificuldade (1 a 3 - fácil para difícil): 2
Digite a quantidade de portais (0 ou 1): 1
Digite o nome do arquivo que você deseja gerar. (Exemplo: 'teste.txt'): teste2.txt
Labirinto gerado e salvo no arquivo 'teste2.txt'
Pressione Enter para continuar...

```

```

TP1 - PAA > ≡ teste2.txt
1 | 10 10 2
2 | 1112121211
3 | 2112232232
4 | 1121111232
5 | 1212212111
6 | 1111121111
7 | 111223214
8 | 1112111141
9 | 2223211221
10 | 2211212111
11 | 1111111102

```

Imagem 13 - Teste 2

```

Bem-Vindo(a) ♥
-----
Opcoes do labirinto:
-----
(1) Carregar novo arquivo de dados.
(2) Processar e exibir resposta.
(3) Gerar um labirinto de teste (Extra).
(4) Observar caminho feito pelo estudante (Extra).
(5 ou qualquer outro caracter) Sair do programa.
-----

Digite um numero: 3

Digite a quantidade de linhas: 10
Digite a quantidade de colunas: 10
Digite a quantidade de chaves: 2
Digite a quantidade de portas: 5
Digite a quantidade de chaves no caminho: 2
Digite o nível de dificuldade (1 a 3 - fácil para difícil): 3
Digite a quantidade de portais (0 ou 1): 0
Digite o nome do arquivo que você deseja gerar. (Exemplo: 'teste.txt'): teste3.txt
Labirinto gerado e salvo no arquivo 'teste3.txt'
Pressione Enter para continuar...

```

```

TP1 - PAA > ≡ teste3.txt
1 | 10 10 2
2 | 1111111211
3 | 1112112122
4 | 1223221123
5 | 1211111111
6 | 2112112321
7 | 1121111211
8 | 1112232211
9 | 1212321121
10 | 1212141241
11 | 2212222202

```

Imagem 14 - Teste 3

2.3.3 - Células Amarelas

Para a implementação das células amarelas no labirinto, as quais indicam a presença de uma chave no local que pode ser recolhida e utilizada para abrir portas, foi necessário adicionar algumas verificações no código.

Cada chave é representada pelo valor **4** no labirinto. Durante a movimentação, a função **movimenta_estudante** é responsável por verificar se a posição analisada contém uma chave, e, caso sim, a variável local criada **novas_chaves** é incrementada, refletindo a quantidade total de chaves que o estudante possui.

Para garantir que as chaves possam ser reutilizadas caso o estudante precise voltar ao caminho anterior, foi implementada uma lógica de 'devolução' das chaves, que

utiliza a recursividade. Dessa forma, assim como quando o estudante passa por uma célula amarela, a variável auxiliar **novas_chaves** é incrementada. Caso o movimento seja inválido, a recursividade é empregada para 'soltar' a chave, já que a variável local passada como parâmetro na próxima chamada recursiva restaura automaticamente o valor anterior.

Além disso, a verificação de acessibilidade das células é feita pelo procedimento **posicaoValida**, que garante que o estudante só se mova para células acessíveis. Para as células amarelas, é adicionada uma nova análise, de modo que, caso a posição seja representada pelo valor 4, a função retorna 1, indicando uma posição válida.

2.3.4 - Criação de Portal

Na função **movimenta_estudante**, há um trecho de código responsável por implementar a lógica de teletransporte no labirinto com portais, representados pelos valores 5 (entrada) e 6 (saída). Quando o estudante chega a uma célula com valor 5, o programa busca, em toda a matriz, a célula correspondente com valor 6. Após localizar essa célula, o estudante é "teletransportado" para ela, e o algoritmo continua a exploração a partir dessa nova posição.

É válido destacar que uma característica importante dessa funcionalidade é a regra de movimentação entre os portais. Quando o estudante está na célula 6 (saída do portal), essa célula é tratada como uma parede, exceto se, ao explorar as células vizinhas ao 6, nenhuma solução for encontrada. Neste caso, o algoritmo é construído de forma a permitir com que o estudante volte ao portal de entrada (5), reiniciando a busca a partir dessa posição.

2.3.5 - Gráfico

Para implementação desta tarefa extra em que se deve plotar um gráfico de **Complexidade** vs **Tempo** para os casos a serem testados, tem-se o cálculo do tempo de processamento de cada um dos labirintos com o auxílio da biblioteca <time.h>, em que se criam variáveis ao longo do Main que contabilizam o tempo de execução de determinados trechos do código - que, nesse caso, estariam nas funções de manipulação dos labirintos. Em seguida, a equipe optou por conferir se os labirintos processados em questão possuem tamanhos quadrados - pois linhas e colunas diferentes atrapalhariam a análise de complexidade dos gráficos gerados e causariam certa desordem. Caso o labirinto seja de ordem quadrada, os dados sobre o seu tamanho e o

tempo de execução para processá-lo são inseridos em um arquivo de saída que será utilizado em um código da linguagem Python que gerará o gráfico correspondente.

Com relação à plotagem do gráfico no código em Python, é importante ressaltar que a biblioteca **matplotlib.pyplot** foi utilizada pelo grupo - sendo que essa deve estar previamente **instalada** no computador, juntamente com a linguagem. Seguindo essa mesma linha, o arquivo contendo as informações do labirinto é lido e estas são adicionadas em vetores de tamanho e tempo e, logo, com o suporte das funções **plot()**, **xlabel()**, **ylabel()**, **title()** e **show()** é definido o plano cartesiano e são criados e mostrados os respectivos gráficos com suas particularidades. Sendo assim, para gerar-lo é necessário executar o arquivo **grafico.py**, navegando até o diretório **Grafico**, onde ele está localizado. Ou seja, se o usuário estiver situado dentro da pasta do presente trabalho é necessário executar o seguinte comando “**python3 Grafico/grafico.py**” no terminal.

Antes de apresentarmos os gráficos resultantes dos testes anteriormente gerados, é importante ressaltar que o tamanho do gráfico não é o fator determinante para definição da complexidade de um labirinto, pois demais aspectos como a sua dificuldade, quantidade de paredes, o fato dele possuir saída ou não e etc são relevantes. Para ilustrar isso, geramos 10 labirintos aleatórios com tamanhos variando de 10 a 100. Em um dos gráficos, o labirinto de tamanho 30 apresenta um tempo de resolução significativamente maior em comparação aos demais, justamente porque não possui saída, o que aumenta consideravelmente a complexidade. Em outro teste, o gráfico segue uma linha de crescimento padrão, evidenciando que, quando o labirinto possui uma estrutura equilibrada e uma saída, o tempo de resolução tende a crescer de forma mais previsível com o aumento do tamanho.

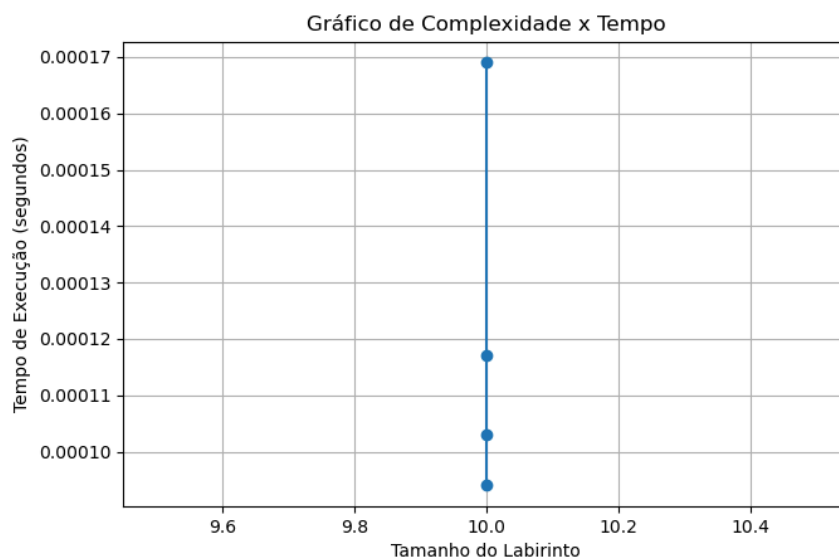


Imagem 15 - Gráficos dos testes anteriores

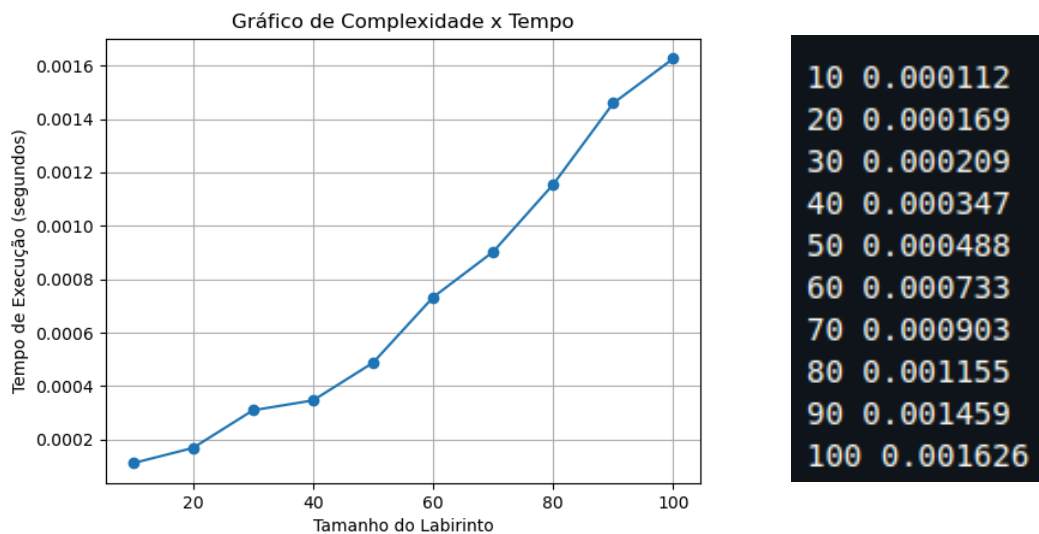


Imagem 16 - Labirintos gerados aleatoriamente

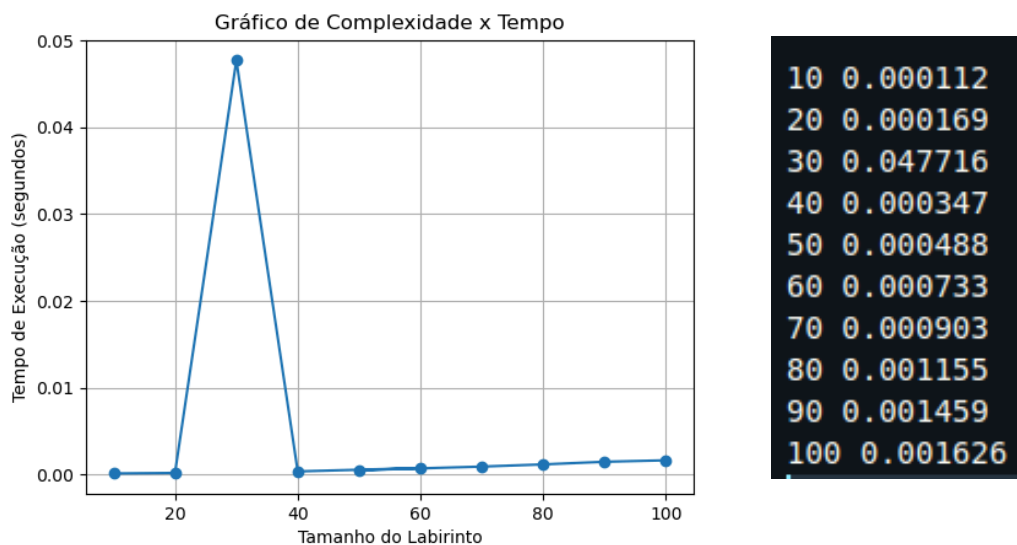


Imagem 17 - Labirintos gerados aleatoriamente (tamanho 30 sem saída)

3. Compilação e Execução

Para a implementação deste trabalho, foi criado um **Makefile** para facilitar o processo de compilação e execução do programa. Para executar, o usuário deve abrir um terminal, navegar até o diretório do projeto e executar o comando **make**, que compila todos os arquivos necessários e gera o executável. Vale ressaltar que todo o desenvolvimento e testes foram realizados no **Visual Studio Code** em um ambiente **Linux**.

4. Resultados

Com o pleno funcionamento do código do trabalho prático, a equipe realizou uma série de testes para análise do desempenho do algoritmo de backtracking implementado, o que permitiu uma avaliação detalhada da estratégia utilizada pelo estudante na resolução do labirinto e da eficiência na busca pela saída.

A seguir, tem-se alguns exemplos de resultados obtidos pelo algoritmo construído pela equipe, a começar pelo labirinto utilizado de exemplo na especificação, seguido dos demais testes que foram gerados no extra implementado “**Labirintos de Teste**”. Todos os labirintos apresentam também o caminho percorrido pelo estudante (tanto em texto, quanto visualmente), a quantidade de movimentos realizados e a coluna em que ele chegou.

```

Processando labirinto...

Caminho do estudante:

Linha: 9 Coluna: 4
Linha: 8 Coluna: 4
Linha: 8 Coluna: 3
Linha: 8 Coluna: 2
Linha: 8 Coluna: 1
Linha: 8 Coluna: 0
Linha: 7 Coluna: 0
Linha: 6 Coluna: 0
Linha: 5 Coluna: 0
Linha: 4 Coluna: 0
Linha: 3 Coluna: 0
Linha: 3 Coluna: 1
Linha: 3 Coluna: 2
Linha: 3 Coluna: 3
Linha: 3 Coluna: 4
Linha: 3 Coluna: 5
Linha: 2 Coluna: 5
Linha: 1 Coluna: 5
Linha: 0 Coluna: 5

0 estudante se movimentou 19 vezes e chegou na coluna 5 da primeira linha

1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2 2 2
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
2 2 2 2 2 2 1 1 1 1
1 1 1 1 1 2 1 1 1 1
2 2 2 2 2 2 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1

```

Imagem 18 - Resultado do Labirinto Teste da Especificação

```

Processando labirinto...

Caminho do estudante:

Linha: 9 Coluna: 8
Linha: 8 Coluna: 8
Linha: 8 Coluna: 7
Linha: 9 Coluna: 7
Linha: 9 Coluna: 6
Linha: 9 Coluna: 5
Linha: 8 Coluna: 5
Linha: 7 Coluna: 5
Linha: 6 Coluna: 5
Linha: 6 Coluna: 6
Linha: 5 Coluna: 6
Linha: 4 Coluna: 6
Linha: 4 Coluna: 7
Linha: 4 Coluna: 8
Linha: 5 Coluna: 8
Linha: 5 Coluna: 9
Linha: 4 Coluna: 9
Linha: 3 Coluna: 9
Linha: 3 Coluna: 8
Linha: 2 Coluna: 8
Linha: 1 Coluna: 8
Linha: 0 Coluna: 8

0 estudante se movimentou 126 vezes e chegou na coluna 8 da primeira linha

1 1 1 2 1 2 1 2 1
2 1 1 2 2 3 2 2 2
1 1 2 1 1 1 1 2 2
1 2 1 2 2 1 2 1
1 1 1 1 1 2
1 1 1 2 2 2 2
1 1 1 2 1 1 4 1
2 2 2 3 2 1 2 2 1
2 2 1 1 2 2 1
1 1 1 1 1 2

Chamadas recursivas: 126
Nível máximo de recursividade: 22

```

Imagem 19 - Resultado do Labirinto teste1.txt

```

Processando labirinto...

Caminho do estudante:

Linha: 9 Coluna: 8
Linha: 8 Coluna: 8
Linha: 8 Coluna: 7
Linha: 9 Coluna: 7
Linha: 9 Coluna: 6
Linha: 9 Coluna: 5
Linha: 8 Coluna: 5
Linha: 7 Coluna: 5
Linha: 6 Coluna: 5
Linha: 6 Coluna: 6
Linha: 5 Coluna: 6
Linha: 4 Coluna: 6
Linha: 4 Coluna: 7
Linha: 4 Coluna: 8
Linha: 5 Coluna: 8
Linha: 5 Coluna: 9
Linha: 4 Coluna: 9
Linha: 3 Coluna: 9
Linha: 3 Coluna: 8
Linha: 2 Coluna: 8
Linha: 1 Coluna: 8
Linha: 0 Coluna: 8

0 estudante se movimentou 126 vezes e chegou na coluna 8 da primeira linha

1 1 1 2 1 2 1 2 1
2 1 1 2 2 3 2 2 2
1 1 2 1 1 1 1 2 2
1 2 1 2 2 1 2 1
1 1 1 1 1 2
1 1 1 2 2 2 2
1 1 1 2 1 1 4 1
2 2 2 3 2 1 2 2 1
2 2 1 1 2 2 1
1 1 1 1 1 2

Chamadas recursivas: 126
Nível máximo de recursividade: 22

```

Imagem 20 - Resultado do Labirinto teste2.txt

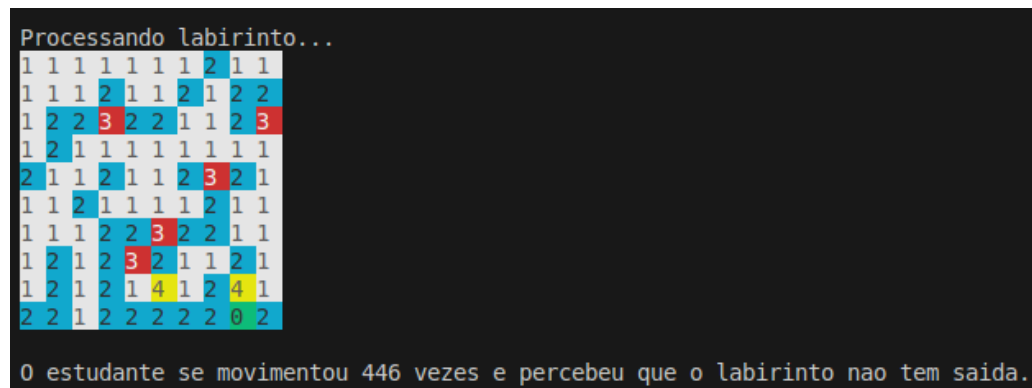


Imagem 21 - Resultado do Labirinto teste3.txt

5. Conclusão

Por fim, infere-se, com a realização deste trabalho prático, que a implementação de um algoritmo para encontrar a saída de um dado labirinto com o auxílio de backtracking foi um sucesso e, apesar de uma certa dificuldade da equipe em lidar com alguns erros gerados pelo paradigma utilizado, os objetivos estabelecidos na especificação foram compreendidos e atendidos corretamente pelo grupo.

6. Referências

- [1] **Github**. Disponível em: <<https://github.com/melissaalanis/TP1-PAA>> Último acesso em: 2024.
- [2] **IDE Visual Studio Code**. Disponível em: <<https://code.visualstudio.com/>>.
- [3] **Extensão Live Share**. Disponível na IDE Visual Studio Code. <<https://visualstudio.microsoft.com/pt-br/services/live-share/>>.
- [4] **UVV. Capítulo 6: Estruturas de Dados**. Universidade Vila Velha. Disponível em: <https://disciplinas.uvv.br/assets/disciplinas/ed1/capitulo06.pdf>. Acesso em: 25 nov. 2024.