Universidade Federal de Ouro Preto - UFOP
Instituto de Ciências Exatas e Biológicas - ICEB
Departamento de Computação - DECOM
Ciência da Computação

Trabalho Prático - Ajudando um rato a sair do labirinto

BCC202 - Estrutura de Dados I

Alunos: Eduardo Silva e Niege Reis
Professor: Pedro Silva

Ouro Preto
15 de junho de 2023

Sumário

1 Introdução

Nesse trabalho, tivemos o desafio de ajudar um rato a sair do labirinto. Com base na proposta, foi solicitado o desenvolvimento de um programa em C que utilize a recursividade para encontrar o menor caminho até a saída de um labirinto. O labirinto possui uma posição inicial para o rato, e a saída está sempre localizada no canto inferior direito.

A abordagem adotada para resolver o problema consiste em considerar cada posição vazia do labirinto como uma possível rota para o rato. Foi implementado um algoritmo que inicia na posição inicial do rato e, para cada caminho possível, chama recursivamente a função Achar Saída. É importante destacar que durante a exploração do labirinto, é impedido que o rato volte por caminhos já percorridos anteriormente. Essa restrição é fundamental, pois caso o rato retorne por um caminho já visitado, o menor caminho até a saída não será encontrado.

Além disso, foi adicionada a funcionalidade de gerar um labirinto aleatório. Para isso, o programa requer apenas as dimensões do labirinto fornecidas pelo usuário, o que resulta na criação de um labirinto aleatório.

2 Implementação

Após compreender o problema proposto inicialmente e desenvolver um algoritmo capaz de encontrar o menor caminho para o rato, começamos a pensar nas estruturas necessárias e suas respectivas funções auxiliares que precisaríamos para a função "encontrarSaída". Com isso, chegamos às estruturas que representariam o Labirinto, a Posição, o Percurso e as Direções Disponíveis.

2.1 Estruturas (ou structs)

2.1.1 Labirinto

A struct "labirinto" tem o objetivo de armazenar informações sobre o tabuleiro do labirinto. Com uma matriz de caracteres "tabuleiro" que armazena os respectivos caracteres de cada posição do labirinto, (esses caracteres sendo "M" representando o rato, representando um possível caminho e ainda um "*"representando uma barreira, ou seja, o rato não conseguiria utrapassa-la), e dois inteiros, "qtdLinhas" que armazena a quantidade de linhas do tabuleiro, "qtdColunas" que armazena a quantidade de colunas do tabuleiro e um caractere "opcao" que representa a opção de saida do programa (podendo ser a listagem dos passos, ou ainda, o próprio caminho apresentado no labirinto), essa é escolhida pelo usuário.

```
struct labirinto {
    char **tabuleiro;
    int qtdLinhas;
    int qtdColunas;
    char opcao;
};
```

Código 1: Estrutura representando o labirinto.

2.1.2 Posição

A struct "posicao" tem o objetivo de armazenar informações sobre uma posição no tabuleiro. Nela possuimos dois inteiros, x representando uma linha e y representando uma coluna desse tabuleiro.

```
struct posicao {
    int x;
    int y;
};
```

Código 2: Estrutura representando a posição.

2.1.3 Percurso

A struct "percurso" representa um caminho percorrido dentro de um labirinto, contendo as informações "posicoes", sendo essa um vetor de Posicoes (mencionada no tópico 2.1.1), e o número de passos dados, que por consequência armazena a quantidade de elementos do vetor de posições.

```
struct percurso {
    Posicao *posicoes;
    int passos;
};
```

Código 3: Estrutura representando o percurso.

2.1.4 Direções Disponíveis

A struct "direcoesDisponiveis" permite armazenar informações sobre as direções possíveis a serem seguidas a partir de uma posição específica no labirinto, fornecendo um vetor de direções (um enumerador mencionado no tópico 2.2.1), e o número total de direções disponíveis "qtdDirecoes".

```
struct direcoesDisponiveis {
    Direcao *direcoes;
    int qtdDirecoes;
};
```

Código 4: Estrutura representando as direções disponíveis.

2.2 Enumeradores (ou Enum's)

Os enum's permite melhorar a legibilidade e manutenabilidade do código, tendo em vista que conseguimos dar nomes a inteiros, ou seja, supondo que temos uma lista de gêneros, masculino e feminino, conseguimos tratar masculino como 0 e feminino como 1, porém, no código referenciariamos como masculino e feminino, facilitando a leitura e a semântica do código.

2.2.1 Direção

Dado que no nosso problema o rato só pode andar para cima, baixo, esquerda ou direita, criamos o enum "direcao" representando justamente as possíveis direções de movimentos, fornecendo uma forma conveniente de representar e trabalhar com as direções possíveis na códificação do problema.

```
enum direcao { CIMA, BAIXO, ESQUERDA, DIREITA };
```

Código 5: Enumerador de direções disponíveis para movimento.

2.3 Funções

As funções são extremamente importantes na programação, tendo em vista que com elas conseguimos evitar duplicidade de código, centralização das operações entre outras. Além, de fazer um papel muito importante quando estamos trabalhando com tipos abstratos de dados, considerando que os tipos tem funções relacionadas. Dado isto implementamos algumas funções para poder auxiliar na solução do problema:

2.3.1 Alocar labirinto

A função "alocarLabirinto" é responsável somente por alocar um labirinto e retornar a que a chamou.

```
Labirinto *alocarLabirinto() {
    Labirinto *labirinto = (Labirinto *)malloc((sizeof(Labirinto)));
    return labirinto;
}
```

Código 6: Função responsável por aloca labirinto.

2.3.2 Desalocar labirinto

A função "desalocarLabirinto" é responsável por liberar a memória alocada para um tipo Labirinto. Ela realiza as seguintes etapas: percorre cada linha do tabuleiro do labirinto, utiliza a função "free" para liberar a memória alocada para cada linha do tabuleiro, em seguida, libera a memória alocada para a matriz de tabuleiro em si, usando novamente a função "free" e por fim, libera a memória alocada para um tipo Labirinto, usando a função "free".

```
void desalocarLabirinto(Labirinto **labirinto) {
    for (int i = 0; i < (*labirinto)->qtdLinhas; i++) {
        free((*labirinto)->tabuleiro[i]);
    }
    free((*labirinto)->tabuleiro);
    free(*labirinto);
}
```

Código 7: Função responsável por desalocar um labirinto.

2.3.3 Desalocar direções disponíveis

A função "desalocarDirecoesDisponiveis" tem como objetivo liberar a memória alocada para um objeto do tipo DirecoesDisponiveis. Ela executa as seguintes etapas, utiliza a função "free" para liberar a memória alocada para o vetor de direções dentro do tipo DirecoesDisponiveis e em seguida, utiliza a função "free" para liberar a memória alocada para o tipo DirecoesDisponiveis.

```
void desalocarDirecoesDisponiveis(DirecoesDisponiveis **direcoesDisponiveis) {
    free((*direcoesDisponiveis)->direcoes);
    free(*direcoesDisponiveis);
}
```

Código 8: Função responsável por desalocar direções disponíveis.

2.3.4 Ler labirinto

A função "leLabirinto" é responsável por ler as informações de um labirinto a partir da entrada padrão (teclado) e armazená-las em uma estrutura do tipo Labirinto. Ela executa as seguintes etapas: lê a quantidade de linhas, colunas e a opção do labirinto e armazena-as no labirinto. Ela ainda aloca a memória necessária para o tabuleiro do labirinto e lê os caracteres do tabuleiro e armazena-os no labirinto, lê um caractere extra (espaço em branco ou nova linha) após cada linha do tabuleiro para descartar o caractere de quebra de linha.

```
void leLabirinto(Labirinto *labirinto) {
    scanf("%d%d", &labirinto->qtdLinhas, &labirinto->qtdColunas);
    scanf(" %c ", &labirinto->opcao);

    labirinto->tabuleiro = (char **)malloc(labirinto->qtdLinhas * (sizeof(char
        *)));
    for (int i = 0; i < labirinto->qtdLinhas; i++) {
        labirinto->tabuleiro[i] = (char *)malloc(labirinto->qtdColunas * (
            sizeof(char)));
    }
    for (int i = 0; i < labirinto->qtdLinhas; i++) {
        for (int j = 0; j < labirinto->qtdColunas; j++) {
            scanf("%c", &labirinto->tabuleiro[i][j]);
        }
        char aux;
        scanf("%c", &aux);
    }
```

```
}
```

Código 9: Função responsável por ler os dados labirinto.

2.3.5 Imprime labirinto

A função "imprimeLabirinto" é responsável por imprimir na saída padrão (console) o conteúdo do tabuleiro armazenado na estrutura do tipo labirinto.

```
void imprimeLabirinto(Labirinto *labirinto) {
    for (int i = 0; i < labirinto->qtdLinhas; i++) {
        for (int j = 0; j < labirinto->qtdColunas; j++) {
            printf("%c", labirinto->tabuleiro[i][j]);
        }
        printf("\n");
    }
}
```

Código 10: Função imprime o tabuleiro do labirinto.

2.3.6 Posição disponível

Ela realiza as seguintes verificações:

- Verifica se o caractere correspondente à posição no tabuleiro do labirinto é um espaço em branco (' '), indicando que a posição está vazia e pode ser percorrida.
- Verifica se a coordenada x da posição está dentro dos limites do labirinto, ou seja, se está entre 0 e o número de linhas - 1.
- Verifica se a coordenada y da posição está dentro dos limites do labirinto, ou seja, se está entre 0 e o número de colunas - 1.

```
bool posicaoDisponivel(Labirinto *labirinto, Posicao *posicao) {
    return (labirinto->tabuleiro[posicao->x][posicao->y] == ' ') &&
           (posicao->x >= 0 && posicao->x <= labirinto->qtdLinhas - 1) &&
           (posicao->y >= 0 && posicao->y <= labirinto->qtdColunas - 1);
}
```

Código 11: Função responsável por verificar se uma determinada posição dentro do labirinto está disponível para ser percorrida.

A função retorna true se todas as condições forem satisfeitas, indicando que a posição está disponível. Caso contrário, retorna false.

2.3.7 Somar X Y na posição

A função "somarXYposicao" recebe uma posição, um valor inteiro x e um valor inteiro y. Ela cria e retorna uma nova posição somando os valores x e y às coordenadas x e y da posição original. Essa função é útil quando se deseja obter uma nova posição a partir da posição atual, adicionando valores a suas coordenadas.

```
Posicao *somarXYposicao(Posicao posicao, int x, int y) {
    Posicao *novaPosicao = criarPosicao(posicao.x + x, posicao.y + y);
    return novaPosicao;
}
```

Código 12: Função responsável por gerar uma nova posição a partir da posição atual, adicionando valores a suas coordenadas de entrada.

2.3.8 Verifica posicao no percurso

A função "verificaPosicaoNoPercurso", similar a função mencionada acima, verifica se uma determinada posição, representada pelas coordenadas x e y, já foi percorrida durante um percurso registrado na estrutura Percurso.

```
bool verificaPosicaoNoPercurso(Percurso *percurso, int x, int y) {
    for (int i = 0; i < percurso->passos; i++) {
        if (percurso->posicoes[i].x == x && percurso->posicoes[i].y == y) {
            return true;
        }
    }
    return false;
}
```

Código 13: Função responsável por verificar se uma posição específica já foi percorrida durante um percurso.

2.3.9 Verificação para pegar direções disponíveis

Função auxiliar a função de pegar as direções disponíveis, criada com o objetivo de reduzir códigos duplicados.

```
bool verificacaoParaPegarDirecoesPossiveis(Labirinto *labirinto, Posicao *
    posicao, Direcao origem, Direcao naoPodeVimDe,
                                           Percurso *percurso) {
    bool novaPosicaoDisponivel = posicaoDisponivel(labirinto, posicao);
    bool naoVeioDe = origem != naoPodeVimDe;
    bool posicaoNaoEstaNoPercurso = !verificaPosicaoNoPercurso(percurso,
        posicao->x, posicao->y);
    return novaPosicaoDisponivel && naoVeioDe && posicaoNaoEstaNoPercurso;
}
```

Código 14: Função auxiliar a função pegar direções possíveis.

2.3.10 Pegar direcoes possíveis

O código acima implementa a função "pegarDirecoesPossiveis" que tem como objetivo determinar as direções disponíveis para o rato se mover a partir de uma determinada posição no labirinto.

Em seguida, são calculadas as posições adjacentes ao rato nas direções correspondentes. Para cada uma dessas posições, é verificado se é possível o movimento, levando em consideração as seguintes condições:

- A posição adjacente está disponível no labirinto
- A direção de origem não é oposta à direção do movimento atual
- A posição adjacente não está presente no percurso atual do rato

Se todas essas condições forem atendidas, para uma determinada direção, então retornará como true, sendo possível a movimentação para o lado. E então é retornado as direções possíveis (estrutura mencionada no tópico 2.1.4)

```
DirecoesDisponiveis *pegarDirecoesPossiveis(Labirinto *labirinto, Posicao
    posicaoRato, Direcao direcaoOrigem,  Percurso *percurso) {
    DirecoesDisponiveis *direcoesDisponiveis = (DirecoesDisponiveis*) malloc(
        sizeof(DirecoesDisponiveis));
    bool podeAndarDireita = false, podeAndarEsquerda = false, podeAndarCima =
        false, podeAndarBaixo = false;

    Posicao *posicaoADireita = somarXYposicao(posicaoRato, 0, 1);
    if(verificacaoParaPegarDirecoesPossiveis(labirinto, posicaoADireita,
        direcaoOrigem, ESQUERDA, percurso))
        podeAndarDireita = true;
    desalocarPosicao(&posicaoADireita);

    Posicao *posicaoAEsquerda = somarXYposicao(posicaoRato, 0, -1);
    if (verificacaoParaPegarDirecoesPossiveis(labirinto, posicaoAEsquerda,
        direcaoOrigem, DIREITA, percurso))
        podeAndarEsquerda = true;
    desalocarPosicao(&posicaoAEsquerda);

    Posicao *posicaoACima = somarXYposicao(posicaoRato, -1, 0);
    if (verificacaoParaPegarDirecoesPossiveis(labirinto, posicaoACima,
        direcaoOrigem, BAIXO, percurso))
        podeAndarCima = true;
    desalocarPosicao(&posicaoACima);

    Posicao *posicaoABaixo = somarXYposicao(posicaoRato, 1, 0);
    if (verificacaoParaPegarDirecoesPossiveis(labirinto, posicaoABaixo,
        direcaoOrigem, CIMA, percurso))
        podeAndarBaixo = true;
    desalocarPosicao(&posicaoABaixo);

    direcoesDisponiveis->qtdDirecoes = podeAndarDireita + podeAndarEsquerda +
        podeAndarCima + podeAndarBaixo;
    direcoesDisponiveis->direcoes = (Direcao *)malloc((direcoesDisponiveis->
        qtdDirecoes) * sizeof(Direcao));

    int contadorIndiceAtual = 0;
    if (podeAndarBaixo)
        direcoesDisponiveis->direcoes[contadorIndiceAtual++] = BAIXO;
    if (podeAndarDireita)
        direcoesDisponiveis->direcoes[contadorIndiceAtual++] = DIREITA;
    if (podeAndarCima)
        direcoesDisponiveis->direcoes[contadorIndiceAtual++] = CIMA;
    if (podeAndarEsquerda)
        direcoesDisponiveis->direcoes[contadorIndiceAtual++] = ESQUERDA;

    return direcoesDisponiveis;
}
```

Código 15: Função responsável por determinar as direções possíveis que o rato pode seguir em uma determinada posição no labirinto.

2.3.11 Copiar percurso

A função "copiarPercurso" tem como objetivo copiar um percurso de uma estrutura de percurso de origem para uma estrutura de percurso de destino.

```
void copiarPercurso(Percurso *destino, Percurso *origem) {
    destino->passos = origem->passos;
    for (int i = 0; i < origem->passos; i++) {
        destino->posicoes[i].x = origem->posicoes[i].x;
        destino->posicoes[i].y = origem->posicoes[i].y;
    }
}
```

Código 16: Função responsável por copiar um percurso de uma variável de uma para outra.

2.3.12 Criar posição

A função "criarPosicao" é responsável por retornar uma posição alocada dinamicamente.

```
Posicao* criarPosicao(int x, int y) {
    Posicaoposicao = (Posicao) malloc(sizeof(Posicao));
    posicao->x = x;
    posicao->y = y;
    return posicao;
}
```

Código 17: Função responsável por criar uma posição.

2.3.13 Imprime percurso

A função "imprimePercurso" é responsável por imprimir as posições de um percurso na ordem em que foram registradas.

```
void imprimePercurso(Percurso *percurso) {
    for (int i = 0; i < percurso->passos; i++) {
        printf("(%d, %d)\n", percurso->posicoes[i].y, percurso->posicoes[i].x)
            ;
    }
}
```

Código 18: Função responsável por imprimir a lista de de percurso.

2.3.14 Copiar posição

A função "copiarPosicao" é responsável por gerar uma cópia (alocada dinamicamente) de uma posição ou seja, alocar e retornar uma nova posição com base em uma outra posição.

```
Posicao *copiarPosicao(Posicao *posicao) {
    return criarPosicao(posicao->x, posicao->y);
}
```

Código 19: Função responsável por copiar uma posição (alocada dinamicamente).

2.3.15 Desalocar posição

A função "desalocarPosicao" é responsável por desalocar uma posição

```
void desalocarPosicao(Posicao **posicao) {
    free(*posicao);
}
```

Código 20: Função responsável por desalocar uma posição.

2.3.16 Desalocar percurso

A função "desalocarPercurso" é responsável por desalocar um percurso e suas posições.

```
void desalocarPercurso(Percurso **percurso) {
    free((*percurso)->posicoes);
    free(*percurso);
}
```

Código 21: Função responsável por desalocar um percurso e suas posições.

2.3.17 Imprime saida

A função "imprimeSaida" tem como objetivo imprimir a saída do programa, exibindo informações sobre o menor percurso encontrado no labirinto.

Um dos requisitos do programa é a saida selecionada pelo usuário, caso ele digite o caractere 'C' deve ser impresso as posições no formato (X, Y) do percurso, caso ele digite 'P' deve ser impresso o percurso no labirinto.

Dessa forma, a função "imprimeSaida" exibe os dados do menor caminho conforme especificado pelo usuário. Isso permite visualizar e compreender o resultado do algoritmo de busca do menor caminho no labirinto.

```
void imprimeSaida(Labirinto *labirinto, Percurso *menorPercurso) {
    printf("%d\n", menorPercurso->passos);
    if (labirinto->opcao == 'c') {
        imprimePercurso(menorPercurso);
    } else {
        imprimeLabirintoEpercurso(labirinto, menorPercurso);
    }
}
```

Código 22: Função responsável por imprimir a saida esperada pelo programa.

2.3.18 Achar saída

A função "acharSaida" é responsável por encontrar a saída do labirinto através de um algoritmo de busca recursiva. A mesma recebe vários parâmetros sendo eles:

- Um ponteiro para a estrutura labirinto, que representa o labirinto a ser percorrido.
- Um ponteiro para a estrutura posicao, que indica a posição atual do rato no labirinto.
- Um inteiro direcao origem, que representa a direção de onde o rato veio (usado pelo enumerador Direcao mencionado no tópico 2.2.1).
- Um ponteiro para um inteiro representado menor caminho, que armazena o tamanho do menor caminho encontrado até o momento.
- Um ponteiro para a estrutura percurso, que armazena o menor percurso encontrado até o momento.
- Um ponteiro para a estrutura percurso, que armazena o percurso atual sendo explorado.

Dentro da função, ocorre o seguinte processo:

- É criado um novo percurso atual utilizando a função alocarPercurso e copiado o conteúdo do percurso atual para ele, utilizando a função copiarPercurso. Isso é necessário para criar uma cópia separada do percurso atual para cada ramo da recursão.
- É verificado se o rato está na última coluna do labirinto e na penultima linha. Se isso for verdade, significa que a saída foi encontrada. Nesse caso, são realizadas as seguintes ações:

- Verifica-se se o número de passos do percurso atual é menor do que o menor caminho encontrado até o momento. Se for, atualiza-se o menor caminho com o valor de passos e copia-se o percurso atual para o menor percurso, utilizando a função auxiliar copiar percurso (mencionada no tópico 2.3.11). Isso garante que sempre armazenamos o menor caminho encontrado.

```c
void acharSaida(Labirinto *labirinto, Posicao *posicaoRato, int direcaoOrigem,
     int *menorCaminho, Percurso *menorPercurso, Percurso *percursoAtual) {
    Percurso *novoPercursoAtual = alocarPercurso(labirinto);
    copiarPercurso(novoPercursoAtual, percursoAtual);

    if (posicaoRato->y + 1 == labirinto->qtdColunas) {
        if (percursoAtual->passos < *menorCaminho) {
            *menorCaminho = percursoAtual->passos;
            copiarPercurso(menorPercurso, percursoAtual);
        }
        desalocarPercurso(&novoPercursoAtual);
        return;
    }

    DirecoesDisponiveis *direcoesPossiveis =
        pegarDirecoesPossiveis(labirinto, *posicaoRato, direcaoOrigem,
            percursoAtual);

    novoPercursoAtual->passos++;
    Posicao *novaPosicao = copiarPosicao(posicaoRato);
    Posicao *posicaoAtualInicial = copiarPosicao(posicaoRato);

    for (int i = 0; i < direcoesPossiveis->qtdDirecoes; i++) {
        Direcao direcao = direcoesPossiveis->direcoes[i];
        moverPosicaoPelaDirecao(novaPosicao, direcao);
        novoPercursoAtual->posicoes[novoPercursoAtual->passos - 1] = *
            novaPosicao;

        if (direcao == BAIXO)
            acharSaida(labirinto, novaPosicao, BAIXO, menorCaminho,
                menorPercurso, novoPercursoAtual);
        if (direcao == DIREITA)
            acharSaida(labirinto, novaPosicao, DIREITA, menorCaminho,
                menorPercurso, novoPercursoAtual);
        if (direcao == ESQUERDA)
            acharSaida(labirinto, novaPosicao, ESQUERDA, menorCaminho,
                menorPercurso, novoPercursoAtual);
        if (direcao == CIMA)
            acharSaida(labirinto, novaPosicao, CIMA, menorCaminho,
                menorPercurso, novoPercursoAtual);

        moverPosicaoParaPosicao(novaPosicao, posicaoAtualInicial);
    }

    desalocarPosicao(&posicaoAtualInicial);
    desalocarPosicao(&novaPosicao);
    desalocarPercurso(&novoPercursoAtual);
    desalocarDirecoesDisponiveis(&direcoesPossiveis);
}
```

Código 23: Função responsável por encontrar a saída do labirinto através de um algoritmo de busca recursiva.

2.3.19 Imprime labirinto e percurso

A função "imprimeLabirintoEpercurso" tem a finalidade de imprimir o labirinto juntamente com o percurso realizado pelo rato.

A função percorre as linhas e colunas do labirinto e, para cada posição, verifica se essa posição faz parte do percurso. Se fizer, imprime um ponto ("."); caso contrário, imprime o caractere correspondente à posição no labirinto.

```
void imprimeLabirintoEpercurso(Labirinto *labirinto, Percurso *percurso) {
    for (int i = 0; i < labirinto->qtdLinhas; i++) {
        for (int j = 0; j < labirinto->qtdColunas; j++) {
            if (verificaPosicaoNoPercurso(percurso, i, j))
                printf(".");
            else
                printf("%c", labirinto->tabuleiro[i][j]);
        }
        printf("\n");
    }
}
```

Código 24: Função responsável por imprimir o labirinto e o percurso.

2.3.20 Encontrar posição do rato

A função "encontrarPosicaoDoRato" tem a finalidade de encontrar a posição atual do rato no labirinto. Ou seja, considerando que o rato no labirinto é a letra M, percorremos a matriz tabuleiro do labirinto e retornamos a posição, caso não seja encontrado, é retornado a posição (-1, -1).

```
Posicao *encontrarPosicaoDoRato(Labirinto *labirinto) {
    Posicao *posicao = (Posicao *)malloc(sizeof(Posicao));
    for (int i = 0; i < labirinto->qtdLinhas; i++) {
        for (int j = 0; j < labirinto->qtdColunas; j++) {
            if (labirinto->tabuleiro[i][j] == 'M') {
                posicao->x = i;
                posicao->y = j;
                return posicao;
            }
        }
    }
    posicao->x = -1;
    posicao->y = -1;
    return posicao;
}
```

Código 25: Função responsável por encontrar a posição atual do rato no labirinto.

2.3.21 Alocar percurso.

A função "alocarPercurso" aloca uma estrutura do tipo percurso (mencionada no tópico 2.1.3) inicializa suas posições proporcionalmente a quantidade de linhas e colunas do labirinto e inicializa a quantidade de passos dados no percurso com 0.

```
Percurso *alocarPercurso(Labirinto *labirinto) {
    Percurso *percurso = (Percurso *)malloc(sizeof(Percurso));
    percurso->posicoes = (Posicao *)malloc((labirinto->qtdLinhas * labirinto->
        qtdColunas * 10) * sizeof(Posicao));
    percurso->passos = 0;
    return percurso;
}
```

Código 26: Função que retorna o número de colunas do labirinto

2.3.22 Imprime direção

A função "imprimirDirecao" foi desenvolvida para auxiliar na depuração do código.

Portanto, a função imprimirDirecao é responsável por imprimir a representação textual de uma direção, com base no valor do parâmetro direcao. Por exemplo, se direcao for igual a CIMA, a função imprimirá "CIMA".

```
void imprimirDirecao(Direcao direcao) {
    switch (direcao) {
    case CIMA:
        printf("CIMA ");
        break;
    case BAIXO:
        printf("BAIXO ");
        break;
    case DIREITA:
        printf("DIREITA ");
        break;
    case ESQUERDA:
        printf("ESQUERDA ");
        break;
    }
}
```

Código 27: Função que imprime a representação textual da direção correspondente

2.3.23 Mover posição pela direção

A função "moverPosicaoPelaDirecao" recebe um ponteiro para uma struct Posicao chamada posicao e uma direção representada por um valor do tipo Direcao. A função atualiza as coordenadas da posição de acordo com a direção especificada.

Dessa forma, a função "moverPosicaoPelaDirecao" permite mover a posição atual para uma nova posição com base na direção especificada.

```
void moverPosicaoPelaDirecao(Posicao *posicao, Direcao direcao) {
    switch (direcao) {
    case CIMA:
        posicao->x--;
        break;
    case BAIXO:
        posicao->x++;
        break;
    case DIREITA:
        posicao->y++;
        break;
    case ESQUERDA:
        posicao->y--;
        break;
    }
}
```

Código 28: Função responsável por mover a posição atual para uma nova posição com base na direção especificada.

2.3.24 Posição está no percurso

A função "posicaoEstaNoPercurso" verifica se uma determinada posição está presente em um percurso específico. Ela recebe uma posição, uma direção e um ponteiro para um percurso (Percurso *percurso).

Resumindo, a função "posicaoEstaNoPercurso" verifica se uma posição, obtida ao mover-se em uma determinada direção a partir de uma posição original, está presente em um percurso. Ela é útil para evitar que o rato retorne a posições anteriores durante a busca de uma saída em um labirinto, garantindo que ele siga um caminho único.

```
bool posicaoEstaNoPercurso(Posicao posicao, Direcao direcao, Percurso *
    percurso) {
    moverPosicaoPelaDirecao(&posicao, direcao);
    for (int i = 0; i < percurso->passos; i++) {
        Posicao posicaoAtual = percurso->posicoes[i];
        if (posicaoAtual.x == posicao.x && posicaoAtual.y == posicao.y)
            return true;
    }
    return false;
}
```

Código 29: Função responsável por verificar se uma posição, obtida ao mover-se em uma determinada direção a partir de uma posição original, está presente em um percurso.

2.3.25 Alocar direcoes disponiveis

A função "alocarDirecoesDisponiveis" é responsável por alocar memória para uma estrutura do tipo direcoes disponiveis e inicializá-la. Ela retorna um ponteiro para a struct alocada.

Em resumo, a função "alocarDirecoesDisponiveis" cria e retorna uma struct DirecoesDisponiveis alocada dinamicamente, com um vetor de direções vazio. Essa estrutura é usada para armazenar as direções disponíveis em um determinado contexto, como as direções possíveis para movimento em um labirinto.

```
DirecoesDisponiveis *alocarDirecoesDisponiveis() {
    DirecoesDisponiveis *direcoesDisponiveis = (DirecoesDisponiveis *)malloc(
        sizeof(DirecoesDisponiveis));
    direcoesDisponiveis->direcoes = (Direcao *)malloc(sizeof(Direcao) * 4);
    direcoesDisponiveis->qtdDirecoes = 0;
    return direcoesDisponiveis;
}
```

Código 30: Função responsável por criar e retornar uma struct DirecoesDisponiveis alocada dinamicamente, com um array de direções vazio.

2.3.26 Inicializa tabuleiro

A função "inicializaTabuleiro" é responsável por inicializar o tabuleiro de um labirinto com o caractere '*'. Ela percorre todas as células do tabuleiro e atribui o caractere a cada uma delas.

Dessa forma, a função inicializaTabuleiro garante que todas as células do tabuleiro do labirinto estejam preenchidas com o caractere '*', indicando que todas as posições são paredes no início.

```
void incializaTabuleiro(Labirinto *labirinto) {
    for (int i = 0; i < labirinto->qtdLinhas; i++) {
        for (int j = 0; j < labirinto->qtdColunas; j++) {
            labirinto->tabuleiro[i][j] = '*';
        }
    }
}
```

Código 31: Função responsável por inicializar o tabuleiro de um labirinto com o caractere.

2.3.27 Esta na borda pela direcao

A função "estaNaBordaPelaDirecao" verifica se uma determinada direcao, a partir de uma posicao, está na borda do labirinto.

Dessa forma, a função "estaNaBordaPelaDirecao" verifica se uma posição, deslocada na direção especificada, está na borda do labirinto, considerando os offsets opcionais para maior flexibilidade na verificação.

```
bool estaNaBordaPelaDirecao(Labirinto *labirinto, Direcao direcao, Posicao
    posicao, int desvioX, int desvioY) {
    moverPosicaoPelaDirecao(&posicao, direcao);
    bool bordaEmX = posicao.x == 0 + desvioX || posicao.x >= labirinto->
        qtdLinhas - 1 - desvioX;
    bool bordaEmY = posicao.y == 0 + desvioY || posicao.y >= labirinto->
        qtdColunas - 1 - desvioY;
    return bordaEmX || bordaEmY;
}
```

Código 32: Função responsável por verificar se uma posição, deslocada na direção especificada está na borda do labirinto.

2.3.28 Pode remover para

A função "podeRemoverPara" verifica se é possível remover uma parede em uma determinada direção, na posição especificada do labirinto.

Dessa forma, a função "podeRemoverPara" verifica se é possível remover uma parede em uma determinada direção, a partir de posição especificada do labirinto, garantindo a consistência e integridade do labirinto resultante.

```
bool podeRemoverPara(Labirinto *labirinto, Direcao direcao, Posicao posicao) {
    bool cimaDireita = true, baixoEsquerda = true, cimaEsquerda = true,
        baixoDireita = true;

    if (posicao.x - 1 >= 0 && posicao.y - 1 >= 0)
        cimaEsquerda = labirinto->tabuleiro[posicao.x - 1][posicao.y - 1] == '
            *';
    if (posicao.x - 1 >= 0 && posicao.y + 1 <= labirinto->qtdColunas)
        cimaDireita = labirinto->tabuleiro[posicao.x - 1][posicao.y + 1] == '*
            ';
    if (posicao.x + 1 <= labirinto->qtdLinhas && posicao.y - 1 >= 0)
        baixoEsquerda = labirinto->tabuleiro[posicao.x + 1][posicao.y - 1] ==
            '*';
    if (posicao.x + 1 <= labirinto->qtdLinhas && posicao.y + 1 <= labirinto->
        qtdColunas)
        baixoDireita = labirinto->tabuleiro[posicao.x + 1][posicao.y + 1] == '
            *';

    bool aEsquerda = true, aDireita = true, aCima = true, aBaixo = true;
    if (posicao.y - 1 >= 0)
        aEsquerda = labirinto->tabuleiro[posicao.x][posicao.y - 1] == ' ';
    if (posicao.y + 1 <= labirinto->qtdColunas)
        aDireita = labirinto->tabuleiro[posicao.x][posicao.y + 1] == ' ';
    if (posicao.x - 1 >= 0)
        aCima = labirinto->tabuleiro[posicao.x - 1][posicao.y] == ' ';
    if (posicao.x - 1 <= labirinto->qtdLinhas)
        aBaixo = labirinto->tabuleiro[posicao.x + 1][posicao.y] == ' ';
    return (cimaDireita + cimaEsquerda + baixoEsquerda + baixoDireita >= 3) &&
           (aEsquerda + aDireita + aCima + aBaixo <= 1);
}
```

Código 33: Função responsável por verificar se é possível remover uma parede em uma determinada direção, na posição especificada do labirinto.

2.3.29 Pode mover para caminho vazio

A função "podeMoverParaCaminhoVazio" verifica se é possível mover-se para uma posição adjacente que contenha um caminho vazio no labirinto.

Dessa forma, a função "podeMoverParaCaminhoVazio" verifica se é possível mover-se para uma posição adjacente que contenha um caminho vazio no labirinto, permitindo a movimentação apenas em direções que não estejam bloqueadas por paredes.

```
bool podeMoverParaCaminhoVazio(Labirinto *labirinto, Direcao direcao, Posicao
    posicao) {
    moverPosicaoPelaDirecao(&posicao, direcao);

    if (posicao.x >= 0 && posicao.x <= labirinto->qtdColunas && posicao.y >= 0
        && posicao.y <= labirinto->qtdLinhas)
        return labirinto->tabuleiro[posicao.x][posicao.y] == ' ';

    return false;
}
```

Código 34: Função responsável por verificar se é possível mover-se para uma posição adjacente.

2.3.30 Contem valor no vetor

A função "contemValorNoVetor" verifica se um determinado valor está presente em um vetor de direções.

Ainda permite verificar se um determinado valor está presente em um vetor de direções, sendo útil para realizar verificações de existência de um valor específico em um conjunto de direções.

```
bool contemValorNoVetor(Direcao *vetor, int tamVetor, int item) {
    for (int i = 0; i < tamVetor; i++) {
        if (vetor[i] == item)
            return true;
    }
    return false;
}
```

Código 35: Função responsável por verificar se um determinado valor está presente em um vetor de direções.

2.3.31 Gerar direções aleatórias

A função "gerarDirecoesAleatorias" gera um vetor de direções aleatórias. Ela aloca memória para o vetor de direções, inicializa todas as posições com um valor inválido (-1) e preenche o vetor com quatro direções aleatórias não repetidas.

```
Direcao *gerarDirecoesAleatorias() {
    Direcao *direcoes = (Direcao *)malloc(sizeof(Direcao) * 4);
    for (int i = 0; i < 4; i++)
        direcoes[i] = -1;
    int i = 0;

    while (i < 4) {
        int direcaoAleatoria = rand() % 4;
        if (contemValorNoVetor(direcoes, 4, direcaoAleatoria))
            continue;
        direcoes[i++] = direcaoAleatoria;
    }
    return direcoes;
}
```

Código 36: Função responsável por gerar um vetor de direções aleatórias.

2.3.32 Mover posição para posição

A função "moverPosicaoParaPosicao" recebe um ponteiro para uma estrutura do tipo posicao e uma nova posição. Ela atualiza a posição atual com os valores da nova posição.

```
void moverPosicaoParaPosicao(Posicao *posicao, Posicao novaPosicao) {
    posicao->x = novaPosicao.x;
    posicao->y = novaPosicao.y;
}
```

Código 37: Função responsável por mover uma posição para outra posição.

2.3.33 Direções que pode criar bifurcações

A função "direcoesQuePodeCriarBifurcacao" verifica se existem condições para criar bifurcações nas direções disponíveis a partir da posição dada.

Nota que a condição para que seja possível criar uma bifurcação é conter uma linha (ou coluna de três caracteres '*') a cada direção, ou seja, para poder criar uma bifurcação para cima é necessário acima da posição possuir uma linha de três asteriscos (estando os três uma linha acima, porêm, um uma coluna a esquerda, um na mesma coluna e o outro na coluna a direita).

```
1  DirecoesDisponiveis *direcoesQuePodeCriarBifurcacao(Labirinto *labirinto,
       Posicao *posicao) {
2      bool podeEmCima = labirinto->tabuleiro[posicao->x - 1][posicao->y - 1] ==
           '*' &&
3                        labirinto->tabuleiro[posicao->x - 1][posicao->y] == '*'
                             &&
4                        labirinto->tabuleiro[posicao->x - 1][posicao->y + 1] ==
                             '*' && posicao->x - 1 > 0;
5      bool podeEmBaixo = labirinto->tabuleiro[posicao->x + 1][posicao->y - 1] ==
           '*' &&
6                         labirinto->tabuleiro[posicao->x + 1][posicao->y] == '*'
                              &&
7                         labirinto->tabuleiro[posicao->x + 1][posicao->y + 1] ==
                              '*' &&
8                         posicao->x + 2 < labirinto->qtdLinhas;
9      bool podeNaDireita = labirinto->tabuleiro[posicao->x + 1][posicao->y + 1]
           == '*' &&
10                         labirinto->tabuleiro[posicao->x][posicao->y + 1] == '
                               *' &&
11                         labirinto->tabuleiro[posicao->x - 1][posicao->y + 1]
                               == '*' &&
12                         posicao->y + 1 < labirinto->qtdColunas;
13     bool podeNaEsquerda = labirinto->tabuleiro[posicao->x + 1][posicao->y - 1]
           == '*' &&
14                          labirinto->tabuleiro[posicao->x][posicao->y - 1] ==
                               '*' &&
15                          labirinto->tabuleiro[posicao->x - 1][posicao->y - 1]
                                == '*' && posicao->y - 1 > 0;
16
17     DirecoesDisponiveis *direcoesDisponiveis = alocarDirecoesDisponiveis();
18     if (podeEmCima)
19         direcoesDisponiveis->direcoes[direcoesDisponiveis->qtdDirecoes++] =
               CIMA;
20     if (podeEmBaixo)
21         direcoesDisponiveis->direcoes[direcoesDisponiveis->qtdDirecoes++] =
               BAIXO;
22     if (podeNaDireita)
23         direcoesDisponiveis->direcoes[direcoesDisponiveis->qtdDirecoes++] =
               DIREITA;
24     if (podeNaEsquerda)
25         direcoesDisponiveis->direcoes[direcoesDisponiveis->qtdDirecoes++] =
               ESQUERDA;
26     return direcoesDisponiveis;
27 }
```

Código 38: Função responsável por verificar quais direções que pode criar bifurcação.

2.3.34 Pode remover para bifurcação

A função "podeRemoverParaBifurcacao" verifica se é possível remover uma parede em uma determinada direção para criar uma bifurcação no labirinto.

Essa função é útil para verificar se é possível criar uma bifurcação removendo uma parede em uma determinada direção do labirinto, considerando as paredes adjacentes necessárias para formar a bifurcação.

Nota: foi feito para que não gere um labirinto muito aberto.

```
bool podeRemoverParaBifurcacao(Labirinto *labirinto, Direcao direcao, Posicao
    posicao) {
    moverPosicaoPelaDirecao(&posicao, direcao);
    int yMainUm = posicao.y + 1;
    int yMenosUm = posicao.y - 1;
    int xMaisUm = posicao.x + 1;
    int xMinusOne = posicao.x - 1;

    bool cimaEsquerda = false, cimaDireita = false, baixoEsquerda = false,
        baixoDireita = false;
    bool aEsquerda = false, aDireita = false, aCima = false, aBaixo = false;

    if (yMenosUm >= 0 && xMinusOne >= 0)
        cimaEsquerda = labirinto->tabuleiro[xMinusOne][yMenosUm] == '*';
    if (yMainUm < labirinto->qtdColunas && xMinusOne >= 0)
        cimaDireita = labirinto->tabuleiro[xMinusOne][yMainUm] == '*';
    if (yMenosUm >= 0 && xMaisUm < labirinto->qtdLinhas)
        baixoEsquerda = labirinto->tabuleiro[xMaisUm][yMenosUm] == '*';
    if (yMainUm < labirinto->qtdColunas && xMaisUm < labirinto->qtdLinhas)
        baixoDireita = labirinto->tabuleiro[xMaisUm][yMainUm] == '*';

    if (yMenosUm >= 0)
        aEsquerda = labirinto->tabuleiro[posicao.x][yMenosUm] == '*';
    if (yMainUm < labirinto->qtdColunas)
        aDireita = labirinto->tabuleiro[posicao.x][yMainUm] == '*';
    if (xMinusOne >= 0)
        aCima = labirinto->tabuleiro[xMinusOne][posicao.y] == '*';
    if (xMaisUm < labirinto->qtdLinhas)
        aBaixo = labirinto->tabuleiro[xMaisUm][posicao.y] == '*';

    return (cimaDireita + cimaEsquerda + baixoEsquerda + baixoDireita >= 2) &&
           (aEsquerda + aDireita + aCima + aBaixo >= 3);
}
```

Código 39: Função responsável por verificar se pode remover para bifurcação.

2.3.35 Criar bifurcação

A função "criarBifurcacao" é responsável por criar uma bifurcação no labirinto a partir de uma posição e direção específicas. Ela recebe um ponteiro para o labirinto, uma posição, uma direção e uma chance de criar a bifurcação.

Essa função é utilizada para criar bifurcações no labirinto, permitindo a geração de caminhos ramificados durante a construção do labirinto. A recursão permite a criação de bifurcações múltiplas, explorando diferentes direções a partir de uma posição inicial. A chance de criar uma bifurcação em cada chamada recursiva é especificada como um parâmetro, permitindo controlar a densidade de bifurcações no labirinto final.

```
1  void criarBifurcacao(Labirinto *labirinto, Posicao posicao, Direcao direcao,
       int chance) {
2      bool estaNaBorda = estaNaBordaPelaDirecao(labirinto, direcao, posicao, 1,
           1);
3      bool naoPodeMoverParaDirecao = !podeRemoverParaBifurcacao(labirinto,
           direcao, posicao);
4
5      moverPosicaoPelaDirecao(&posicao, direcao);
6      if (estaNaBorda || naoPodeMoverParaDirecao || !((rand() % 100) + 1 <=
           chance))
7          return;
8
9      labirinto->tabuleiro[posicao.x][posicao.y] = ' ';
10
11     Direcao *direcoesAleatorias = gerarDirecoesAleatorias();
12     for (int i = 0; i < 4; i++) {
13         Direcao direcaoAtual = direcoesAleatorias[i];
14         criarBifurcacao(labirinto, posicao, direcaoAtual, chance - 5);
15     }
16     free(direcoesAleatorias);
17 }
```

Código 40: Função responsável por criar uma bifurcação no labirinto.

2.3.36 Direções disponíveis

A função "direcoesDisponiveisParaMovimento" é responsável por retornar as direções disponíveis para movimento a partir de uma posição específica no labirinto.

Essa função é utilizada para determinar as direções em que um agente no labirinto pode se mover a partir de uma determinada posição. Ela identifica as direções em que há espaços vazios no labirinto, permitindo ao agente escolher uma direção para seu próximo movimento.

```
1  DirecoesDisponiveis *direcoesDisponiveisParaMovimento(Labirinto *labirinto,
       Posicao posicao) {
2
3      bool aEsquerda = labirinto->tabuleiro[posicao.x][posicao.y - 1] == ' ';
4      bool aDireita = labirinto->tabuleiro[posicao.x][posicao.y + 1] == ' ';
5      bool aCima = labirinto->tabuleiro[posicao.x - 1][posicao.y] == ' ';
6      bool aBaixo = labirinto->tabuleiro[posicao.x + 1][posicao.y] == ' ';
7      DirecoesDisponiveis *direcoesDisponiveis = alocarDirecoesDisponiveis();
8      if (aCima)
9          direcoesDisponiveis->direcoes[direcoesDisponiveis->qtdDirecoes++] =
               CIMA;
10     if (aBaixo)
11         direcoesDisponiveis->direcoes[direcoesDisponiveis->qtdDirecoes++] =
               BAIXO;
12     if (aDireita)
13         direcoesDisponiveis->direcoes[direcoesDisponiveis->qtdDirecoes++] =
               DIREITA;
14     if (aEsquerda)
15         direcoesDisponiveis->direcoes[direcoesDisponiveis->qtdDirecoes++] =
               ESQUERDA;
16     return direcoesDisponiveis;
17 }
```

Código 41: Função responsável por retornar as direções disponíveis para movimento a partir de uma posição no labirinto.

2.3.37 Percorrer caminho

A função "percorrerCaminho" é responsável por percorrer o labirinto a partir de uma determinada posição e registrar o percurso realizado. Função semelhante a achar saida (mencionada no tópico 2.3.12), porém, foi criada pois é usada em contextos diferentes, para não acabar gerando efeitos colaterais.

Essa função é utilizada para percorrer o labirinto a partir de uma posição inicial, explorando todas as direções possíveis e criando bifurcações ao longo do caminho.

```
void percorrerCaminho(Labirinto *labirinto, Posicao posicao, Percurso *
    percursoAtual) {
    percursoAtual->posicoes[percursoAtual->passos++] = posicao;

    if (posicao.x == labirinto->qtdLinhas - 2 && posicao.y == labirinto->
        qtdColunas - 2) {
        return;
    }

    DirecoesDisponiveis *direcoesDisponiveis =
        direcoesDisponiveisParaMovimento(labirinto, posicao);
    Posicao posicaoInicial = posicao;
    for (int i = 0; i < direcoesDisponiveis->qtdDirecoes; i++) {
        Direcao direcaoAtual = direcoesDisponiveis->direcoes[i];
        bool podeMoverParaDirecaoAtual = podeMoverParaCaminhoVazio(labirinto,
            direcaoAtual, posicao);
        bool posicaoNaoEstaNoPercurso = !posicaoEstaNoPercurso(posicao,
            direcaoAtual, percursoAtual);

        if (podeMoverParaDirecaoAtual && posicaoNaoEstaNoPercurso)
            moverPosicaoPelaDirecao(&posicao, direcaoAtual);
        else
            continue;

        percorrerCaminho(labirinto, posicao, percursoAtual);
    }
    desalocarDirecoesDisponiveis(&direcoesDisponiveis);
    moverPosicaoParaPosicao(&posicao, &posicaoInicial);
    DirecoesDisponiveis *direcoesDisponiveisParaBifurcar =
        direcoesQuePodeCriarBifurcacao(labirinto, &posicao);
    if (direcoesDisponiveisParaBifurcar->qtdDirecoes == 0) {
        desalocarDirecoesDisponiveis(&direcoesDisponiveisParaBifurcar);
        return;
    }
    Direcao direcao = direcoesDisponiveisParaBifurcar->direcoes[rand() %
        direcoesDisponiveisParaBifurcar->qtdDirecoes];
    criarBifurcacao(labirinto, posicao, direcao, 100);
    desalocarDirecoesDisponiveis(&direcoesDisponiveisParaBifurcar);
}
```

Código 42: Função responsável por percorrer caminho.

2.3.38 Criar saída

A função "criarSaida" é responsável por criar a saída do labirinto.
Essa função utiliza uma abordagem recursiva para percorrer o labirinto e criar a saída. Ela mover-se aleatoriamente até encontrar a saída.

```
void criarSaida(Labirinto *labirinto, Posicao posicao, Percurso *percursoAtual
    , bool *criouSaida) {
    labirinto->tabuleiro[posicao.x][posicao.y] = ' ';
    percursoAtual->posicoes[percursoAtual->passos++] = posicao;
    Direcao *direcoesAleatorias = gerarDirecoesAleatorias();

    if (posicao.x == labirinto->qtdLinhas - 2 && posicao.y == labirinto->
        qtdColunas - 2) {
        *criouSaida = true;
        free(direcoesAleatorias);
        return;
    }

    Posicao posicaoInicial = posicao;
    for (int i = 0; i < 4; i++) {
        Direcao direcaoAtual = direcoesAleatorias[i];

        bool estaNaBorda = estaNaBordaPelaDirecao(labirinto, direcaoAtual,
            posicao, 0, 0);
        bool naoPodeMoverParaDirecao = !podeRemoverPara(labirinto,
            direcaoAtual, posicao);
        bool jaPassouPorAqui = posicaoEstaNoPercurso(posicao, direcaoAtual,
            percursoAtual);
        if (estaNaBorda || naoPodeMoverParaDirecao || jaPassouPorAqui)
            continue;
        moverPosicaoPelaDirecao(&posicao, direcaoAtual);
        criarSaida(labirinto, posicao, percursoAtual, criouSaida);
        if (*criouSaida) {
            free(direcoesAleatorias);
            return;
        }
        moverPosicaoParaPosicao(&posicao, &posicaoInicial);
    }
    free(direcoesAleatorias);
    labirinto->tabuleiro[posicao.x][posicao.y] = '*';
}
```

Código 43: Função responsável por criar saída do labirinto.

2.3.39 Gerar labirinto aleatório

A função "gerarLabirintoAleatorio" é responsável por gerar um labirinto aleatório com base nos parâmetros fornecidos.

Essa função utiliza as funções criarSaida e "percorrerCaminho" para criar a saída e percorrer o caminho do labirinto, respectivamente. O tabuleiro do labirinto é inicializado com obstáculos em todas as posições e, em seguida, é gerado um caminho simples até a saida e então gerado bifurcações.

```
1  Labirinto *gerarLabirintoAleatorio(Labirinto *labirinto, int l, int c, char
       opcao) {
2      labirinto->qtdColunas = c;
3      labirinto->qtdLinhas = l;
4
5      labirinto->tabuleiro = (char **)malloc(sizeof(char *) * l);
6      for (int i = 0; i < l; i++)
7          labirinto->tabuleiro[i] = (char *)malloc(sizeof(char) * c);
8
9      labirinto->opcao = opcao;
10
11     Percurso *percursoAtual = alocarPercurso(labirinto);
12     incializaTabuleiro(labirinto);
13     bool criouSaida = false;
14     Posicao *posicao = criarPosicao(1, 1);
15     criarSaida(labirinto, *posicao, percursoAtual, &criouSaida);
16     desalocarPercurso(&percursoAtual);
17
18     Percurso *percursoPercorridoAtual = alocarPercurso(labirinto);
19     percorrerCaminho(labirinto, *posicao, percursoPercorridoAtual);
20     desalocarPercurso(&percursoPercorridoAtual);
21
22     desalocarPosicao(&posicao);
23
24     labirinto->tabuleiro[l - 2][c - 1] = ' ';
25     labirinto->tabuleiro[1][1] = 'M';
26     return labirinto;
27 }
```

Código 44: Função responsável por gerar labirinto aleatório.

3 Testes

Realizando os testes do algoritmo, obtivemos os seguintes resultados:

3.1 Teste com tabuleiro fornecido pelo usuário

Dadas as opções disponíveis para o usuário, onde ele pode escolher "p" para imprimir o percurso e "c" para imprimir as coordenadas do rato até a saída. Foram feitos testes para ambos os casos.

3.1.1 Imprimindo o percurso

Obtivemos como o resultado o único caminho disponível.

Teste 1

```
15 15
p
***************
*   *   *     *
*** * * * *** *
*M* * *     * *
* * *** ***** *
* *   * *     *
* *** * * *****
*     * * *   *
* ***** * * * *
* *     *   * *
* *** ******* *
*   *     *   *
*** ******* * *
*           *
***************
```

Código 45: Entrada do teste 1.

```
27
***************
*   *   *     *
*** * * * *** *
*M* * *     * *
*.* *** ***** *
*.*   * *     *
*.*** * * *****
*.    * * *   *
*.***** * * * *
*.*     *   * *
*.*** ******* *
*...*     *...*
***.*******.*.*
*  .........*..
***************
```

Código 46: Saída do teste 1.

3.1.2 Imprimindo as coordenadas

Obtivemos como o resultado as coordenadas do caminho.

Teste 2

```
15 15
c
***************
*   *   *     *
*** * * * *** *
*M* * *     * *
* * *** ***** *
* *   * *     *
* *** * * *****
*     * * *   *
* ***** * * * *
* *     *   * *
* *** ******* *
*   *     *   *
*** ****** * * *
*           *
***************
```

Código 47: Entrada do teste 2.

```
27
(1, 4)
(1, 5)
(1, 6)
(1, 7)
(1, 8)
(1, 9)
(1, 10)
(1, 11)
(2, 11)
(3, 11)
(3, 12)
(3, 13)
(4, 13)
(5, 13)
(6, 13)
(7, 13)
(8, 13)
(9, 13)
(10, 13)
(11, 13)
(11, 12)
(11, 11)
(12, 11)
(13, 11)
(13, 12)
(13, 13)
(14, 13)
```

Código 48: Saída do teste 2.

3.2 Primeiro caso de teste alterado

Após alterar o primeiro caso de teste adicionando um caminho menor obtivemos o seguinte resultado:

Teste 3

```
15 15
p
***************
*   *   *     *
*** * * * *** *
*M* * *     * *
* * *** ***** *
* *   * *     *
* *** * * *****
*             *
* ***** * * * *
* *     *   * *
* *** ******* *
*   *     *   *
*** ******* * *
*           *
***************
```

Código 49: Entrada do teste 3.

```
23
***************
*   *   *     *
*** * * * *** *
*M* * *     * *
*.* *** ***** *
*.*   * *     *
*.*** * * *****
*.............*
* ***** * * *.*
* *     *   *.*
* *** *******.*
*   *     *  .*
*** ******* *.*
*           *..
***************
```

Código 50: Saída do teste 3.

3.3 Tabuleiro fornecido pelo usuário sem saida

Na sequência alteramos o caminho o deixando sem saida e obtivemos como esperado "Epic Fail":

Teste 4

```
7 11
p
***********
*         *
******* * *
*M* * * * *
* * * * * *
*       * *
***********
```

Código 51: Entrada do teste 4.

```
EPIC FAIL!
```

Código 52: Saída do teste 4.

3.4 Tabuleiro gerado aleatóriamente

Na sequência para finalizar os possíveis casos de testes testamos a função de gerar labirinto em conjunto com a função de achar saida e obtivemos o seguinte resultado:

Teste 5

```
57
***************
*M..***...*****
***.....*..* **
**********.* **
*.....*    .* **
*.***.****.* **
*...*......* **
***.******** **
**.. *    *** **
**.**   *   ** **
**.*   * * ....*
**.* ** **.**.*
**.***.....* .*
**.....***** ..
***************
```

Código 53: Saída do teste 5.

4 Análise

Foram realizados testes de tempo das funções "acharSaida" e "gerarLabirintoAleatorio" assim sendo, fizemos alguns testes com tabuleiros de diferentes dimensões. Foram feitos quinze testes para cada caso, levando em consideção a média de gasto de tempo de todos eles.

Realizamos os testes de tempo usando a seguinte máquina:

Processador	11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz 2.80 GHz
RAM	24,0 GB (23,7 GB usable)
Arquitetura	64-bit operating system, x64-base processor

Tabela 1: Configurações da máquina.

Obtivemos os seguintes resultados:

4.0.1 Gerar Labirinto Aleatório

Dimensões	Média de tempo de execução em segundos
50 por 50	0
100 por 100	0.20
150 por 150	0.80
200 por 200	2.93
250 por 250	6.73
300 por 300	13.53
350 por 350	13.53
400 por 400	41.60
450 por 450	60.13

Tabela 2: Média de tempo nas execuções em segundos.

4.0.2 Achar saída

Dimensões	Média de tempo de execução em segundos
50 por 50	0
100 por 100	0
150 por 150	0.20
200 por 200	0.80
250 por 250	2.60
300 por 300	5.86
350 por 350	7.40
400 por 400	13.06
450 por 450	17.73

Tabela 3: Média de tempo nas execuções em segundos.

Com isso, conseguimos perceber que a função "gerarLabirintoAleatorio" por fazer mais operações tem seu tempo aumentando mais rápido que a função "acharSaida".

5 Conclusão

Durante o desenvolvimento deste trabalho, foi possível visualizar na prática o funcionamento da "call stack" (pilha de chamadas) em conjunto com a recursão. Por meio disso, foi possível observar um algoritmo da família dos algoritmos de força bruta em ação.

No entanto, surgiram algumas dificuldades ao lidar com certos cenários. Por exemplo, quando tínhamos um labirinto "aberto", sem obstáculos, o número de caminhos possíveis se tornava muito grande. Isso acabava aumentando significativamente o tempo necessário para a execução do algoritmo e, em alguns casos, resultava em problemas de loop.

Além disso, a criação da lógica para gerar um labirinto aleatório também foi um desafio. As abordagens iniciais não conseguiram gerar um labirinto com obstáculos suficientes para formar um caminho coerente. Foi necessário pensar fora da caixa e desenvolver um algoritmo que garantisse a geração de um labirinto aleatório válido.

Essa experiência nos proporcionou um entendimento mais aprofundado sobre como podemos usar uma função recursiva para dividir um problema grande em problemas menores. A recursão e a manipulação de estruturas de dados foram fundamentais para a criação e resolução do labirinto. Compreender os desafios e limitações desses algoritmos nos permitiu explorar soluções alternativas e aprender com os problemas encontrados.

Em conclusão, o desenvolvimento deste trabalho nos proporcionou um valioso aprendizado sobre a aplicação da recursão em algoritmos de busca e resolução de labirintos. Através dos desafios enfrentados, pudemos aprimorar nossas habilidades de resolução de problemas e compreender melhor as vantagens e limitações dos algoritmos de força bruta.

Referências

Aulas teórias e práticas da disciplina de Estrutura de Dados 1.