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

# Trabalho Prático 01: Projeto e análise de algoritmos

*Pedro Henrique Dias Quintão - 5916*

*Artur Gil Luiz - 5924*

*Tariky Rodrigues campos - 5758*

*Leonardo Henrique de Oliveira - 5913*

**18 de outubro de 2025**

# 1. Introdução

Este documento é uma documentação completa para o trabalho prático 01 da disciplina projeto e análise de algoritmos. O objetivo do software é simular a exploração de uma nave espacial em um mapa 2D desconhecido. A nave tem uma missão, mas também tem recursos limitados.

Imagine que você é o piloto. Você começa em um ponto de partida (**'X'**) e sua missão é chegar ao destino (**'F'**). No entanto, a nave possui um escudo de "Durabilidade" (**D**) que se desgasta a cada movimento que você faz (**Dp**).

Para sobreviver, você pode encontrar "Peças" (**'P'**) espalhadas pelo mapa, que recarregam sua durabilidade (**A**). Além disso, o mapa possui "terrenos perigosos" (**'B'**) que causam dano extra e caminhos restritos que só permitem movimentos específicos.

O objetivo do programa é encontrar **um caminho válido** que leve a nave do início ao fim, ou que a permita coletar 4 peças, antes que sua durabilidade chegue a zero.

# 2. O Mundo do Jogo

O universo do jogo é carregado de um arquivo de texto. Este arquivo define não apenas o layout do mapa, mas também as regras de sobrevivência da nave.

### 2.1 Os Parâmetros de Sobrevivência

A primeira linha do arquivo define três valores essenciais:

- **D (Durabilidade Inicial):** A "vida" total da sua nave.
- **Dp (Dano por Passo):** O custo em durabilidade para se mover para uma nova célula.
- **A (Bônus por Peça):** A quantidade de durabilidade que a nave *recupera* ao encontrar uma peça 'P'.

### 2.2 A Lenda do Mapa

O mapa é composto pelos seguintes caracteres, cada um com uma regra:

- **X**: O ponto de partida da nave.
- **F**: O destino final. Atingir este ponto completa a jornada.
- **P**: Uma peça. Ao pousar aqui, a durabilidade aumenta em **A** e o contador de peças restantes diminui.
- **B**: Um setor perigoso .Causa o **dobro de dano** (2 * Dp) ao entrar.
- **|** (Barra vertical): Um "corredor" que só permite movimento na **vertical** (para cima ou para baixo).
- **-** (Hífen): Um "túnel" que só permite movimento na **horizontal** (esquerda ou direita).
- **+** (Sinal de Mais): Uma "interseção" que permite movimento em **todas as 4 direções**.
- **.**: Uma parede ou obstáculo intransponível.

### 2.3 Objetivos da Missão

A jornada termina com sucesso se **uma** destas duas condições for atingida:

1. A nave chega à célula **'F'**.
2. A nave coleta 4 peças **'P'** (o contador pecas_restantes chega a 0).

A jornada falha se a durabilidade da nave chegar a 0 ou menos.

# 3. O Cérebro da Nave: O Algoritmo de Backtracking

Como a nave "decide" para onde ir? Ela não sabe o caminho de antemão. Para isso, usamos um algoritmo clássico de "tentativa e erro" chamado Backtracking, que é uma forma de Busca em Profunidade (DFS).

### 3.1 A Lógica da Tentativa e Erro

Pense no algoritmo como um explorador em um labirinto escuro com um novelo de lã:

1. **Dê um passo:** A nave escolhe uma direção válida (Norte, Sul, Leste, Oeste) que ainda não visitou.
2. **Marque o caminho:** Ao entrar em uma nova célula, a nave "deixa um rastro" (marca visitado[l][c] = 1).
3. **Recalcule a vida:** A nave perde durabilidade (Dp ou 2*Dp se for 'B'). Se for uma peça 'P', ela ganha durabilidade (A).
4. **Chegou?** Se a célula atual for o 'F' ou se as peças acabaram, a nave avisa: "Encontrei a solução!" e para de procurar.
5. **Continue explorando:** Se não for o fim, a nave chama a si mesma (recursão) para explorar a partir desta nova célula.

6. **Beco sem Saída (O Backtrack):** Se a nave ficar sem durabilidade ou se todos os vizinhos já foram visitados (um beco sem saída), ela "volta" para a célula anterior. Ao voltar, ela **pega seu rastro de volta** (marca visitado[l][c] = 0).

Esse último passo é a "mágica" do backtracking. Ao desmarcar a célula como visitada no retorno, a nave permite que *outros caminhos* possam usar aquela mesma célula para tentar encontrar a solução.

### 3.2 Encontrando o Primeiro Caminho, Não o Melhor

É importante notar que este algoritmo **não garante encontrar o caminho mais curto** ou o caminho que termina com mais durabilidade. Ele foi projetado para parar assim que encontra **a primeira solução válida**.

Isso é feito pela variável global achou_destino. Assim que ela se torna 1, todas as chamadas recursivas ativas param de explorar e retornam imediatamente, economizando processamento.

## 4. Análise Detalhada do Código

Nesta seção, vamos analisar mais a fundo as funções-chave que fazem o projeto funcionar.

### 4.1 Função `lerArquivo` (em `Mapa.c`)

Esta é a função que constrói o nosso mundo. Ela é responsável por abrir o arquivo de texto e alocar dinamicamente toda a memória necessária para a simulação.

- Ela primeiro lê os parâmetros de durabilidade (D, Dp, A) e as dimensões (linhas, colunas).
- Em seguida, ela aloca a memória para os ponteiros de ponteiros (por exemplo, m->mapa = malloc(m->linhas * sizeof(char *));).
- Dentro de um laço, ela aloca a memória para cada linha individual (m->mapa[i] = malloc(...);).
- Ela também aloca e zera as matrizes visitado e coletada usando calloc.
- Enquanto lê o mapa, ela aproveita para identificar a posição inicial 'X' e armazená-la.
- O código dessa função está apresentado na parte final dessa documentação na seção Figuras, como figura 1.

---

### 4.2 Função `podeMover` (em `Nave.c`)

Esta função é a "guarda de trânsito" da simulação. Ela garante que a nave obedeça às regras de movimento dos corredores. A lógica é dividida em duas partes:

1. **Verificação de Saída:** O código primeiro verifica se a *célula atual* permite sair na direção desejada. Por exemplo, se a célula atual for **'-'** e a direção for vertical (0 ou 1), a função retorna 0 (falso) imediatamente.
2. **Verificação de Entrada:** Se a saída for válida, o código verifica se a *próxima célula* permite a entrada a partir da direção de origem. Por exemplo, se a próxima célula for **'|'** e o movimento for horizontal (2 ou 3), a função retorna 0 (falso).

Somente se a saída da célula atual **e** a entrada na próxima célula forem válidas, a função retorna 1 (verdadeiro).

O código dessa função está apresentado na parte final dessa documentação na seção Figuras, como figura 2.

---

### 4.3 Função `movimentar` (em `Nave.c`)

Este é o coração do algoritmo de backtracking. A função é longa, mas podemos dividi-la em partes lógicas:

1. **Registro de Estado:** A função registra o passo atual e marca a célula como visitado[l][c] = 1.
2. **Processamento da Célula:** Ela verifica se a célula é uma peça 'P'. Se for, aplica o bônus de durabilidade 'A' e decrementa o contador pecas_restantes.
3. **Verificação dos Casos Base (Sucesso):** A função verifica se a missão terminou com sucesso. Ela retorna (e ativa achou_destino) se a célula atual for 'F' **ou** se pecas_restantes for 0.
4. **Verificação dos Casos Base (Falha):** A função verifica se a nave "morreu". Se dur <= 0, ela simplesmente retorna, iniciando o processo de backtracking.
5. **Passo Recursivo:** A função entra em um laço for para testar os 4 vizinhos. Para cada vizinho, ela:
    - Verifica se é um movimento válido (usando podeMover e se já foi visitado).
    - Calcula a nova durabilidade, aplicando o dano normal (Dp) ou o dano dobrado (se for 'B').
    - Chama a si mesma (`movimentar(...)`) para o vizinho.
    - **Corte de Exploração:** Se a chamada recursiva retornar e achou_destino for verdadeiro, ela para de testar os outros vizinhos e retorna imediatamente.

6. **Backtracking (Limpeza):** Se a função retornar (seja por um beco sem saída ou falha), ela "limpa seus rastros", resetando visitado[l][c] = 0 e coletada[l][c] = 0 para seus valores originais.

O código dessa função está apresentado na parte final dessa documentação na seção Figuras, como figuras 3 e 4.

---

# 5. Estrutura do Projeto (Visão Geral dos Arquivos)

O projeto é modularizado em diferentes arquivos .c e .h para manter o código organizado.

- **Main.c:** O ponto de entrada. Ele gerencia o menu principal, pergunta o nome do arquivo, pergunta sobre os modos (Visual e Análise),gera um mapa aleatório e inicia a jornada.
- **Mapa.h / Mapa.c:** Responsável por tudo relacionado ao mapa.
    - Mundo: A struct gigante que armazena o mapa, as matrizes de estado (`visitado`, `coletada`) e os valores de durabilidade.
    - lerArquivo: Abre o arquivo de texto e aloca dinamicamente a memória para o mapa e as matrizes.
    - liberarMapa: Libera toda a memória alocada.
    - mostrarMapa: Imprime o mapa no console, usando cores ANSI para destacar a posição atual da nave e as áreas visitadas (se o MODO_VISUAL estiver ativo).
- **Nave.h / Nave.c:** O "cérebro" do projeto.
    - Step: Uma struct auxiliar para guardar os dados de um passo (posição, durabilidade, peças) e salvar o caminho final.
    - movimentar: A função recursiva de backtracking.
    - podeMover: A função que verifica as regras de movimento dos corredores.
    - iniciarJornada: Prepara as variáveis globais, aloca memória para os caminhos (current_steps e final_steps) e dispara a primeira chamada da `movimentar`.
- **Gerar_Testes.h / Gerar_Testes.c:** Um módulo utilitário que gera aleatoriamente arquivos de mapa para facilitar os testes.

# 6. Tarefas extras

## Tarefa Extra 1: Modo Visual (Interface Alternativa)

Atendendo à "Tarefa extra 1", que sugere a criação de diferentes formas de exibir o resultado, nosso grupo implementou um **"Modo Visual"** opcional.

Este modo, controlado pela constante MODO_VISUAL, oferece uma visualização passo a passo do progresso da nave diretamente no terminal, em vez de apenas exibir o resultado final.

Neste tópico mostraremos as atividades extras implementadas pelo grupo:

Primeiramente temos a função de mostrar as etapas do mapa.Este modo, controlado pela constante MODO_VISUAL, oferece uma visualização passo a passo do progresso da nave diretamente no terminal, em vez de apenas exibir o resultado final.

A função mostrarMapa é o coração dessa funcionalidade. Quando o modo visual está ativo, ela é chamada a cada movimento da nave e faz o seguinte:

1. **Exibição Colorida:** Utiliza códigos de cores ANSI (padrão em terminais Linux/macOS) para diferenciar os elementos no mapa, melhorando a legibilidade:
    - **X Verde:** Destaca a posição **atual** da nave.
    - **B Vermelho:** Destaca os buracos negros existem no mapa.
    - **Azul:** Marca as células que já foram **visitadas**.
    - **Cor Padrão:** Células ainda não exploradas.
2. **Animação Passo a Passo:** A função usleep(300000) introduz uma pequena pausa (0.3 segundos) após imprimir o mapa. Isso retarda a execução propositalmente, criando um efeito de "animação" que permite ao usuário acompanhar visualmente o caminho que o algoritmo está testando em tempo real.

O código criado para a resolução dessa tarefa se encontra na seção de figuras, como figura 8.

## Tarefa Extra 2: Gerador de Mapas de Teste

Para atender à solicitação de criar um programa para a geração de arquivos de teste, utilizamos a função gerar_mapa_teste como base. Ao adicionar uma função main que aceita argumentos de linha de comando, criamos um utilitário robusto chamado Gerar_Testes.c.

Este programa permite criar uma infinidade de mapas de teste com parâmetros configuráveis, salvando-os no formato exato esperado pelo aplicativo principal.

### Tarefa Extra 3: Complicação Adicional (Buracos Negros)

Para tornar a aventura mais desafiadora, conforme sugerido pela terceira tarefa extra, nosso grupo introduziu uma nova dinâmica no mapa: **Buracos Negros**.

**A Ideia**

A complicação não é simplesmente adicionar um novo tipo de obstáculo, mas sim uma "zona de perigo" que afeta a movimentação da nave em sua vizinhança.

1. **Representação:** Um buraco negro é representado no mapa pelo caractere B.
2. **A Regra:** A nave não pode entrar em nenhuma célula que seja **adjacente** (horizontal, vertical ou diagonalmente) a um buraco negro.
3. **O Desafio:** Isso força o algoritmo de busca a ser mais inteligente. Células que parecem livres (como ., P ou F) tornam-se intransitáveis se estiverem na "área de efeito" de um B. A nave deve, portanto, calcular rotas que contornem não apenas o buraco negro em si, mas toda a sua zona de influência de 8 células ao redor.

# 7. Exemplo de Uso

## 7.1 Exemplo de Arquivo de Mapa

Vamos criar um pequeno mapa de exemplo para ilustrar:

```
60 10 25
5 8
X-+-P..-
..|....F
B-+-..|-
..|..P..
P-+.B..-
```

**Tradução:**

- A nave começa com **60** de durabilidade.
- Cada passo custa **10**.
- Cada peça recupera **25**.
- O mapa tem 5 linhas e 8 colunas, com o layout mostrado.

## 7.2 Compilando e Executando

Para compilar o projeto (sem o gerador de testes), você usaria um comando similar a este em um terminal Windows:

mingw32-make

Para usuários Linux, apenas digite:
Make compile

Ou se quiser compilar na mão utilize:
gcc Main.c Mapa.c Nave.c Entradas_Extras/Gerar_Testes.c -o app.exe

Para executar:

./app.exe

ou

Make run

O programa então perguntará:

==== PROJETO E ANALISE DE ALGORITMOS - TRABALHO 1 ====

Digite o nome do arquivo de entrada: entrada.txt
Deseja ativar o modo visual (1 = sim, 0 = nao)? 1
Deseja ativar o modo de análise (1 = sim, 0 = nao)? 1

Se o **Modo Visual** estiver ativo, você verá o mapa sendo redesenhado a cada passo da nave, permitindo acompanhar a exploração em tempo real, as figuras com exemplos da execução com modo visual estão no final da documentação nas figuras 6 e 7.

Se o **Modo de Análise** estiver ativo, ao final da execução, o programa exibirá estatísticas sobre o desempenho do algoritmo:

===== MODO DE ANÁLISE =====
Chamadas recursivas: X
Nível máximo de recursão: X
==========================

# 8. Conclusão

O trabalho é considerado um excelente exemplo prático do algoritmo de backtracking, especificamente a Busca em Profundidade. O principal resultado e aprendizado do projeto foi demonstrar como um problema complexo de busca de caminho em um labirinto, que inclui regras dinâmicas como durabilidade, coleta de peças, dano variável e restrições de movimento, pode ser resolvido de forma elegante utilizando recursão.

# 9. Referências

As principais referências utilizadas para o desenvolvimento do trabalho foram:

- Documentação da linguagem C Disponivel em: https://devdocs.io/c/
- Material da disciplina de Projeto e Análise de Algoritmos (CCF 330).
- Notas de aula e exemplos práticos discutidos em sala

# 10. Figuras

```c
void lerArquivo(Mundo *m, int *x_inicio, int *y_inicio, const char *nomeArquivo) {
    FILE *arq = fopen(nomeArquivo, "r");
    if (!arq) {
        printf("Erro ao abrir o arquivo %s!\n", nomeArquivo);
        exit(1);
    }

    fscanf(arq, "%d %d %d", &m->D, &m->Dp, &m->A);
    fscanf(arq, "%d %d", &m->linhas, &m->colunas);

    m->mapa = malloc(m->linhas * sizeof(char *));
    m->visitado = malloc(m->linhas * sizeof(int *));
    m->coletada = malloc(m->linhas * sizeof(int *));
    for (int i = 0; i < m->linhas; i++) {
        m->mapa[i] = malloc((m->colunas + 1) * sizeof(char));
        m->visitado[i] = calloc(m->colunas, sizeof(int));
        m->coletada[i] = calloc(m->colunas, sizeof(int));
        fscanf(arq, "%s", m->mapa[i]);
        for (int j = 0; j < m->colunas; j++) {
            if (m->mapa[i][j] == 'X') {
                *x_inicio = i;
                *y_inicio = j;
            }
        }
    }
    fclose(arq);
}
```

Figura 1:código da função lerArquivo

```c
int podeMover(char atual, char prox, int dir) {
    if (prox == '.' || prox == '\0') return 0;

    int eh_horizontal = (dir == 2 || dir == 3);
    int eh_vertical   = (dir == 0 || dir == 1);

    int saida_valida = 0;
    if (atual == '-' && eh_horizontal) saida_valida = 1;
    else if (atual == '|' && eh_vertical) saida_valida = 1;
    else if (atual == '+' || atual == 'P' || atual == 'F' || atual == 'X' || atual == 'B')
        saida_valida = 1;

    if (!saida_valida) return 0;

    int entrada_valida = 0;
    if (prox == '-') entrada_valida = eh_horizontal;
    else if (prox == '|') entrada_valida = eh_vertical;
    else if (prox == '+' || prox == 'P' || prox == 'F' || prox == 'X' || prox == 'B')
        entrada_valida = 1;

    return entrada_valida;
}
```

Figura 2: código da função podeMover

```c
void movimentar(Mundo *m, int l, int c, int dur, int nivel, int pecas_restantes) {
    chamadas_recursivas++;
    if (nivel > max_nivel_recursao) max_nivel_recursao = nivel;

    // registra passo atual na sequência
    if (current_steps) {
        current_steps[nivel-1].l = l;
        current_steps[nivel-1].c = c;
        current_steps[nivel-1].dur = dur;
        current_steps[nivel-1].pecas = pecas_restantes;
    }

    int coletada_antes = m->coletada[l][c];

    m->visitado[l][c] = 1;
    // mostra mapa se modo visual ativo
    mostrarMapa(m, l, c);

    if (m->mapa[l][c] == 'P' && m->coletada[l][c] == 0) {
        m->coletada[l][c] = 1;
        dur += m->A;
        pecas_restantes--;
        // atualiza o registro do passo, pois D e peças mudaram neste mesmo passo
        if (current_steps) {
            current_steps[nivel-1].dur = dur;
            current_steps[nivel-1].pecas = pecas_restantes;
        }
    }

    if (m->mapa[l][c] == 'F') {
        achou_destino = 1;
        copiarCaminhoFinal(nivel);
        return;
    }

    if (pecas_restantes == 0) {
        achou_destino = 1;
        copiarCaminhoFinal(nivel);
        return;
    }
```

Figura 3: parte 1 do código da função movimentar

```c
    if (dur <= 0) {
        // restaura e volta
        m->visitado[l][c] = 0;
        m->coletada[l][c] = coletada_antes;
        return;
    }

    // tenta os 4 vizinhos
    for (int i = 0; i < 4; i++) {
        int nl = l + movimentos[i][0];
        int nc = c + movimentos[i][1];

        if (nl >= 0 && nl < m->linhas && nc >= 0 && nc < m->colunas) {
            char prox = m->mapa[nl][nc];

            if (!m->visitado[nl][nc] && podeMover(m->mapa[l][c], prox, i)) {
                int novo_dur = dur;

                if (prox == 'B') {
                    printf("Entrando em um setor perigoso! Dano extra!\n");
                    novo_dur -= m->Dp * 2;
                }
                else {
                    if (pecas_restantes > 0) novo_dur -= m->Dp;
                }

                movimentar(m, nl, nc, novo_dur, nivel+1, pecas_restantes);

                if (achou_destino) return; // corta demais explorações quando já encontrou solução
            }
        }
    }

    // backtracking: desfaz marcações no retorno
    m->visitado[l][c] = 0;
    m->coletada[l][c] = coletada_antes;
}
```

Figura 4: parte 2 do código da função movimentar

```
Digite o nome do arquivo de entrada: entrada_padrao.txt
Deseja ativar o modo visual (1 = sim, 0 = nao)? 0
Deseja ativar o modo de análise (1 = sim, 0 = nao)? 1
Linha: 2, Coluna: 3; D: 20, peças restantes: 4
Linha: 3, Coluna: 3; D: 15, peças restantes: 4
Linha: 3, Coluna: 4; D: 10, peças restantes: 4
Linha: 3, Coluna: 5; D: 15, peças restantes: 3
Linha: 3, Coluna: 6; D: 10, peças restantes: 3
Linha: 3, Coluna: 7; D: 5, peças restantes: 3
Linha: 4, Coluna: 7; D: 10, peças restantes: 2
Linha: 5, Coluna: 7; D: 5, peças restantes: 2

A tripulação finalizou sua jornada.
```

Figura 5: execução do código sem modo visual com o txt dentro do projeto

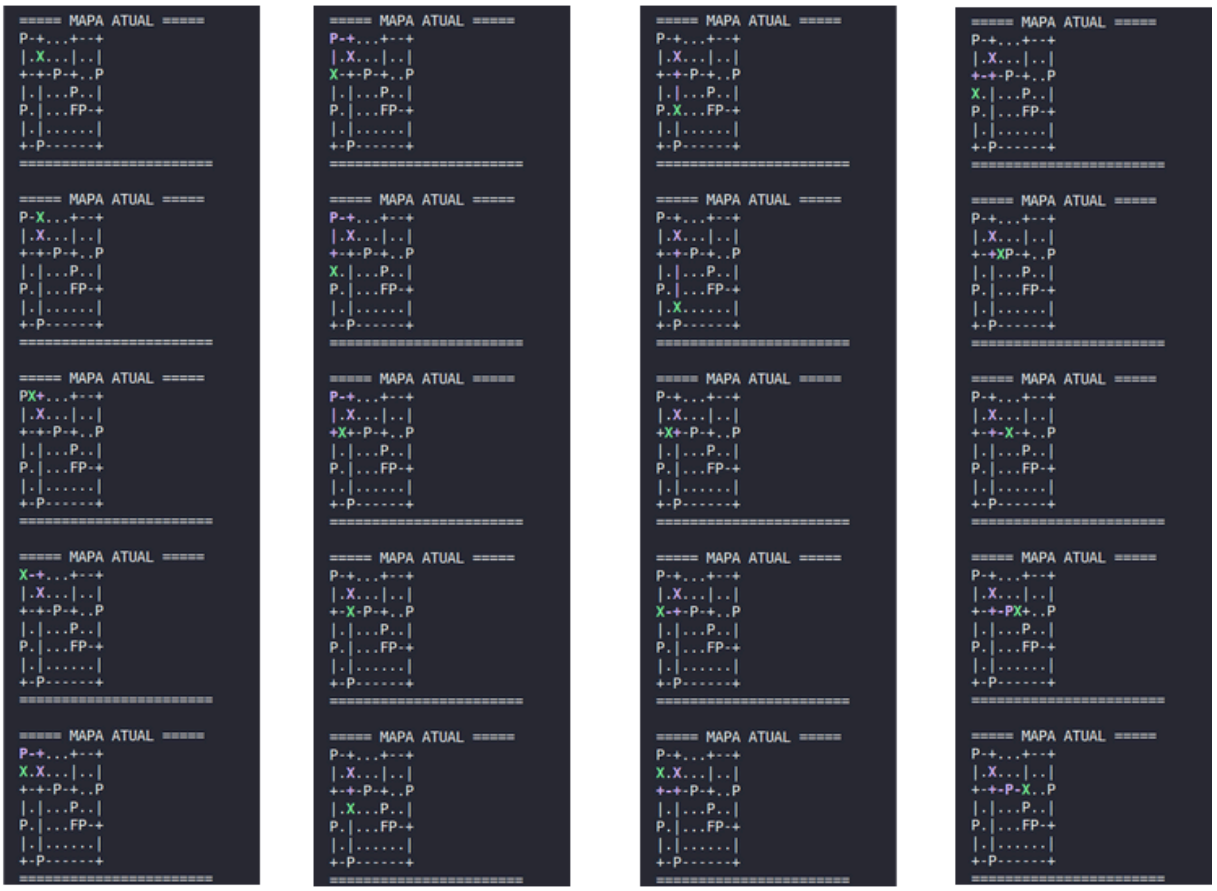


Figura 6: parte 1 da execução com modo visual

```
===== MAPA ATUAL =====
P-+...+--+
|.X...|..|
+-+-P-+..P
|.|...X..|
P.|...FP-+
|.|......|
+-P------+
======================

===== MAPA ATUAL =====
P-+...+--+
|.X...|..|
+-+-P-+..P
|.|...P..|
P.|...XP-+
|.|......|
+-P------+
======================
Linha: 2, Coluna: 3; D: 20, peças restantes: 4
Linha: 3, Coluna: 3; D: 15, peças restantes: 4
Linha: 3, Coluna: 4; D: 10, peças restantes: 4
Linha: 3, Coluna: 5; D: 15, peças restantes: 3
Linha: 3, Coluna: 6; D: 10, peças restantes: 3
Linha: 3, Coluna: 7; D: 5, peças restantes: 3
Linha: 4, Coluna: 7; D: 10, peças restantes: 2
Linha: 5, Coluna: 7; D: 5, peças restantes: 2

A tripulação finalizou sua jornada.

===== MODO DE ANÁLISE =====
Chamadas recursivas: 23
Nível máximo de recursão: 8
============================
```

Figura 7: parte 2 da execução com modo visual

```c
void mostrarMapa(Mundo *m, int atualL, int atualC) {
    if (!MODO_VISUAL) return;

    printf("\n===== MAPA ATUAL =====\n");
    for (int i = 0; i < m->linhas; i++) {
        for (int j = 0; j < m->colunas; j++) {
            if (i == atualL && j == atualC)
                printf("\033[1;32mX\033[0m");
            else if (m->mapa[i][j] == 'B')
                printf("\033[1;31mB\033[0m");
            else if (m->visitado[i][j])
                printf("\033[1;34m%c\033[0m", m->mapa[i][j]);
            else
                printf("%c", m->mapa[i][j]);
        }
        printf("\n");
    }
    printf("======================\n");
    usleep(300000);
}
```

Figura 8: função que imprime o mapa do modo visual