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

Trabalho 2 - Projeto e análise de algoritmos

Bruno Vicentini Ribeiro - 5907
Erich Pinheiro Amaral - 5915
Fabrício Henrique Viana Albino - 5925
Pedro Paulo Paz do Nascimento - 5937
Vitor Mathias Andrade - 5901

Florestal - MG
2025

SUMÁRIO

INTRODUÇÃO

Este trabalho tem como objetivo a implementação de um algoritmo baseado no paradigma programação dinâmica, projetado para determinar o caminho ótimo que permita que a tripulação do Expresso Interestelar chegue até Nikador, o titã do conflito, com o menor desgaste possível de suas forças. A proposta dá continuidade à narrativa iniciada no primeiro trabalho, em que a nave viajava em busca do Planeta das Festividades, mas agora enfrenta uma missão ainda mais desafiadora no planeta Amphoreus.

Nesta nova jornada, a tripulação deve percorrer um mapa repleto de obstáculos, monstros e âncoras temporais, que possibilitam o deslocamento entre dois tempos distintos — o presente e o passado. O desafio consiste em escolher, entre os caminhos possíveis, aquele que maximize a força final da tripulação ao chegar ao destino, considerando tanto os custos de energia ao enfrentar inimigos quanto os benefícios de descanso em determinadas regiões.

Para isso, o algoritmo desenvolvido emprega os conceitos fundamentais da programação dinâmica, permitindo armazenar e reutilizar resultados de subproblemas já resolvidos, evitando recomputações desnecessárias e melhorando a eficiência da solução. Dessa forma, é possível determinar o percurso ideal entre os mapas fornecidos, respeitando as restrições de movimentação e os efeitos das âncoras temporais.

Além de consolidar os conhecimentos sobre o paradigma de programação dinâmica, este trabalho visa aprofundar a compreensão sobre otimização de rotas e redução de complexidade computacional, aplicando técnicas eficientes de resolução de problemas em cenários com múltiplas variáveis e interdependências.

ORGANIZAÇÃO

Na figura 1, observa-se o panorama geral da organização do projeto, no qual estão elencados os arquivos necessários para o trabalho. Esses arquivos foram separados em pastas para garantir melhor organização e funcionamento do código.

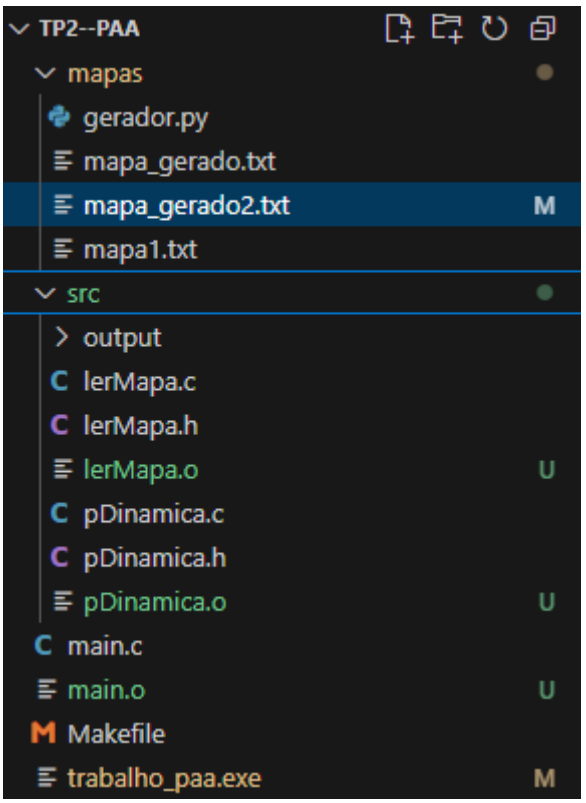

Figura 1: Repositório do projeto.

DESENVOLVIMENTO

- EXPLICAÇÃO DO ALGORITMO

O algoritmo de programação dinâmica foi desenvolvido com o objetivo de encontrar o melhor caminho possível para que a tripulação do Expresso Interestelar chegue até Nikador, o titã do conflito, com a maior força possível.

Diferente do primeiro trabalho, que utilizava *backtracking* para explorar todas as rotas possíveis, nesta implementação é aplicada uma abordagem otimizada, que armazena e reutiliza resultados intermediários, evitando repetições desnecessárias de cálculos.

O problema é representado por dois mapas de mesmo tamanho: um do presente e outro do passado. Cada célula pode conter:

- um número inteiro (representando inimigos que reduzem a força da tripulação),
- o valor 000 (indicando uma área de descanso, onde parte da força é recuperada),
- o símbolo AAA (representando uma “âncora temporal”, que transporta entre passado e presente),

- ou * (regiões intransponíveis).

O algoritmo analisa cada célula do mapa de forma sequencial, coluna por coluna, avaliando as três posições anteriores possíveis — acima, na mesma linha ou abaixo — e determinando qual delas oferece o melhor valor de força acumulado até aquele ponto. Quando uma célula contém uma âncora temporal, o cálculo é feito considerando a transição entre mapas (do presente para o passado e vice-versa).

A força total é atualizada conforme o tipo de célula:

- Se for um inimigo, a força é reduzida pelo valor indicado;
- Se for uma área livre, a tripulação recupera D unidades de força;
- Se for uma âncora, o tempo é alternado sem alterar a força atual.

Dessa forma, o algoritmo percorre toda a matriz até a última coluna, determinando o valor máximo de força possível para cada célula e registrando as melhores decisões em cada etapa. Ao final, a maior força encontrada na última coluna representa o resultado ideal. Se não houver caminho viável (força ≤ 0), é exibida a mensagem:

"A calamidade de Nikador é inevitável".

Caso haja um caminho possível, o algoritmo compara a força final da tripulação (F) com a força do inimigo (N), resultando em uma das mensagens:

"A ruína de Nikador é iminente" (se $F \geq N$),
ou
"Será necessário mais planejamento para parar a calamidade" (se $F < N$).

- IMPLEMENTAÇÃO DO ALGORITMO

A função principal do algoritmo é resolverPD() (figura 2), responsável por preencher a estrutura de programação dinâmica com os valores de força correspondentes a cada célula dos mapas.

Primeiramente, a função inicializa a primeira coluna com base na força inicial da tripulação (forcaTripulacao), aplicando os efeitos de inimigos ou áreas de descanso. Em seguida, percorre as colunas seguintes e, para cada célula (i, j), calcula a nova força com base na melhor força das três posições anteriores válidas:

- (i-1, j-1) – movimento diagonal superior direita;
- (i, j-1) – movimento para a direita;
- (i+1, j-1) – movimento diagonal inferior direita.

```
void resolverPD(EstruturaPD *pd)
{
    int h = pd->linhas;
    int w = pd->colunas;

    for (int i = 0; i < h; i++)
    {
        int valorPresente = getValorCelula(0, i, 0);
        if (valorPresente != INTRANSPONIVEL)
        {
            pd->melhorForca[0][i][0] = calcularNovaForca(tropa->forcaTripulacao, valorPresente);
        }

        // Tempo 1 (Passado)
        int valorPassado = getValorCelula(1, i, 0);
        if (valorPassado != INTRANSPONIVEL)
        {
            pd->melhorForca[1][i][0] = calcularNovaForca(tropa->forcaTripulacao, valorPassado);
        }
    }

    for (int j = 1; j < w; j++)
    {
        for (int i = 0; i < h; i++)
        {

            int valorPresente = getValorCelula(0, i, j);
            int valorPassado = getValorCelula(1, i, j);

            int maxAnteriorPresente = -1;
            int maxAnteriorPassado = -1;

            if (valorPresente != INTRANSPONIVEL)
            {
                if (valorPresente == ANCORA)
                {
                    maxAnteriorPresente = max3(
                        getForcaOrigem(pd, 1, i - 1, j - 1), // tempo = 1
                        getForcaOrigem(pd, 1, i, j - 1),     // tempo = 1
                        getForcaOrigem(pd, 1, i + 1, j - 1)  // tempo = 1
                    );
                }
                else
                {
                    maxAnteriorPresente = max3(
                        getForcaOrigem(pd, 0, i - 1, j - 1), // tempo = 0
                        getForcaOrigem(pd, 0, i, j - 1),     // tempo = 0
                        getForcaOrigem(pd, 0, i + 1, j - 1)  // tempo = 0
```

Figura 2: Função resolverPD.

A função calcularNovaForca() (figura 3) é usada para determinar a força resultante ao mover-se para a célula atual, considerando o tipo de terreno e os valores definidos no mapa.

Quando a célula contém uma âncora temporal (AAA), a origem é buscada no outro mapa (passado ou presente), simulando a troca de tempo.

```c
int calcularNovaForca(int forcaAnterior, int valorCelula)
{
    if (forcaAnterior <= 0)
    {
        return -1;
    }

    if (valorCelula == ANCORA)
    {
        return forcaAnterior;
    }
    else if (valorCelula == 0)
    {
        return forcaAnterior + tropa->forcaDescanso;
    }
    else if (valorCelula > 0)
    {
        return forcaAnterior - valorCelula;
    }
    return -1;
}
```

Figura 3: Função calcularNovaForca.

Após preencher toda a matriz, a função imprimirCaminho() (figura 4) é responsável por reconstruir o percurso ótimo. Ela identifica a célula de destino com maior força e percorre o caminho inverso até o início, armazenando as coordenadas em um vetor de estruturas do tipo Ponto.

O caminho final é então exibido no formato “linha coluna”, seguido da mensagem apropriada de acordo com o resultado obtido.

```c
void imprimirCaminho(EstruturaPD *pd)
{
    int h = pd->linhas;
    int w = pd->colunas;

    int forcaFinal = -1;
    int linhaFinal = -1;
    int tempoFinal = -1;

    for (int i = 0; i < h; i++)
    {
        if (pd->melhorForca[0][i][w - 1] > forcaFinal)
        {
            forcaFinal = pd->melhorForca[0][i][w - 1];
            linhaFinal = i;
            tempoFinal = 0;
        }
        if (pd->melhorForca[1][i][w - 1] > forcaFinal)
        {
            forcaFinal = pd->melhorForca[1][i][w - 1];
            linhaFinal = i;
            tempoFinal = 1;
        }
    }

    if (forcaFinal <= 0)
    {
        printf("A calamidade de Nikador é inevitável\n");
        return;
    }

    // 3. Reconstruir o caminho
    Ponto *caminho = malloc(w * sizeof(Ponto));
    if (caminho == NULL)
        exit(1);

    int linhaAtual = linhaFinal;
    int tempoAtual = tempoFinal;

    for (int j = w - 1; j >= 0; j--)
    {
        caminho[j].i = linhaAtual;
        caminho[j].j = j;

        if (j == 0)
            break;
```

Figura 4: Função imprimirCaminho.

Outras funções auxiliares utilizadas no programa incluem:

- criarEstruturaPD() – aloca dinamicamente a matriz tridimensional de programação dinâmica (melhorForca).
- liberarEstruturaPD() – libera toda a memória alocada após a execução.
- getValorCelula() – obtém o conteúdo de uma célula específica em um dos mapas.
- getForcaOrigem() – retorna a melhor força obtida na célula de origem.
- max3() – retorna o maior valor entre três possíveis origens de movimento.

Essa implementação permite que o programa encontre o melhor caminho possível de forma eficiente e determinística, garantindo uma solução ótima para o problema proposto.

- ESTRUTURA DE DADOS

A `struct` EstruturaPD (figura 5) possui todas as informações necessárias para o funcionamento do algoritmo de programação dinâmica. Ela armazena as linhas e colunas da matriz, além da matriz tridimensional "melhorForca", em que o primeiro indice representa o tempo, o segundo a linha e o terceiro a coluna.

```
typedef struct
{
    int linhas;
    int colunas;
    int ***melhorForca; // melhorForca[tempo][linha][coluna]

} EstruturaPD;
```

Figura 5: Estrutura "EstruturaPD".

A `struct` Tripulação (figura 6) armazena os valores da força da tripulação e o valor de força recuperada no descanso.

```
typedef struct Tripulacao
{
    int forcaTripulacao; // F
    int forcaDescanso;   // D
} Tripulacao;
```

Figura 6: Estrutura "Tripulação".

A `struct` Mapa (figura 7) armazena os valores da altura e largura do mapa, além de apontadores para os mapas do presente e passado.

```c
typedef struct Mapa
{
    int altura;  // h
    int largura; // w
    int **passadoGrid;
    int **presenteGrid;
} Mapa;
```

Figura 7: Estrutura “Mapa”.

- INTERFACE

A interface é baseada em entrada e saída no terminal, tornando a interação com o usuário simples e direta. Após a execução do programa, o usuário recebe os resultados da execução (figura 8).

```
PS D:\UFV\4º período\PAA\TP2\TP2--PAA> .\trabalho_paa .\mapas\mapa1.txt
----Mapas e Variveis----
h: 5, w: 5, F: 2, D: 2, N: 500
Presente:
 10  -1  -1  -2  -1
 -1  -2   5  -1   6
  5   0  -1   3   5
 -1  -1   3  -1   4
 10   0  -1  -1  -1
Passado:
  5  -1  -1  -2  -1
 -1  -2   0  -1  10
 -1   0  -1   8   5
 -1  -1   3   7  -1
 -1   2  -1   0  -1
A calamidade de Nikador e inevitavel
```

Figura 8: Interface do programa.

COMPILAÇÃO E EXECUÇÃO

Para facilitar a compilação e execução do programa, foi utilizado um arquivo *Makefile.* Para compilar, utiliza-se o seguinte comando:

```
None
make
```

Após isso, o programa é executado seguindo o seguinte formato:

```
None
 .\trabalho_paa .\mapas\NomeDoMapa.txt
```

Na figura 9 é possível ver um exemplo de comando de compilação e execução.

```
PS D:\UFV\4º período\PAA\TP2\TP2--PAA> make
gcc -Wall -g -std=c99 -Isrc -c src/pDinamica.c -o src/pDinamica.o
gcc -Wall -g -std=c99 -o trabalho_paa main.o src/lerMapa.o src/pDinamica.o
PS D:\UFV\4º período\PAA\TP2\TP2--PAA> .\trabalho_paa .\mapas\mapa1.txt
```

Figura 9: Execução e seleção do arquivo de entrada.

RESULTADOS

Para avaliar o desempenho e comportamento do algoritmo de programação dinâmica, foram realizados testes utilizando diferentes mapas, variando tanto o tamanho das matrizes, quanto a disposição dos elementos (áreas intransponíveis, âncoras temporais, etc).

Durante os testes, observou-se que o programa é capaz de encontrar corretamente o caminho ótimo que maximiza a força final da tripulação, respeitando as restrições de movimentação (apenas para a direita, diagonal superior direita e diagonal inferior direita) e a troca entre os dois tempos (presente e passado) ao atingir uma âncora temporal.

A saída gerada segue o formato definido na especificação do trabalho, exibindo as coordenadas do percurso (linha e coluna) da esquerda para a direita, seguidas da mensagem final, conforme o resultado obtido.

- EXECUÇÃO BEM-SUCEDIDA

  Quando a tripulação consegue chegar até Nikador com força maior que a dele, é exibida a mensagem “A ruina de Nikador e iminente” (figura 10).

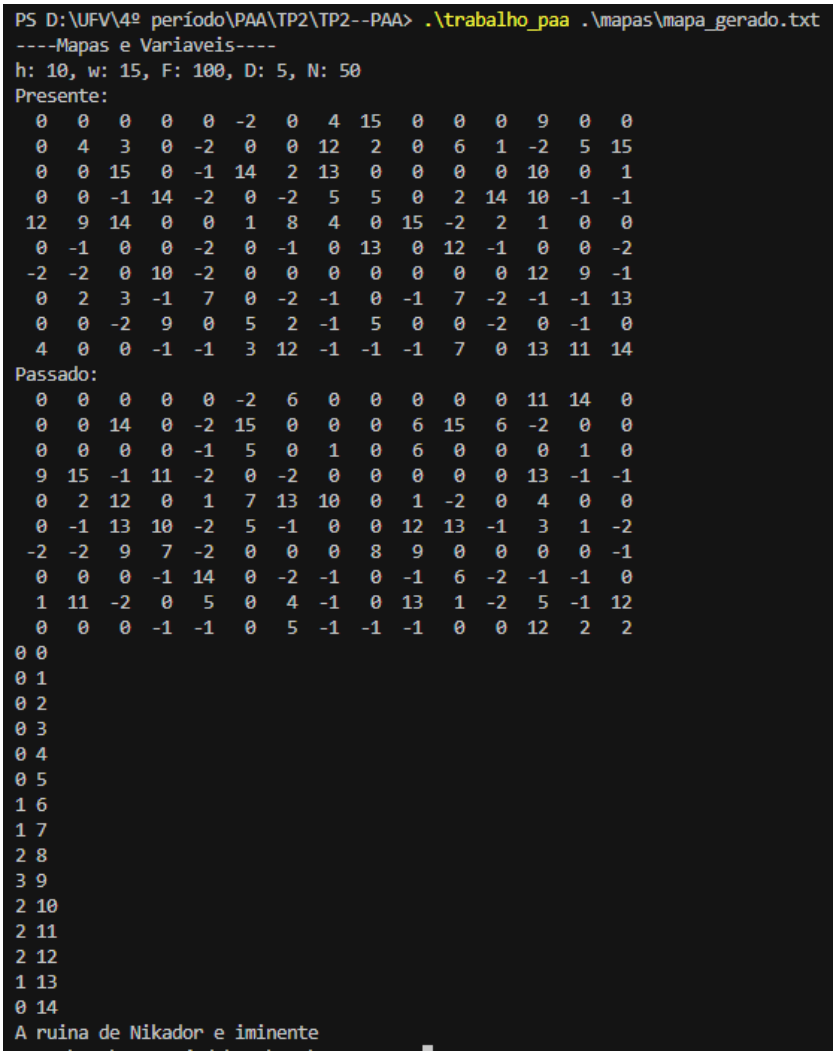

Figura 10: Execução bem sucedida.

- EXECUÇÃO COM FORÇA DA TRIPULAÇÃO ESGOTADA

Quando a força da tripulação se esgota antes de chegar em Nikador, é exibido para o usuário a mensagem: “A calamidade de Nikador e inevitavel” (figura 11).

Figura 11: Execução com força esgotada.

- EXECUÇÃO COM FORÇA MENOR A DE NIKADOR

Quando a tripulação consegue chegar até Nikador, porém sua força é menor que a dele, é exibido na tela: “Sera necessario mais planejamento para parar a calamidade” (figura 12).

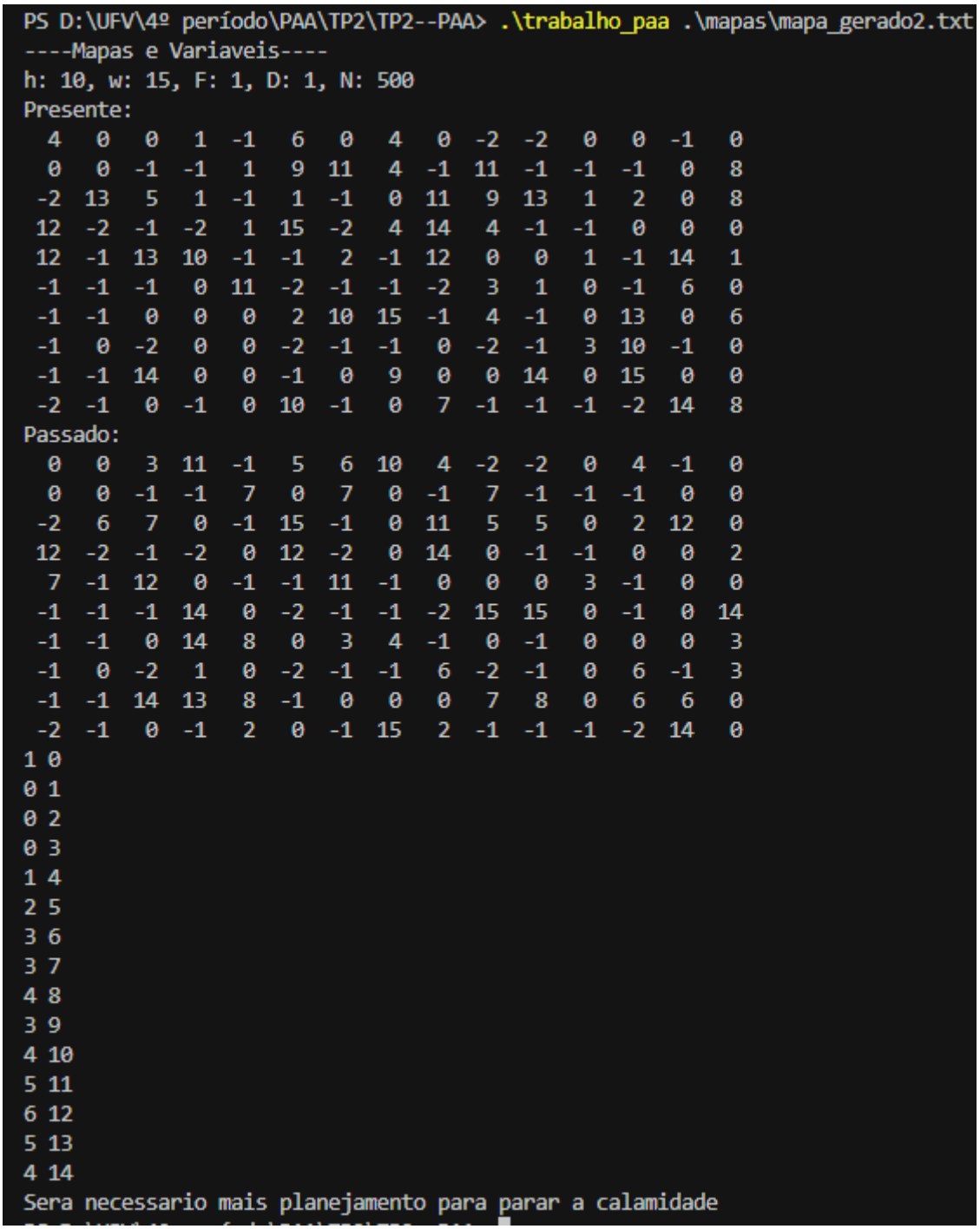

Figura 12: Execução com força insuficiente.

CONCLUSÃO

A implementação do algoritmo baseado em programação dinâmica possibilitou compreender, de forma prática, como esse paradigma pode ser utilizado para resolver problemas de otimização, reduzindo o custo computacional por meio do reaproveitamento de subproblemas já resolvidos.

No contexto do trabalho, o algoritmo mostrou-se eficiente ao determinar o melhor caminho possível para que a tripulação do Expresso Interestelar chegasse até Nikador, equilibrando perdas e ganhos de força ao longo do percurso e lidando corretamente com as âncoras temporais, que permitem alternar entre o passado e o presente.

Os resultados demonstraram que a solução proposta é capaz de identificar com precisão se a jornada é bem-sucedida, apresentando mensagens coerentes com a força final obtida. Além disso, foi possível observar o ganho de desempenho em relação à abordagem do primeiro trabalho, uma vez que a programação dinâmica evita a repetição de cálculos e torna o processo significativamente mais rápido para mapas de maior dimensão.

Conclui-se, portanto, que o trabalho atingiu seus objetivos ao aplicar e consolidar o entendimento sobre programação dinâmica, reforçando a importância de escolher o paradigma de resolução mais adequado ao tipo de problema enfrentado. A atividade também contribuiu para aprimorar a análise de desempenho e o raciocínio lógico necessário para a construção de algoritmos mais eficientes e escaláveis.

REFERÊNCIAS

- W3Schools. Dynamic Programming – DSA Reference. Disponível em: https://www-w3schools-com.translate.goog/dsa/dsa_ref_dynamic_programming.php?_x_tr_sl=en&_x_tr_tl=pt&_x_tr_hl=pt&_x_tr_pto=tc. Acesso em: 06 nov. 2025.