Universidade Federal De São João Del Rei – (UFSJ)
Departamento De Ciência Da Computação
Curso De Ciência Da Computação

Adélson Oliveira do Carmo Júnior
Renan Rodrigues Sousa

Documentação Trabalho Prático 2 – AEDS III

São João del Rei, MG
2023

1. Introdução

Um personagem deve percorrer um grid S, composto por L linhas e C colunas, em busca de um artefato. O personagem deve chegar até o artefato, de modo que sua energia nunca atinja ou fique abaixo de zero durante a jornada. Caso a energia do personagem chegue a zero ou menos, ele falhará na missão.

Quando o personagem se encontra em uma célula S[i][j] com um valor negativo, sua energia é reduzida em um valor igual. Por outro lado, quando ele encontra um valor positivo na célula (i, j), sua energia é aumentada no mesmo valor. O personagem sempre inicia sua jornada na célula (1,1) e o artefato está localizado na célula (L, C). Além disso, o personagem só pode se mover para baixo ou para a direita a partir de uma célula, não sendo permitido sair do grid mágico. A Figura 1 apresenta um exemplo do grid, com as diferentes movimentações possíveis, bem como o objetivo destacado, onde o personagem deve chegar.

Figura 1. Exemplo de grid

O objetivo do problema é determinar qual é a energia mínima necessária para que o personagem inicie sua jornada e consiga manter uma energia positiva ao longo de todo o percurso.

Embora o problema seja um desafio fictício, é possível considerar múltiplas situações do mundo real que possam ser modeladas de forma semelhante. Um exemplo do mundo real é o roteamento logístico, no qual é necessário encontrar o caminho ideal para minimizar a utilização inicial de combustível dos caminhões. Este trabalho apresenta duas abordagens distintas para encontrar a solução ótima: a abordagem baseada em recursão e a abordagem baseada em programação dinâmica.

2. Recursividade

Quando um problema requer a execução repetida de uma determinada ação, é possível utilizar a recursão, que consiste em criar uma função capaz de chamar a si mesma. Essa abordagem simplifica a escrita do código, permitindo a sua implementação em um código pequeno. Esse processo conhecido como recursividade é geralmente considerado o método mais intuitivo para lidar com esse tipo de situação. Isso ocorre devido à natureza da recursão, **que pode realizar cálculos repetidos o que causa um aumento significativo no tempo de execução**, especialmente para entradas grandes.

Contudo, para problemas pequenos, ele é uma boa opção devido a sua implementação simples. Além disso, por ser semelhante a uma abordagem de força bruta, a recursão garante que não haverá um erro quanto ao seu resultado. Em suma, a afirmação “Muitos algoritmos úteis são recursivos em sua estrutura: para resolver um dado problema, eles chamam a si mesmos recursivamente uma ou mais vezes para lidar com subproblemas intimamente relacionados.“ (Cormen, 2012).

3. Programação Dinâmica

A programação dinâmica é um método que busca encontrar a solução geral de um problema, através da análise de uma sequência de subproblemas mais simples em relação ao problema original. Esse método é aplicado quando ocorre sobreposição de subproblemas em uma estrutura recursiva, levando a cálculos repetidos. Os subproblemas menores são processados, os resultados são armazenados e, em seguida, são utilizados para resolver o problema geral. Essa abordagem evita que as soluções sejam recalculadas várias vezes, o que otimiza significativamente o tempo de execução do algoritmo.

Há algumas abordagens na programação dinâmica. Na abordagem “bottom-up”, a solução ótima começa a ser calculada a partir do subproblema mais trivial. Essa estratégia é, de certa forma, uma tradução iterativa de um processo recursivo. Para esse método ser aplicado o problema precisa ter uma estrutura recursiva, pois a solução de cada instância contém a solução de uma subinstância.

4. Tipo Abstrato de Dados

O problema abordado neste trabalho foi tratado usando apenas um tipo abstrato de dado. A TAD Matriz é composta por três atributos: linha e coluna, que são inteiros que representam as dimensões da matriz criada, e o atributo valor, que é um ponteiro de ponteiros. Esse ponteiro armazena a própria matriz, na qual cada espaço contém um valor inteiro. A estrutura pode ser vista na Figura 2.

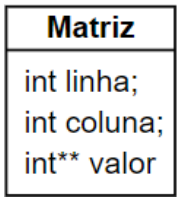

Figura 2. Tipo Abstrato de Dados

Cada valor inteiro presente na matriz representa a quantidade de vida que o personagem ganha ou perde ao se mover para aquela determinada célula.

5. Listagem de Rotinas

Matriz.h

- **Alocação dinâmica da matriz - Matriz alocarMatriz (int linhas, int colunas):** aloca dinamicamente uma matriz de inteiros com dimensão (linhas+1,colunas+1).
- **Criar a matriz de valores - Matriz criarMatriz (FILE *entrada):** essa função lê um arquivo e atribui os valores lidos aos espaços da matriz de valores.
- **Criar a matriz da programação dinâmica - Matriz criarMatrizPD (Matriz matriz):** essa função cria uma matriz com as mesmas dimensões da matriz de valores e inicializa todos os espaços com 0.
- **Liberar as matrizes alocadas - void freeMatriz (Matriz matriz):** libera as matrizes alocadas dinamicamente.

Recursividade.h

- **Calcula a vida mínima por recursão - int vida_minima (Matriz matriz, int i, int j):** testa todos os caminhos e calcula a solução, vida mínima, chamando a si mesma recursivamente.

ProgDinamica.h

- **Função para apenas uma direção - int valorUnico (int valorPD, int valorOriginal):** calcula a vida mínima de uma célula quando o personagem pode se mover apenas para uma direção.
- **Escolhe o melhor caminho - int compararValores (int dir, int abx, int valorMatriz):** calcula a vida mínima de uma célula quando o personagem pode se mover para duas direções. Escolhe o caminho com a menor vida mínima.
- **Calcula a vida mínima - int vidaMinimaPD (Matriz matriz, Matriz matrizPD):** função que calcula a solução, vida mínima, utilizando programação dinâmica.

6. Solução do Problema

Para solucionar o problema, foram adotadas duas abordagens implementadas em linguagem C. Devido à natureza recursiva do problema, optamos por utilizar uma solução baseada em recursão pura. Além disso, como alternativa para otimizar o processo recursivo, empregamos uma estratégia de programação dinâmica. Usando o grid da figura 1, é possível observar todos os caminhos calculados pelo método recursivo, ilustrado na figura 3, com destaque para os cálculos repetidos que a programação dinâmica otimiza.

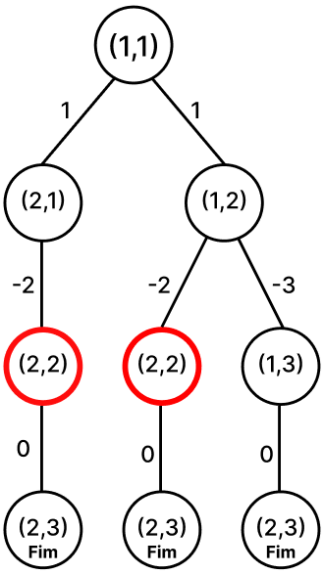

Figura 3. Todos os caminhos possíveis

Para ambas as soluções, é necessário criar uma matriz que represente o grid pelo qual o personagem irá percorrer. É fornecido um arquivo de entrada que apresenta a seguinte estrutura: na primeira linha há um número inteiro que representa a quantidade de grids que serão testados. Na próxima linha estão os valores da dimensão do primeiro grid (L,C). Nas próximas L linhas, cada linha contém C números, representando os valores de cada célula da matriz. Os dados são lidos e a matriz criada.

No momento da alocação da matriz, é alocada uma linha e uma coluna a mais. Esses espaços adicionais são preenchidos com zeros e nunca são acessados. A única

função desses espaços extras é permitir que o personagem comece na célula (1,1) da matriz.

6.1. Solução 1 - Recursividade

Conforme mencionado anteriormente, a recursão é a abordagem inicial que vem à mente ao lidar com um problema dessa natureza. Sabendo que a navegação segue na direção da esquerda para a direita ou de cima para baixo, é necessário mapear todas as rotas possíveis. Sendo assim, a função deve ser chamada para o elemento à direita da célula em análise e, em seguida, para o elemento abaixo dele, quando possível, como mostrado na figura 4. Dessa forma, podemos garantir que todos os caminhos são explorados e que o caminho com a menor perda de vida seja escolhido como o principal.

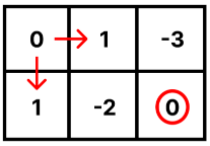

Figura 4. Movimentos possíveis na recursão

Primeiramente, é importante considerar que estamos buscando um valor que seja capaz de sobreviver até o final. Portanto, quando encontrarmos um número negativo que faça o saldo atual ficar igual ou abaixo de zero, devemos inverter o valor do saldo. Para simplificar essa etapa, podemos inverter todos os valores da matriz, de modo que que o resultado final não precise de modificações adicionais. É necessário ainda, que o valor inicial seja 1, pois é o menor possível. A figura 5 exemplifica bem esse processo:

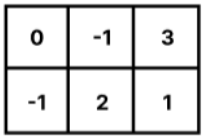

Figura 5. Exemplo da modificação dos valores da matriz

Em seguida, é realizada uma comparação entre os dois caminhos possíveis: o caminho para a direita e o caminho para baixo. Escolhemos o menor valor entre os dois, em seguida acrescentamos o oposto do valor do elemento naquela posição ao saldo atual, figura 6. Após essa etapa, se o saldo atual for maior que zero, significa que é necessário ter aquele valor para sobreviver. Isso ocorre porque, naquele caminho, existe um valor negativo alto o suficiente que poderia "matar" o personagem se não houvesse essa vida mínima.

Figura 6. Escolha do menor elemento para ser o caminho até a célula

Se a vida mínima atual for menor ou igual a zero, significa que, ao longo daquele caminho, não há necessidade de utilizar um valor maior que 1 para garantir a sobrevivência. Nesse caso, podemos utilizar a menor vida mínima possível, que é 1. Esse processo, exemplificado pela imagem x, será repetido para os outros espaços da matriz até chegar à posição inicial. Em seguida, a resposta será retornada para o sistema e escrita no arquivo de saída.

Figura 7. Exemplo de comportamento da matriz para números positivos

Todo processo recursivo necessita de um critério de parada ou ele executará indefinidamente. No problema tratado quando a posição atual das colunas for igual ao número de colunas total, quer dizer que há apenas um caminho possível, para baixo, já que se encontra na borda à direita e não há mais elementos a partir dela. O mesmo pode ser dito quanto à posição atual das linhas for igual ao total de linhas, o que corresponde à borda inferior, sendo o único caminho possível, para a direita. Como consequência disso, é necessário atribuir um valor a essa posição inexistente que não afete os cálculos. Nesse caso, opta-se por utilizar um número suficientemente grande, que corresponde a um número negativo extremamente pequeno considerando a inversão dos valores dos elementos. Desta forma o personagem nunca sai do grid.

6.2. Solução 2 - Programação Dinâmica abordagem “bottom-up”

Como já mencionado anteriormente, a abordagem “bottom-up” consiste em calcular as soluções mais simples primeiro. No contexto desse problema específico, isso significa calcular a vida mínima para o ponto de destino primeiro e, em seguida, calcular para os espaços adjacentes que podem alcançar aquele espaço usando o valor já calculado. Esse processo é repetido até chegarmos ao valor do espaço (1,1), que representa a solução do problema que estamos procurando.

No início da solução, é realizado o processo já mencionado para criar a matriz de valores lendo o arquivo de entrada. Isso é feito utilizando a função "criarMatriz", que irá gerar o grid pelo qual o personagem deve se mover. Em seguida, um processo semelhante é executado para criar outra matriz com as mesmas dimensões da matriz de valores. Esse processo é realizado utilizando a função "criarMatrizPD". Nessa nova matriz, chamada de matrizPD, serão armazenados os resultados dos cálculos de vida mínima para cada célula durante a programação dinâmica. A matrizPD é inicializada com o valor 0 em todos os espaços. Um exemplo da matriz de valores pode ser visualizado na **Figura 1** e da matrizPD pode ser visualizado na **Figura 8**.

0	0	0
0	0	0

Figura 8. Exemplo de matrizPD inicial

Para realizar o cálculo da solução, é utilizada a função "vidaMinimaPD", que recebe como parâmetros a matriz de valores e a matrizPD. Dois loops *for* aninhados são utilizados para percorrer ambas as matrizes e realizar os cálculos de vida mínima. Conforme mencionado anteriormente, começamos pelo espaço para o qual desejamos chegar. Como o personagem deve chegar nesse espaço com vida positiva e o custo para entrar nele é sempre 0, a vida mínima para esse espaço será sempre 1. O caso base pode ser visualizado na figura 9.

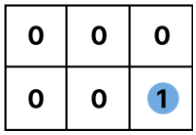

Figura 9. Caso base da matrizPD

Para o restante da matriz, há três casos possíveis: no caso 1, se o espaço está na última linha, o personagem pode se mover apenas para a direita; no caso 2, se o espaço está na última coluna, o personagem pode se mover apenas para baixo; no caso 3, se o espaço tem as duas opções de movimento, o melhor caminho deve ser escolhido.

Nos casos 1 e 2, o cálculo da vida mínima ocorre da mesma maneira. A fórmula utilizada para encontrar a vida mínima é: *VM* = [*VM calculada na posição para onde o personagem se move*] - [*valor de vida ganho ou perdido no espaço atual*]. Essa fórmula, feita na função "valorUnico", leva em consideração o valor previamente calculado na posição para a qual o personagem está se movendo e subtrai o valor de vida que é ganho ou perdido no espaço atual. Caso o valor calculado seja menor ou igual a zero a vida mínima recebe o menor limite possível de 1. Usando o exemplo das figuras 1 e 9 o resultado para esses casos pode ser visto na figura 10.

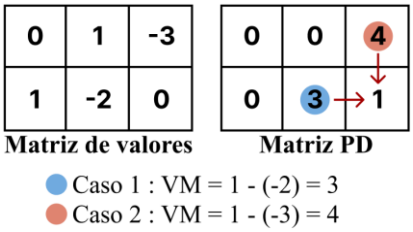

Figura 10. Exemplo do caso 1 e caso 2

No caso 3 a mesma fórmula é usada para calcular para os dois espaços que o personagem pode se mover e como o objetivo é encontrar a vida mínima escolhemos apenas o menor resultado. A figura 11 ilustra o caso 3.

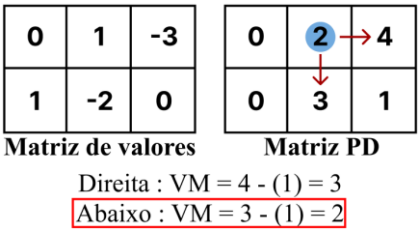

Figura 11. Exemplo do caso 3

Ao final da execução, obtemos uma matriz como pode ser observado na figura 12. O elemento destacado nessa matriz representa a solução final do problema, ou seja, a vida mínima que o personagem precisa ter para alcançar o destino com vida positiva durante todo o percurso.

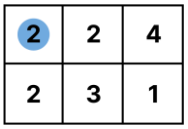

Figura 12. MatrizPD completa – Solução do problema

A solução é impressa no arquivo de saída. Esse processo é repetido para todos os grids no arquivo de entrada. Após a solução de cada grid, as matrizes alocadas são liberadas.

7. Análise de Complexidade

Nesta seção será realizada a análise de complexidade de tempo das soluções propostas nesse trabalho. Serão considerados o tempo de execução das operações mais significativas. As variáveis consideradas na análise são "L" (número de linhas) e "C" (número de colunas), que representam as dimensões da matriz.

- **Função para alocar a matriz - O(L):** Aloca o número de linhas e depois as percorre para alocar as colunas.
- **Função para criar a matriz - O(L*C):** Preenche a matriz.
- **Função para criar a matriz da PD - O(L*C):** Criar uma matriz com as mesmas dimensões da matriz de valores.
- **Função que libera as matrizes - O(L):** Percorre o número de linhas, liberando as colunas e em seguida libera as linhas.
- **Função que retorna o valor único - O(**1**):** Apenas executa uma comparação, sem repetir o processo.
- **Função que compara os valores - O(**1**):** Apenas executa duas comparações, sem repetir o processo.

7.1. Análise de complexidade da solução por recursão

A solução por recursão é composta pelas funções criarMatriz e freeMatriz além da principal que será analisada mais a fundo: vida_minima.

Tendo em vista o quão difícil é encontrar a equação de recorrência, é possível contornar essa situação e encontrar diretamente a função que domina assintoticamente esse método. Considerando que há duas direções possíveis, para direita ou para baixo, cada célula da matriz tem 2 candidatos em potencial como caminhos, como mostrado na imagem 3, isso estabelece uma relação exponencial 2 elevado a um valor. Pelos motivos citados e desconsiderando as interações que chegarem à borda, já que está sendo buscado um valor aproximado, assume-se que a complexidade dessa função é $O(2^{L+C})$, já que L+C corresponde à profundidade do maior caminho da matriz.

Quanto às demais funções que estão presentes na solução, olharemos quem é a dominante para a solução geral: $O(2^{L+C}, L * C, L) = O(2^{L+C})$. Portanto, a complexidade final é $\boldsymbol{O(2^{L+C})}$**.**

7.2. Análise de complexidade da solução por programação dinâmica

A função principal do algoritmo, vidaMinimaPD, é composta por dois laços "for", o que nos permite presumir que a complexidade será O(L*C). No entanto, ao analisar mais detalhadamente, há as comparações realizadas pelos "if" e seus respectivos "else". Devido a essas comparações estarem aninhadas, é menos intuitivo identificar o comportamento. As tabelas na figura 13 é usada para observar esse comportamento.

Iteração	Resultado
[1,1]	1
[1,2]	3
[1,3]	5
[1,4]	7
[1,5]	9

Iteração	Resultado
[2,1]	3
[2,2]	8
[2,3]	13
[2,4]	18
[2,5]	23

Iteração	Resultado
[3,1]	5
[3,2]	13
[3,3]	21
[3,4]	29
[3,5]	37

Iteração	Resultado
[4,1]	7
[4,2]	18
[4,3]	29
[4,4]	40
[4,5]	51

Iteração	Resultado
[5,1]	9
[5,2]	23
[5,3]	37
[5,4]	51
[5,5]	65

Figura 13. Tabela de para analisar o comportamento

Observe que quando $L = 1$, a progressão é de 2 em 2, já em $L = 2$ ela passa a ser de 5 em 5. Nas demais iterações temos os seguintes comportamentos: $L = 3$ a progressão é de 8 em 8, $L = 4$ a progressão é de 11 em 11, $L = 5$ a progressão é de 14 em 14. Por conta disso, podemos assumir que à medida que o número de linhas aumenta, as comparações triplicam, e o mesmo pode ser dito para aumento do número de colunas. É possível supor uma possível equação que representa essa determinada conta: $3(L * C) - L - C$.

Prova real: Toma-se as iterações [1,1] ,[3,2] e [4,4] para averiguar a veracidade da suposição:

$[1,1]\ em\ 3(L * C) - L - C = 3(1 * 1) - 1 - 1 = 3 - 1 - 1 = 1$

$[3,2]\ em\ 3(L * C) - L - C = 3(3 * 2) - 3 - 2 = 18 - 3 - 2 = 13$

$[4,4]\ em\ 3(L * C) - L - C = 3(4 * 4) - 4 - 4 = 48 - 4 - 4 = 40$

Todos os valores estão de acordo com os representados na tabela, dessa forma, temos que a complexidade da função da vidaMinimaPD é $O(3(L * C) - L - C)$ ou simplesmente $O(L * C)$, comprovando as suposições feitas no início.

A solução ainda se utiliza das funções criarMatriz, criarMatrizPD e freeMatriz presentes em matriz.c e matriz.h, logo como mostrado anteriormente, tem-se: $O(MAX(L * C\ , L * C, L * C, L))\ O(L * C)$. Portanto a complexidade final para essa solução é $\boldsymbol{O(L * C)}$.

8. Resultados

Para avaliar o programa, foram utilizados diversos grids com diferentes dimensões. Foi possível obter o tempo de usuário e de sistema por meio da função 'getrusage', utilizando as funções contidas no arquivo 'tempo.c'. Esses tempos permitem ilustrar o comportamento assintótico dos métodos utilizados.

Para analisar a mudança no tempo de execução à medida que aumentamos algumas das dimensões da matriz de entrada, optamos por utilizar gráficos tridimensionais. Isso nos permite visualizar de forma mais clara o impacto das dimensões da matriz no tempo de execução. Vale ressaltar que foi utilizado os tempos de usuário já que o tempo de sistema precisa de entradas grandes para aparecer.

Analisando a solução por recursão, foram usadas 37 entradas para plotar o gráfico ilustrado na figura 14. Nesse gráfico, é possível observar que as variações de tempo não são muito significativas quando aumentamos apenas uma das dimensões da matriz. À medida que aumentamos ambas as dimensões da matriz, é observado um aumento rápido nas variações de tempo. Por isso, os testes para matrizes com dimensões maiores que

19x19 começam a se tornar inviáveis. **É possível notar um salto considerável no tempo de execução ao comparar uma matriz de 19x19 com uma de 20x20, o que faz sentido com a complexidade exponencial calculada.**

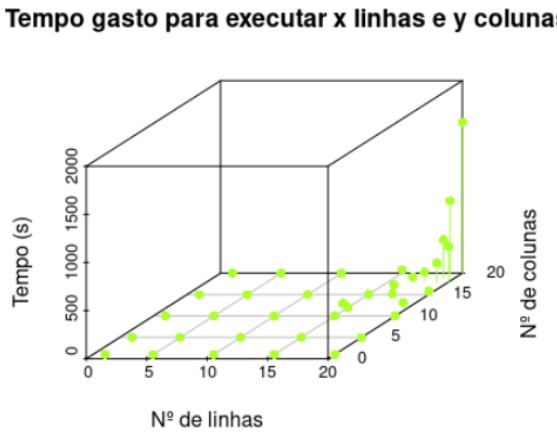

Figura 14. Gráfico de tempo para a recursão

Na solução por programação dinâmica, apenas 25 entradas são suficientes para descrever de forma precisa o comportamento assintótico. Como pode ser observado no gráfico da figura 15, mesmo com uma matriz de 20x20 a resposta é quase instantânea. O impacto do aumento das dimensões da matriz na variação de tempo é semelhante ao da solução recursiva. As variações são significativas quando as duas dimensões aumentam.

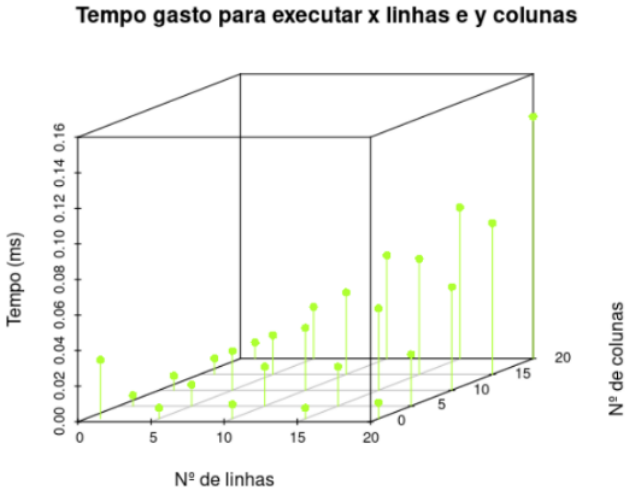

Figura 15. Gráfico de tempo para a programação dinâmica

Para demonstrar a eficiência do método de programação dinâmica, realizamos testes com entradas de até 5000x5000, cujos resultados foram calculados em um tempo quase instantâneo. O gráfico correspondente a essas entradas pode ser observado na Figura 16 e está em alinhado com a complexidade calculada para o método de programação dinâmica. Para poder gerar um gráfico bidimensional consideramos o número de linhas e colunas iguais.

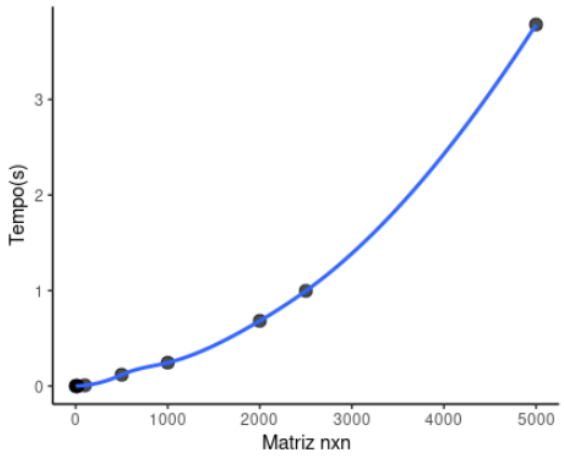

Figura 16. Gráfico para matrizes quadradas na programação dinâmica

Quanto ao tempo de sistema, ele apresenta o mesmo comportamento do tempo de usuário, porém há diversos empecilhos que impedem que muitos testes com ele sejam gerados, tais como tempo de execução para a primeira solução e uma matriz muito grande para a segunda.

9. Conclusão

Com os resultados obtidos e com funções de complexidade, foi possível chegar em algumas conclusões. Conforme ilustrado na seção 8, a abordagem recursiva apresenta uma grande limitação em relação ao tamanho da matriz de entrada. Isso ocorre devido aos cálculos repetitivos e à complexidade de tempo exponencial. Portanto, embora sua implementação seja simples, **a abordagem recursiva é ineficiente e só é viável para entradas pequenas.**

Por outro lado, a abordagem de programação dinâmica se destaca ao apresentar alta eficácia, mesmo para entradas consideravelmente maiores. Em um teste realizado com uma matriz de dimensão 5000 x 5000, a execução da solução por programação dinâmica foi concluída em apenas alguns segundos.

No fim, o trabalho demonstrou de forma esclarecedora como a programação dinâmica pode ser utilizada para transformar um problema com estrutura recursiva em um programa iterativo. Os resultados obtidos mostram uma grande otimização do método recursivo. Isso evidencia que um programa iterativo é consideravelmente mais eficiente do que um programa recursivo.

10. Referências

T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein “ALGORITMOS teoria e prática”, 3ª edição. São Paulo GEN LTC 2012

Victor Matheus R. de Carvalho (2019) “Programação Dinâmica”, Disponível em: https://lamfo-unb.github.io/2019/05/30/Programacao-Dinamica/, abril.