Ciência da Computação

Algoritmos e Estruturas de Dados III

Documentação Trabalho Prático 2 - Harry Potter

Davi dos Reis de Jesus, Gabriel de Paula Meira

2023.1

1 Introdução

O mago mais famoso do planeta, Harry Potter, necessita vencer um jogo que consiste em um tabuleiro onde é preciso percorrê-lo da melhor maneira possível, de forma que obtenha o tesouro que está no fim desse tabuleiro.

Cada posição (i, j) do tabuleiro fornecido com dimensão $(m \times n)$ possui um valor que interfere na vida total do personagem que a percorre. O propósito é conseguir sair do ponto inicial $(1, 1)$ e alcançar o objetivo final (m, n) com o menor custo possível, ou seja, necessitando do mínimo de vida inicial possível para percorrer toda a matriz com movimentos apenas para baixo e para direita. A Figura 1 exemplifica um possível tabuleiro para o problema descrito.

0	7	-2	1	-5	-7
-3	4	-2	5	2	0
3	-8	-8	4	3	6
-2	5	7	-1	-3	-4
9	-3	9	0	6	0

Figura 1: Exemplo de matriz de entrada

Encontrar um caminho de custo mínimo com valores negativos representa um desafio adicional em comparação aos problemas convencionais de busca do menor caminho entre dois pontos. Os algoritmos comumente utilizados, como o de Djikstra, não são adequados para lidar com esse tipo de situação. Além disso, outro empecilho para o problema é a restrição dos movimentos, que deve ser respeitada durante toda a busca.

Foi desenvolvido um programa que possui duas estratégias diferentes para encontrar a solução ao mesmo tempo que devem garantir a legibilidade e manutenibilidade do código, o que pode ser alcançado por meio de boas práticas de programação e documentação adequada, assim como demonstrado adiante.

2 Arquitetando a solução

É necessário, antes de iniciar a programação, planejar o funcionamento do programa em cada uma das etapas, a fim de garantir a qualidade do software do início ao fim do projeto, facilitando o processo de depuração e manutenção.

O programa foi dividido em quatro etapas principais ilustradas pelo fluxograma da Figura 2. Essa divisão proporciona uma visão clara e estruturada do processo, permitindo uma abordagem mais organizada e eficiente na implementação do programa.

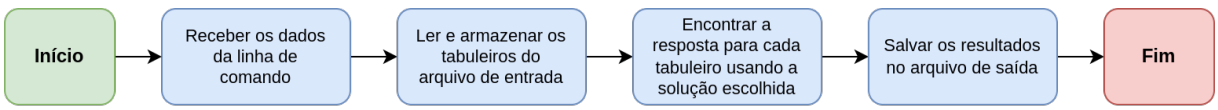

Figura 2: Fluxograma de funcionamento geral

2.1 Receber dados da linha de comando

Como mencionado anteriormente, o programa deve permitir que o usuário escolha a estratégia a ser utilizada para encontrar as soluções de todos os tabuleiros contidos no arquivo de entrada. Esta operação requer o uso dos argumentos da linha de comando assim como no exemplo abaixo.

```
./tp2 {estrategia} {arquivoEntrada}
```

Para receber esses dados é necessário utilizar os parâmetros **argc** e **argv**, recebidos pela função **main()**, verificando se as entradas são válidas para o funcionamento correto do programa.

2.2 Ler tabuleiros do arquivo de entrada

Após obter o endereço do arquivo de entrada, é efetuada a leitura completa do mesmo utilizando a função **fscanf()**, pertencente à biblioteca **stdlib.h**.

A primeira linha contém a quantidade total de casos de teste. O programa deve alocar o espaço necessário para armazenar todos os tabuleiros, cada qual com suas respectivas dimensões, preenchendo os dados de cada posição da matriz.

2.3 Encontrar a solução para cada tabuleiro

Cada tabuleiro é único, não existindo um padrão entre a valoração de suas casas, ou seja, cada um possui sua própria solução. Entretanto, um tabuleiro pode conter caminhos diferentes que possuem soluções em comum, assim como mostra a Figura 3, não sendo possível afirmar qual o melhor, mas sim a vida mínima necessária para concluir o percurso.

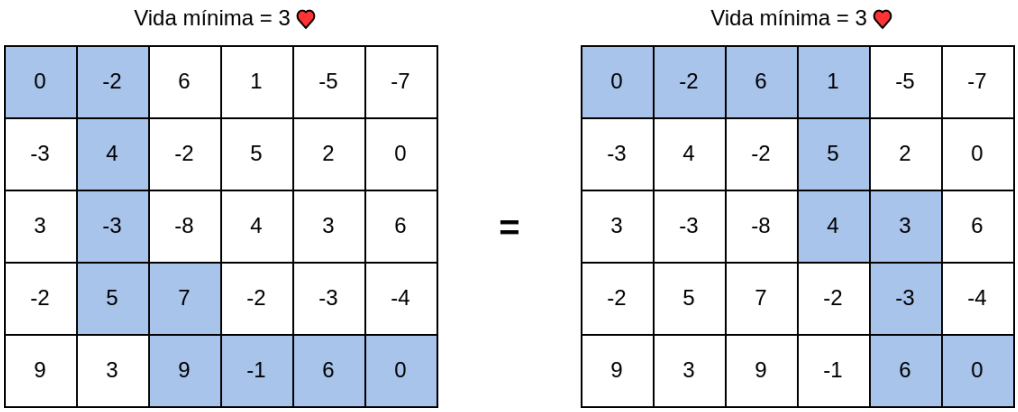

Figura 3: Exemplos de soluções encontradas

O funcionamento das duas estratégias formuladas para resolver o problema é descrito com detalhes na seção 4 da documentação.

2.4 Salvar os resultados no arquivo de saída

A solução encontrada em cada tabuleiro é salva em um arquivo de saída padronizado com o nome “saida.txt”usando a função **fprintf**, também da biblioteca **stdio.h**.

Após a conclusão do programa, todas as estruturas de dados dinamicamente alocadas são liberadas para não ocorrer vazamentos de memória indesejados.

3 Estruturas de dados

Visando garantir uma organização eficiente e uma maior escalabilidade do software, foram criadas estruturas de dados destinadas unicamente para armazenar os tabuleiros. Isso permite que os atributos - dimensões e dados das matrizes - possam ser facilmente acessados e manipulados durante toda a execução do programa.

```c
// boardlib.h

typedef int Energy;

typedef struct {
    Energy energy;
} Square;

typedef struct {
    int rows, columns;
    Square** matrix;
} Board;

typedef struct {
    int length;
    Board* boards;
} BoardsArray;
```

Em conjunto com a implementação dos tipos abstratos de dados, foram desenvolvidos os métodos para a criação (alocação de memória) e exclusão (liberação de memória) dos tabuleiros.

3.1 Modelando um tabuleiro em grafo

Um tabuleiro, apesar de ser uma matriz, para ser solucionado pode ser modelado como um grafo, no qual cada vértice representa uma casa (R, C) e possui arestas direcionadas para os elementos que se encontram uma linha abaixo $(R + 1, C)$ e uma coluna à direita $(R, C + 1)$, assim como mostra a Figura 4.

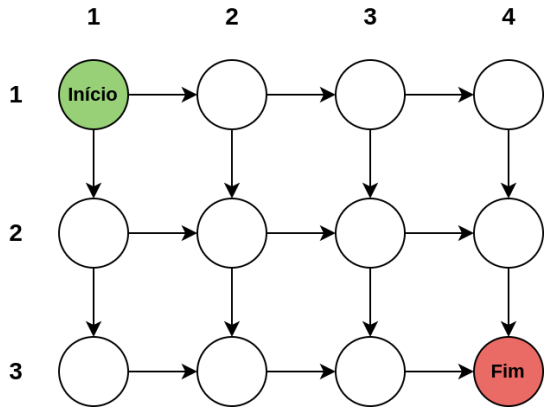

Figura 4: Modelagem da matriz em grafo

Esse método facilita a visualização e permite a implementação de um algoritmo mais eficiente de força bruta, desenvolvido como uma das estratégias para resolver o problema.

4 Estratégias de resolução

Conforme visto em seções anteriores, foram elaboradas duas estratégias distintas para encontrar as soluções dos tabuleiros de entrada: um algoritmo de força bruta (recursivo) e um algoritmo de programação dinâmica (iterativo).

4.1 Força bruta

A estratégia mais simples é testar todos os caminhos possíveis para percorrer a matriz, armazenando a menor vida mínima encontrada durante toda a busca. O que possibilita a implementação eficaz desse algoritmo é a modelagem da matriz em grafo, mencionada na seção anterior.

A função de busca inicia na posição $(1, 1)$ executando as instruções representadas pelo fluxograma da Figura 5. O processo de repete até que a última casa seja encontrada, indicando que o caminho chegou ao fim e uma nova possibilidade de trajeto deve ser testada, retornando ao nível anterior na pilha de execução.

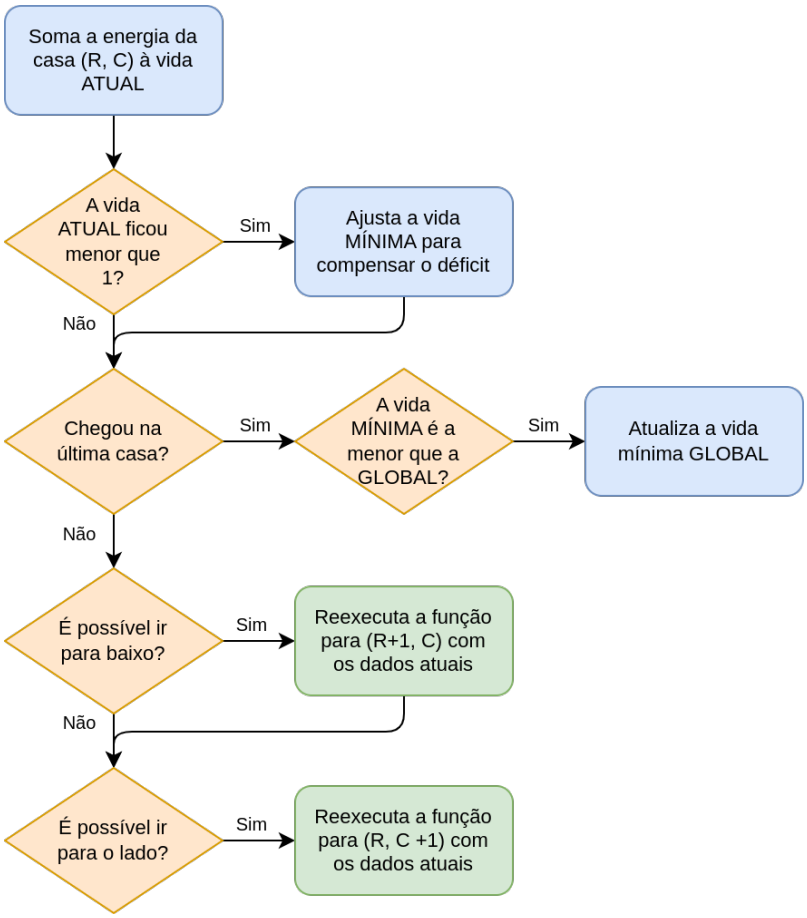

Figura 5: Fluxograma do algoritmo de força bruta

Como é possível observar, o algoritmo possui um funcionamento simples e elegante. Apesar disso, a força bruta se torna ineficiente para matrizes de grandes dimensões, tendo em vista que o número de chamadas recursivas aumenta exponencialmente ao ponto de tornar o algoritmo intratável. A seção 5 traz as análises matemáticas que comprovam as afirmações sobre o desempenho da estratégia.

4.1.1 Poda (*Branch and Bound*)

Uma alternativa para melhorar a execução do algoritmo de força bruta é a utilização de recursos para evitar a análise de caminhos que se distanciam da solução sem que seja preciso prosseguir nas chamadas recursivas que não irão melhorar o resultado.

Existem várias abordagens para realizar podas, como, por exemplo, comparar o caminho em construção com o menor caminho encontrado até o momento, parando a busca toda vez que o caminho supera ou iguala a menor vida mínima já encontrada. Esse processo não altera a ordem de complexidade do algoritmo, apenas adiciona uma comparação.

Outra possível melhoria é combinar um algoritmo guloso à estratégia visando obter uma solução aproximada antes da execução da busca por exaustão, sendo utilizada também como base para comparação com os demais caminhos possíveis durante sua formação. A Figura 6 ilustra o funcionamento de tal algoritmo.

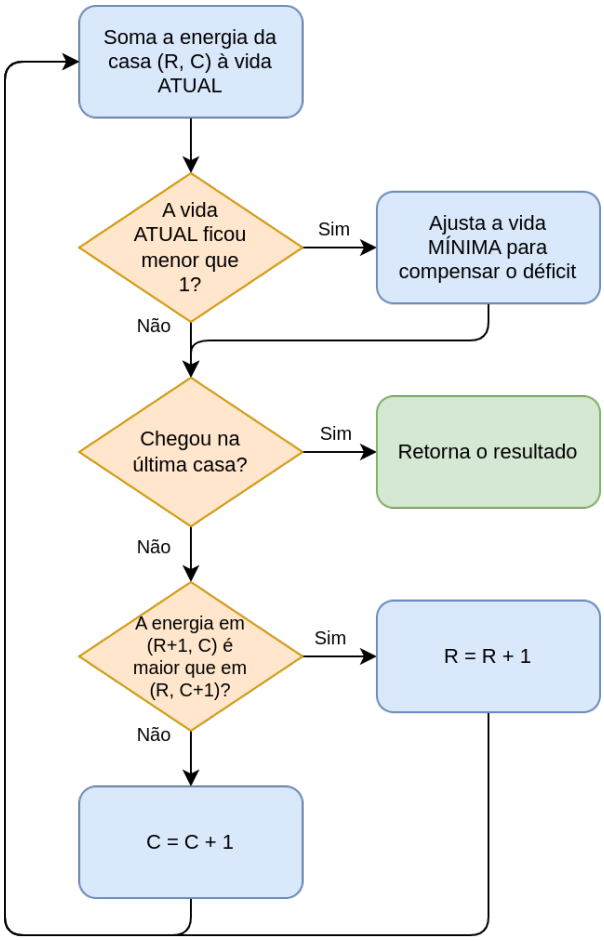

Figura 6: Fluxograma do algoritmo guloso

Apesar de ser uma busca adicional, o custo do algoritmo guloso nessa estratégia é praticamente insignificante, considerando que a complexidade é linear e o número de chamadas recursivas evitadas em um caso médio é bastante considerável.

As ideias apresentadas apenas conseguem reduzir a ineficiência da estratégia, não sendo capazes de melhorar a complexidade, visto que ainda há margem para casos desfavoráveis, muito próximos ao algoritmo de força bruta. Por outro lado, no geral, a melhoria é extremamente significativa, como mostrado na seção 6.

4.2 Programação dinâmica

Um algoritmo de programação dinâmica consiste em dividir o problema em subproblemas menores e solucioná-los em cada iteração, armazenando os resultados obtidos para auxiliar na resolução dos próximos subproblemas e assim conseguir encontrar a solução global ótima para o problema.

A primeira etapa do algoritmo é a criação de uma matriz adicional de mesmas dimensões do tabuleiro a ser resolvido. Essa matriz armazena a vida mínima necessária para concluir o caminho a partir de cada casa, de modo que a solução final seja construída a partir das informações obtidas nas iterações anteriores.

É necessário que o tabuleiro seja percorrido da última casa (m, n) até a primeira $(1, 1)$, armazenando as soluções que tomam por base os dados previamente registrados (casas à direita e abaixo). O fluxograma da Figura 7 ilustra o funcionamento da função que preenche os dados da matriz auxiliar.

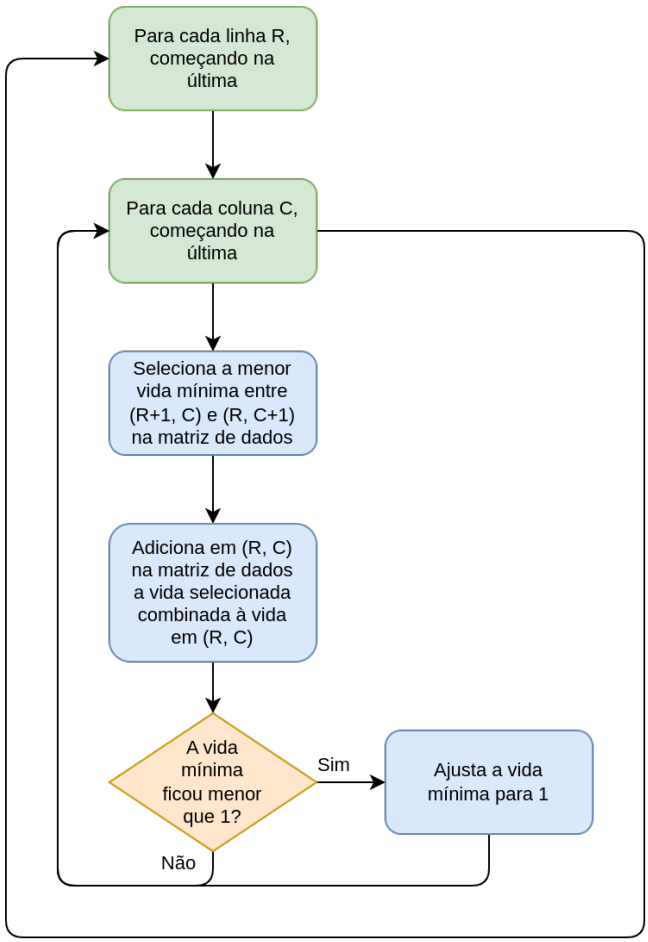

Figura 7: Fluxograma do algoritmo de programação dinâmica

Vale destacar que cada casa é visitada apenas uma vez, não havendo chamadas recursivas que possam impactar no tempo de execução do programa, o que torna essa estratégia mais eficiente que a anterior, conforme mostrado nas próximas seções.

Outro ponto importante é que essa estratégia possibilita uma forma mais simples de obter o caminho em que a vida mínima corresponde à solução encontrada. Para realizar essa tarefa, a matriz de dados auxiliar pode ser percorrida por meio de um algoritmo guloso (ver seção 6.2).

5 Análises matemáticas

Os dois algoritmos apresentados na seção anterior possuem funcionamentos notavelmente diferentes, o que resulta em tempos de execução distintos. Por isso, é importante realizar uma análise matemática detalhada do funcionamento de ambos.

5.1 Análise do algoritmo de força bruta

Tal como foi mencionado anteriormente, o algoritmo realiza chamadas recursivas que testam todas as possibilidades de se chegar à última casa do tabuleiro começando da primeira. Observando atentamente a Figura 8 é possível observar um padrão que resulta na quantidade de maneiras diferentes se percorrer o tabuleiro: cada casa (R, C) possui $(R-1, C) + (R, C-1)$ caminhos distintos partindo da origem.

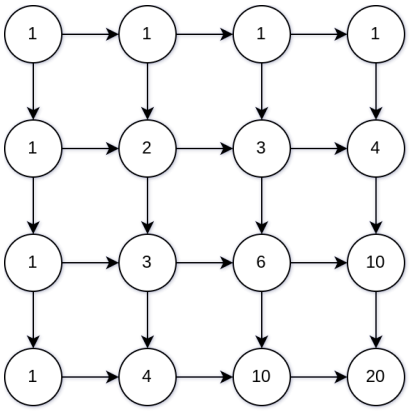

Figura 8: Possibilidades de chegar em uma determinada posição de uma matriz 4x4

Faz-se necessário, portanto, obter a função que fornece essa quantidade. Para percorrer uma matriz $(m \times n)$, todos os caminhos possíveis necessitam de $m-1$ movimentos para baixo e $n-1$ movimentos para a direita. Isso pode ser expresso por:

$$m + n - 2$$

Para calcular o número total de trajetos possíveis, pode-se utilizar a fórmula de coeficiente binomial, definida por:

$$C_{k,p} = \frac{k!}{p! \times (k-p)!}$$

Considerando que k seja $m + n - 2$ (movimentos de cada caminho) e p seja os $m - 1$ ou $n - 1$ movimentos necessários para uma direção, obtém-se a função que define a quantidade de possibilidades de trajetos:

$$f(m, n) = \frac{(m+n-2)!}{(m-1)! \times (n-1)!}$$

Desconsiderando os termos de menor relevância assintótica, é possível identificar a ordem de complexidade da função:

$$f(m, n) = O((m+n)!)$$

5.2 Análise do algoritmo de programação dinâmica

Diferentemente do anterior, o algoritmo de programação dinâmica não necessita que todas as possibilidades sejam testadas; a solução é construída gradativamente conforme cada elemento da matriz é analisado, resultando na seguinte função de complexidade em uma matriz $(m \times n)$:

$$g(m, n) = m \times n$$

A ordem de complexidade da estratégia é facilmente constatada nesse caso, sendo classificada como quadrática.

$$g(m, n) = O(m \times n)$$

5.3 Comparação dos algoritmos

A discrepância na complexidade entre os dois algoritmos é evidente, já que é fácil perceber que expressões fatoriais tendem a crescer significativamente à medida que as entradas aumentam. Para exemplificar a diferença, é possível testar as duas funções com uma matriz (20×20):

$$f(20, 20) = \frac{38!}{19!^2} = 35345263800$$

$$g(20, 20) = 20 \times 20 = 400$$

Além da expressiva diferença dos resultados, é importante destacar que o cálculo para o algoritmo de força bruta não leva em consideração o tempo gasto pelo sistema operacional para realizar as chamadas recursivas, que prejudica ainda mais o tempo real de execução.

6 Testes de Software

Foram realizados testes para avaliar de forma qualitativa os algoritmos desenvolvidos para resolver o problema, considerando sua eficiência nos tempos de execução. O objetivo desses testes foi realizar um estudo detalhado do desempenho de cada algoritmo.

6.1 Monitoramento do tempo de execução

Para monitorar o tempo de execução total de cada algoritmo na busca de soluções, foi utilizada a biblioteca **getrusage.h**, que permite acompanhar especificamente o tempo de CPU, evitando inconsistências na medição decorrentes do uso do tempo físico como parâmetro.

Um conjunto de testes foi realizado empregando matrizes quadradas como casos de teste para facilitar a documentação dos resultados. Os dados temporais obtidos foram utilizados para construir gráficos informativos sobre a eficiência das duas estratégias, que proporcionaram uma análise mais aprofundada das informações obtidas.

6.1.1 Tempo do algoritmo de força bruta

Ao observar o gráfico da Figura 9, que retrata os tempos registrados das execuções do algoritmo de força bruta, é possível constatar que as análises matemáticas realizadas na seção anterior estão corretas pois é perceptível o comportamento exponencial do tempo, aumentando de tal forma que a estratégia se torna inviável.

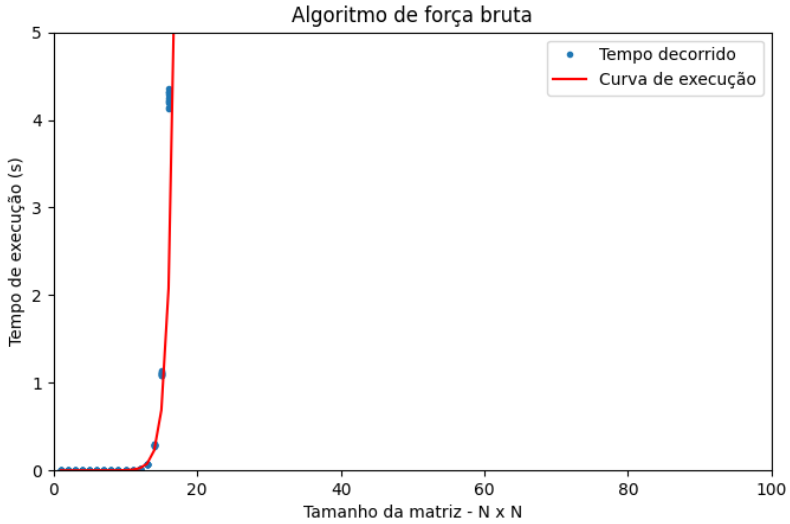

Figura 9: Gráfico dos tempos de execução do algoritmo de força bruta

Por testar todas as possibilidades, o tempo tende a ser próximo para todos os casos de mesmo tamanho. Para matrizes de dimensão até 17 o tempo total é tolerável, não superando 5 segundos decorridos. A partir desse ponto, o aumento se torna grande a ponto de que matrizes de dimensão 20 demorem mais de 900 segundos.

6.1.2 Tempo do algoritmo de programação dinâmica

Em contrapartida, o algoritmo de programação dinâmica, em matrizes de dimensão até 100, não excede 0,175ms para solucionar os tabuleiros, conforme se pode observar na Figura 10. Com isso, pode-se observar uma melhora considerável no tempo gasto para encontrar a solução, mesmo para matrizes de ordem elevada.

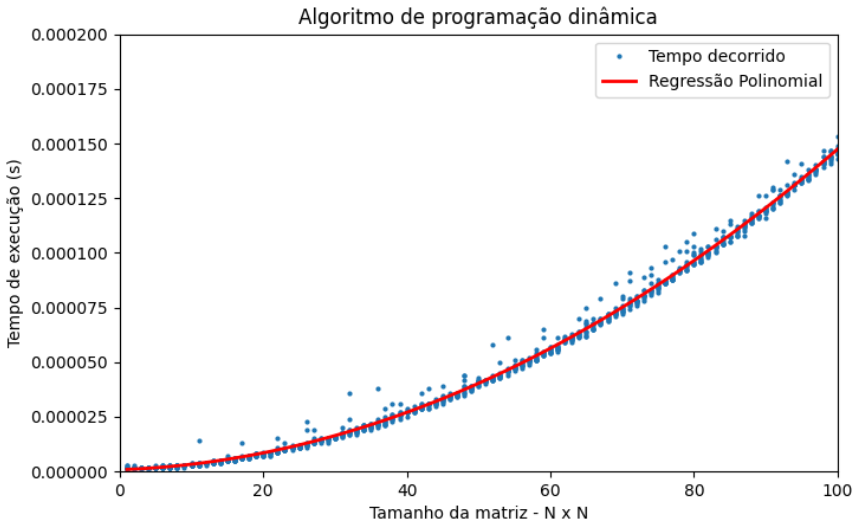

Figura 10: Gráfico dos tempos de execução do algoritmo programação dinâmica

Analisando a regressão linear do gráfico, evidencia-se a constância no funcionamento do algoritmo, de forma que, com o aumento das dimensões das matrizes de entrada, o tempo de execução aumenta de forma quadrática.

Portanto, comparando os resultados obtidos do desempenho de cada algoritmo, pode-se concluir que o algoritmo de busca em exaustão se torna inviável em relação ao de programação dinâmica, dado que há uma diferença considerável no tempo gasto por cada um para encontrar a solução de matrizes de dimensão maior que 20.

6.1.3 Tempo do algoritmo de força bruta com poda

Apesar de possuir um pior caso idêntico à força bruta convencional, a implementação das podas consegue reduzir consideravelmente o tempo na maioria dos casos, viabilizando o algoritmo, como é possível constatar no gráfico da Figura 11.

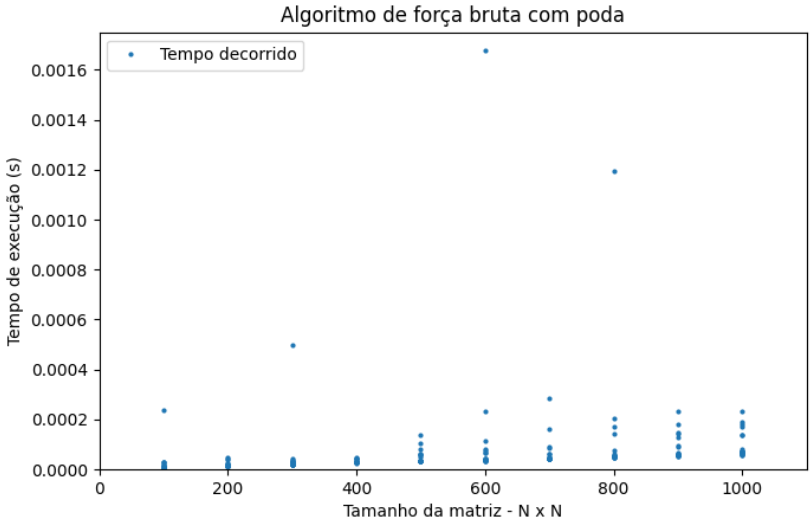

Figura 11: Gráfico dos tempos de execução do algoritmo força bruta com poda

Nota-se que os tempos apresentam instabilidade e imprevisibilidade, com variações significativas em alguns casos de teste, relacionadas à eficiência da poda para a disposição dos elementos na matriz. No entanto, a técnica utilizada permite, em situações favoráveis, a análise de matrizes de grandes dimensões.

6.2 Visualização dos caminhos formados

De nada adianta obter os tempos de execução sem garantir que a resposta final esteja correta, dessa forma, foi desenvolvida uma forma de visualizar graficamente os caminhos formados. É possível armazenar os percursos que levam à melhor solução em ambas as estratégias elaboradas, contudo, o algoritmo de programação dinâmica facilita o processo para encontrá-los, tendo em vista que são geradas matrizes auxiliares com as soluções dos tabuleiros.

Um algoritmo guloso é capaz de percorrer cada matriz auxiliar imprimindo as coordenadas de cada casa com pequenos ajustes para melhor exibição no plano cartesiano. Os pontos gerados podem ser facilmente inseridos na calculadora gráfica online **Desmos**, gerando uma exibição muito semelhante com os tabuleiros de entrada, assim como a demonstrada na Figura 12. Esse recurso possibilitou que o tempo e esforço destinados a criar uma janela de renderização do zero fossem investidos em melhorias para o código principal.

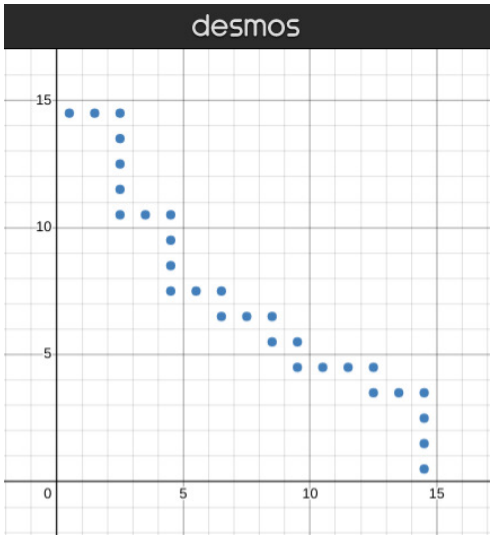

Figura 12: Exemplo de visualização gráfica de uma solução

7 Conclusão

Ao comprovar as análises matemáticas realizadas previamente durante os testes, fica evidente a importância de considerar várias soluções para um mesmo problema. Isso se deve ao fato de que um algoritmo pode enfrentar um caso de execução muito desfavorável, tornando-se inviável devido ao tempo total decorrido para encontrar a resposta.

Para o problema em questão, o desempenho do algoritmo de força bruta é precário para situações em que a ordem da matriz de entrada é maior que 20, conforme exposto na seção 6. Dessa forma, a existência do algoritmo de programação dinâmica fornece a possibilidade de encontrar a solução para matrizes de entrada grandes, ou seja, a obtenção da resposta para o problema não é significativamente prejudicada pelo tamanho da entrada.

Por outro lado, a implementação de podas, apesar de não reduzir a complexidade do algoritmo de força bruta, melhora consideravelmente o tempo de execução do caso médio para um tabuleiro, mesmo em grandes dimensões. Dessa forma, pode-se comprovar a importância de se conhecer os paradigmas de programação e técnicas de otimização, dado que, com base neles, é possível elaborar soluções muito mais eficientes para problemas já conhecidos.

Em todas as áreas do desenvolvimento de software, garantir a rapidez e eficiência dos algoritmos é uma tarefa crucial de todo programador. A redução do tempo de processamento não é importante apenas para melhorar a experiência do usuário com a aplicação, mas também pode reduzir significativamente os custos de operação. Quanto mais eficiente um algoritmo, menor é a quantidade de recursos necessários para executá-lo.

Referências

[1] Thomas H. Cormen. *Algoritmos: Teoria e prática*. LTC, 2012.

[2] Orunmila. Modular code and how to structure an embedded C project. https://www.microforum.cc/blogs/entry/46-modular-code-and-how-to-structure-an-embedded-c-project/, 2019.

[3] Jayme L. Szwarcfiter and Lilian Markenzon. *Estruturas de Dados e Seus Algoritmos 3ª edição*, volume 53. GEN, 2015.