Projeto e Análise de Algoritmos
Trabalho Prático 1 - Dama do Vovô

Felipe Caldas de Paiva

Abril de 2025

1 Introdução

Este trabalho tem como objetivo principal aplicar os conhecimentos adquiridos na disciplina de Projeto e Análise de Algoritmos para resolver de forma computacional um problema envolvendo o tradicional Jogo de Damas.

Dado um tabuleiro retangular e a disposição das peças em um pré-determinado estado do jogo, busca-se encontrar o número máximo de capturas possíveis a partir de uma peça aliada em uma só jogada, seguindo, obviamente, as regras do jogo.

Apesar de aparentar ser simples, tal problema envolve uso significativo de recursos computacionais, como recursão, otimização com programação dinâmica e avaliações de desempenho; logo, foram usadas duas estratégias diferentes a fim de comparar resultados, sendo essas:

- Força Bruta : *Backtracking* exaustivo de cada sequência de capturas
- Programação Dinâmica: memoriza resultados de subproblemas para evitar recomputação.

2 Desenvolvimento

2.1 Fluxo do Programa

Primeiramente, foi necessário arquitetar o esboço da solução e definir qual seria o fluxo de execução do programa, no intuito de facilitar o entendimento do projeto e organizar de forma clara as etapas de execução. Sendo assim, foram definidas etapas, como visto na figura 1 abaixo:

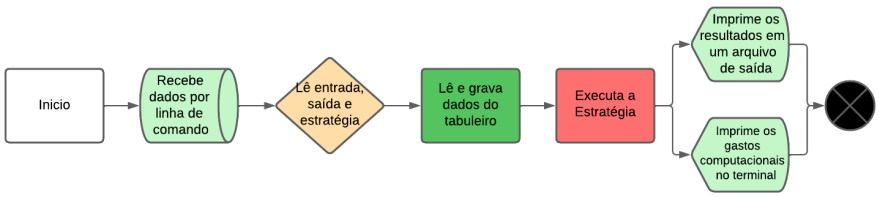

Figure 1: Fluxo de execução do programa

2.2 Estruturas de dados

Foi usada apenas uma estrutura de dados, referente ao Tabuleiro do jogo de Damas, sendo esta capaz de armazenar as dimensões do tabuleiro, além de uma matriz de acordo com a descrição do estado atual do jogo e a tabela de máximo de capturas possíveis a partir de cada posição usada na programação dinâmica.

```
{typedef struct Tabuleiro{
    int N,M;
    int **casas;
    int **tabela; //tabela usada na PD
}tabuleiro;}
```

2.3 Modularização

O projeto foi estruturado de forma que cada módulo ficasse responsável por uma tarefa específica definida pelas etapas acima, permitindo assim clareza na solução e fácil realização de ajustes para casos futuros.

2.3.1 Módulo de entrada e saída

Responsável por receber os argumentos passados pela linha de comando através do modelo exigido:

```
./tp1 {inputPath} {outputPath} {estrategia(1 ou 2)}
```

verificando se são válidos para a execução do programa. Após a execução escreve os resultados em um arquivo de saída.

2.3.2 Módulo do Tabuleiro

Após receber os endereços dos arquivos de entrada e saída e a estratégia escolhida, lê as dimensões do tabuleiro na primeira linha e a disposição das peças **jogáveis** no tabuleiro na linha seguinte até atingir a condição de parada(N = M = 0, sendo N e M as dimensões de um tabuleiro). Além disso, verifica os critérios que definem os limites do problema. Estes são:

- $3 \leq N \leq 20, 3 \leq M \leq 20$
- N $\times$ M ≤ 200
- Cada descrição do tabuleiro possui (N $\times$ M)/2 inteiros
- Cada jogador possui no máximo (N $\times$ M)/4 peças

Não ocorrendo erro nas verificações, armazena a disposição do tabuleiro em uma matriz alocada dinamicamente referente às casas do tabuleiro, em que -1 é atribuído às casas não jogáveis, 0 às casas jogáveis vazias, 1 às casas com peças aliadas e 2 às casas com peças adversárias. Além disso, caso escolhida a estratégia que utiliza Programação Dinâmica, cria uma matriz preenchida por valores -1 para uso futuro na implementação.

2.3.3 Módulo das Estratégias

Neste módulo foram implementadas todas as funções necessárias para a execução das estratégias abordadas:

- **capturar:** recebe a posição de uma peça e uma posição à diagonal desta para verificar a possibilidade de captura de uma peça adversária. Primeiramente checa a existência de uma peça inimiga na posição à diagonal, caso exista, checa se a posição diagonalmente adjacente a ela está vazia. Caso afirmativo, realiza a captura e atualiza o tabuleiro para a nova disposição.
- **maxCapturasPara1:** função genérica que recebe o tabuleiro e um ponteiro para a função correspondente à estratégia escolhida para execução. Percorre a matriz referente ao tabuleiro e para cada ocorrência de uma peça aliada, realiza a chamada da função passada como parâmetro e verifica se o número máximo de capturas é maior que o armazenado até então. No fim retorna o máximo de capturas possíveis.
- **maxCapturasFB:** função de força bruta que percorre exaustivamente todas as sequências possíveis de capturas, utilizando *backtracking* recursivo. A cada chamada testa as quatro direções diagonais e chama de forma recursiva a função, restaurando o tabuleiro após o fim de cada chamada.
- **maxCapturasPD:**Utiliza da mesma lógica inicial da Força Bruta, fazendo chamadas recursivas da própria função, porém armazena o máximo de sequências a partir de cada posição em uma matriz auxiliar, garatindo que cada posição seja explorada uma única vez, logo quando chamada para uma posição já explorada, a função apenas retornar o valor já encontrado, sem a necessidade de recalcular.

2.3.4 Módulo de ánalise de resultados

Este módulo utiliza das funções *gettimeofday()* e *getrusage()* a fim de verificar os tempos de execução e memória gasto durante a execução do programa para que estas possam ser comparadas e avaliadas em diferentes cenários.

3 Estratégias abordadas

Foram implementadas duas estratégias diferentes para abordar o problema proposto: Força bruta e Programação Dinâmica. Ambas as estratégias possuem suas vantagens e desvantagens e possibilitaram análises comparativas entre suas complexidades para diferentes descrições do problema.

3.1 Força Bruta

A forma mais simples e direta, a qual checa todas as sequências de capturas para todas as peças aliadas. A figura 3 exemplifica de forma intuitiva o funcionamento desta estratégia:

1. Tenta fazer uma primeira captura
2. Atualiza o tabuleiro e chama a função novamente para outra diagonal da nova posição
3. Após atingir uma posição em que não é possível realizar novas capturas, volta para a posição anterior e restaura o tabuleiro
4. Repete o processo exaustivamente para cada peça aliada no tabuleiro
5. A cada execução, compara o número de capturas feitas com a variável que armazena o valor referente a maior sequência de capturas feitas até o momento e, em caso desta ser maior, atualiza o valor.

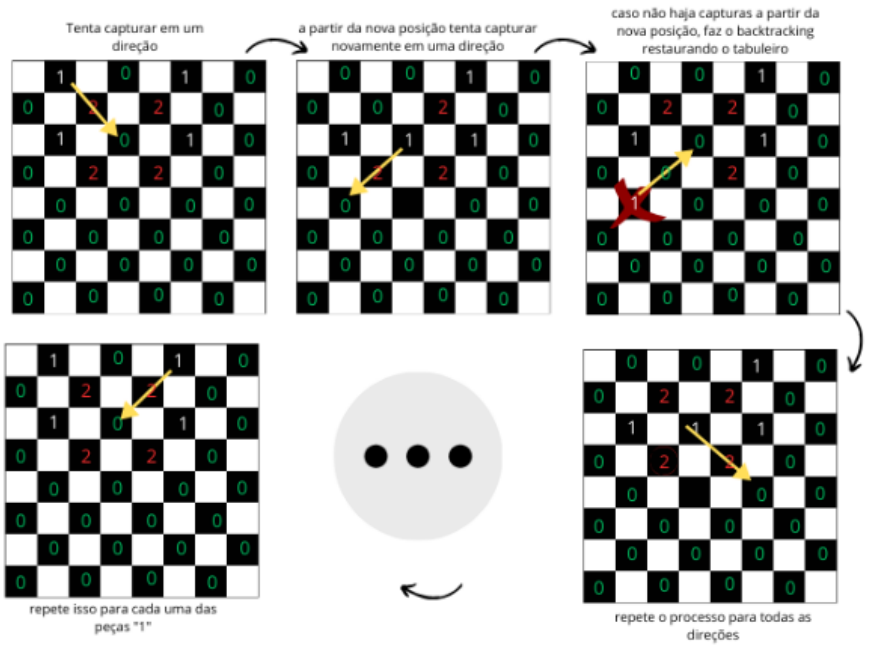

Figure 2: Visualização gráfica do funcionamento da força Bruta

Apesar de ser uma abordagem simples e direta de fácil implementação, acaba sendo prejudicada em situações em que ocorrem múltiplas sequências de capturas por diversas peças, já que para cada nova captura realizada, são exploradas até 4 direções, atribuindo, assim, uma complexidade $O(4^k)$, onde k é o número máximo de capturas sequenciais, aumentando consideravelmente seu gasto computacional.

3.2 Programação Dinâmica

A estratégia de programação dinâmica permite evitar recomputações redundantes, armazenando os resultados de subproblemas previamente calculados. Usando como base o mesmo algoritmo implementado na estratégia de força bruta, foram feitas melhorias em prol da otimização do programa, abordando o conceito de memoization para guardar os valores correspondentes à sequência máxima de capturas a partir de uma determinada posição alcançada, garantindo que ao atingir esta posição o programa possa apenas retornar o valor já salvo na tabela, como pode ser observado na Figura 3. Apesar de possuir, a princípio, a

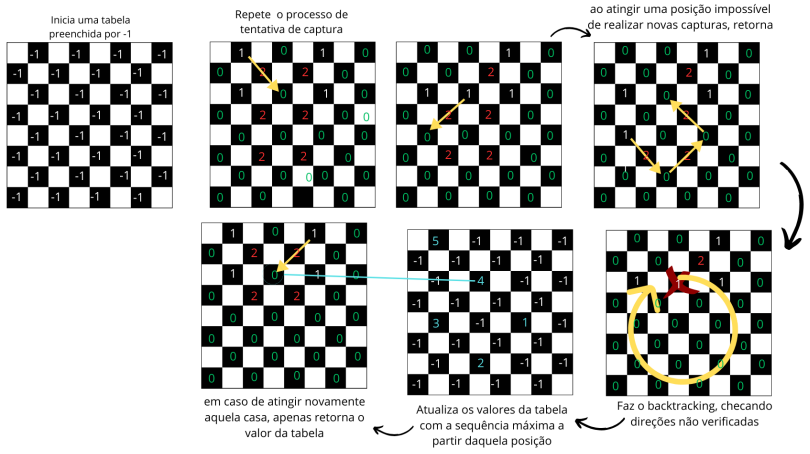

Figure 3: Visualização gráfica do funcionamento da Programação Dinâmica

mesma lógica de verificação da abordagem de força bruta, o armazenamento dos resultados previamente calculados permite grande ganho computacional em relação à outra estratégia, e mesmo com um mínimo aumento no gasto de memória devido à alocação de espaço para a tabela utilizada, o ganho em tempo de execução e a diminuição de chamadas recursivas supera em muito esse fator, permitindo uma complexidade de $O(N \times M \times D)$ mesmo no pior caso, sendo D as possíveis direções de capturas.

4 Análise de Complexidade

A análise da complexidade de algoritmos permite prever a eficiência do programa e avaliar os melhores e piores cenários em que cada estratégia pode ser utilizada.

4.1 Melhor Caso

Considerando um tabuleiro em que não existem peças adversárias, impossibilitando qualquer tipo de captura, os algoritmos se comportam de forma semelhante, tendo cada um apenas que percorrer o tabuleiro inteiro uma vez e realizando verificações rápidas para cada peça aliada encontrada, se existente, garantindo uma complexidade de $O(N \times M)$

4.2 Pior Caso

Entretanto, ao colocarmos ambos os algoritmos em situações de estresse máximo, é possível perceber uma grande superioridade daquele que usa Programação Dinâmica. Considere um tabuleiro 13x13 com 1 peça aliada no centro e peças adversárias ocupando as diagonais de forma a possibilitar inúmeras sequências de capturas possíveis, conforme exibido na figura 4:

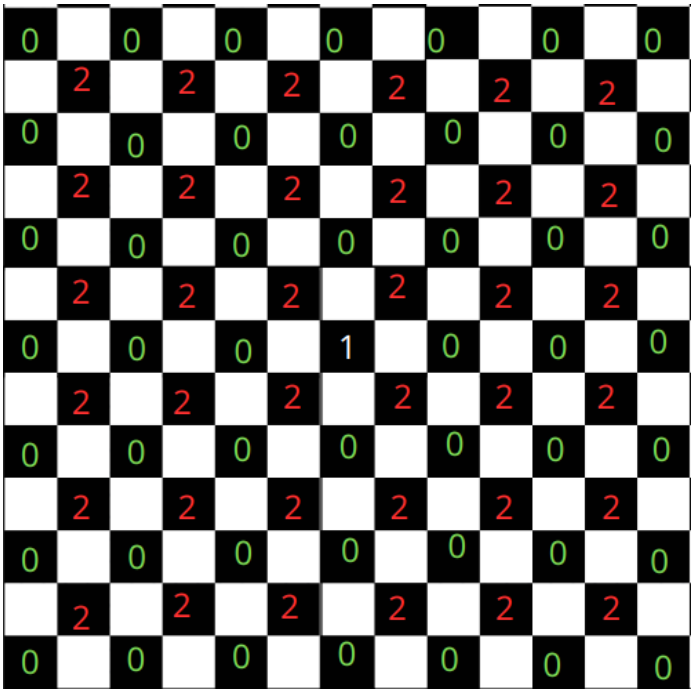

Figure 4: Exemplo de tabuleiro de estresse máximo

Para tal cenário, há aumentos significativos na complexidade do algoritmo de força bruta, já que a cada posição jogável explorada é necessário verificar outras 4 direções, das quais serão verificadas 4 outras direções para cada uma,

sendo assim, tendo que realizar até

$$\frac{NM}{2}\sum_{i=1}^{k} 4^i$$

chamadas recursivas, onde k seria o nível de profundidade atingido durante a sequência de capturas. Com isso o algoritmo de força bruta atinge uma complexidade exponencial de $O(4^k)$.

Enquanto isso, para o algoritmo de Programação Dinâmica, mesmo para o pior caso, o algoritmo ainda precisa apenas percorrer cada posição uma única vez para até 4 direções, evitando a recomputação repetitiva, garantindo uma complexidade de *O(N × M x 4)*. Para efeito de comparação, usando de exemplo um tabuleiro 13x13 com possíveis 10 capturas sequenciais, encontra-se os seguintes resultados:

- Força bruta:

$$\frac{13 \times 13}{2} \times \sum_{i=0}^{10} 4^i = \frac{4^{11} - 1}{3} = \frac{4194304 - 1}{3} \approx 118.139.534 \; chamadas$$

- Programação Dinâmica:

$$\frac{13 \times 13}{2} \times 4 = 676 \; chamadas$$

5 Comparação de Resultados

Foram realizados testes durante todo o desenvolvimento do projeto, avaliando desde as funções mais básicas até as estratégias implementadas.

Devido à natureza do problema, são possíveis infinitas entradas diferentes para a execução, logo, para melhor clareza e entendimento, os testes serão classificados em níveis de estresse, tendo em vista os seguintes critérios:

- Estresse mínimo: casos em que o tabuleiro se encontra vazio ou são realizadas poucas ou nenhuma captura, se fazendo trivial o número de chamadas do programa.
- Estresse médio: cenários em que ocorrem algumas sequências simples de capturas, sem muito nível de profundidade
- Estresse máximo: cenário com tabuleiro distribuído de forma densa, com várias sequências de captura e nível de profundidade alto para cada sequência (ex.: tabuleiro 13 × 13 da figura 4)

5.1 Métodos de Análise

Para poder realizar uma análise concisa e bem organizada, foram usadas as funções *gettimeofday* e *getrusage* a fim de checar os tempos de execução e uso de memória de cada estratégia. Os valores foram salvos em um arquivo *.txt* que foi usado para criar uma tabela e, a partir desta, um gráfico para melhor visualização. Todos os testes foram realizados em um sistema *Ubuntu 22.04 (64-bits), Processador Intel(R) Core(TM) i5-9300H CPU @ 2.40GHz(4 nucleos, 8 threads), 32GB de RAM*

5.2 Análise do tempos de execução

Níveis de Estresse	Força Bruta(ms)	Prog. Dinâmica(ms)
1	0	0
2	0.0001	0
3	0.0002	0
4	0.0005	0.0001
5	0.001	0.0001
6	0.0025	0.0001
7	0.005	0.0002
8	0.012	0.0003
9	0.025	0.0004
10	0.118	0.0011

Tabela 1: Tempos de execução para diferentes níveis de estresse

Como já comentado anteriormente e agora devidamente observado na tabela 1, os tempos de execução crescem significativamente mais rápido para o algoritmo de força bruta, chegando a ser 10 vezes maior que o tempo de execução da estratégia otimizada no nível de estresse mais alto verificado. Entretanto, para níveis mais baixos de estresse, em que, por exemplo, os tabuleiros foram mais esparsos ou houveram poucas capturas em sequência, os valores de ambos os algoritmos se aproximam, existindo uma diferença trivial entre eles.

Também é possível observar o comportamento das funções através de um gráfico de curvas, que deixa evidente o crescimento exponencial do algoritmo de força bruta implementado.

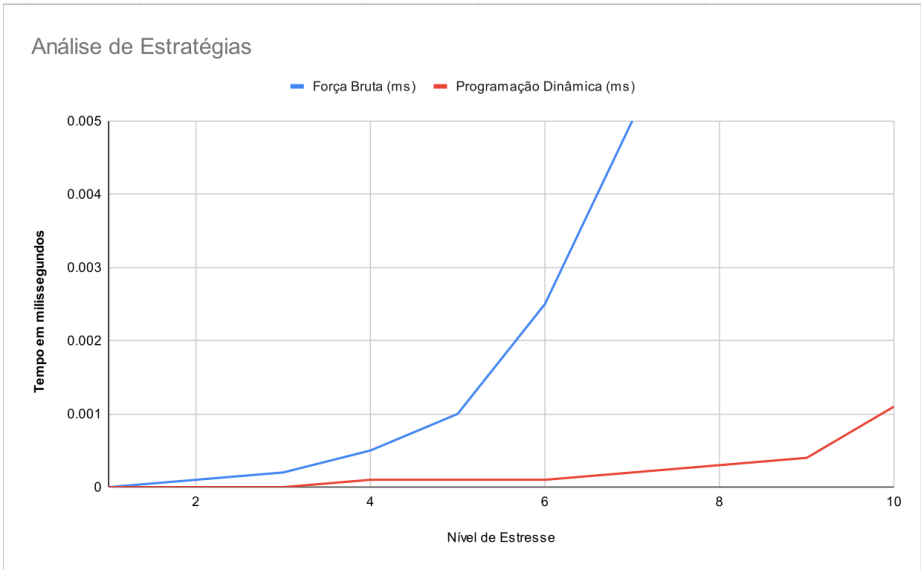

Figure 5: Gráfico de curvas relativo ao desempenho dos algoritmos

6 Conclusão

Com os resultados obtidos nos testes realizados com as distintas estratégias é possível evidenciar vantagens e desvantagens de cada método utilizado. Inicialmente, analisando a estratégia de força bruta, esta possui fácil implementação e funciona de forma intuitiva. Porém possui uma complexidade de tempo $O(4^k)$, podendo aumentar de forma extremamente rápida seu gasto computacional.

Por outro lado, o algoritmo que usa Programação Dinâmica através dos conceitos de *memoization* se provou mantendo uma complexidade constante mesmo para casos em que seu funcionamento é levado ao limite. Diferente da abordagem simples e direta do Força Bruta, esta estratégia alternativa exige mais conhecimento computacional, vista a necessidade de manipular a tabela para *memoization*. Além disso, possui um consumo minimamente maior de memória, devido à alocação de memória para esta.

Observa-se, também, que para descrições mais simples do tabuleiro, ambos os algoritmos apresentam desempenho semelhante, logo, devido à maior facilidade de implementação, pode ser mais vantajoso usar o Força Bruta caso seja previamente conhecido o caráter do problema.

Conclui-se então que cada estratégia possui suas vantagens e desvantagens, e o desempenho de cada uma pode variar dependendo das características específicas do sistema e do conjunto de dados. Critérios como tamanho, dispersão do tabuleiro,, número de peças afetam diretamente o desempenho de cada algoritmo, sendo necessária análise da situação para devido discernimento sobre a escolha da solução. Por fim, é possível relacionar o problema proposto neste

projeto em diversas áreas da tecnologia, como inteligência artificial para jogos, a qual auxilia na toma de decisões. Além da relação explícita na área de jogos, é possível utilizar os conhecimentos aplicados durante a resolução deste problema em situações como a otimização combinatória para maximizar desempenho e analisar riscos, podendo assim alocar recursos de forma adequada, planejar etapas/rotas de acordo com o problema e prever possíveis interrupções que podem vir a ocorrer.

References

[1] CORMEN, Thomas H.; LEISERSON, Charles E.; RIVEST, Ronald L.; STEIN, Clifford. *Algoritmos: teoria e prática*. 3. ed. Rio de Janeiro: Elsevier, 2012.

[2] SEDGEWICK, Robert; WAYNE, Kevin. *Algoritmos*. 4. ed. São Paulo: Pearson Addison Wesley, 2013.

[3] BRASSARD, Gilles; BRATLEY, Paul. *Fundamentos de algoritmos*. Rio de Janeiro: LTC, 2002.

[4] FOGEL, David B. *Blondie24: jogando no limite da inteligência artificial*. São Paulo: Makron Books, 2003.