



Universidade Federal de São João del Rei
Campus Tancredo Neves - Ciência da Computação

TRABALHO PRÁTICO 1

Oscar Alves Jonson Neto
Geraldo Arthur Detomi

São João del-Rei
2025

Sumário

1. Introdução	3
1.1 Enunciado	3
1.2 Análise de entradas e saídas	3
2. Análise do Problema	3
2.1 Regras personalizadas e Modelagem do tabuleiro	3
2.2 Estratégia da Resolução	4
3. Desenvolvimento do Código	5
3.1 Tabuleiro.c	6
3.2 Strategybacktracking.c	6
3.2 Strategybruteforce.c	7
3.3 Main.c	7
4. Testes	8
4.1 Análise de Complexidade Backtracking	8
4.1.1 Função calcular_maximo_capturas_tabuleiro_backtracking	8
4.1.2 Função calcular_maximo_captura_por_peca	8
4.2.1 Função calcular_maximo_capturas_tabuleiro_brute	9
4.2.2 Função simular_todos_os_caminhos	10
4.2.3 Função calcular_maximo_capturas	10
4.3 Análise de Resultados	10
5. Conclusão	12
6. Referências	12

1. Introdução

Este projeto tem como objetivo apresentar o desenvolvimento do Trabalho Prático 1 da disciplina de Projeto e Análise de Algoritmos, ofertada pelo docente Leonardo Chaves Dutra da Rocha.

1.1 Enunciado

Este programa tem como objetivo determinar, de forma computacional, o número máximo de peças do oponente que podem ser capturadas em um único movimento contínuo no jogo de damas, considerando regras personalizadas e um tabuleiro retangular.

1.2 Análise de entradas e saídas

O programa recebe como entrada dois inteiros N e M , que indicam respectivamente o número de linhas e o número de colunas do tabuleiro, e após isso recebe uma descrição do estado do jogo que consiste de $(N*M)/2$ inteiros, com os seguintes valores: 0 representa uma casa vazia, 1 representa uma de suas peças e 2 representa uma peça inimiga, tendo no máximo $(N*M)/4$ peças de cada jogador no tabuleiro, o final da entrada sendo indicado quando $N = M = 0$. A saída deve ser um inteiro indicando a quantidade máxima de peças do oponente que podem ser capturadas em uma única jogada, além dos tempos de usuário e de sistema que serão usados para comparação.

2. Análise do Problema

2.1 Regras personalizadas e Modelagem do tabuleiro

O problema proposto utiliza uma variação do jogo de damas tradicional, com regras personalizadas. O tabuleiro pode ser retangular, com dimensões variáveis N e M , e as peças podem capturar oponentes em qualquer direção diagonal — para frente ou para trás, capturas múltiplas são possíveis desde que as condições do tabuleiro permitam, e nenhuma peça pode ser capturada mais de uma vez na mesma jogada, segue um exemplo de entrada e de como seria montado esse tabuleiro.

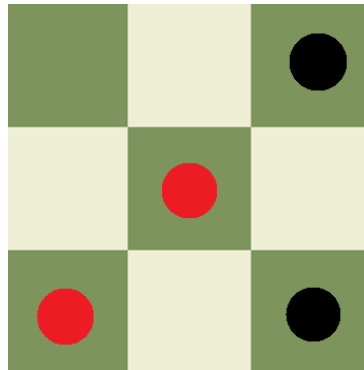
Entrada:

3 3

2 1 2 0 1

Como montar o tabuleiro com essa entrada?

Já sabemos o significado de cada informação da entrada: os dois primeiros números correspondem, respectivamente, a N e M, ou seja, temos um tabuleiro de 3x3. Os valores da segunda linha indicam a posição das peças no momento atual do jogo a ser analisado. Essas peças são posicionadas da esquerda para a direita, seguindo uma orientação à escolha. Neste caso, adotaremos a orientação de baixo para cima. A seguir, apresentamos um exemplo visual da construção do tabuleiro:



Peças vermelhas são oponentes, já as pretas são suas

Agora que temos o tabuleiro montado, precisamos começar a pensar na resolução do problema em si, e qual será a estratégia da resolução.

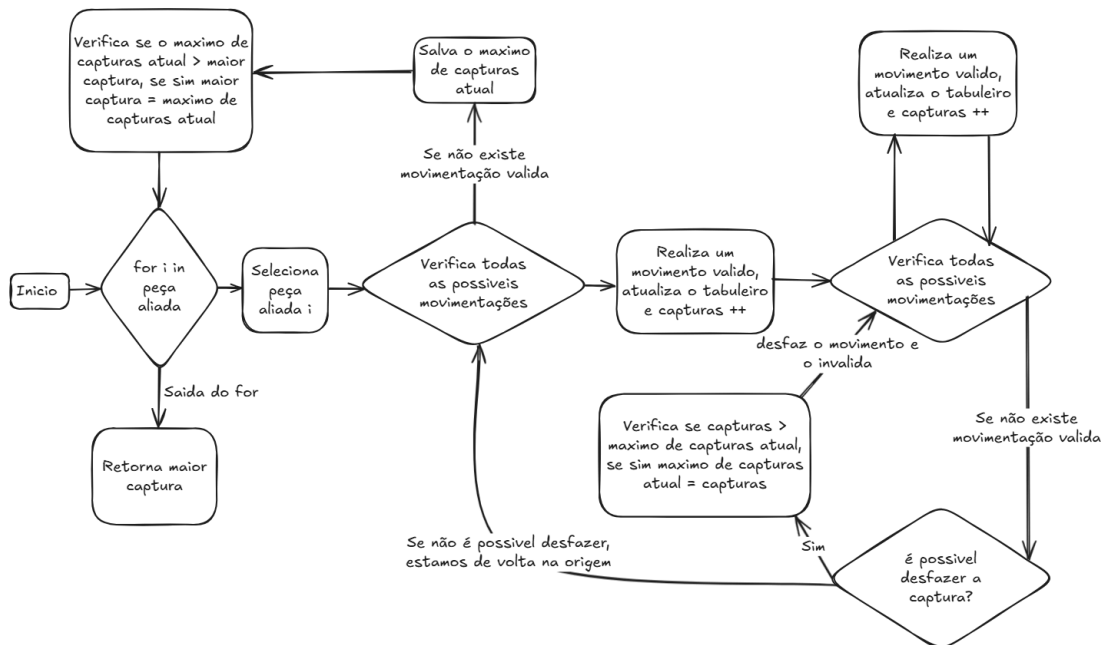
2.2 Estratégia da Resolução

Antes de iniciar a implementação, é fundamental compreender a lógica de resolução do problema e definir a estratégia a ser aplicada. A ideia central consiste em, para cada peça do jogador, explorar todos os movimentos válidos que ela pode realizar. Ao encontrar um movimento possível, a peça é deslocada, o tabuleiro é atualizado, e novamente são verificados os próximos movimentos válidos a partir da nova posição. Esse processo continua recursivamente até que não haja mais movimentos possíveis. A cada caminho percorrido, registra-se a quantidade de capturas realizadas.

Quando não for mais possível continuar a sequência de jogadas, desfaz-se o último movimento, invalidando-o temporariamente, e busca-se outras alternativas. Esse processo se repete até que todas as possibilidades a partir de uma peça sejam esgotadas. Ao final, verifica-se se o número de capturas obtido com aquela peça foi o maior até o momento. Todo esse procedimento é repetido para cada peça do

jogador, e ao final é obtido o maior número possível de capturas em uma única jogada contínua.

Logo abaixo, apresenta-se um diagrama ilustrando visualmente o funcionamento desse raciocínio e a sequência de decisões tomadas durante a resolução do problema.



3. Desenvolvimento do Código

Diante da estratégia de resolução, percebemos que a abordagem mais adequada para explorar todos os caminhos possíveis de captura de peças é o **backtracking**. Essa técnica nos permite simular cada sequência de movimentos válidos de uma peça até o limite possível, retrocedendo sempre que necessário, o que se encaixa perfeitamente no comportamento esperado para resolver o problema apresentado. No enunciado também é requisitado uma maneira de resolução que envolva força bruta, a explicação dessa resolução encontra-se em seu respectivo módulo.

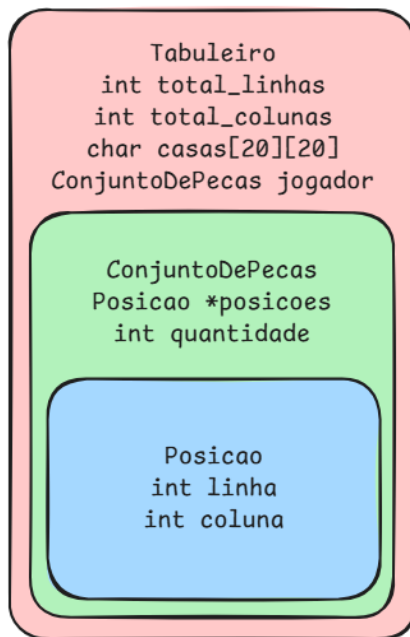
Para facilitar o desenvolvimento e entendimento do código, dividimos o algoritmo em módulos:

- entrada.c
- direcoes.c
- strategybacktracking.c
- strategybruteforce.c
- tabuleiro.c
- tempo.c

- main.c

Mas os módulos principais são `tabuleiro.c`, `strategybacktracking.c`, `strategybruteforce.c` e `main.c`, que vamos analisar agora.

3.1 Tabuleiro.c



O módulo `tabuleiro.c` gerencia a criação, uso e destruição do tabuleiro de damas, com alocação dinâmica de memória para o tabuleiro e as peças do jogador. Ele define três structs interligadas que são demonstradas visualmente ao lado, e usa a função **criar_tabuleiro** para alocar e inicializar o tabuleiro, marcando as casas inválidas com `CASA_BRANCA`. Calcula a quantidade máxima de posições válidas e aloca memória para armazenar as peças. Em caso de falha, libera os recursos alocados. Caso contrário, retorna o tabuleiro pronto para uso. Para a implementação de força bruta também é utilizado as funções **posicao_esta_dentro_tabuleiro** que verifica se

uma posição está dentro dos limites do tabuleiro, e **copiar_casas_tabuleiro** que copia todo o conteúdo do tabuleiro para uma matriz para que não se perca a referência original. E por fim temos a função **destruir_tabuleiro** libera toda a memória e evita vazamentos ao ajustar os ponteiros para `NULL`.

3.2 Strategybacktracking.c

O módulo `Strategybacktracking.c` implementa a lógica “otimizada” para determinar o número máximo de peças do oponente que podem ser capturadas em um único movimento contínuo. Ele define direções de movimento diagonais válidas para capturas, tanto para localizar a peça do oponente quanto a casa vazia logo após ela. A função **atualizar_tabuleiro** altera o estado do tabuleiro ao simular uma captura, enquanto **restaurar_tabuleiro** desfaz essa alteração, permitindo a exploração de outras possibilidades.

E temos a função principal do módulo, **calcular_maximo_captura_por_peca**, que, para uma peça específica, testa todas as direções possíveis de captura, simula os movimentos, contabiliza as capturas e retrocede o estado do tabuleiro para explorar

outros caminhos. Por fim, a função **calcular_maximo_capturas_tabuleiro_backtracking** percorre todas as peças do jogador no tabuleiro e determina qual delas permite a maior sequência de capturas válidas.

3.2 Strategybruteforce.c

O módulo Strategybruteforce.c implementa a lógica “bruta” para determinar o número máximo de peças do oponente que podem ser capturadas em um único movimento contínuo. Temos a função **calcular_maximo_capturas_tabuleiro_brute** que aloca dinamicamente um ponteiro com o tamanho relativo ao máximo de peças inimigas que podem ser capturadas, e então chama a função **simular_todos_os_caminhos** que gera todas as combinações de movimentos possíveis, de forma a capturar todas as peças do oponente, e para cada combinação a função **calcular_maximo_capturas** simula a captura das peças por peça do jogador, seguindo até o fim da combinação, ou parando antes quando um movimento inválido for encontrado.

3.3 Main.c

O código executa a função principal do programa, responsável por ler os dados de entrada, processar as instâncias do jogo e medir o tempo de execução. Inicialmente, os argumentos da linha de comando são validados; em caso de erro, uma mensagem é exibida e a execução é encerrada.

Com os parâmetros válidos, o programa tenta abrir o arquivo de entrada. Se falhar, trata o erro. Um temporizador é iniciado para medir o tempo total de execução. Em seguida, o programa entra em um laço: a cada iteração, são lidas as dimensões de um novo tabuleiro. Caso ambas sejam zero, o laço termina; caso contrário, o tabuleiro é criado dinamicamente com **criar_tabuleiro** e preenchido com **carregar_tabuleiro_arquivo**.

Um segundo temporizador mede o tempo de execução daquela instância. Com base no parâmetro da linha de comando, o programa escolhe entre as estratégias força bruta ou otimizada, chamando **calcular_maximo_capturas_tabuleiro_brute** ou **calcular_maximo_capturas_tabuleiro_backtracking**. O resultado é armazenado em maximo_capturas, impresso na tela junto ao tempo, e também salvo no arquivo de saída. Por fim, o tabuleiro é destruído com **destruir_tabuleiro**, e o processo se

repete até que $N = M = 0$. Ao final, o tempo total é exibido e o arquivo de entrada é fechado.

4. Testes

Agora que entendemos como o programa funciona, precisamos rodá-lo e analisar seus resultados e também sua complexidade.

4.1 Análise de Complexidade Backtracking

Vamos começar analisando a função **calcular_maximo_capturas_tabuleiro_backtracking**, ela percorre todas as peças do jogador, e para cada uma chama recursivamente **calcular_maximo_captura_por_peca**, precisamos ter em mente para essa análise algumas variáveis importantes:

- n = número total de casas no tabuleiro ($linhas * colunas$)
- p = número de peças do jogador ($1 \leq p \leq (linhas * colunas) / 4$)
- d = número de direções possíveis (sempre 4, constante)

4.1.1 Função **calcular_maximo_capturas_tabuleiro_backtracking**

Começando por **calcular_maximo_capturas_tabuleiro_backtracking**, temos que sua complexidade é $O(p)$, pois para cada peça ele irá executar **calcular_maximo_captura_por_peca**.

4.1.2 Função **calcular_maximo_captura_por_peca**

A função é recursiva e, a cada chamada, percorre até 4 direções. Se uma captura for possível, chama-se novamente com a nova posição.

Pior caso: A cada nível, uma peça é capturada e isso pode ocorrer em até 4 direções, com no máximo $n/4$ peças capturáveis .

Complexidade: $O(4^{(n/4)})$

Melhor caso: Nenhuma captura é possível, mas todas as direções são testadas.

Complexidade: $O(4)$

Caso médio: Apenas parte das peças permite sequências de captura. A profundidade da recursão ainda é limitada por $n/4$, mas poucas direções realmente levam a novas chamadas, reduzindo o número de ramificações por nível.

Complexidade: $O(c^{(n/4)})$, com $1 \leq c < 4$, refletindo a média de direções úteis por nível.

Essa complexidade é exponencial, e cresce muito rápido com o aumento do número de peças aliadas no tabuleiro. Na prática, isso limita o uso da abordagem para casos pequenos.

4.2 Análise de Complexidade Força Bruta

Vamos agora analisar a função **calcular_maximo_capturas_tabuleiro_brute**, que busca simular todas as combinações possíveis de movimentos que podem ser realizadas pelas peças do jogador. Para isso, precisamos considerar as seguintes variáveis:

- n = número total de casas no tabuleiro (linhas * colunas)
- p = número de peças do jogador ($1 \leq p \leq n/4$)
- d = número de direções possíveis (sempre 4, constante)
- $PATH_SIZE_MAX$ = número máximo de peças do oponente que podem ser capturadas

4.2.1 Função **calcular_maximo_capturas_tabuleiro_brute**

Essa função é responsável por alocar memória para o caminho de movimentos, inicializar a variável **maximo_capturas** e chamar a função **simular_todos_os_caminhos**, que realiza a geração de todos os caminhos e a simulação para cada um. Como o grosso do trabalho está em **simular_todos_os_caminhos**, a complexidade aqui será dominada por essa função.

Complexidade: $O(1)$

4.2.2 Função **simular_todos_os_caminhos**

Esta é a função responsável por gerar todas as combinações de caminhos possíveis com até $PATH_SIZE_MAX$ movimentos. Para cada posição do vetor

current_path, tenta-se 4 direções, formando uma árvore de chamadas com fator de ramificação 4 e profundidade PATH_SIZE_MAX, Para cada caminho gerado, chama-se **calcular_maximo_capturas**.

Complexidade: $O(4^{PATH_SIZE_MAX})$

4.2.3 Função **calcular_maximo_capturas**

Para cada combinação oferecida, percorre todas as peças do jogador e para cada peça, simula até PATH_SIZE_MAX capturas no caminho atual. Dentro da simulação, a função realiza verificações e cópias de tabuleiro que, apesar de parecerem pesadas, são constantes (dimensão fixa 20x20).

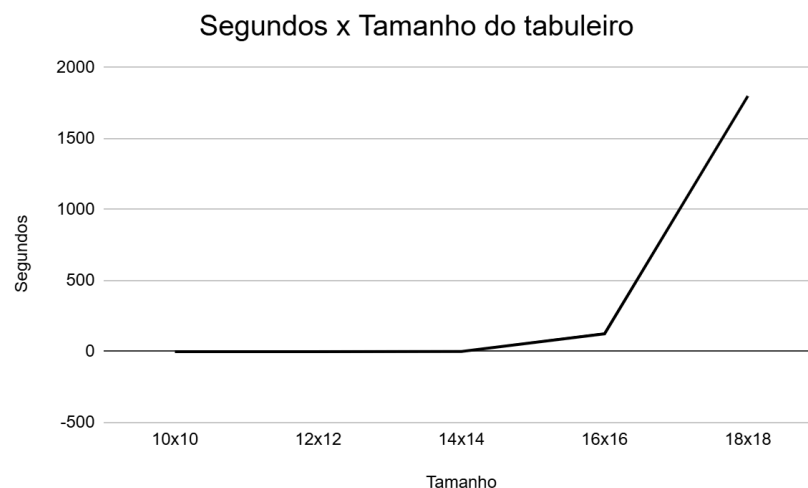
Complexidade: $O(p \cdot PATH_SIZE_MAX)$

Essa complexidade também é exponencial, porém seu crescimento é relativo ao número de peças inimigas. Mas não diminui o fato de que na prática, isso limita o uso da abordagem para casos pequenos.

4.3 Análise de Resultados

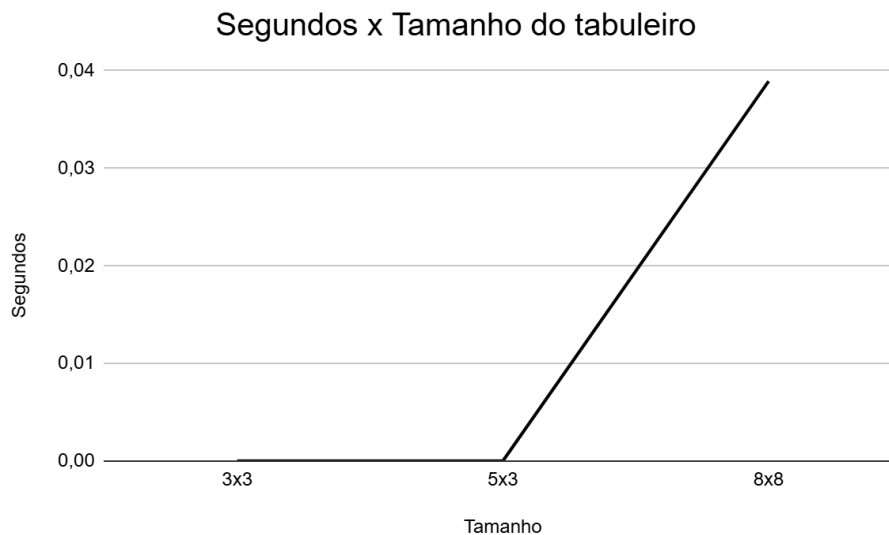
Agora que entendemos a complexidade do código, vamos rodar o mesmo e analisar os resultados, primeiro vamos analisar o método do backtracking:

Criamos tabuleiros de tamanho 10x10, 12x12, 14x14, 16x16 e 18x18, e eles possuem um formato que tendem ao pior caso, e os resultados dos mesmos pode ser observado através do seguinte gráfico:



Como podemos ver, o problema é exponencial sendo que o tabuleiro 18x18, após 30 minutos de execução não havia terminado sua execução, porém com isso sabemos que a execução levaria no mínimo 30 minutos.

Já na análise do força bruta utilizamos as entradas fornecidas pelo enunciado do problema, e os resultados podem ser observados no gráfico abaixo:



Novamente vemos uma curva exponencial, mas como a complexidade da força bruta mesmo sendo exponencial, é mais alta que a do backtracking, seu valor escalona de maneira extremamente mais elevada.

Ou seja, mesmo que sejam valores pequenos, os mesmos representam a mesma curva exponencial dos demais, comprovando com evidências, que no pior caso dos algoritmos sua resolução é **exponencial**.

5. Conclusão


A análise cuidadosa das regras de captura e da estrutura do tabuleiro permitiu a implementação de uma solução eficiente baseada em backtracking. A simulação de todas as sequências possíveis de capturas possibilitou avaliar o comportamento do algoritmo em diferentes cenários.

Apesar da complexidade elevada no pior caso, os limites estabelecidos no enunciado garantem tempos de execução viáveis na prática, mas que se esses limites não existissem, seria necessária uma outra estratégia de resolução.

6. Referências

[CORMEN et al., 2012] CORMEN, T. H., Leiserson, C., Rivest, R., and Stein, C (2012). Algoritmos: teoria e prática. LTC.

BACKES, A. LINGUAGEM C: COMPLETA E DESCOMPLICADA. [s.l.] Elsevier, 2012.

 Força Bruta e Backtracking - LPC I 2021

[Getting getrusage\(\) to measure system time in C - Stack Overflow](#)