



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

<b>1. Introdução</b>	<b>3</b>
1.1 Enunciado	3
1.2 Análise de entradas e saídas	3
<b>2. Análise do Problema</b>	<b>3</b>
2.1 Regras personalizadas e Modelagem do tabuleiro	3
2.2 Estratégia da Resolução	4
<b>3. Desenvolvimento do Código</b>	<b>5</b>
3.1 Tabuleiro.c	6
3.2 Jogo.c	7
3.3 Main.c	7
<b>4. Testes</b>	<b>8</b>
4.1 Análise de Complexidade	8
4.1.1 Função calcular_maximo_capturas_tabuleiro	8
4.1.2 Função calcular_maximo_captura_por_peca	8
4.2 Análise de Resultados	9
<b>5. Conclusão</b>	<b>11</b>
<b>6. Referências</b>	<b>12</b>

# 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.

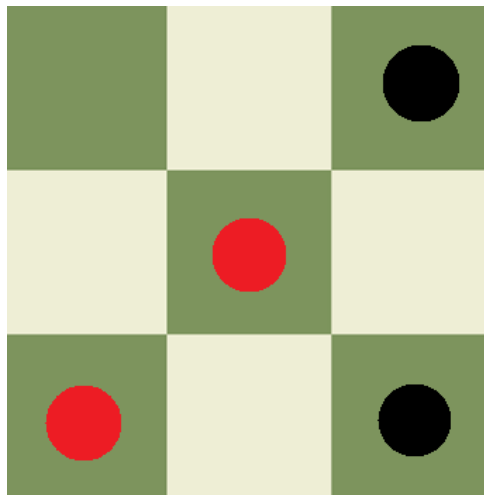
Entrada:

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.

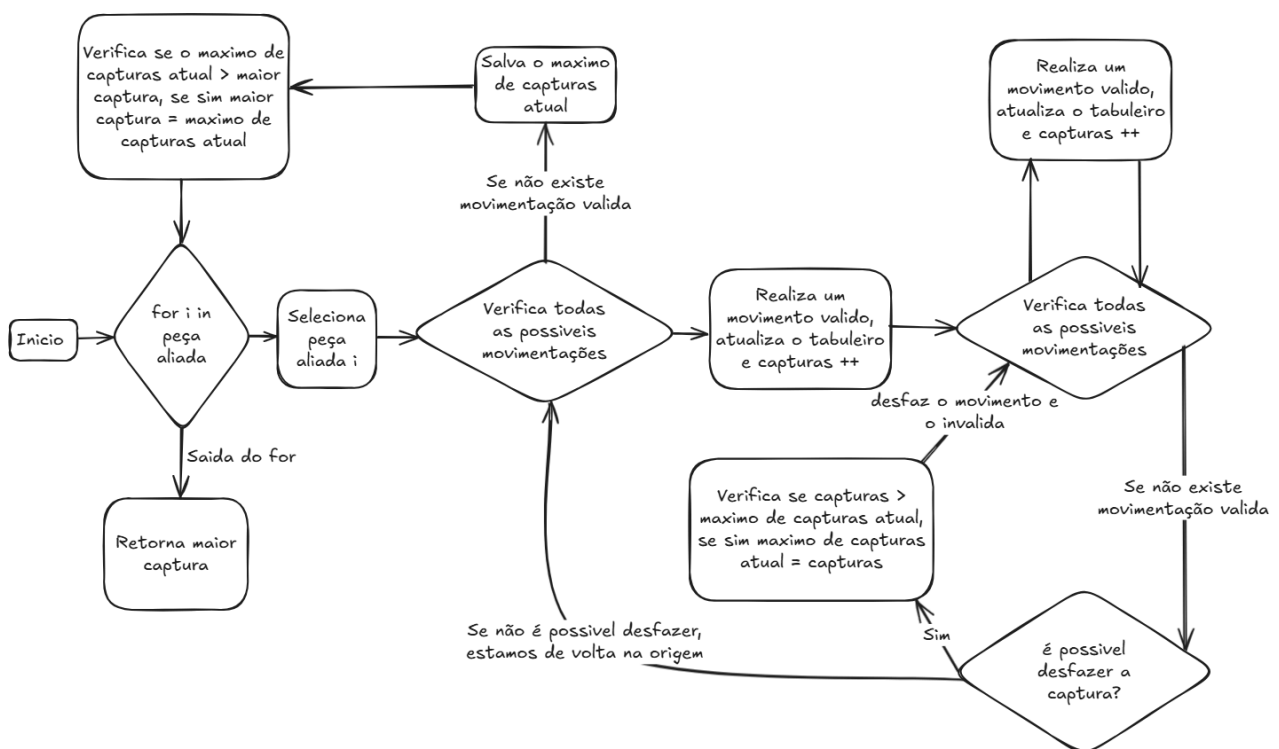
## 2.2 Estratégia da Resolução

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.

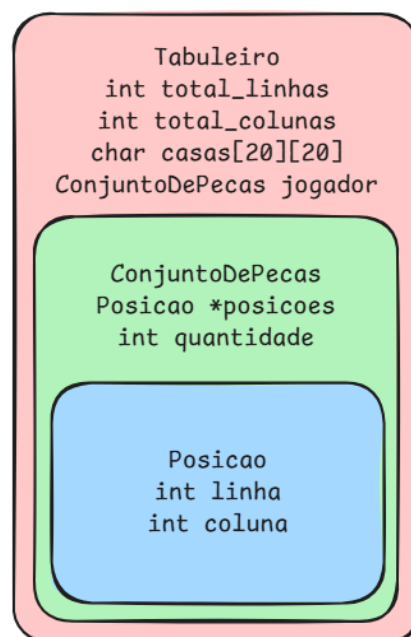
Para facilitar o desenvolvimento e entendimento do código, dividimos o algoritmo em módulos:

- entrada.c
- jogo.c
- tabuleiro.c
- tempo.c
- main.c

Mas os módulos principais são tabuleiro.c , jogo.c e main.c, que vamos analisar agora.

### 3.1 Tabuleiro.c

O módulo tabuleiro.c implementa a criação, manipulação e destruição do tabuleiro de damas, utilizando alocação dinâmica de memória para gerenciar o tabuleiro e as peças do jogador. O módulo também implementa três structs que estão interligadas, e são representadas visualmente a seguir:



A função principal, **criar\_tabuleiro**, aloca memória para a estrutura **Tabuleiro**, define suas dimensões e inicializa as casas com `CASA_BRANCA`, representando posições inválidas. Em seguida, calcula a quantidade máxima de posições válidas com a fórmula  $(\text{linhas} * \text{colunas}) / 2 + 1$  e aloca memória para armazenar as posições das peças com a função **alocar\_posicoes**. Se qualquer alocação falhar, a memória previamente alocada é liberada e a função retorna `NULL`.

Se tudo ocorrer corretamente, o conjunto de peças é associado ao tabuleiro, e a função retorna um ponteiro para ele já inicializado. Por fim, **destruir\_tabuleiro** libera toda a memória alocada, ajustando os ponteiros para NULL a fim de evitar acessos inválidos e vazamentos de memória.

### 3.2 Jogo.c

O módulo `jogo.c` implementa a lógica central para determinar o número máximo de peças do oponente que podem ser capturadas em um único movimento contínuo no jogo de damas com regras personalizadas. 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.

A função **posicao\_eh\_valida** garante que todas as posições acessadas estejam dentro dos limites do tabuleiro, e por fim, o núcleo da lógica está na função recursiva **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** percorre todas as peças do jogador no tabuleiro e determina qual delas permite a maior sequência de capturas válidas.

### 3.3 Main.c

O código realiza a função principal do programa, a qual é responsável pela leitura dos dados de entrada, pelo processamento das instâncias do jogo e pela medição do tempo de execução. Ele começa validando os argumentos passados na linha de comando para verificar se o formato é correto e se o caminho do arquivo de entrada foi passado, caso contrário, imprime uma mensagem de erro e encerra a execução do programa.

Uma vez validados os parâmetros, o programa tenta abrir o arquivo fornecido para leitura, o que, se falhar, trata o erro. Um temporizador é iniciado para medir o tempo total de execução do programa. Então, o código entra em um laço, e a cada iteração é lido do arquivo as dimensões de um novo tabuleiro, se ambas forem zero, o laço é

encerrado, do contrário, um novo tabuleiro é criado dinamicamente com as dimensões lidas usando a função **criar\_tabuleiro** e o conteúdo é carregado dentro do tabuleiro pela função **carregar\_tabuleiro\_arquivo**. Um segundo temporizador é iniciado para medir o tempo daquela instância. A seguir, a função **calcular\_maximo\_capturas\_tabuleiro** é chamada e após toda a lógica explicada anteriormente, os resultados serão salvos na variável `maximo_capturas`, e por fim os resultados e tempos de execução são impressos na tela, em seguida o tabuleiro é destruído pela função **destruir\_tabuleiro**, e o processo se repete até que  $N = M = 0$ , após isso, o tempo total da execução é impresso na tela, 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

Vamos analisar a função principal do programa, a **calcular\_maximo\_capturas\_tabuleiro**, ela percorre todas as peças do jogador, e para cada uma chama recursivamente **calcular\_maximo\_captura\_por\_pecas**, 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**

Começando por **calcular\_maximo\_capturas\_tabuleiro**, temos que sua complexidade é  $O(p)$ , pois para cada peça ele irá executar **calcular\_maximo\_captura\_por\_pecas**.

#### 4.1.2 Função **calcular\_maximo\_captura\_por\_pecas**

Como essa é uma função recursiva, cada chamada tem o seguinte efeito:

- Percorre as direções válidas = 4.
- Em cada direção, se a captura for válida, chama recursivamente a si mesma com a nova posição da peça.



Com isso temos a seguinte análise:

**Pior caso (recursão máxima):** A cada nível de recursão, uma peça do oponente é capturada, e o número total de capturas possíveis é limitado ao número de peças do oponente no tabuleiro (máximo  $O(n/4)$ ), e isso pode acontecer para cada direção possível (4).

Assim a complexidade no pior caso para uma única peça fica:

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

**Melhor caso:** No melhor caso, nenhuma peça pode capturar, ou seja não há chamadas recursivas, mas mesmo assim todas as direções são verificadas para todas as peças.

Assim a complexidade no melhor caso para uma única peça é:

$$O(d)$$

**Caso médio:** No caso médio, apenas parte das peças do oponente permite capturas em sequência. Isso significa que, na prática, nem toda direção testada leva a uma nova captura, e a recursão se aprofunda menos vezes do que no pior caso. Embora a profundidade continue limitada ao número máximo de peças do oponente, a quantidade real de chamadas recursivas é bem menor, pois nem sempre há continuidade nas jogadas. Assim, a função explora só uma parte das possibilidades em cada nível.

Assim a complexidade no caso médio para uma única peça, fica em torno de:

$$O(c^{(n/4)})$$

Com  $c$  estando entre 1 e 4, refletindo a média de direções que realmente geram capturas

## 4.2 Análise de Resultados

Agora que entendemos a complexidade do código, vamos rodar o mesmo e analisar os resultados, temos o seguinte como arquivo de entrada:

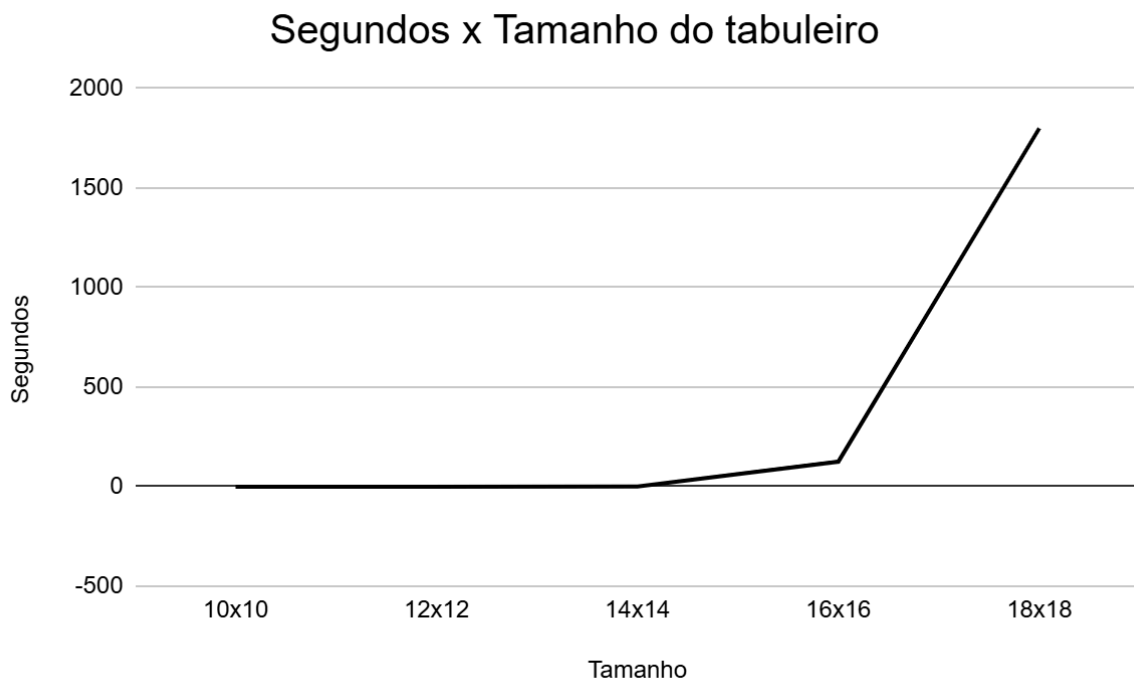
```
10 10
1 0 0 0 0 2 2 2 2 2 0 0 0 0 0 2 2 2 2 2 0 0 0 0 0 2 2 2 2 2 0 0 0 0 0 2 2 2 2 2 0 0 0 0 0 0 0 0 0 0
12 12
1 0 0 0 0 0 2 2 2 2 2 2 0 0 0 0 0 0 2 2 2 2 2 2 0 0 0 0 0 0 2 2 2 2 2 2 0 0 0 0 0 0 2 2 2 2 2 2 0 0
0 0 0 0 2 2 2 2 2
```

```

14 14
1 0 0 0 0 0 0 2 2 2 2 2 2 2 0 0 0 0 0 0 0 2 2 2 2 2 2 2 0 0 0 0 0 0 0 2 2 2 2 2 2 2 0 0 0 0 0 0 0 2 2 2 2 2 2 2 0 0 0 0 0 0 0
0 2 2 2 2 2 2 2 0 0 0 0 0 0 0 2 2 2 2 2 2 2 0 0 0 0 0 0 0 2 2 2 2 2 2 2
16 16
1 0 0 0 0 0 0 0 2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 2 2 2 2 2 2 2
2 2 0 0 0 0 0 0 0 2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 2 2 2 2
2 2 2 2
18 18
1 0 0 0 0 0 0 0 0 2 2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0 2 2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0 2 2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0
0 2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0 2 2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0 2 2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0 2 2 2 2 2 2 2
2 2 0 0 0 0 0 0 0 0 2 2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0 2 2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0 2 2 2 2 2 2 2 2 2

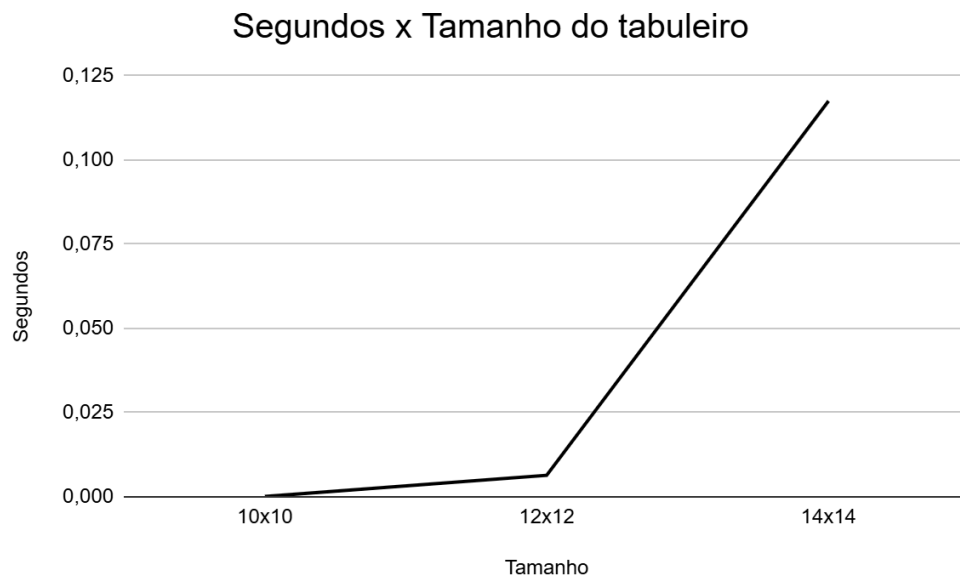
```

Os tabuleiros 10x10, 12x12, 14x14, 16x16 e 18x18, 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, a mesma sendo parada, porém com isso sabemos que a execução levaria no mínimo 30 minutos.

Como os valores de 10x10, 12x12 e 14x14 são muito pequenos, segue gráfico para melhor visualização dos mesmos:



Mesmo que sejam valores pequenos, os mesmos representam a mesma curva exponencial dos demais, comprovando com evidências, que no pior caso do algoritmo 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](#)