Análise do melhor caminho em um tabuleiro de damas

Matheus Henrique, Antônio Carlos

Setembro, 2022

1 Introdução

A proposta do trabalho consiste em encontrar, sem demorar, o melhor caminho em um tabuleiro de damas, dado um estado corrente do jogo. Foi definido como melhor caminho aquele em que é possível capturar o maior número de peças em sequência.

A captura deve respeitar as regras. Só é permitido capturar para frente e para trás, saltando sobre a peça do oponente para uma casa vazia, em casas adjacentes diagonalmente alinhadas. Apenas uma peça deve ser capturada por salto, mas é permitido realizar múltiplos saltos. A maior sequência de saltos forma o melhor caminho. Ao realizar uma captura, a peça do oponente é retirada do tabuleiro e a casa é considerada vazia, impossibilitando capturar mais de uma vez uma mesma peça.

Para solucionar a proposta, duas estratégias computacionais serão aplicadas.

2 Formato de entrada e saída

2.1 Entrada

A entrada é um arquivo, entrada.txt, que descreve um tabuleiro com um estado corrente do jogo, com o seguinte padrão:

---

1. Linha Coluna
2. Peça i Peça i+1 Peça i+2 ... Peça n
3. 0 0

---

1. São dois inteiros indicando o número de linhas e colunas do tabuleiro, tal que $3 \leq N \leq 20, 3 \leq M \leq 20, N * M \leq 200$.
2. São inteiros de 0 a 2, tais que 0 representa uma casa vazia, 1, representa uma peça do jogador, 2 representa uma peça do oponente.
3. Indicam o final do arquivo de entrada.

O tabuleiro é montado da esquerda para a direita, de cima para baixo, intercalando um valor da entrada e uma casa vazia. São $\lceil \frac{(N*M)}{2} \rceil$ peças no tabuleiro de forma que há no máximo $\lfloor \frac{(N*M)}{4} \rfloor$ peças de cada jogador.

2.2 Saída

É impresso apenas o valor do melhor caminho de cada tabuleiro em um arquivo de saída, saída.txt.

3 Modelagem do tabuleiro

A estrutura de dados que mais se assemelha a um tabuleiro é uma matriz. Como as dimensões do tabuleiro mudam a cada entrada, foi utilizada uma matriz dinâmica para representar o tabuleiro.
As seguintes funções operam essa estrutura:

1. **create_board(int **, int, int)** recebe um ponteiro para ponteiro do tipo inteiro, o número de linhas e o número de colunas do tabuleiro a ser criado. Ao criar o tabuleiro, essa função o preenche com zeros e cria uma camada de borda nas 4 laterais preenchidas com o número 3. Esse tratamento de bordas nos auxilia nos algoritmos que vão percorrer o tabuleiro buscando o melhor caminho.

2. **fill_board(int **, int, int)** recebe um tabuleiro já criado e suas dimensões, então o preenche com valores lidos do arquivo de entrada.

3. **copy_board(int **, int **, int, int)** recebe um tabuleiro que será a cópia, um tabuleiro que será copiado e as dimensões de ambos. Essa cópia é utilizada nos algoritmos que serão apresentados a seguir.

4. **validate_board(int **, int, int)** recebe um tabuleiro já preenchido e realiza a validação de acordo com as regras descritas na secção 2. Se o tabuleiro é valido o programa segue para o próximo passo, caso contrário o tabuleiro inválido é descrito na saída e o programa é encerrado.

O processo de modelagem do tabuleiro segue os passos, criar tabuleiro - preencher tabuleiro - validar tabuleiro. Ao final desse processo a entrada será modelada como no exemplo a seguir:

Entrada:

8 8

2 2 2 2 0 0 0 0 2 2 2 2 0 0 0 0 2 2 2 2 0 0 0 0 2 2 2 2 0 1 0 0

Tabuleiro:

$$\begin{bmatrix} 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \\ 3 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 3 \\ 3 & 2 & 0 & 2 & 0 & 2 & 0 & 2 & 0 & 3 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 \\ 3 & 2 & 0 & 2 & 0 & 2 & 0 & 2 & 0 & 3 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 \\ 3 & 2 & 0 & 2 & 0 & 2 & 0 & 2 & 0 & 3 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 \\ 3 & 2 & 0 & 2 & 0 & 2 & 0 & 2 & 0 & 3 \\ 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \end{bmatrix}$$

Note que ao redor dos valores que compõe a entrada, existe uma borda criada pela função que cria o tabuleiro. Essa borda é um tratamento realizado no tabuleiro para que ao testar possíveis capturas próximas a ela o algoritmo não teste posições inválidas de memória.

4 Soluções

Para encontrar o melhor caminho é necessário testar todos os caminhos do tabuleiro, isso nos leva a solução implementada no Algoritmo I.

4.1 Algoritmo I

A ideia do algoritmo I é percorrer todo tabuleiro procurando as peças "1"do jogador. Ao encontrar ele compara as 4 diagonais possíveis em busca do padrão 1 - 2 - 0 (peça do jogador, pela do oponente, casa vazia). Ao realizar essa comparação existem 3 possibilidades:

1. Encontrar 1 diagonal com o padrão.
2. Encontrar mais de 1 diagonal com o padrão.
3. Encontrar nenhuma diagonal como o padrão.

Se for encontrado o padrão em apenas uma diagonal ele captura a peça do oponente, transformando essa diagonal em uma nova sequência, 0 - 0 - 1 (casa vazia, casa vazia, peça do jogador) e analisa a 4 novas diagonais possíveis.

Se for encontrado o padrão em mais de uma diagonal, o endereço (linha, coluna) será salvo como um checkpoint, além de uma copia do tabuleiro naquele estado. Isso acontece apenas no primeiro endereço em que mais de 1 padrão é encontrado. Feito isso, o algoritmo escolhe uma diagonal para capturar, captura e analisa novamente as 4 diagonais possíveis. Ele repete o processo de captura e análise até chegar em um endereço que não possua peças para capturar. Então, ele coloca uma flag (o inteiro que representa a letra y na tabela ASCII) nesse endereço e retorna até o checkpoint.

Se nenhum padrão for encontrado, então o algoritmo verifica se existem flags em uma das 4 possíveis diagonais. Caso existam, ele conta as flags a partir dali. Se existir um checkpoint o algoritmo retorna para ele, caso contrário encerra.

Estados do tabuleiro usado na secção 3, com a aplicação do algoritmo I.

Inicio:
$$\begin{bmatrix} 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \\ 3 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 3 \\ 3 & 2 & 0 & 2 & 0 & 2 & 0 & 2 & 0 & 3 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 \\ 3 & 2 & 0 & 2 & 0 & 2 & 0 & 2 & 0 & 3 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 \\ 3 & 2 & 0 & 2 & 0 & 2 & 0 & 2 & 0 & 3 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 \\ 3 & 2 & 0 & 2 & 0 & 2 & 0 & 2 & 0 & 3 \\ 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \end{bmatrix}$$

Checkpoint:
$$\begin{bmatrix} 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \\ 3 & 0 & 0 & 0 & x & 0 & 0 & 0 & 0 & 3 \\ 3 & 2 & 0 & 2 & 0 & 2 & 0 & 2 & 0 & 3 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 \\ 3 & 2 & 0 & 2 & 0 & 2 & 0 & 2 & 0 & 3 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 \\ 3 & 2 & 0 & 2 & 0 & 2 & 0 & 2 & 0 & 3 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 \\ 3 & 2 & 0 & 2 & 0 & 2 & 0 & 2 & 0 & 3 \\ 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \end{bmatrix}$$

Captura 1:
$$\begin{bmatrix} 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \\ 3 & 0 & 0 & 0 & x & 0 & 0 & 0 & 0 & 3 \\ 3 & 2 & 0 & 0 & 0 & 2 & 0 & 2 & 0 & 3 \\ 3 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 3 \\ 3 & 2 & 0 & 2 & 0 & 2 & 0 & 2 & 0 & 3 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 \\ 3 & 2 & 0 & 2 & 0 & 2 & 0 & 2 & 0 & 3 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 \\ 3 & 2 & 0 & 2 & 0 & 2 & 0 & 2 & 0 & 3 \\ 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \end{bmatrix}$$

Captura 2:
$$\begin{bmatrix} 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \\ 3 & 0 & 0 & 0 & x & 0 & 0 & 0 & 0 & 3 \\ 3 & 2 & 0 & 0 & 0 & 2 & 0 & 2 & 0 & 3 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 \\ 3 & 2 & 0 & 0 & 0 & 2 & 0 & 2 & 0 & 3 \\ 3 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 3 \\ 3 & 2 & 0 & 2 & 0 & 2 & 0 & 2 & 0 & 3 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 \\ 3 & 2 & 0 & 2 & 0 & 2 & 0 & 2 & 0 & 3 \\ 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \end{bmatrix}$$

Captura 3:
$$\begin{bmatrix} 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \\ 3 & 0 & 0 & 0 & x & 0 & 0 & 0 & 0 & 3 \\ 3 & 2 & 0 & 0 & 0 & 2 & 0 & 2 & 0 & 3 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 \\ 3 & 2 & 0 & 0 & 0 & 2 & 0 & 2 & 0 & 3 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 \\ 3 & 2 & 0 & 0 & 0 & 2 & 0 & 2 & 0 & 3 \\ 3 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 3 \\ 3 & 2 & 0 & 2 & 0 & 2 & 0 & 2 & 0 & 3 \\ 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \end{bmatrix}$$

Coloca flag:
$$\begin{bmatrix} 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \\ 3 & 0 & 0 & 0 & x & 0 & 0 & 0 & 0 & 3 \\ 3 & 2 & 0 & 0 & 0 & 2 & 0 & 2 & 0 & 3 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 \\ 3 & 2 & 0 & 0 & 0 & 2 & 0 & 2 & 0 & 3 \\ 3 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 3 \\ 3 & 2 & 0 & y & 0 & 2 & 0 & 2 & 0 & 3 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 \\ 3 & 2 & 0 & 2 & 0 & 2 & 0 & 2 & 0 & 3 \\ 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \end{bmatrix}$$

Captura 4:
$$\begin{bmatrix} 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \\ 3 & 0 & 0 & 0 & x & 0 & 0 & 0 & 0 & 3 \\ 3 & 2 & 0 & 0 & 0 & 2 & 0 & 2 & 0 & 3 \\ 3 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 3 \\ 3 & 2 & 0 & 0 & 0 & 0 & 0 & 2 & 0 & 3 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 \\ 3 & 2 & 0 & y & 0 & 2 & 0 & 2 & 0 & 3 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 \\ 3 & 2 & 0 & 2 & 0 & 2 & 0 & 2 & 0 & 3 \\ 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \end{bmatrix}$$

Captura 5:
$$\begin{bmatrix} 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \\ 3 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 3 \\ 3 & 2 & 0 & 0 & 0 & 0 & 0 & 2 & 0 & 3 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 \\ 3 & 2 & 0 & 0 & 0 & 0 & 0 & 2 & 0 & 3 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 \\ 3 & 2 & 0 & y & 0 & 2 & 0 & 2 & 0 & 3 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 \\ 3 & 2 & 0 & 2 & 0 & 2 & 0 & 2 & 0 & 3 \\ 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \end{bmatrix}$$

Coloca flag:
$$\begin{bmatrix} 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \\ 3 & 0 & 0 & 0 & x & 0 & 0 & 0 & 0 & 3 \\ 3 & 2 & 0 & 0 & 0 & y & 0 & 2 & 0 & 3 \\ 3 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 3 \\ 3 & 2 & 0 & 0 & 0 & 0 & 0 & 2 & 0 & 3 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 \\ 3 & 2 & 0 & y & 0 & 2 & 0 & 2 & 0 & 3 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 \\ 3 & 2 & 0 & 2 & 0 & 2 & 0 & 2 & 0 & 3 \\ 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \end{bmatrix}$$

Captura 6:
$$\begin{bmatrix} 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \\ 3 & 0 & 0 & 0 & x & 0 & 0 & 0 & 1 & 3 \\ 3 & 2 & 0 & 0 & 0 & y & 0 & 0 & 0 & 3 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 \\ 3 & 2 & 0 & 0 & 0 & 0 & 0 & 2 & 0 & 3 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 \\ 3 & 2 & 0 & y & 0 & 2 & 0 & 2 & 0 & 3 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 \\ 3 & 2 & 0 & 2 & 0 & 2 & 0 & 2 & 0 & 3 \\ 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \end{bmatrix}$$

Coloca flag:
$$\begin{bmatrix} 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \\ 3 & 0 & 0 & 0 & x & 0 & 0 & 0 & 0 & 3 \\ 3 & 2 & 0 & 0 & 0 & y & 0 & y & 0 & 3 \\ 3 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 3 \\ 3 & 2 & 0 & 0 & 0 & 0 & 0 & 2 & 0 & 3 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 \\ 3 & 2 & 0 & y & 0 & 2 & 0 & 2 & 0 & 3 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 \\ 3 & 2 & 0 & 2 & 0 & 2 & 0 & 2 & 0 & 3 \\ 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \end{bmatrix}$$

Captura 7:
$$\begin{bmatrix} 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \\ 3 & 0 & 0 & 0 & x & 0 & 0 & 0 & 0 & 3 \\ 3 & 2 & 0 & 0 & 0 & y & 0 & y & 0 & 3 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 \\ 3 & 2 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 3 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 3 \\ 3 & 2 & 0 & y & 0 & 2 & 0 & 2 & 0 & 3 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 \\ 3 & 2 & 0 & 2 & 0 & 2 & 0 & 2 & 0 & 3 \\ 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \end{bmatrix}$$

Captura 8:
$$\begin{bmatrix} 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \\ 3 & 0 & 0 & 0 & x & 0 & 0 & 0 & 0 & 3 \\ 3 & 2 & 0 & 0 & 0 & y & 0 & y & 0 & 3 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 \\ 3 & 2 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 3 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 \\ 3 & 2 & 0 & y & 0 & 2 & 0 & 0 & 0 & 3 \\ 3 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 3 \\ 3 & 2 & 0 & 2 & 0 & 2 & 0 & 2 & 0 & 3 \\ 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \end{bmatrix}$$

Captura 9:
$$\begin{bmatrix} 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \\ 3 & 0 & 0 & 0 & x & 0 & 0 & 0 & 0 & 3 \\ 3 & 2 & 0 & 0 & 0 & y & 0 & y & 0 & 3 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 \\ 3 & 2 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 3 \\ 3 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 3 \\ 3 & 2 & 0 & y & 0 & 0 & 0 & 0 & 0 & 3 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 \\ 3 & 2 & 0 & 2 & 0 & 2 & 0 & 2 & 0 & 3 \\ 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \end{bmatrix}$$

Durante do processo de captura, uma váriavel "max"armazena o maior valor de capturas até o endereço (linha, coluna), incluindo as possíveis flags.

Repare que nessa representação de estados, após a captura 3, é colocada a primeira flag e a peça "1"retorna para a posição imediata antes da captura. No entanto, não é isso que acontece no algoritmo. O que na realidade acontece é que o algoritmo repete o caminho desde o checkpoint até a posição anterior a flag. Quando ele encontra a flag, ele é obrigado a seguir para outro caminho ou retornar para o checkpoint, sempre deixando uma flag no último endereço visitado.

Isso resulta em uma repetição de caminhos já visitados, até que tenham flags em todas as diagonais do checkpoint e não existam mais caminhos para percorrer.

Pseudocódigo para o algoritmo I apenas com as principais operações:

```
function algoritmoI(int **board, int linhas, int colunas):
    for i = 0 to linhas do:
        for j = 0 to colunas do:
            if(board[i][j] == 1) then:
                copy_board(copyboard, board, linhas, colunas);
                check_diagonals(copyboard, i, j);
            endif
        endfor
    endfor
endfunction

function check_diagonals(int **board, int i, int j):
```

```
    if(board[i+1][j+1] == 1 && board[i+1][j+1] == 2) then:
        board[i+1][j+1] = 0; board[i+1][j+1] = 0; board[i+2][j+2] = 1;
        check_diagonals(copyboard, i+2, j+2);
    else if(board[i-1][j-1] == 1 && board[i-1][j-1] == 2) then:
        board[i-1][j-1] = 0; board[i-1][j-1] = 0; board[i-2][j-2] = 1;
        check_diagonals(copyboard, i-2, j-2);
    else if(board[i+1][j-1] == 1 && board[i+1][j-1] == 2) then:
        board[i+1][j-1] = 0; board[i+1][j-1] = 0; board[i+2][j-2] = 1;
        check_diagonals(copyboard, i+2, j-2);
    else if(board[i-1][j+1] == 1 && board[i-1][j+1] == 2) then:
        board[i-1][j+1] = 0; board[i-1][j+1] = 0; board[i-2][j+2] = 1;
        check_diagonals(copyboard, i-2, j+2);
    else if(nmb_diagonals(board, i, j, 'y', 0) > 0) then:
        count_flags(board, i, j);
        if("existe checkpoint") then:
        check_diagonals(point_board, i_point, j_point);
        else return 0
    else return 0;
endfunction
```

A função nmb_diagonals recebe um tabuleiro, um endereço (linha, coluna), e dois inteiros a serem pesquisados nas diagonais daquele endereço. A existência de um checkpoint é controlada por uma variável que só é incrementada quando o checkpoint existir. A movimentação das peças é realizada em uma cópia do tabuleiro, a fim de preservar o original para jogadas de outras peças.

4.2 Algoritmo II

O algoritmo II utiliza uma estrutura de dados do tipo árvore para criar os diversos caminhos do tabuleiro. Uma posição do tabuleiro pode ter até 4 peças para capturar, considerando a posição inicial da captura como o nodo pai, armazenando nele o valor 0, e as possíveis capturas como os nodos filhos, armazenando neles o valor 1, é possível criar uma árvore 4-ária, em que cada nodo tem de 0 a 4 filhos.

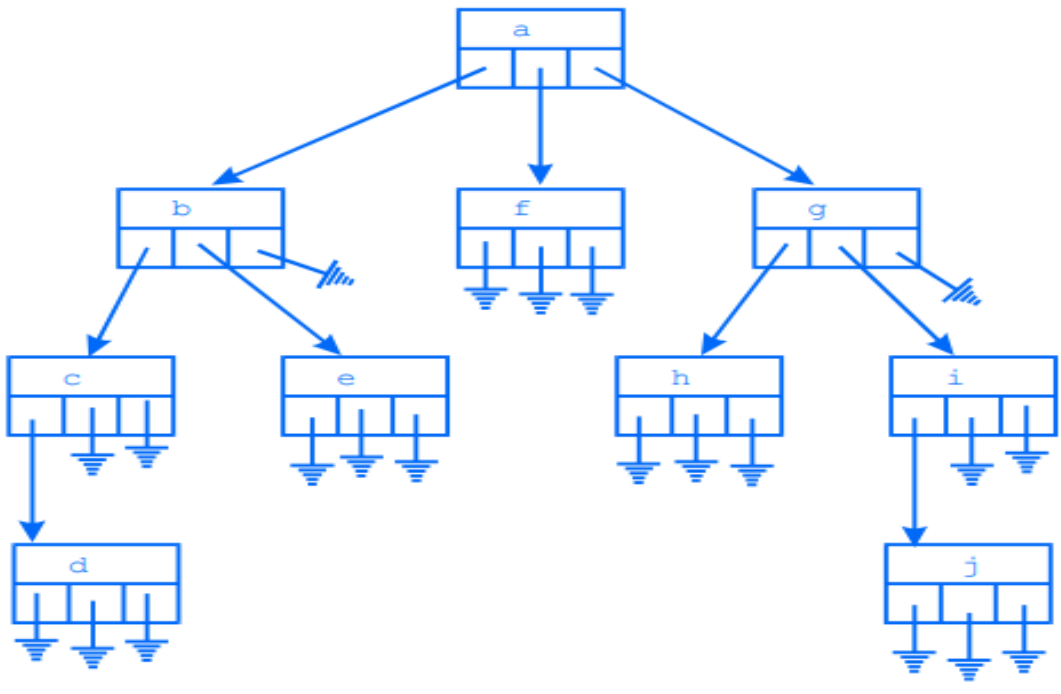

Figura 1: Árvore 3-ária

A desvantagem dessa implementação é o desperdício de memória para os ponteiros alocados que não serão usados, na maioria dos casos em um estado corrente do jogo uma peça tem no máximo 2 possibilidades de captura. Deste modo, um alternativa para esse desperdício é inserir nodos que armazenam o mesmo valor como uma lista de filhos do nodo pai.

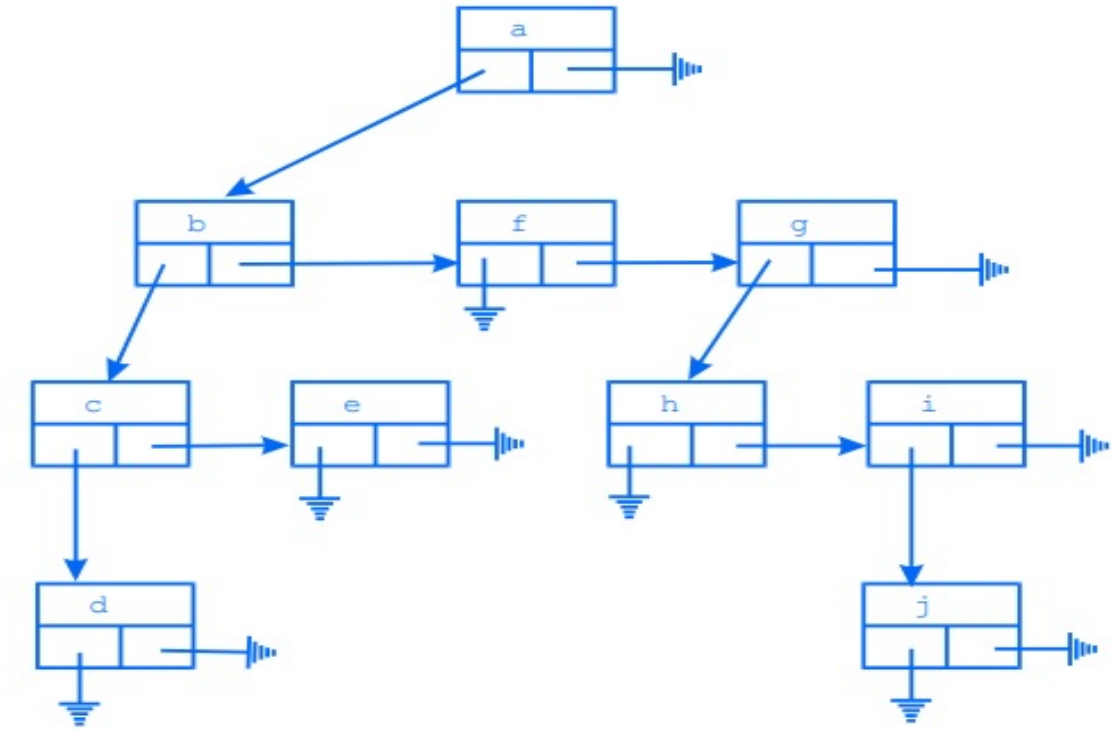

Figura 2: Árvore binária

Repare que, como a legenda da figura 2 apresentou, a lista de filhos torna a árvore n-ária em uma árvore binária desbalanceada. Utilizando o fator de desbalanceamento da árvore a nosso favor, é possível colocar sempre a esquerda o primeiro valor de captura da peça e os demais valores, iguais, a direita, como os próximos elementos da lista. Assim, é possível realizar o caminhamento pré ordem da árvore e encerrar quando atingir o último nodo mais a esquerda, pois é nele que estará o maior valor de captura.

A estrutura de dados da árvore foi implementada com a seguinte TAD:

```
struct nodo {
    int value;
    struct nodo *left;
    struct nodo *right;
};

typedef struct nodo nodo;

int create_tree(nodo **tree);
int insert(nodo **tree, int value);
int get_max(nodo **tree, int *max);
int free_tree(nodo **tree);
```

A ideia do algoritmo II é criar uma árvore que a cada possibilidade de captura insira um nodo que armazene o valor atual daquela jogada. Para mapear todas as capturas, simultaneamente, são utilizadas diversas cópias do tabuleiro, que em seguida são liberadas.

Para isso, o tabuleiro é percorrido em busca das peças "1"do jogador. Ao encontrar, o algoritmo cria uma cópia do tabuleiro (para preservar o original para futuras jogadas), cria a árvore com o valor 0 na raiz e busca as diagonais em que há possibilidade de capturas.

A busca de diagonais é realizada pela função ways, que incrementa o contador toda vez que é chamada. A função busca as diagonais que oferecem capturas e ao encontrar utiliza a função insert para inserir o valor incrementado na árvore. Feito isso, uma cópia do tabuleiro é criada, essa cópia é alterada (realizada a captura) e a função ways é chamada novamente para buscar diagonais da nova posição da peça 1.

As funções que liberam a memória alocada para a cópia do tabuleiro, são empilhadas para que após a execução completa da função ways não existam leaks de memoria.

A inserção na árvore acontece sempre a esquerda se o valor do contador for maior do que o valor do nodo comparado e direita caso contrário.

O algoritmo encerra quando as 4 diagonais da primeira chamada da função são analisadas.

Uma árvore gerada pelo algoritmo II, do exemplo da secção 3, tem o seguinte formato:

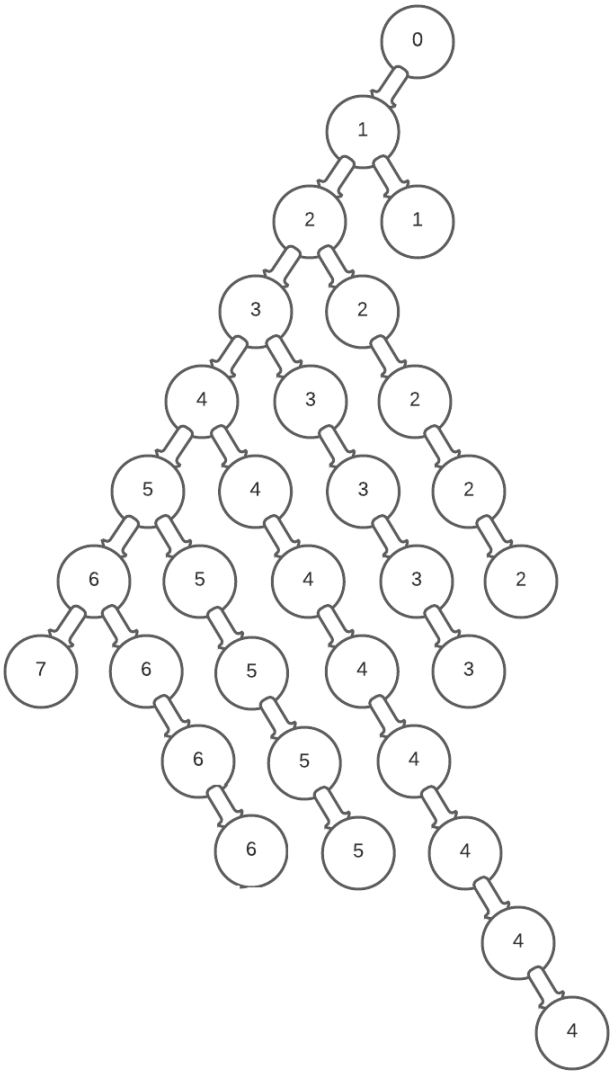

Figura 3: Árvore desbalanceada de um tabuleiro 8x8 com 7 capturas

Após a montagem, a função get_max percorre a árvore em um encaminhamento pré ordem e para quando encontra o ultimo nodo mais a esquerda, que representa o maior valor de captura.

Pseudocódigo para o algoritmo II apenas com as principais operações:

```
function algoritmoII(int **board, int linhas, int colunas):
    for i = 0 to linhas do:
        for j = 0 to colunas do:
            if(board[i][j] == 1) then:
                create_tree(&tree);
                copy_board(copyboard, board, linhas, colunas);
                ways(copyboard, i, j);
                get_max(&tree, max);
            endif
        endfor
    endfor
endfunction

function ways(int **board, int i, int j):
    if(board[i+1][j+1] == 1 && board[i+1][j+1] == 2) then:
        insert(tree, count);
        copy_board(copy_board, board, n, m);
        board[i+1][j+1] = 0; board[i+1][j+1] = 0; board[i+2][j+2] = 1;
        ways(copy_board, i+2, j+2);
    endif

    if(board[i-1][j-1] == 1 && board[i-1][j-1] == 2) then:
        insert(tree, count);
        copy_board(copy_board, board, n, m);
        board[i-1][j-1] = 0; board[i-1][j-1] = 0; board[i-1][j-1] = 1;
        ways(copy_board, i-1, j-1);
    endif

    if(board[i+1][j-1] == 1 && board[i+1][j-1] == 2) then:
        insert(tree, count);
        copy_board(copy_board, board, n, m);
        board[i+1][j-1] = 0; board[i+1][j-1] = 0; board[i+1][j-1] = 1;
        ways(copy_board, i-1, j-1);
    endif

    if(board[i-1][j+1] == 1 && board[i-1][j+1] == 2) then:
        insert(tree, count);
        copy_board(copy_board, board, n, m);
        board[i-1][j+1] = 0; board[i-1][j+1] = 0; board[i-1][j+1] = 1;
        ways(copy_board, i-1, j+1);
    endif
endfunction

function insert(nodo **root, int value)
    if(*root == NULL) then:
        *root = (nodo *)malloc(sizeof(nodo));
        (*root)->right = NULL; (*root)->left = NULL; (*root)->value = value;
    else if(value > ((*root)->value)) then:
        insert(&(*root)->left, value); return 0;
    else insert(&(*root)->right, value); return 0;
endfunction

funcion get_max(nodo **root, int *max)
    if((*root)->value >= *max) then: *max = (*root)->value;
    if(*root == NULL) then: return 0;
    get_max(&(*root)->left, max);
```

5 Tempo de execução e memória

A análise de tempo e memória foi realiza em um arquivo de entrada com 300 tabuleiros **aleatórios** gerados pelo algoritmo geracasos (o uso desse algoritmo será mencionado nas instruções de compilação e execução). As funções getrusage e gettimeofday foram usadas para medir o tempo de sistema e o tempo de usuário, respectivamente. Cada algoritmo foi executado 10 vezes e foi realizada uma média aritmética dos tempos apresentados. Para medir o uso de memória, foi utilizado o programa valgrind, que também verificou possíveis leaks de memória.
As tabelas a seguir demonstram tempo e memória, para casos gerais, de cada algoritmo.

Tempos		
	Algoritmo I	Algoritmo II
Tempo de sistema	0.00216489975 sec	0.0014684999 sec
Tempo de ususário	0.05055769644 sec	0.0325579971 sec

Tabela 1

Memória	
Algoritmo I	Algoritmo II
17.036.464 bytes allocated	11.201.952 bytes allocated

Tabela 2

O valgrind não apresentou leaks de memória.

6 Análise de complexidade

O pior caso estabelecido é o caso tal qual $N * M = 200$ e que todas as operações de cada algoritmo sejam utilizadas, ou seja, que existam flags e checkpoint para o algoritmo I e o maior número possível de inserções para o algoritmo II. O número máximo de capturas nesse tabuleiro é 36.

6.1 Algoritmo I

O algoritmo I utiliza funções auxiliares para encontrar o melhor caminho. Definindo os termos que dão complexidade ao algoritmo temos:
n como o número de capturas possíveis, sejam de peças ou de flags.
x como o número de linhas do tabuleiro.
y como número de colunas do tabuleiro.
A função que conta as flags tem ordem de complexidade $O(n)$.
A função que checa as diagonais do tabuleiro tem complexidade $O(n)$.
A função que percorre o tabuleiro tem complexidade $O(x * y)$.
Desta forma, a complexidade geral do algoritmo I, considerando o pior caso, é $O(xyn^2)$.

6.2 Algoritmo II

O algoritmo II utiliza funções auxiliares para encontrar o melhor caminho. Definindo os termos que dão complexidade ao algoritmo temos:
h como a altura da árvore binária.
n como o número de capturas possíveis, que são os nodos da árvore.
x como o número de linhas do tabuleiro.
y como número de colunas do tabuleiro.
A função que busca o maior valor da árvore (o ultimo nodo mais a esquerda) é $O(h)$.
A função que insere um nodo na árvore tem complexidade $O(n)$.

A função que checa as diagonais do tabuleiro tem árvore tem complexidade $O(n)$.
A função que percorre o tabuleiro tem complexidade $O(x * y)$.
Desta forma a complexidade geral do algoritmo II, considerando o pior caso, é $O(hxyn^2)$.

6.3 Testes para o pior caso

Como descrito no inicio da secção, o pior caso para ambos os algoritmos é o caso do tabuleiro de 200 peças com 36 possibilidades de captura. Na secção 5, foi apresentada a análise de tempo e memória para casos gerais, gerados por um gerador de casos aleatório. Nesta subsecção apresentaremos o tempo e memória para o pior caso.

Entrada:
20 10
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 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 2 2 2 2 2 0 0 0 0 0 2 2 2 2 2 0 0 0 0 1

Pior caso		
	Algoritmo I	Algoritmo II
Tempo de sistema	0.0000000000* sec	0.0048159999 sec
Tempo de ususário	0.0054150000 sec	8.7666521072 sec
Memória	174.604 bytes allocated	4.921.847.548 bytes allocated

Tabela 3
*Em 10 execuções do algoritmo em apenas uma foi apresentado tempo de sistema relevante.

7 Discussão final

7.1 Comparação dos algoritmos

Os algoritmos I e II cumprem o objetivo primário de encontrar o melhor caminho para todos os casos teste. Quanto ao objetivo secundário, encontrar sem demorar, cada algoritmo apresenta particularidades. A análise de complexidade somada aos testes gerais e de pior caso, nos revelam uma disparidade entre os algoritmos para diferentes situações.

Nos casos gerais, o algoritmo II se mostrou mais eficiente tanto em tempo quanto em memória. Para esses casos, que estão distantes do pior caso, seu uso é recomendado majoritariamente.

Para o pior caso, o algoritmo I mostrou grande superioridade, tanto em tempo quanto em memória, para esse tipo de caso seu uso é amplamente recomendado.

Apenas as dimensões do tabuleiro não nos dizem se estamos enfrentando um pior caso, elas apenas indicam a possibilidade. É possível que um tabuleiro $NxM \approx 200$ tenha poucas jogadas o que o descaracteriza como um pior caso (no arquivo de entrada há tabuleiros com mais de 190 peças que tem poucas possibilidades de captura e tempo de execução dentro da média dos casos gerais). Para confirmar a possibilidade, precisamos testa-la. Deste modo, para todos os tabuleiros com dimensões próximas de 200 casas, recomendamos o uso do algoritmo I.

7.2 Possíveis melhorias

Os dois algoritmos podem se tornar mais eficientes. Uma boa melhoria para o algoritmo I é tratar as repetições de caminhos com programação dinâmica, realizar retornos com o backtracking ou podas com o branch and bound. Para o algoritmo II existe a possibilidade de não inserir nodos "primos"de mesmo valor e transformar a árvore binária em uma lista encadeada. Essas melhorias seriam refletidas no uso de memória dos algoritmos, mas as recomendações da secção anterior seriam mantidas.

8 Instruções para compilação e execução

É fortemente recomendado o uso de um ambiente Linux para compilação e execução do programa. Neste ambiente, para compilar e executar, basta utilizar em sequência os seguintes comandos:

1. make
2. ./tp1 -i entrada.txt -o saida.txt

Para gerar casos testes, utilizar o comando ./gerarcasos numero de casos desejados. Ex.: ./gerarcasos 10. Os casos teste serão impressos no terminal. Para limpar os arquivos .o, executáveis e de saída, utilizar o comando make clean.

9 Bibliografia

1. N. ZIVIANI, Projeto de Algoritmos com Implementações em Pascal e C, 3a edição Editora Cengage Learning, 2010
2. D. E. KNUTH. The Art of Computer Programming, Volume 1: Fundamental Algorithms, Addison-Wesley, 1997.