Universidade Federal
de São João del-Rei

Trabalho Prático 1

Disciplina: Projeto e Análise de Algoritmos
Integrantes: Gabriel da Silva Souza
José Vitor Santos Alves
Professor: Leonardo Rocha

São João del-Rei - Abril, 2025

Sumário

1. O Problema

O jogo de damas adaptado por um avô apresenta regras que permitem capturas diagonais tanto para frente quanto para trás, em tabuleiros retangulares de dimensões N×M, usando apenas casas alternadas (preto/branco). O objetivo é, dado um estado inicial do tabuleiro com peças do jogador (valor 1) e peças adversárias (valor 2), determinar o número máximo de peças adversárias que podem ser capturadas em um único movimento, considerando duas abordagens:

- **Busca em Força Bruta**: recursão com backtracking que explora todas as sequências possíveis de capturas.
- **Movimentos Mínimos**: abordagem iterativa usando pilha dinâmica para gerenciar estados de tabuleiro.

Este problema visa comparar desempenho e consumo de memória de cada estratégia, equilibrando tempo de execução e uso de recursos.

2. Módulos do Projeto

O projeto está organizado em três módulos principais, cada um com arquivos .h e .c correspondentes.

2.1. Módulo io.h / io.c

Funções Principais:

Função	Declaração	Descrição
lerEntrada	char* lerEntrada (char *nomeEntrada);	Lê todo o conteúdo de um arquivo de entrada até encontrar a linha "0 0" ignorando linhas vazias, retornando uma string com os dados juntos.
escreverSaida	void escreverSaida (char *nomeSaida, char *conteudo);	Grava a string de resultados no arquivo de saída especificado.

Fluxo de leitura (lerEntrada):

1. Abre o arquivo em modo leitura.
2. Lê linha a linha com fgets, interrompendo ao encontrar "0 0".
3. Ignora linhas em branco.
4. Concatena cada linha válida em buffer dinâmico, retornando o ponteiro.

Fluxo de escrita (escreverSaida):

1. Abre (ou cria) o arquivo em modo escrita.
2. Usa fprintf para gravar todo o conteúdo.
3. Fecha o arquivo.

2.2. Módulo tabuleiro.h / tabuleiro.c

Estrutura de Dados:

```
typedef struct tabuleiro{
    int N;      // número de linhas
    int M;      // número de colunas
    int *estado; // vetor que guarda as casas válidas
}Tabuleiro;
```

Funções:

Função	Declaração	Descrição
inicializaTabuleiro	Tabuleiro* inicializaTabuleiro (char **entrada);	Constroi dinamicamente um tabuleiro lendo dimensões e valores de casas a partir da string de entrada. Avança o ponteiro de leitura.
liberaTabuleiro	void liberaTabuleiro (Tabuleiro *tab);	Libera a memória alocada para o vetor estado e pra estrutura Tabuleiro.

lerCasa	int lerCasa (Tabuleiro *tab, int linha, int coluna);	Retorna o valor de uma casa (0, 1, 2) ou -1 se for inválida.
gravarCasa	void gravarCasa (Tabuleiro *tab, int linha, int coluna ,int valor)	Atribui valor a uma posição válida

Observação: Apenas casas onde (linha+coluna)%2==0 são válidas; outras representam posições que não existem no tabuleiro.

2.3. Módulo algoritmos.h / algoritmos.c

Funções Principais:

Função	Declaração	Descrição
buscaForcaBruta	char* buscaForcaBruta (Tabuleiro *tabuleiro);	Explora recursivamente todas as sequências de captura para cada peça do jogador, retornando string com máximo de capturas.
buscaMovimentosMinimos	char* buscaMovimentosMinimos (Tabuleiro *tabuleiro);	Usa abordagem com pilha para explorar estados

		iterativamente, retornando string com máximo de capturas.

Funções Auxiliares:

Função	Declaração	Descrição
calcularMaxCapturas	int calcularMaxCapturas (Tabuleiro *tab, int linha, int coluna);	Para uma peça em (linha,coluna), retorna o número máximo de capturas via backtracking recursivo.
duplicarEstado	int* duplicarEstado (Tabuleiro *tab);	Aloca e copia o vetor estado, usado para simular estados em pilha.
lerValorCasa	int lerValorCasa (int *estado, int totalLinhas, int totalColunas, int linha, int coluna);	Lê valor de casa em vetor de estado, retornando -1 se inválido.
gravarValorCasa	void gravarValorCasa (int *estado, int totalLinhas, int totalColunas, int linha, int coluna, int valor);	Grava valor em vetor de estado.

3. Algoritmos em Detalhe

3.1. Busca em Força Bruta (recursão e backtracking)

Fluxo geral:

1. Para cada célula (i, j) do tabuleiro, se lerCasa (tab, i, j) == 1, chama calcularMaxCapturas (tab, i, j).
2. Mantém o maior valor retornado.
3. Constroi string com resultado e retorna.

Função:

```
int calcularMaxCapturas(Tabuleiro *tab, int linha, int coluna){
    int melhorResultado = 0;
    for(int d = 0; d < 4; d++){
        int midLinha = linha + movimentos[d][0];
        int midColuna = coluna + movimentos[d][1];
        int destLinha = linha + 2 * movimentos[d][0];
        int destColuna = coluna + 2 * movimentos[d][1];
           if (lerCasa(tab, midLinha, midColuna) == 2 && lerCasa(tab,
destLinha, destColuna) == 0){
            gravarCasa(tab, linha, coluna, 0);
            gravarCasa(tab, midLinha, midColuna, 0);
            gravarCasa(tab, destLinha, destColuna, 1);
                int capturas = 1 + calcularMaxCapturas(tab, destLinha,
destColuna);
            melhorResultado = max(melhorResultado, capturas);
            // restaura o estado original
            gravarCasa(tab, linha, coluna, 1);
            gravarCasa(tab, midLinha, midColuna, 2);
            gravarCasa(tab, destLinha, destColuna, 0);
        }
    }
    return melhorResultado;
}
```

3.2. Busca de Movimentos Mínimos (pilha dinâmica)

Fluxo geral:

1. Para cada peça do jogador, inicializa pilha de estados.
2. Estado contém (linha, coluna, capturas_ate_agora, copiaEstado).

3. Itera: desempilha estado atual, atualiza máximo global, gera novos estados para cada captura válida (clone de estado + atualização de casas) e empilha.
4. Libera memória de cada estado ao desempilhar.

4. Testes e Resultados

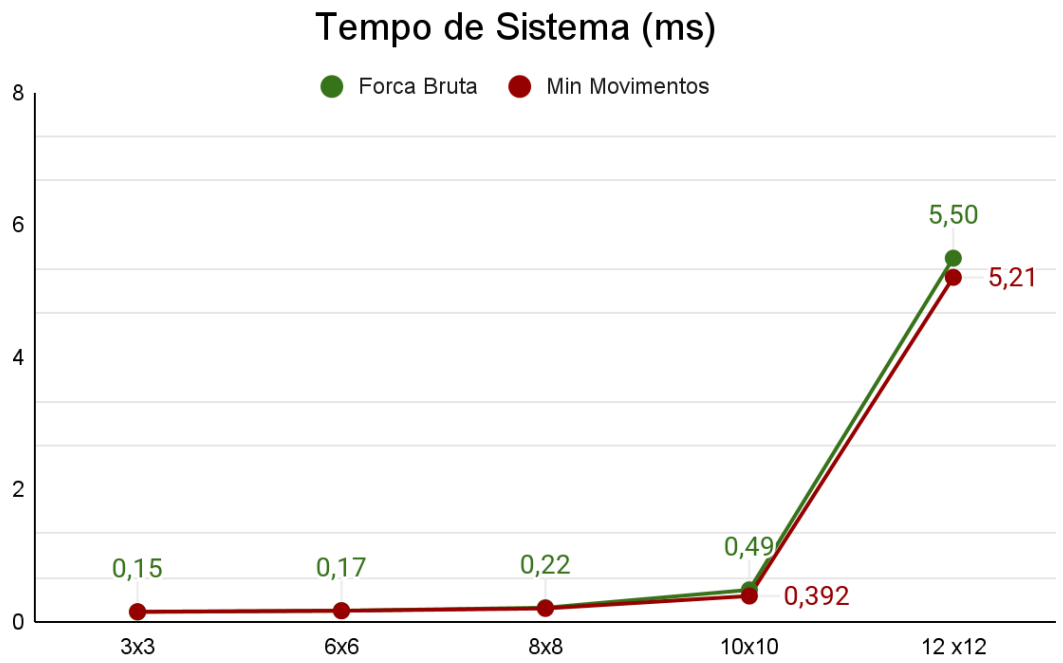
Ambiente de Teste:

- Ubuntu 22.04 LTS, Intel Core i5-8250U, 8 GB RAM.

Tamanho N x M	Estado Inicial	Resultado Força Bruta	Resultado Movimentos Mínimos	Tempo (Força Bruta / Mínimos Movimentos)
3x3	2 1 2 0 1	1	1	0,15/0,15
6x6	2 2 2 0 0 0 2 2 2 1 0 0 2 2 2 0 0 0	4	4	0,17/0,17
8x8	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	7	7	0,22/0,22
10x10	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 1 0	16	16	0,48/0,39
12x12	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 2 0 0	21	21	5,50/5,21

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

Gráfico com Resultados:



5. Compilação e Execução

Para compilar é utilizado o Makefile:

```
objetos = io.o tabuleiro.o algoritmos.o main.c

run: main
    ./main -i entrada.txt

main: $(objetos)
    gcc $(objetos) -o main

algoritmos.o: algoritmos.h algoritmos.c
    gcc -c algoritmos.c

tabuleiro.o: tabuleiro.h tabuleiro.c
    gcc -c tabuleiro.c

io.o: io.h io.c
    gcc -c io.c
```

```
clean:
    rm -rf  saida.txt io.o tabuleiro.o  algoritmos.o main
```

6. Saída Gerada

O programa gera um arquivo “saida.txt” que cada linha corresponde a um caso de teste no arquivo de entrada e imprime no terminal o tempo de execução.

7. Análise de Complexidade

7.1. calcularMaxCapturas

- **Pior Caso**: O(4^n). A pilha armazena estados para cada movimento válido. No pior caso, cada estado gera até 4 novos estados, resultando em crescimento exponencial.
- **Melhor Caso**: O(1), nenhuma direção gera recursão.

7.2. buscaForcaBruta

- **Pior Caso**: O((N x M)·4^n), verifica cada peça (N x M / 2) e chama backtracking exponencial.
- **Melhor Caso**: O(N×M), nenhuma captura possível.

7.3. buscaMovimentosMinimos

- **Pior Caso**: O(4^n). A pilha armazena estados para cada movimento válido. No pior caso, cada estado gera até 4 novos estados, resultando em crescimento exponencial.
- **Melhor Caso**: O(N x M), pilha quase vazia.

8. Conclusão

Ambas as estratégias conseguem alcançar o mesmo objetivo: identificar o número máximo de capturas possíveis. A abordagem de força bruta apresenta um uso mais contido de memória, mas tem um custo alto em termos de tempo, especialmente conforme aumenta a profundidade da análise. Por outro lado, a busca por movimentos mínimos utiliza mais memória devido à clonagem de estados, mas oferece vantagens como maior clareza para depuração e o

potencial para implementar heurísticas de otimização no futuro. Para tabuleiros maiores, seria mais eficiente adotar técnicas avançadas, como poda heurística ou programação dinâmica.