



UNIVERSIDADE FEDERAL DE SÃO JOÃO DEL-REI

DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

TRABALHO PRÁTICO 1 – DAMA COM VOVÔ

PROJETO E ANÁLISE DE ALGORITMOS

Discentes: Júlia Martins Silva, Ramon Paula

São João Del-Rei, abril de 2025


1. Documentação – Algoritmo DFS

Motivação e Descrição Geral

Este código implementa uma estratégia utilizando uma abordagem de busca em profundidade (DFS), para encontrar o número máximo de capturas consecutivas que um jogador pode realizar em um tabuleiro de damas aleatório. O programa lê as configurações de tabuleiro de um arquivo e calcula a sequência máxima de capturas para o jogador 1.

1.1. Estruturas de Dados Utilizadas


1. Position



```
1 typedef struct {
2     int row;
3     int col;
4 } Position;
```

Armazena coordenadas (linha e coluna) de uma posição no tabuleiro, sendo 'row' correspondente a linha, e 'col' a coluna.

2. Board



```
1 typedef struct {
2     int board[MAX_DIM][MAX_DIM];
3     int rows;
4     int cols;
5 } Board;
```

Representa o tabuleiro de damas:

- board: matriz de inteiros onde 0 = casa vazia, 1 = peça do jogador 1, 2 = peça do jogador 2
- rows: número de linhas do tabuleiro

- cols: número de colunas do tabuleiro

1.2. Funções Principais

1. createBoard(int n, int m)

- **Descrição:** Esta função aloca memória para um novo tabuleiro e inicializa todas as casas como vazias (0). O tabuleiro é representado como uma matriz bidimensional.
- **Complexidade:** $O(n*m)$ - precisa percorrer todas as células para inicializá-las com 0.

2. readBoard(Board *t, FILE *file)

- **Descrição:** Esta função lê os valores do tabuleiro a partir de um arquivo, preenchendo apenas as casas pretas (de acordo com a lógica do jogo).
- **Complexidade:** $O(n*m)$ - percorre todas as células válidas do tabuleiro.

3. isValidPosition(Board *t, int row, int col)

- **Descrição:** Verifica se uma posição (linha, coluna) está dentro dos limites do tabuleiro, retornando um valor booleano.
- **Complexidade:** $O(1)$ - operações de comparação constantes.

4. dfsCapture(Board *t, int row, int col, bool captured[MAX_DIM][MAX_DIM], int *max_captures, int current_captures, int player)

- **Descrição:** Implementa uma busca em profundidade (DFS) para encontrar todas as sequências possíveis de capturas a partir de uma posição. O backtracking é implementado para reverter movimentos e explorar outras possibilidades.
 - Verifica todas as 4 direções diagonais possíveis para capturas
 - Para cada captura válida, realiza o movimento e continua a busca recursivamente
 - Mantém registro das peças já capturadas para evitar repetição
 - Atualiza o máximo de capturas encontrado

- **Complexidade:** $O(4^d)$, onde d é a profundidade máxima da árvore de capturas (no pior caso). No tabuleiro de damas, d é limitado pelo tamanho do tabuleiro.

5. findMaxCaptures(Board *t, int player)

- **Descrição:** Esta função itera sobre todas as peças do jogador e chama a função dfsCapture para cada uma delas, buscando a captura máxima possível.
 - o Para cada peça do jogador no tabuleiro, inicia uma DFS
 - o Mantém o controle do máximo global de capturas encontrado
- **Complexidade:** $O(n*m * 4^d)$ - executa DFS para cada peça do jogador.

6. main(int argc, char *argv[])

- **Descrição:** Função principal que gerencia a leitura do arquivo, processamento dos tabuleiros e saída dos resultados.
 1. Verifica argumentos de linha de comando
 2. Abre o arquivo de entrada
 3. Para cada tabuleiro no arquivo:
 - Cria o tabuleiro
 - Lê a configuração
 - Calcula a captura máxima
 - Imprime o resultado
 4. Libera recursos

1.3. Análise de Complexidade Geral

A complexidade geral do algoritmo é dominada pela função findMaxCaptures, que para cada peça do jogador executa uma DFS. Considerando:

- n, m : dimensões do tabuleiro (limitadas por MAX_DIM)
- p : número de peças do jogador no tabuleiro (no pior caso, $p = O(n*m)$)
- d : profundidade máxima da árvore de capturas (no pior caso, $d = O(n)$ ou $O(m)$)

A complexidade total é $O(p * 4^d)$, que no pior caso pode ser exponencial em relação ao tamanho do tabuleiro. Isso ocorre porque o algoritmo precisa explorar todas as possíveis sequências de capturas.

1.4. Análise dos Resultados

O método implementado apresenta várias qualidades importantes:

1. **Exaustividade:** A DFS garante que nenhuma jogada potencial seja deixada de lado
2. **Controle de estado:** O algoritmo mantém registro preciso de quais peças já foram capturadas
3. **Atualização contínua:** Sempre que encontra uma sequência melhor, atualiza o recorde

Pontos que merecem atenção:

- Em situações com muitos movimentos possíveis, o tempo de execução pode aumentar significativamente
- O consumo de memória também cresce com a profundidade da busca

Sugestões para evolução:

- Trocar a DFS por BFS poderia revelar mais rapidamente as sequências mais curtas com máximo de capturas
- Introduzir critérios para descartar antecipadamente ramos de busca pouco promissores
- Dividir a análise entre várias peças simultaneamente poderia acelerar o processo

1.5. Considerações Finais

O código apresentado é uma implementação robusta para calcular a captura máxima em um jogo de tabuleiro. Utilizando conceitos de alocação dinâmica, estruturas de dados e algoritmos de busca, ele demonstra como a programação em C pode ser aplicada para resolver problemas complexos de forma eficiente. A modularidade do código facilita a manutenção e a compreensão, tornando-o um excelente exemplo de boas práticas de programação.

1. Documentação Algoritmo Força Bruta

Funções Principais

1. readVector(int vector[], FILE *arq);

- **Descrição:** Essa função é utilizada para leitura de N e M, é guardado num vetor NM[2] respectivamente onde a primeira posição ocupa o tamanho de linhas e a segunda o tamanho das colunas. Foi utilizado fscanf para leitura.

2. verify(int* vector);

- **Descrição:** Esta função é a responsável por verificar as condições estabelecidas no documento do tp, como condição de parada, tamanho máximo e mínimo das linhas e colunas assim como número máximo de casas. Caso ela retorne 0 a função quebra e caso retorne 1 o programa continua normalmente.

3. algForBru(int linha, int coluna, int matriz[linha][coluna]);

- **Descrição:** Esta função cria uma variavel aux e max e as inicializa elas em 0, então percorre a matriz utilizando 2 for, quando é achada uma peça 1 é criado um vetor de duas posições que contêm linha e coluna em que a peça 1 é achada, aux recebe a função verifyDia e após isso testa-se se aux é maior que max caso sim max recebe o valor de aux, quando percorre toda a matriz max é retornado;

4. verifyDia(int linha, int coluna, int matriz[linha][coluna], int i, int j);

- **Descrição:** Esta função recebe a matriz o tamanho dela e a posição atual em que uma das minhas peças foi encontrada, ela então cria uma matriz constante de inteiros [4][2] onde cada linha recebe uma diagonal possível, por isso o nome da matriz foi diagonaisSimples, após isso é criada uma variável melhor caminho que é zerada. Então um For que percorre 4 posições(4 diagonais). Dentro do For primeiro criamos duas variáveis(di, dj) inteiros que recebem uma diagonal de acordo com o For, por exemplo "*int di = diagonaisSimples[dia][0]*". Após isso é criado a posilnil e posilniJ que utiliza a posição da peça 1 que é passada no

cabeçalho então soma com as respectivas diagonais simples, segue o exemplo *"int posiInil = i+di"*. Então se verifica a posição além da inimiga, cria-se posiVagal posiVagaJ que recebem a posição atual da peça 1 mais a diagonal simples 2 segue o exemplo *"int posiVagal = i+(2di)"*. após isso se verifica se as posições do inimigo e a vaga estão dentro do limites do tabuleiro. então temos um IF que verifica se a posição do inimigo dentro da matriz é = a 2 e a posição vaga é igual a 0 ent uma função captura é criada ela se inicializa recebendo 1 + verifyDia(linha, coluna, matriz, posiVagal, posiVagaJ); pois essa função chama recursivamente e parte da posição que você "se encontra" após comer a primeira peça inimiga a partir da peça 1 que foi passada originalmente, após isso há um teste para saber se captura é maior que o melhorCaminho se for melhorCaminho recebe captura. Após o for o retorno é melhorCaminho, assim encerrando o algoritmo de força bruta.

Análise de Complexidade Geral

Análise de complexidade tem que estar focado na principal função do programa e o que ela realiza, no algoritmo de força bruta seria a função "verifyDia", logo é a função onde a análise ocorre.

Toda vez que a função é chamada, ela testa todas as 4 possíveis diagonais. Isso ocorre no IF que verifica a posição do inimigo e a posição vaga necessária para que seja viável o movimento, mas, independente de ser viável ou não, essa comparação vai sempre existir, o que torna o algoritmo péssimo. $T(N)$ então no pior caso é $O(4^n)$, é 4 porque existem 4 operações fixas independentemente de N , e como ela se torna recursiva, quando se chama uma segunda vez é 4^2 e assim sucessivamente " $4^2+4^3...4^n$ ", logo chegamos ao final da análise tendo o resultado que o algoritmo de força bruta é $O(4^n)$.

Análise de Resultado

Devido a ser um algoritmo recursivo que não utiliza o backtracking, acontece que uma entrada um pouco maior explode a pilha. Com um tabuleiro menor, normalmente 3x3 ou 3x4, é completamente viável e soluciona o problema, mas quando chegamos a uma entrada média, o algoritmo se torna incapaz de produzir resultados satisfatórios. Diferentemente do outro algoritmo, que utiliza ferramentas para que essa explosão não aconteça.

Com entradas relativamente pequenas, ocorre que o tempo de execução não tem uma variação muito alta, devido à grande semelhança dos algoritmos, porém é óbvio qual é mais desejável e qual dos dois deveria ser implementado em um sistema.