Report for Programming Problem 1 – 2048

Equipa:

N.º Estudante: 2018295474 Nome: Bruno Ricardo Leitão Faria
N.º Estudante: 2018282583 Nome: Diogo Alves Almeida Flórido

1. O Algoritmo

Ao analisar o problema do 2048 deparamo-nos logo com o facto de podermos efetuar 4 operações diferentes para transitar de estado para estado do tabuleiro formando uma árvore imaginária do seguinte tipo:

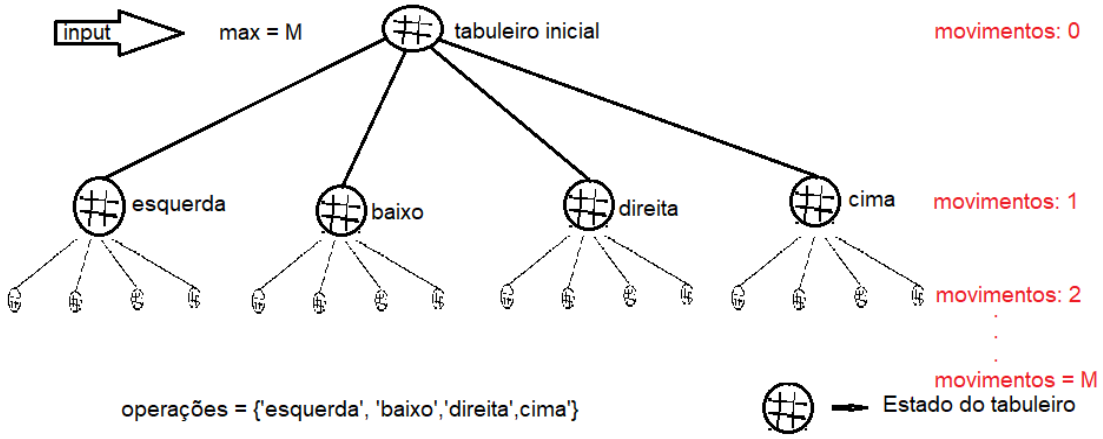


Figura 1. Árvore imaginária - cada nó com 4 filhos. Cresce sempre até o tamanho ultrapassar o máximo dado.

E foi exatamente isto que fizémos logo ao início: uma função exaustiva e recursiva que se chama a si própria $\sum_{n=0}^{max} 4^n$ vezes (sendo max o número máximo de jogadas permitido recebido no input) mas que, pelo menos, nos permitiu chegar a uma resposta certa sem grandes dores de cabeça.

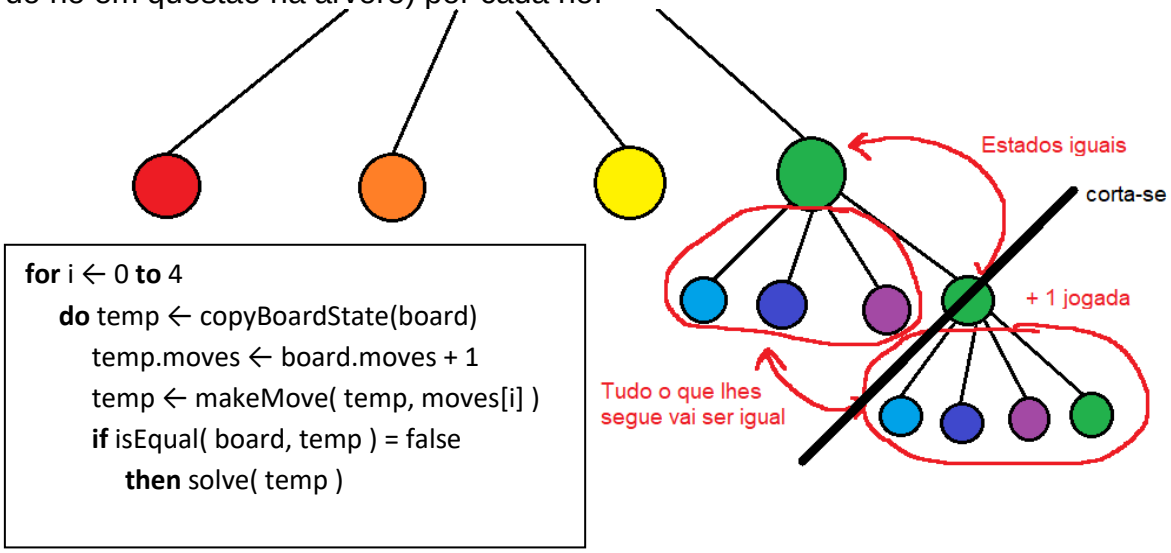
```
moves ← { ‘u’, ’r’, ’d’, ’l’ }
min_moves ← max_moves + 1
solve( board )
        num_of_elements ← getNumberOfElements( board )
        if num_of_elements < 2
                then if min_moves > board.moves
                        then min_moves ← board.moves
                return;
        if board.moves > max_moves
                then return;
        for i ← 0 to 4
                do board.moves ← board.moves + 1
                    board ← makeMove( board, moves[i] )
                    solve( board )
```

*Figura 2. Pseudocódigo da função recursiva inicial. **BoardState** é uma classe com o atributo **moves** que indica o número de jogadas feitas até chegar ao estado que ela possui, **min_moves** é um valor inteiro que indica o melhor resultado alcançado e **max_moves** o número máximo de jogadas permitidas dado no input. **makeMove**(BoardState board, char direction) é um método que faz as operações de slides num tabuleiro.*

Para tratar das operações de slide limitámo-nos a implementar um método/função de complexidade O(n*(n-1)) (sendo n o tamanho do lado do tabuleiro) que dependendo da direção percorre elemento a elemento do tabuleiro por uma ordem específica e faz ações diferentes caso encontre dois números iguais seguidos, dois números diferentes ou espaços vazios.

1.1 Melhorando o algoritmo

Há ocasiões em que certa operação não altera o estado do tabuleiro, ou seja, ocasiões em que gastamos uma jogada em vão. Retirando esses nós da árvore imaginária poupamos $\sum_{n=0}^{max-h} 4^n$ chamadas da função (sendo h a altura do nó em questão na árvore) por cada nó!




```
for i ← 0 to 4
    do temp ← copyBoardState(board)
       temp.moves ← board.moves + 1
       temp ← makeMove( temp, moves[i] )
       if isEqual( board, temp ) = false
          then solve( temp )
```

Figura 3. Ilustração visual da estratégia e alteração do código recursivo na função anterior. Destaque para o if que impede a função de se voltar a chamar caso o estado anterior seja igual ao atual. copyBoardState é um método que copia os elementos dum tabuleiro para outro e isEqual é um método que verifica se dois tabuleiros são iguais.

Considerámos retirar estados repetidos que não tivessem seguidos, mas isso implicaria guardar os tabuleiros todos da árvore em memória (deixaria de ser imaginária).

Outra forma de poupar chamadas é não chamar de todo pois há muitos casos em que o tabuleiro é **impossível** de se resolver mesmo com jogadas infinitas e isso dá para se saber logo ao início: basta ir somando os números repetidos do tabuleiro e ver se acabamos só com um número no final. Concluímos que isto é equivalente a

Se a soma dos números não for 0 nem uma potência de 2, o tabuleiro é impossível.

Assumindo que o número mínimo de jogadas necessárias para terminar um tabuleiro é de $Log_2 N$, sendo N o número de elementos no tabuleiro, chegámos ainda aos seguintes casos de rejeição:

-Se o número mínimo de jogadas necessárias para terminar o tabuleiro for maior do que o número de jogadas restantes, então já não é possível resolver o tabuleiro por esse nó.

-Se o número mínimo de jogadas necessárias para terminar o tabuleiro for maior do que o número de jogadas restantes **até alcançar o melhor resultado obtido** menos um, então já não é possível ultrapassar esse resultado por esse nó.

2. Estruturas de dados

Neste problema utilizámos a linguagem C++ e criámos duas classes:

BoardState
Atributos: **int Moves** – indica o número de jogadas feitas para chegar a este estado; **int board[20][20]** – array bidimensional com os elementos do tabuleiro num determinado estado.

Game
Atributos: **int max_moves** – número máximo de jogadas dado pelo input; **int min_slides** – número de jogadas do melhor resultado alcançado; **int board_size** – tamanho do lado do tabuleiro; **char moves[4]** – array com parâmetros a passar para o método makeMove; **BoardState initial_state** – tabuleiro inicial dado pelo input; **BoardState temp** – tabuleiro temporário para verificar se houve alterações em relação ao tabuleiro anterior.
Métodos: **Game(BoardState init_state, int brd_size, int max_plays)** – construtor; **solve(BoardState current)** – resolve o tabuleiro inicial chamando-se recursivamente e alterando-o; **copyBoardState**(**BoardState previous)** – devolve um BoardState com a mesma board e moves+1; **getNumberOfElements(BoardState board)** – devolve o número de elementos não zero no tabuleiro; **isEqual(BoardState board, BoardState new_board)** – devolve se duas boards têm os mesmos elementos ou não; **makeMove(BoardState board, char direction)** – realiza as operações de slides num BoardState;

3. Porque é que o nosso algoritmo está correto?

Pensamos que o sucesso do nosso algoritmo no Mooshak se deva ao facto de:

-O algoritmo ser bem implementado e, de facto, dar as respostas certas;

-Utilizamos sempre os mesmos objetos Game e BoardState cujos atributos vamos alterando ao longo da execução, ou seja, não precisamos de guardar mais que dois tabuleiros em memória;

-O espaço em memória necessário é bastante reduzido porque apenas usámos ints e chars dentro das classes e também não precisamos de perder tempo com procuras e inserções de tabuleiros, por exemplo;

-Os nossos casos de rejeição evitam que se façam muitas chamadas recursivas desnecessárias. Cada um evita um valor de $\sum_{n=p}^{max} 4^n$ de chamadas à recursão, sendo p o nível onde foi efetuado o corte.

4. Complexidades temporal e espacial do algoritmo

Complexidade Temporal: $O(4^N * k^2)$

Complexidade Espacial: $O(k^2)$

sendo N o número máximo de movimentos e k o tamanho do lado do tabuleiro