Relatório para o Problema 1 de Programação - 2048

Equipa

N°Estudante: *2018284515* **Nome:** *Ana Rita Rodrigues*
N°Estudante: *2018233092* **Nome:** *Dylan Gonçalves Perdigão*

1. Descrição do Algoritmo

Após uma tentativa de "procura em largura primeiro" (***B**readth **F**irst **S**earch*) que nos foi impossibilitada pelo limite de memória do *mooshak*, foi escolhida a "procura em profundidade primeiro" (***D**epth **F**irst **S**earch*) de forma recursiva. Esta técnica torna-se lenta uma vez que procura primeiro as soluções mais profundas, e logo piores, necessitando de otimizações para evitar ao máximo casos desnecessários.

Foram desenvolvidas quatro funções que permitem movimentar os números nas 4 direções. No caso do movimento para a esquerda, o algoritmo trata cada uma das linhas da matriz individualmente de acordo com o pseudo-código apresentado:

```
function caseLeft(board[1..n], size, used_moves, sum):
    for i ← 0 to size
        current ← 0;  next ← 1;  write ← 0;  line ← i * size
        while next < size do
            if board[line + current] = 0 then
                current ← current+1;  next ← next+1
            else if board[line + next] = 0 then
                next ← next+1
            else if board[line + current] = board[line + next] then
                board[line + write] ← board[line + current] << 1
                if board[line + write] = sum then
                    minMoves ← used_moves + 1
                    return board[0..n]
                write ← write+1;  current ← next+1;  next ←next+2
            else
                board[line + write] ← board[line + current]
                current ← next;  next ← next+1;  write ← write+1
        if current < size then
            board[line + write] ← board[line + current]
            write ← write+1
        while write < size do
            board[line + write] ← 0
            write ← write+1
    return board[0..n]
```

Os restantes movimentos foram realizados de forma análoga.

O algoritmo de “procura em profundidade primeiro” necessitou de várias otimizações para passar nos testes do *mooshak* e alcançarmos os 200 pontos. Concretamente:

1. Verifica-se antes de lançar o algoritmo de procura se o “*bitwise and*” da soma de todos os números do tabuleiro é igual à mesma soma subtraída de um é igual a 0:

   $$soma \,\&\, (soma - 1) = 0$$

   Caso isto não se verifique, a soma dos números no tabuleiro não é uma potência de 2 e portanto não há solução possível.

2. A recursão não pode ir mais além do número máximo de jogadas ou do número de jogadas da melhor solução até ao momento encontrada, sendo este atualizado ao longo da execução.

3. Sabendo que a solução terá como único número a soma de todos os números inicialmente no tabuleiro, e que em cada jogada cada número pode duplicar apenas uma vez, é possível saber qual o número mínimo de jogadas a realizar antes de se atingir uma solução. Caso a soma desse número mínimo de jogadas com o número de jogadas já realizadas seja igual ou superior ao número de jogadas da melhor solução encontrada, não é necessário continuar a recursividade a partir desse momento.

4. Se o movimento resultar no mesmo vetor que na jogada anterior, não é necessário continuar a recursão nessa direção, uma vez que os movimentos que lhe seguirão, já terão sido verificados, evitando assim, jogadas inúteis.

5. A solução é procurada durante as transformações evitando verificações posteriores. Antes do início da procura foi calculada a soma de todos os números no tabuleiro, caso ao juntar dois números o resultado seja esta soma, sabe-se que a transformação encontrou a solução, o mínimo de movimentos é atualizado e a transformação é parada uma vez que não é necessário continuar a procurar a partir daí.

2. Estruturas de Dados

As principais estruturas de dados utilizadas foram os vetores que existem na linguagem *C++* que permitem representar as casas do tabuleiro de jogo. O vetor é unidimensional sendo que representa mais concretamente a concatenação das linhas do tabuleiro. O motivo de adotarmos um vetor unidimensional e não um vetor de vetores foi a elevada quantidade de cópias necessárias. Ao utilizar um vetor de vetores, seria necessário realizar tantas cópias quantas linhas existem na matriz para cada passo recursivo, assim o número de cópias passa a apenas uma, poupando tempo pela redução do número de alocações de memória.

A segunda estrutura de dados implementada é uma *struct* chamada “*Board*” contendo o vetor referido anteriormente (*matrix*), o tamanho das linhas do tabuleiro (*size*) e o número máximo de movimentos para o tabuleiro em questão (*max_size*).

3. Exatidão do Algoritmo

O algoritmo que implementámos está correto, uma vez que, testa exaustivamente todas as soluções. No entanto e como já referido anteriormente estão implementadas otimizações que impedem a procura de soluções piores que as já encontradas anteriormente ou a descida desnecessária até determinados nós, tornando assim o algoritmo mais eficiente mas não incorreto.

A primeira otimização referida, em que se verifica se a soma dos números no tabuleiro é uma potência de 2 não torna o algoritmo incorreto porque para haver solução é necessário o *merge* (soma) de todos os elementos do tabuleiro, que resultam em potências de 2 apenas.

A implementação da 3ª otimização não afeta a eficácia do algoritmo. Uma vez que o número mínimo de movimentos necessários para atingir uma solução é de $log_2(sum(matrix)/min(matrix))$, em que a função *min* devolve o valor mais baixo diferente de zero, e que o número máximo de jogadas úteis é dado pela diferença entre a melhor solução encontrada e o número de jogadas já usadas, qualquer estado cujo número mínimo de movimentos para atingir a solução seja superior ao número máximo de jogadas úteis é inútil.

4. Análise do Algoritmo

Considerando *n* o tamanho do lado do tabuleiro e *m* o número máximo de movimentos, sabe-se que para cada jogada serão geradas outras 4, até um máximo de *m* jogadas. Assim, o número de casos gerados será de 4^m. Em cada um destes casos é aplicada uma transformação a cada linha ou coluna do tabuleiro de complexidade $O(n)$, para uma complexidade total da transformação de $O(n^2)$. Assim a complexidade do algoritmo é de $O(4^m * n^2)$.

Best case: $O(n^2)$, solução obtida à partida, ou solução impossível por soma dos números não ser potência de 2

Worst case: $O(4^m * n^2)$, percorreu todos os ramos não tendo nenhuma das otimizações sucesso em reduzir ramos na procura.

5. Referências

- Rosen, K. (2016). *Discrete Mathematics and Its Applications* (Seventh Ed). McGraw-Hill Higher Education - VST E+p.
- Costa, E., & Simões, A. (2008). *Inteligência artificial: fundamentos e aplicações*. FCA.