Report for Programming Problem 1 - 2048

Team: EA 2018274233_2018295452

Student ID: 2018274233 Name Ana Beatriz Marques

Student ID: 2018295452 Name Bárbara Gonçalves

1. Algorithm description

Função main: Recebe o input do utilizador, armazenando o número de tabelas recebidas na variável n. Para cada tabela, recebemos também a sua dimensão (*dim*) e o número máximo de jogadas,que vai ser guardado na variável global *nplays.* As tabelas vão ser guardadas num vetor bidimensional de tamanho *dim x dim*. Seguidamente, verificamos se há solução possível para a tabela atual, recorrendo à função check. Caso seja, entramos na função play, que vai calcular o melhor caso possível para o tabuleiro. Caso esse número seja menor ou igual a *nplays* esse número é impresso. Caso contrário, dizemos que não há solução possível.

Função recursiva (*play*): A função recursiva é a *play*(), que tem como argumentos o vetor "pai", o vetor "avô", a dimensão do vetor e o número de moves atual. Quando a função é chamada pela primeira vez no main, os vetores "pai" e "avô" são iguais. Nesta função chamamos as várias funções de movimentos (move right, up, down e left), que retornam o vetor após ser efetuado o respectivo movimento, e verificamos se aconteceu alguma alteração relativamente ao vetor "pai" e ao vetor "avô". Caso tenha acontecido é chamada a função play novamente, onde o vetor "pai" vai corresponder ao vetor que acabou de sofrer alterações e o vetor avô vai corresponder ao vetor "original". A dimensão mantém-se constante durante todo o programa e o número de moves é incrementado. Isto repete-se até encontrarmos o primeiro término do tabuleiro, com auxílio da função *contazeros.* Esta função devolve o número de zeros do tabuleiro passado por parâmetro. Assim que for encontrado um tabuleiro com apenas um número diferente de zero, e caso o número de moves seja melhor do que a variável global "best" (que inicialmente é inicializada a 1000), atualizamos a variável best para o novo valor. Caso futuramente se verifique outro tabuleiro com apenas um número diferente de zero e com um "best" melhor, este volta a ser atualizado. Caso nos encontremos num ramo onde o número de moves é maior ou igual ao best -1 e o tabuleiro ainda não estiver terminado, este ramo é abandonado. Igualmente, se o número de moves atual for maior do que o número máximo de jogadas definido pelo utilizador, o ramo é abandonado.

Funções de movimentos (*move up, right, left e down*): Tomemos como exemplo a função move left. Percorremos cada linha, verificando se o número da posição em que nos encontramos é diferente de 0. Caso seja igual a 0 é ignorado. Caso seja diferente, esse número é guardado na variável *number* e a sua posição na variável *position* (estas são inicializadas a -1). Continuamos a percorrer o vetor. Caso o número atual seja igual a zero continuamos a ignorá-lo, caso seja diferente do número guardado na variável *number* este passa a ser o novo *number.* A variável *position* é também atualizada para a

posição deste mesmo. Caso o número atual seja igual à variável *number*, estes vão ser somados, ou seja, o número que se encontra na posição *position* for igual a *number* + *number* e o número em que nos encontramos atualmente vai desaparecer, transformando-se num 0.

Após termos todos os números possíveis somados de uma linha, temos de organizar os zeros. Percorremos novamente a linha. Se o número atual for 0, incrementamos a variável *countz.* Se for diferente de 0 e o contador já tiver sido inicializado, o número atual é reposicionado para a posição atual - *countz* (= número de zeros à esquerda deste). Na posição onde se encontrava o número atual passamos a ter um 0.

As funções de movimentos seguem todas este raciocínio. As diferenças são que, no caso das funções move right e move left trabalhamos com as linhas, no caso das funções move up e move down com as colunas. Nas funções move left e move up percorremos as linhas/colunas do início ao fim, quando nas funções move right e move down percorremos do fim ao início.

Funções auxiliares:

diffmtrx*:* Recebe duas matrizes como parâmetro e verifica se são iguais.

contazeros**:** Conta os zeros de um tabuleiro.

printvector: Imprime o tabuleiro.

check: Percorre o tabuleiro, somando todos os seus elementos. Caso a soma final seja uma potência de 2, continuamos o programa, entrando na função play. Caso contrário, avisamos o utilizador que não há solução possível.

2. Data structures

Para este problema precisávamos de uma estrutura que representasse um tabuleiro *NxN.* Achamos que a melhor abordagem seria um vetor bidimensional, sendo a dimensão *N* passada por parâmetro pelo utilizador.

3. Correctness

Para determinar se é possível sequer realizar o tabuleiro verificamos se a soma de todos os elementos do tabuleiro é igual a $2^n,\ n \in \mathbb{N}$(ou seja, pertence ao conjunto dos números {2, 4, 8, 16,....., 2048} pois para obter uma solução válida o tabuleiro deve terminar com apenas um elemento igual a essa soma). Desta forma, conseguimos assim descartar antes da recursividade, tabuleiros que sabemos que não tem solução.

Antes de obtermos 200 pontos, tivemos a ideia inicial de mover dentro do vetor os 0's consoante o movimento (fazendo remove e push), realizando a soma dos elementos e remoção dos 0's dentro do mesmo ciclo, teoricamente reduzindo a complexidade mas, como altera o tamanho do vetor constantemente, demora mais tempo a realizar e por isso obtivemos "Time Limit Exceeded" apesar do output ser o correto.
Ao apercebemo-nos desse problema, alteramos o nosso funcionamento de cada movimento de forma a não ser necessário mudar o tamanho do vetor:

realizando a soma dos elementos iguais, e após isso contar zeros e trocá-los de posição.

Além disso, para evitar movimentos infinitos sem alteração, comparamos com o pai, nenhum tabuleiro pode ser igual ao pai, e com o avô, movimentos de apenas uma linha ou coluna que se repetem (ex: uma linha que se movimenta esquerda direita esquerda) se não gerem evolução, isto é, são iguais ao avô. Desta forma conseguimos descartar possibilidades/movimentos que não irão levar ao melhor resultado.

Finalmente, apesar de não representado na classificação do mooshak (esta alteração não melhorou a pontuação mas devemos considerar como melhoria no nosso código), caso já exista um best, ignoramos todas as recursividades com moves igual a best - 1, pois, apesar de ainda ser possível terminar o tabuleiro em próximas jogadas, o valor mínimo de moves seria sempre igual ou superior a best, o que é desnecessário verificar.

Tendo tudo isto em conta, a nossa solução final apresentada acima é, a nosso ver, a solução ideal.

4. Algorithm Analysis

$T(n) = 4T(n - 1) + Tq = 16T(n - 2) + 4Tq + Tq = 4^k .T(n - k) . (4^k - 1)/3Tq = 4^k .T(0) . (4^n - 1)/3Tq \in O(4^{(n+1)})$. É esta a complexidade da função recursiva. Quanto ao nosso base-case, a complexidade seria apenas de $O(1)$, visto que a recursividade não é utilizada nem são realizados quaisquer movimentos. Estes têm uma complexidade de $O(n^2)$.

5. References

(Provide all the bibliography and internet links that you have used to support the development of your approach)

slides da disciplina de Estruturas Algorítmicas
https://play2048.co/
https://www.bigocheatsheet.com/