Report for Programming Problem 1 - 2048

Team:

Student ID: 2018293871 Name: Sofia Margarida Ribeiro da Silva
Student ID: 2018296218 Name: Sofia Meireles Fonseca Costa

1. Algorithm description

Movimentos:
Para fazer os movimentos de forma eficiente, começámos por guardar o primeiro valor e a sua posição. Percorrendo a linha ou a coluna da matriz, dependendo do movimento, verificamos se o número seguinte é igual ou diferente e diferente de 0. Caso seja igual, somamos esse ao valor guardado, colocamos na posição guardada e na posição atual colocamos o valor 0. Caso seja diferente e não seja 0, guardamos esse valor e essa posição. Seguidamente, compomos a linha/coluna, isto é, metemos os 0 todos de um lado, conforme o movimento.

Função recursiva:
Esta função recebe o vetor tabuleiro (v), o número de movimentos (n_moves) e os 4 vetores tabuleiro anteriores (old1, old2, old3, old4). A função *check_end()* verifica se o jogo já acabou, isto é, se já só existe um valor diferente de 0 e a função *check_equal()* verifica se os tabuleiros são iguais. Inicialmente, criamos uma cópia do tabuleiro. Seguidamente, verificamos se o jogo chegou ao fim ou se o número de movimentos é maior que o número máximo de movimentos (best-case). Caso uma destas condições se confirme, se o número de movimentos for menor que o best, definimos o best como o número de movimentos e damos return. Se não, damos apenas return. Depois, começamos os movimentos (recursive step). A ordem que escolhemos foi direita, cima, esquerda e baixo. Usamos o copy2 como “teste”, para verificar se o movimento é vantajoso. Esta verificação passa por comparar o tabuleiro atual com anteriores e ver se houve alguma alteração. No caso de passar nas verificações, atualizamos os tabuleiros anteriores e o nosso tabuleiro atual passa a ser o copy2, e o número de movimentos incrementa. É chamada novamente a função.

Speed-up tricks:
De formar a cortar ramos da recursividade e melhorar, assim, a eficiência da nossa solução, utilizámos as seguintes condições:

- Se a soma dos valores dos tabuleiros não der um número que seja uma potência de 2, imprime-se logo “no solution” uma vez que é impossível;
- Se o tabuleiro atual for igual a qualquer um dos últimos 4, não avança mais e dá return;

- Se o número de movimentos for igual ao best-1, dá return uma vez que qualquer movimento a partir daí não vai ser melhor que o best.

A ordem dos movimentos escolhida foi direita, cima, esquerda e baixo, uma vez que nos pareceu mais vantajoso do que ter os movimentos horizontais e os verticais agrupados.

2. Data structures

Para a realização deste projeto, apenas utilizámos vetores bidimensionais tanto para armazenar o tabuleiro como os tabuleiros antigos de forma a realizar os movimentos.

3. Correctness

De forma, a alcançar os 200 pontos, começámos por encontrar uma solução que chegasse ao resultado esperado mas com um tempo bastante superior ao desejável. De seguida, de forma a solucionar este problema, implementámos algumas melhorias. Inicialmente começámos por verificar se seria possível resolver o tabuleiro e de seguida tentámos descobrir qual seria a melhor ordem para executar os movimentos. Por fim, comparámos com os tabuleiros anteriores.
Para isso, arranjámos diversos casos-teste que nos permitiram verificar se não estávamos a cortar a recursão em casos que não era suposto.
Devido às caraterísticas da função recursiva, conseguimos garantir que todos os ramos necessários são percorridos.

4. Algorithm Analysis

Relativamente à complexidade da solução, os movimentos têm uma complexidade temporal de O(N^2). O nosso base-case, uma vez que não realiza nenhum movimento e não chama recursivamente a função, apresenta complexidade temporal O(1). Desenrolando a recursividade, concluímos que a complexidade da função recursiva é:

$T(n) = 4\,T(n-1) + Tq = 16T(n-2) + 4Tq + Tq = \ldots = 4^k T(n-k) + (4^k - 1)/3Tq = \ldots = 4^n T(0) + (4^n - 1)/3Tq \in O(4^{n+1})$

Em relação à memory, como guardamos o array 6 vezes, ocupa $6n^2$.

5. References

Materiais Teóricos fornecidos pelo professor.