Report for Programming Problem 1 - 2048

Team:

Student ID: 2018285164 Name: Eduardo Cruz

Student ID: 2018298209 Name : Rodrigo Sobral

1. Algorithm description

O nosso algoritmo baseia-se essencialmente em 2 passos para a resolução do tabuleiro.

1) Primeiramente, na fase de leitura do input, preenchemos uma estrutura auxiliar (lista) com o número de ocorrências de cada elemento. Através desta lista efetuamos o somatório de todos os elementos presentes no tabuleiro inicial e verificamos se o resultado deste somatório (count) é efetivamente uma base de 2; caso essa condição não se verifique, damos o tabuleiro como impossível, imprimindo 'no solution'. O facto de o somatório calculado ser uma base de 2 afere a possibilidade de decompor a totalidade dos elementos em 2's, 2count vezes, o que nos permite facilmente descartar todo o conjunto de tabuleiros que não verifique esta condição, antes mesmo de iniciarmos a tentativa, computacionalmente dispendiosa, de encontrar uma solução através de recursão, para tabuleiros claramente impossíveis.
2) De seguida entramos num processo recursivo de escolha de *slidings*, onde os tabuleiros que passaram no 1º teste de rejeição seguem para uma fase onde o jogo só não será concretizável se a posição dos números não for favorável. Por isso, a cada recursão garantimos que

ainda não foi ultrapassado o limite de *slidings* estabelecido, que não estamos perante uma matriz já registada na nossa *HashTable*, caso contrário, registamo-la, e que armazenamos o tabuleiro resultante dos *slides* nas diversas direções para o passar à recursão seguinte.

2. Data structures

Foram, ao todo, utilizadas duas estruturas de dados:

1) **Matriz**: o mais indicado foi utilizar uma estrutura de dados que fosse de rápido acesso e facilmente manipulável, por isso decidimos armazenar o tabuleiro do jogo numa matriz com *X* por *X* sendo *X* o tamanho definido pelo utilizador. Esta estrutura permite-nos aceder a uma qualquer célula, em tempo constante, O(1).
2) ***HashTable***: a cada recursão, a matriz atual é registada na *HashTable,* desta forma evitamos situações onde possamos encontrar uma matriz que já tinha sido registada anteriormente, o que significa que não estamos a fazer recursões desnecessárias que mantêm o tabuleiro no mesmo estado.

3. Correctness

As duas ocasiões que mais se destacaram eram relativas a matrizes de maior dimensão. Para estas, o programa demorava a executar ou por vezes, nem respondia. Foi então que tivemos a ideia de aplicar a estratégia anteriormente referida no ponto **1** de fazer uma análise prévia não à matriz mas sim aos números presentes na matriz, uma vez que, para matrizes grandes e com uma grande quantidade de números, há uma maior probabilidade de nos depararmos com uma não concretizável devido à falta de *merging* de algum elemento. Nas funções de slide evitámos ao máximo efetuar operações que considerámos serem desnecessárias para a obtenção do resultado pretendido, como por exemplo, no caso do slideLeft, após efetuarmos o slide, não procedemos à inversão da matriz dado que, ao nível do número mínimo

de slides necessário para a resolver, são equivalentes. Para reduzirmos a complexidade dos ciclos efetuados para percorrer a matriz, após cada slide, o tamanho N da matriz é reduzido sempre que possível. Por fim, a utilização de uma variável global (referenciada nas aulas Teóricas como 'best'), teve um papel fundamental na redução de chamadas recursivas, que seriam desnecessárias, efetuadas.

4. Algorithm Analysis

Base case: *recursiveTries(…,slide_count=max_slide+1,…,max_slide)*
Sem considerar a utilização dos testes de rejeição descritos acima, que têm um papel crucial na redução do número de slides necessários para chegar a uma solução, para um *m* número de recursões teríamos $\sum_{i=0}^{m} 4^i$ slides. Cada slide tem uma complexidade de O(n^2+*n*) em que *n* é o tamanho da matriz. Sempre que possível, reduzimos o *n,* dado que através dos slides o número de elementos na matriz diminui (*merge*) e os elementos resultantes ocupam lugares próximos do início (índices menores). É complicado determinar ao certo a complexidade temporal conseguida através dos vários testes de rejeição implementados, dado que estão diretamente relacionados com as especificidades de cada tabuleiro. Escolhemos utilizar uma hash table por nos permitir manter uma complexidade temporal de O(1), tanto para a inserção como para a procura.
Ao nível da complexidade espacial, a matriz tem uma complexidade O(n^2). A hash table tem uma complexidade de O(n) assim como a estrutura auxiliar (lista) utilizada nas funções de slide.

5. References

Para este projeto não recorremos a nenhum tipo de bibliografia. Todo o código e estratégia algorítmica foram pensados e projetados única e exclusivamente por nós.