Report for Programming Problem 1 - 2048

Team:

Student ID: 2018285621 | Name: Nuno Marques da Silva
Student ID: 2018285632 | Name: Pedro Tiago dos Santos Marques

1. Descrição do Algoritmo

Após a análise do problema (e por recomendação do professor) o grupo optou por tomar uma abordagem recursiva na resolução do mesmo. Sendo assim, uma das principais otimizações residia no quão cedo se conseguia parar a recursividade e chegar ao resultado correto. Para isso, obedecemos a várias estratégias que permitiram melhorar imenso a *performance* do programa. Note-se que todo o código foi desenvolvido em C++.

O algoritmo inicia com a leitura do *input*, gerando um vetor (tabuleiro), 'N' que corresponde ao número de linhas/colunas e 'M' que equivale ao número máximo de *swipes* permitidos. Também é criada uma variável auxiliar chamada "limiar" cujo valor inicial é 'M' + 1. De seguida, o tabuleiro passa por alguns testes, antes de se começar a aplicar *swipes* para encontrar a solução.

Primeiramente, é verificado se o tabuleiro é, de facto, possível. Para isso, é aplicado um *sort* ao tabuleiro, permitindo assegurar (de maneira eficiente) que ao dar *merge* em todos os números compatíveis no tabuleiro é sequer possível chegar a uma solução. Caso o tabuleiro não seja solucionável, devolve logo "no solution" e não são aplicados quaisquer *swipes*.

De seguida, é feita outra verificação onde se encontra o número mínimo de jogadas que se pode fazer para resolver o tabuleiro. Para tal, é utilizado uma versão *sorted* do tabuleiro e aplicamos o número máximo de *merges* possíveis, ou seja, (['N'/2] * 'N') junções por iteração até chegar à solução. Este *best case scenario* dá um valor útil que será usado em mais ocasiões. A verificação anteriormente referida reside na seguinte lógica, se é pedido para resolver um tabuleiro em menos jogadas do que o *best case scenario*, então a função retorna logo "no solution". Por exemplo, se no melhor dos casos só se consegue resolver um determinado tabuleiro em 10 jogadas, então nem vale a pena tentar encontrar uma solução para esse mesmo tabuleiro cujo 'M' seja 8, porque nunca se vai encontrar uma solução.

1.1 Passo Recursivo

Se o *input* passar por estes dois testes, então vai entrar na função recursiva que tem como objetivo encontrar a menor solução possível para o problema. Para tal, também foram implementadas várias otimizações. Note-se que esta função recebe como parâmetros o nível onde se encontra (corresponde ao número de *swipes* efetuados), o tabuleiro (podendo já lhe ter sido aplicado *swipes* ou não) e um inteiro (*path*) que identifica a direção do próximo *swipe*, '1' corresponde esquerda, '2' corresponde direita, '3' corresponde cima e '4' corresponde baixo.

Ao entrar na função, é feita logo uma rejeição. Se o nível atual for maior do que o 'limiar' ('M'+1 inicialmente ou a melhor solução atual) então para logo a recursividade. Se não se verificar esse caso, é aplicado o *swipe* correspondente (consoante o *path*). A função que está encarregue de fazer o swipe funciona da seguinte maneira:

1. Mover todos os elementos da primeira linha ou coluna para a direção pretendida (note-se que existe uma função *swipe* para todas as direções.
   Exemplo: [2][0][0][2] → [0][0][2][2] (*swipeRight*)

2. Fazer *merge* dos primeiros números iguais consecutivos.
   Exemplo: [0][0][2][2] → [0][0][2][4] (*swipeRight*)

3. Andar o resto da linha (tudo o que estiver a trás do número que levou *merge*) uma casa para a direita.
   Exemplo: [0][0][2][4] → [0][0][0][4] (*swipeRight*)

4. Repetir os paços 2. e 3. Até não ser possível realizar mais *swipes* nessa linha/coluna.

5. Voltar ao passo 1. Por cada linha/coluna existente no tabuleiro consoante a direção pretendida

Os *swipes* implementados colocam sempre o primeiro elemento do vetor a '-1' caso o tabuleiro não tenha sofrido nenhuma alteração. Após o *swipe*, verifica-se se o primeiro elemento do vetor é, de facto, '-1'. Se for, então não se continua a recursividade, visto que a melhor solução não contém *swipes* que não geram nenhuma alteração no vetor (tabuleiro).

Na eventualidade de encontrar alterações no tabuleiro, é feita uma verificação para ver se se encontrou uma solução. A função responsável por isso apenas percorre linearmente o tabuleiro e verifica se encontra apenas um elemento diferente de zero. Se sim, então atualiza-se o 'limiar' e para-se a recursão. Caso contrário, averigua-se se o próximo nível é menor do que o 'limiar' - 1 (para ver se se deve continuar na recursão). Se a premissa anterior for verdadeira, é feita uma última verificação onde se verifica se o *best case scenario* é maior do que o número de jogadas restantes. Por exemplo, se no melhor dos casos se resolver o tabuleiro atual com 5 jogadas, então não vale a pena continuar a recursividade se já só tivermos 4 jogadas restantes. Finalmente, se esta restrição for validada, então continua-se a recursão.

1.2 Estratégias descartadas

Aquando o desenvolvimento do programa, implementámos uma *hash table* que guardava passos intermédios do tabuleiro e as combinações já testadas nesse mesmo estado. Para tal, no passo recursivo fazia-se sempre uma pesquisa do tabuleiro atual na *hash table*. Se já existisse alguma entrada desse tabuleiro, verificava-se se já se tinha testado o 'path' atual. Se sim, parava-se a recursão, caso contrário, esse tabuleiro era atualizado com a direção do *swipe* atual e continuava-se a normal execução do programa. O objetivo desta estratégia era evitar repetir swipes em tabuleiros que já tinham sido anteriormente analisados. Todavia, esta estratégia demonstrou um ganho de eficiência quase nulo pelo que foi posta de parte da versão final do programa, face à quantidade de memória que utilizava.

2. Estruturas de Dados

No desenvolvimento deste programa foi apenas utilizado um vetor de inteiros que continha o tabuleiro todo. Para tal, as funções que operavam sob a mesma foram elaboradas de tal forma que se distinguia sempre em que linha ou coluna se estava a trabalhar.

3. Correctness

Para alcançar os 200 pontos no *Mooshak* começámos por calcular todas as combinações possíveis chegando a uma solução ótima. Não sendo o suficiente para chegar à cotação máxima, tivemos de pensar em estratégias que iriam encurtar o tempo necessário para achar a solução ótima:

Começámos por limitar o nível de profundidade da recursividade caso tivéssemos uma solução mais ótima, isto é, caso tivéssemos uma solução no nível 9 não é necessário calcular mais combinações para além do nível 9, inclusive.

Depois pensámos que é possível verificar logo à partida se uma matriz é solucionável ou não, tanto pelos números que estão na matriz como pelo limite de jogadas dado pelo utilizador.

Com estas otimizações não foi possível chegar à marca dos 200 pontos então vimo-nos obrigados a pensar numa otimização, em cada nível de recursividade, que verificaria qual dos ramos de combinações deveria ignorar pois esse não teria solução. O algoritmo também gera sempre a solução correta visto que testamos sempre todas as possibilidades que, de facto, podem gerar uma solução melhor que a atual que temos.

Com isto batemos a cotação pretendida. Optámos por uma boa abordagem dado que conseguimos otimizar bem o nosso programa sem que este dê soluções erradas.

4. Análise do Algoritmo

Num passo recursivo a complexidade temporal é $O(2n^2)$ dado que qualquer função *swipe* (simboliza uma jogada) e a função *best case scenario* partilham da mesma complexidade temporal $O(n^2)$. Já quanto à complexidade espacial, o passo é $O(n)$ devido ao facto de não usarmos nenhuma estrutura de dados adicional com informações dos últimos passos da recursão. Para além disso, não usamos funções recursivas dentro do passo recursivo o que garante a complexidade temporal anteriormente referida.

No caso base a complexidade temporal é $O(n^2)$ devido às verificações iniciais (cálculo do *best case scenario*) e a complexidade espacial é $O(n)$, pois não criamos estruturas adicionais para além da matriz inicial e também não usamos funções recursivas.

5. Referências

Todo o código apresentado foi desenvolvido pelo grupo sendo que apenas completamos a informação do relatório com informação proveniente dos slides disponibilizados na cadeira de Estratégias Algorítmicas. Finalmente, todas as estratégias de otimização foram introduzidas por nós ou pelo professor nas aulas teóricas.