Relatório para o Problema de Programação 1 – 2048

Equipa:

N.º de Estudante: 2018285941 Nome: Cláudia Filipa Peres Saraiva de Campos
N.º de Estudante: 2018275530 Nome: Dário Filipe Torres Félix

1. Descrição do Algoritmo

1.1 Eliminação à priori de tabuleiros/casos impossíveis:

Para determinar se um tabuleiro possui pelo menos uma solução, isto é, se um dado tabuleiro se pode reduzir a uma única peça independentemente das suas posições, determinamos se a soma de todos os seus elementos é uma potência de 2:

- Se a afirmação anterior se confirmar, então este possui uma possível solução e chama a função recursiva.

- Caso contrário concluímos que, o tabuleiro não tem solução, ou seja, nunca sobrará apenas uma peça no tabuleiro e por isso a execução termina sem chamar a função recursiva e imprime *"no solution".*

1.2 Backtracking

Não chamamos a função recursiva para os movimentos que não originem modificações no tabuleiro, ou seja, caso o movimento gere um tabuleiro igual ao tabuleiro original a função recursiva não é chamada.

Como pretendemos minimizar o número de movimentos necessários para que o tabuleiro termine com uma peça, possuímos uma variável "**best**" que irá conter o menor número de movimentos "atual". Esta variável será atualizada durante as várias chamadas da função recursiva. E para limitar o número de chamadas recursivas e assim optimizar este processo, não chamamos a função recursiva para as chamadas que sejam iguais ou que ultrapassem esse valor "**best**".

Contamos quantas jogadas/recursões ainda necessitamos, nas melhores das hipóteses, para reduzir o tabuleiro a uma e uma só peça, num dado momento, a partir da função $ceil(log_2(n\ elementos))$ = jogadas ainda necessárias. Se, num dado momento, o número de jogadas ainda necessárias for superior ao número efetivo de jogadas que ainda nos resta (M - numJogadaAtual), saímos da recursão porque mesmo que o tabuleiro tenha solução, nas melhores hipóteses, o tabuleiro nunca conseguirá chegar a uma solução antes do M (número máximo de jogadas) se esgotar.

1.3 Heurísticas

Para descobrir uma solução possível em menos tempo, isto é, em menos passos recursivos, calculamos o número de peças para cada tabuleiro resultante de cada movimento (cima, baixo, esquerda, direita) e ordenamos as "chamadas recursivas" (os tabuleiros que serão os parâmetros das próximas chamadas recursivas) por esse número de peças com o algoritmo de ordenamento da biblioteca do C++.

Ou seja, o movimento que causar uma maior redução no número de peças do tabuleiro será processado primeiro pelo passo recursivo, dando-nos uma maior probabilidade de encontrar uma possível solução em menos passos recursivos, reduzindo a complexidade temporal.

1.4. Recursão

Qualquer tabuleiro com que possua apenas 1 peça é caso base da função recursiva uma vez que é a condição de aceitação.
Por sua vez, o nosso passo recursivo consiste em aplicar cada um dos movimentos ao tabuleiro original obtendo assim os tabuleiros que resultam de cada um desses movimentos e chamar a função recursiva para cada um deles.

Por fim, rejeita-se a solução parcial quando o número de jogadas é superior ao número máximo de jogadas (**M**), ou quando o número de jogadas é igual ou superior ao menor número de jogadas atual (**best**), ou quando um movimento não origina mudanças no tabuleiro, ou quando o número de jogadas ainda necessárias for superior ao número efetivo de jogadas restantes.

2. Estruturas de Dados

Para representar um tabuleiro do jogo 2048 recorremos a uma matriz quadrada n por n, representada por um array unidimensional que armazena n^2 números inteiros. Se, em determinada posição, um dos elementos do array for zero, significa que não existe qualquer peça nessa posição. Caso contrário, se um dos elementos do array for positivo significa que há uma peça com esse mesmo valor nessa posição.

3. Verificação do Algoritmo (Correctness)

$$
\begin{aligned}
&\textbf{Function } calcGame(board, boardSize, countPlays, countNum)\\
&\quad \textbf{if } countPlays > maxPlaysNum \textbf{ or } countPlays \geq bestNum \textbf{ then}\\
&\quad\quad \textbf{return}\\
&\quad \textbf{else if } countNum = 1 \textbf{ then}\\
&\quad\quad bestPlayNum \leftarrow countPlays\\
&\quad\quad \textbf{return}\\
&\quad boards \leftarrow \textbf{new } array[1..4] \textbf{ of object Board}\\
&\quad initializeMovesBoards(boards, board, boardSize)\\
&\quad \textbf{sort } boards \textbf{ by } Board.countNum \textbf{ in ascending order}\\
&\quad \textbf{for each } b \textbf{ in } boards \textbf{ do}\\
&\quad\quad \textbf{if } b.isEqual = \textbf{false and } countPlays < bestPlayNum - \\
&1 \textbf{ and } b.countPlaysNeeded \leq (maxPlaysNum - countPlays) \textbf{ then}\\
&\quad\quad\quad calcGame(b.board, boardSize, countPlays + 1, b.countNum)
\end{aligned}
$$

A função *isSolvable2* é correta porque se cada peça do tabuleiro corresponde a uma potência de 2 e se quando duas peças se fundem, a peça resultante corresponde à sua soma e sendo que essa soma é também uma potência de 2. Portanto se um tabuleiro se puder reduzir a uma peça, então essa mesma peça terá de ser uma potência de 2, ou seja, calculando o logarítmo de base 2 da soma de todos os elementos irá determinar corretamente se o tabuleiro terá uma possível solução.

Assumindo que os movimentos esquerda, cima, direita e baixo estão corretos, concluímos que a função recursiva na sua forma básica é correta porque aplicando todos os 4 movimentos ao tabuleiro chama para cada um desses 4 movimentos, a função recursiva, ou seja, não excluí qualquer caminho.

Dito isto, nós aplicamos um ordenamento a estas chamadas recursivas com base no número de peças reduzidas (heurística) do tabuleiro

correspondente. Pelo facto de termos utilizado uma função de ordenamento da biblioteca standard do C++, assumimos que o ordemanento efetuado é correto.

A cada passo recursivo o algoritmo verifica se o número atual de jogadas é superior ao número máximo de jogadas **M** ou se o número atual de jogadas é igual ou superior ao menor número de jogadas já registado (**best**). Também verifica se algum tabuleiro resultante de um dos 4 movimentos resulta num tabuleiro igual ao original, ou se o número de jogadas restantes é inferior ao número de jogadas necessárias, na melhor das hipóteses, para chegar uma solução. Caso alguma destas condições seja verdadeira, então o programa retorna (condições de rejeição).

Abordemos com detalhe a última condição referida no parágrafo anterior: num tabuleiro de tamanho n, existem x número de peças. Na melhor das hipóteses, cada movimento, apenas pode reduzir em metade o número de peças, pela razão de que a redução numa só peça necessita sempre de 2 peças originais. Assim, se aplicarmos esta estratégia em movimentos sucessivos estaremos a calcular o $\log_2 x$, em que o resultado deste cálculo indica, na melhor das hipóteses, o número mínimo e necessário de movimentos que irá permitir à redução do tabuleiro a uma só peça.

Quando um determinado tabuleiro é reduzido a uma só peça e quando as condições descritas anteriormente não se verificam, atualiza o valor da variável que contém o menor número de jogadas **(best)** e retorna uma vez que já satisfaz as condições do caso base.

4. Análise Algorítmica

4.1. Complexidade Temporal

$T(M) = 4T(M-1) + 4N + C$

$\Leftrightarrow T(M) = 4(4T(M-2) + 4N + C) + 4N + C$

$\Leftrightarrow T(M) = 16T(M-2) + 20N + 5C$

$\Leftrightarrow T(M) = 16(4T(M-3) + 4N + C) + 20N + 5C$

$\Leftrightarrow T(M) = 64T(M-3) + 84N + 21C$

$\Leftrightarrow T(M) = 4^k T(M-k) + \sum_{i=1}^{k} 4^i N + \sum_{i=1}^{k} 4^{i-1} C$

$\Leftrightarrow T(M) = 4^M T(0) + \sum_{i=1}^{M} 4^i N + \sum_{i=1}^{M} 4^{i-1} C\ , \subseteq O(4^{M+1} N)$

$\Leftrightarrow O(4^M N)$

M corresponde ao nº máximo de jogadas;

N corresponde ao tamanho da matriz;

C corresponde a uma constante;

O valor **4** corresponde aos movimentos possíveis;

$\boldsymbol{T(M-1)}$ corresponde à chamada da função recursiva;

O algoritmo possui uma função adicional invocada sempre antes da função recursiva de complexidade $O(N)$. Como o este valor é muito inferior ao calculado anteriormente, então $O(4^M N + N) = O(4^M N)$.

4.2. Complexidade Espacial

A cada passo recursivo alocamos 4 vetores de tamanho **N** mais uma dada constante, **C**. Como a árvore de recursão tem profundidade **M**, então, a complexidade espacial corresponde a $O(4N \times M) = O(NM)$.

5. Referências

Slides disponibilizados na disciplina de Estratégias Algorítmicas. 😉