Report for Programming Problem 1

Team: 2019217030_2019227240

Student ID: 2019227240 Name: Sofia Botelho Alves

Student ID: 2019217030 Name: João Dionísio

1. Algorithm description

O algoritmo implementado começa por guardar as peças recebidas do input num vetor (“**pool**”), que, no nosso caso, irá servir como uma espécie de “saco” onde guardamos inicialmente todas as peças. Ao guardar a peça no saco, guardamos também as restantes configurações da mesma. Deste modo, no final, o nosso saco irá ter o quádruplo das peças (Figura 1).

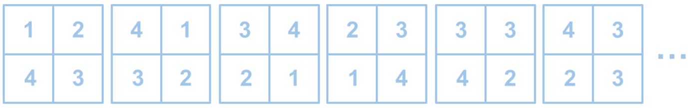

Figura 1 - Representação gráfica do vetor pool

Simultâneamente, vamos guardando todas as cores distintas num array (“**colors**”), que será depois usado para determinar se o puzzle é impossível ou não.

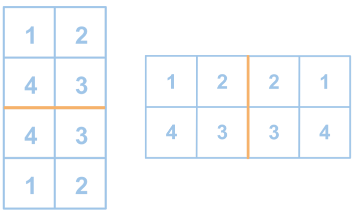

Figura 2 - Representação gráfica da combinação entre peças

De seguida, percorremos as peças que se encontram no saco e, para cada uma, voltamos a percorrer o saco calculando as peças compatíveis com a peça atual, isto é, aquelas que possuem lados em comum. Neste passo, estamos a comparar o lado de baixo da primeira peça com o lado de cima da segunda e o lado direito da primeira com o lado esquerdo da segunda (Figura 2).

Por fim, guardamos este conjunto de peças num vetor (“**compatibles**”) que ficará associado à peça atual. Este passo é crucial na otimização do problema, uma vez que, deste modo, em vez de estarmos a observar a totalidade das peças, analisamos um conjunto mais restrito e com uma maior probabilidade de encaixar no puzzle.

Para além disto, também eliminamos logo à partida alguns dos casos impossíveis através do array “**colors**” anteriormente referido**.** Sabendo que a cardinalidade de uma cor pode ser ímpar, no máximo, apenas quatro vezes (corresponde aos quatro cantos) e esta, só pode ser igual a dois para as cores que se encontram na borda do tabuleiro, conseguimos, logo à partida, achar alguns casos impossíveis.

Seguidamente, colocamos a primeira peça no tabuleiro (anchor piece) e chamamos a função que resolve o problema (“**solve_puzzle_v2”**). Esta é a função recursiva do nosso algoritmo. Nesta, verificamos onde é que a peça se encontra no tabuleiro, e, consoante a sua posição, averiguamos se esta se encaixa no tabuleiro. Se esta não encaixar, retornamos 0. Se encaixar, avançamos para um ciclo onde percorremos todas as peças do vetor “**compatibles”** da peça atual no tabuleiro e chamamos a própria função para cada uma delas. A função só retorna 1 se a iteração recursiva seguinte retornar 1, o que acontece quando todas as peças já estiverem colocadas no tabuleiro. Para verificar se as peças encaixam, poupamos imenso tempo ao remover as rotações, transformando-as em simples peças.

2. Data structures

Neste projeto foram usadas como estruturas de dados vetores, estruturas e arrays. A seguir, apresenta-se uma breve descrição de cada estrutura utilizada.

- **Vetor “pool”**: O “saco” onde colocamos, inicialmente, todas as peças com as restantes três configurações.
- **Vetor “table”**: Onde vamos colocando as peças à medida que elas vão encaixando. Representa o tabuleiro.
- **Vetor “colors”**: Guarda o número de vezes que cada cor aparece no tabuleiro.
- **Estrutura ”piece”**: Representa uma peça.
    - **Vetor “compatibles”**: O conjunto de peças compatíveis com a peça representada na estrutura.
    - **Array “config”**: Valores das cores da peça representada na estrutura.

3. Correctness

A nossa abordagem está correta porque desde o princípio que tivemos preocupação com a eficiência, rapidez e complexidade genérica do código.

Todos os “cortes” no nosso código garantem que a solução é a mais otimizada. O “corte” mais eficaz foi, sem dúvida, a utilização das rotações como peças normais do tabuleiro, uma vez que permitiu restringir a procura da peça seguinte às peças compatíveis com a última peça colocada no tabuleiro. Antes desta otimização, estava-se a considerar percorrer todas as peças que ainda não teriam sido utilizadas e a compará-las com a última peça do tabuleiro, o que consistia em analisar várias peças sem qualquer relação com a peça atual, gastando assim várias chamadas recursivas para testar com estas mesmas peças.

4. Algorithm Analysis

Para o cálculo da complexidade temporal, tivemos apenas em conta a função "**solve_puzzle_v2**", uma vez que as restantes linhas de código possuem uma complexidade que não é significativa comparando com a totalidade do algoritmo.

Na função "**solve_puzzle_v2**", apenas foi considerado o último ciclo *for* com a chamada recursiva no seu interior. Tivemos em conta o pior caso possível, que, assumindo que existem peças repetidas, teríamos uma complexidade $O(4^{n-1}(n-1)!) = O((n-1)!)$, onde n representa o número total de peças. Como dentro deste ciclo estamos a percorrer o vetor "**compatibles"**, o pior caso seria se este fosse constituído por todas as restantes peças e todas as suas configurações, portanto com um tamanho igual a $4(n-1)$, isto na primeira chamada recursiva. À medida que vamos avançando no puzzle, peças vão sendo colocadas no tabuleiro, passando a ficar com a sua variável "**used"** igual a 1, para indicar que não poderão voltar a ser usadas enquanto estiverem colocadas no tabuleiro. Deste modo, o número de peças disponíveis no vetor "**compatibles"** vai sendo menor a cada chamada recursiva, $4(n-1) * 4(n-2) * 4(n-3)$... , chegando assim à expressão $4^{n-1}(n-1)!$. Se assumirmos que não existem peças iguais, não seria possível com que todas as configurações de todas as peças pertencessem ao vetor "**compatibles"**. Assim, não teríamos o fator 4 a multiplicar.

No caso da complexidade espacial, analisando todas as estruturas de dados criadas, chegamos à expressão:

$$4n + n + 999 + (4(n-1))$$

"pool" "tab" "colors" "compatibles"

Deste modo, a complexidade espacial será $O(n)$.

5. References

https://www.cplusplus.com/
Powerpoints da disciplina disponibilizados pelo docente