Report for Programming Problem 1

Team:

Student ID: 2018298933 | **Name:** Gustavo Toste Bizarro
Student ID: 2019219057 | **Name:** Thomas Pereira Fresco

1. Algorithm description

O algoritmo desenvolvido pode ser dividido em duas partes. A primeira parte consiste numa espécie de pré-processamento do input inicial:

- Em primeiro lugar, são reunidas as ocorrências brutas dos números presentes nos cantos de todas as peças, sendo armazenadas num map do tipo {número:ocorrências}. Ao testar a divisão inteira por 2 em todas as ocorrências, se houver mais do que 4 que não sejam 0 (ou seja, que não sejam ocorrências pares), é possível perceber que um puzzle é impossível mesmo antes de começar a tentar resolvê-lo. No limite, apenas podem existir 4 números de cantos com frequência ímpar (os 4 cantos do tabuleiro/puzzle, que não precisam de encaixar com nada).

- Em segundo lugar, à medida que são criadas as peças, guardam-se ponteiros para esses registos em 2 mapas, um do tipo {par de cantos:peças que possuem o par} e outros {trio de cantos:peças que possuem o trio}. Este processo permite reduzir drasticamente o tempo de procura, acedendo ao mapa de pares para inserção de peças na 1ª linha ou na 1ª coluna (1 superfície de encaixe a satisfazer), e ao mapa de trios para inserção no "meio" do puzzle (2 superfícies de encaixe a satisfazer). Esta técnica surge em alternativa à primeira tentativa de resolução do problema, em que se guardavam as peças todas num único vetor.

A segunda parte inicia-se com a inserção da primeira peça, seguindo-se o processo de resolução, baseado em Backtracking. Em cada ciclo:

- verifica-se se a última posição da board está ocupada, ou seja, se puzzle está resolvido. Em caso afirmativo, o problema tem solução.
- Procura-se, nas peças já colocadas, a subsequência necessária para a inserção da nova: o par correspondente à face direita da peça à esquerda, no caso de inserção na 1ª linha, ou o par correspondente à face de baixo da peça acima, no caso de inserção na 1ª coluna. Tem em consideração a peça acima e a do lado esquerdo (trio) no caso de inserção no "meio" do puzzle.

- Encontra-se uma peça disponível com essa subsequência, roda-se até ficar na posição que encaixa, coloca-se na board, atualiza-se o seu estado para “usada” e guardam-se as últimas coordenadas modificadas do tabuleiro;
- Chama-se a função recursivamente;
- Se não houver uma peça possível de encaixar, retira-se a última peça colocada, modifica-se o seu estado para “disponível” e repõem-se as coordenadas da última posição do tabuleiro alterada, voltando aos valores anteriores.

2. Data structures

1. Class Piece - Representa cada uma das peças existentes.
    - vector<int> cores - vetor de 4 posições que guarda o valor dos quatro cantos/cores da peça;
    - bool used - assinala se a peça está a ser usada no tabuleiro ou não.

2. Class Board - Representa a construção atual do puzzle, onde vão sendo colocadas as peças.
    - int lSize, cSize - dimensões do tabuleiro/puzzle;
    - int l,c - coordenadas da posição atual a resolver da board;
    - map<int,int> contagem - guarda as ocorrências brutas dos números presentes nos cantos de todas as peças;
    - map<vector<int>,vector<Piece*>> pares - guarda as peças organizadas pelos pares que possuem;
    - map<vector<int>,vector<Piece*>> trios - guarda as peças organizadas pelos trios que possuem;
    - vector<vector<Piece>> board - matriz de objetos Piece que representa o tabuleiro/puzzle em si.

3. Correctness

A solução desenvolvida obteve a pontuação máxima (200 pontos) na plataforma Mooshak. A obtenção do máximo de pontos dever-se-á à utilização de uma técnica de recursividade Backtracking, aliada à pesquisa rápida de peças proporcionada pela organização do input em dicionários, bem como o descartar de soluções, à partida, impossíveis.

No entanto, o código construído apresenta imperfeições. Um aspeto facilmente melhorável seria a secção relativa à busca das rotações para o encaixe das peças. Com o intuito de cobrir as várias posições de encaixe no caso em que as peças apresentam 3 números iguais (2 posições possíveis), o algoritmo, de uma forma genérica, força uma primeira rotação para qualquer peça, antes sequer de

verificar se a posição atual serve. Numa solução melhorada, esta rotação extra poderia apenas ser executada nos casos em que, efetivamente, a peça em causa apresenta 3 números iguais, em vez de ser aplicada a todas as peças. Desta maneira, seria possível reduzir um pouco o tempo de execução.

4. Algorithm Analysis

A complexidade espacial do algoritmo depende maioritariamente das estruturas auxiliares construídas no princípio de cada caso de teste. O tamanho destas estruturas, bem como do "tabuleiro", depende exclusivamente do número de peças dadas no input, pelo que se traduz numa complexidade espacial O(n).

A construção dos 3 dicionários auxiliares ao problema tem complexidade temporal O(n).

No melhor caso, onde todas as peças são colocadas apenas uma vez e o puzzle é resolvido logo na primeira tentativa, estamos perante uma complexidade temporal O(n).

No pior caso, onde são testadas todas as combinações possíveis de peças e o puzzle apenas fica resolvido na última tentativa possível, estamos perante uma complexidade temporal O(n!).

Nos casos base, ou seja, quando é encontrado o puzzle completo ou é atingido o momento de recusar a peça a ser testada, temos um peso computacional pequeno e constante, pois, nesses casos a ação a ser concretizada trata-se apenas de um return true/false.

5. References

- S. , Digvijay, disponível a 8 de março de 2022 em: https://cppsecrets.com/users/14799115971199710810110010510311810510697121554864103109971051084699111109/Program-To-Left-Rotate-Array-In-CPP.php
- anmolsharmalbs, disponível a 12 de março de 2022 em: https://www.geeksforgeeks.org/slicing-a-vector-in-c/