Problema A - Impossible puzzle!

Nome: Fábio Miguel Rodrigues Vaqueiro **Nº estudante:** 2019222451

Nome: Marta Carreira Santos **Nº estudante:** 2019220054

Nome Mooshak: 2019222451_2019220054

1. Descrição do Algoritmo

Numa primeira abordagem sobre o problema proposto, começámos por apenas tentar introduzir as peças em linha. Inicialmente, coloca-se a primeira peça sem qualquer rotação, de seguida percorremos todas as peças comparando estas com a peça já colocada anteriormente. De modo a concluir o puzzle foi necessário implementar uma função que fizesse a rotação das peças, ficando assim concluída a primeira linha do puzzle.

De forma à execução ficar mais eficiente, recorre-se à recursividade ao criar uma função para colocar as peças sempre à direita umas das outras e testando as 3 rotações possíveis para cada uma. Foi necessário implementar duas funções recursivas que possibilitassem a adição de peças não apenas à direita, mas também em baixo na primeira coluna de cada linha, que retornam um valor booleano. Caso este seja *True* a recursividade prossegue e tenta colocar a peça seguinte, caso se verifique o contrário (*False*) a última peça colocada é substituída por outra peça que encaixe nessa posição.

Feita esta implementação, verifica-se se todas as pecas percorridas já foram colocadas no tabuleiro do puzzle (concluindo o problema) ou se existe pelo menos uma peça que não foi posta no tabuleiro. Sendo feita esta verificação nas funções recursivas e se o caso base for aceite, se na função de adicionar peças à direita existir uma peça na última coluna e na função de adicionar peças em baixo existir uma peça na última linha, fica concluído o puzzle com todas as peças colocadas e é feita a impressão deste, se existirem peças por colocar o output será “impossible puzzle”.

Em seguida, foi implementado um pré-processamento para reduzir a complexidade temporal para grandes casos de teste, sendo verificado consoante o input introduzido. Calcula-se a quantidade de vezes que cada número aparece nas peças do puzzle. Se no conjunto das quantidades a presença de números ímpares for superior a 4 (possibilidade de números sem correspondência nos cantos do puzzle) a construção do puzzle é impossível, existindo peças sem correspondência emitido o output “impossible puzzle”.

Por fim, adicionou-se um hashmap e uma função auxiliar para preencher a mesma procurando as peças compatíveis entre si. Consoante a última peça colocada são guardados os 2 números correspondestes ao lado, como chaves, onde podem ser colocadas peças em seguida (à direita ou em baixo).

Na função *compatibilidades()* são percorridas todas as peças e comparadas as peças que estabelecem correspondência com esse par de valores, caso necessário são feitas rotações nessas peças. Assim, os valores das mesmas são guardados na hashmap para posteriormente esta ser percorrida nas funções recursivas.

2. Estruturas de dados

Para as estruturas de dados foram implementadas um hashmap e uma classe Peca. A primeira, de modo a melhorar a eficiência do algoritmo, ao adicionar uma nova peça ao puzzle apenas é necessário percorrer as peças compatíveis ao invés de percorrer todas as peças. Sendo assim, a complexidade temporal terá um valor menor.

A segunda, é constituída por variáveis com os valores da peca, com a flag se a peça já foi posta no puzzle e a função de rotação da peça. Esta foi introduzida dentro da classe pois facilita a complexidade espacial, não sendo necessário guardar os valores das peças rodadas posto isto os valores das peças são alterados na própria estrutura.

3. Correções

Tendo em conta o algoritmo implementado e a cotação máxima alcançada no mooshak, a abordagem sobre o problema feita foi a correta.

Para se obter este resultado positivo foi essencial a implementação de um hashmap utilizado nas funções recursivas, pois para casos de testes maiores este reduz significativamente o número de comparações a realizar aquando se tenta introduzir uma nova peça, e de um pré-processamento que verifica previamente se através do input é vantajoso fazer comparações de modo a construir o puzzle.

4. Análise do Algoritmo

Na complexidade espacial teremos de ter em conta todas as chamadas das funções recursivas. A cada peça posta teremos 2^n chamadas de função e em cada chamada o maior gasto de memória vem do armazenamento das peças introduzidas anteriormente, para realização do backtracking que neste caso terá complexidade $O(n^2)$. Desta forma, com variáveis constantes teremos: $\mathrm{S(n)} \in O(2^n * n^2)$.

Na complexidade temporal teremos de ter em conta a utilização de um hashmap que terá complexidade, no pior caso, de $O(n * \log n)$. A utilização de funções recursivas também terá impacto, pois depende do input, tendo como complexidade, no pior dos casos, $O(n)$. Assim a complexidade temporal para este algoritmo, no pior dos casos, será: $O(n^2)$.

5. Referências

- Material disponibilizado no UCStudent
- Documentação c++
- Documentação sobre Hashmap
- https://www.geeksforgeeks.org/
- https://en.algorithmica.org/hpc/
- https://hackingcpp.com/cpp/cheat_sheets.html
- http://jeffe.cs.illinois.edu/teaching/algorithms/book/01-recursion.pdf