Relatório do Problema 1

2019215917 Inês Marçal

2019219433 Noémia Gonçalves

1. Descrição do algoritmo Pré

processamento:

Foi colocado o índice das peças num unordered_map onde a key correspondia aos possíveis lados das mesmas, sendo estes representados por um array {int,int}.

Percorrendo o unordered map, para cada par era efetuada a comparação entre o número de ocorrências e o seu complementar, marcando o mesmo como visitado para tornar o processo mais eficiente. Ao mesmo tempo, eram também contabilizados os pares sem correspondência no par complementar. Caso este número fosse superior ao perímetro do tabuleiro, dado por 2 * nColunas + 2 * nLinhas, o puzzle seria logo impossível e avançava-se para o próximo caso de teste, sem entrar no backtracking novamente.

Backtracking:

Foi definido como caso de paragem o número de peças já colocadas no tabuleiro ser igual ao número de peças do input.

Posteriormente, de acordo com a posição onde a peça pudesse vir a ser inserida definiu-se três casos, a peça encontrar-se na primeira coluna, encontrar-se na primeira linha ou então numa das restantes posições. Inicialmente ainda é efetuada uma verificação para o caso em que o y (a coluna em que nos encontramos) seja superior ao número de colunas pressuposto pelo puzzle (mudança de linha no tabuleiro), incrementando assim o x (linhas) e o y (colunas) regressa a 0.

Relativamente aos 3 casos referidos, se a peça se encontrar na primeira linha verifica-se a compatibilidade apenas com a que se encontra no lado esquerdo, se se encontrar na primeira coluna verifica-se a compatibilidade com a peça acima e para as restantes a compatibilidade com ambas as peças (a acima e à esquerda). Apesar disso a abordagem é semelhante, começa-se por percorrer o array, criado no pré processamento, onde se encontram os índices das peças compatíveis, se a peça se encontrar inserida no tabuleiro continua-se para a seguinte. Caso contrário, para cada rotação verifica-se se encaixa na posição em causa e se for o caso coloca-se no tabuleiro. Antes de se iniciar a chamada recursiva a peça é marcada como utilizada e incrementa-se o número de peças presentes no tabuleiro. Assim é iniciada a chamada recursiva com a próxima posição do tabuleiro, caso esta retorne *false* o número de peças é decrementado e a peça é novamente marcada como não usada. Se as peças forem todas percorridas sem nunca retornar *true* a função retorna *false*.

2. Estruturas de dados

As estruturas de dados utilizadas foram as seguintes:

-Struct piece: esta contem um array[4] para guardar os lados da peça e um bool isOnBoard para verificar se a peça se encontra ser utilizada.

-Struct pieceSide: esta contem um vector<int> para guardar os índices das peças que compreendem o lado (a,b) e um bool isVisited para otimizar o pré-processamento.

-Array [2500][piece]: este permite guardar as peças.

-Array[50][50][4]: este representa o tabuleiro e as peças inseridas.

-Unordered map<char,int[4]>: onde se relaciona a posição de cada número nas 4 rotações permitidas com a posição do mesmo na peça original.

-Unordered map<int[2], pieceSide>: onde se guarda a informação dos lados e das peças que os contêm.

3. Correctness

No sentido de melhorar o algoritmo desenvolvido foram implementadas as seguintes técnicas:

- Um puzzle impossível pode ser detetado no pré processamento sem que a recursão seja iniciada.
- A separação das peças por lados diminui significativamente as possibilidades para cada posição do tabuleiro.
- A rotação de peças é feita sem recorrer a atribuições, utilizando apenas a peça original e o unordered map dos índices de rotação.

4. Análise do algoritmo

Analisando a complexidade temporal do pré-processamento e assumindo o pior caso da mesma (a peça tem todos os lados diferentes), a inserção dos lados terá complexidade O(n) e o tamanho do unordered_map ao ser 4N (sendo o N o número de peças), percorrê-lo levará a uma complexidade O(4n). Deste modo o pré processamento todo ficará com uma complexidade O(5n).

No backtracking, assumindo as condições do enunciado (primeira peça já colocada), o passo recursivo tem complexidade temporal:

$T(n) = (n-1)* T(n-1) + c$

$= (n-1) * [(n -2) * T(n-2) + c] + c$

$= (n-1) * [(n-2) * [(n-3) * T(n-3) + c] + c] + c$

$= \ldots$

deste modo ao fim de k chamadas recursivas a expressão final será:

$= \frac{(n-1)!}{(n-k)!} * T(n-k) + \sum_{k=1}^{n-k} \frac{(n-1)!}{(n-k)} c$

$= \ldots.$

$= (n-1)! * T(0) + \sum_{k=1}^{n-1} \frac{(n-1)!}{(n-k)} c$

Onde, T(n) representa a complexidade temporal total, c o custo computacional adicional de cada chamada recursiva e o n o número de peças.

Na primeira linha, começou-se com (n-1) chamadas recursivas (pior caso), onde posteriormente T(n-1) irá dar origem a (n-2) chamadas recursivas e assim sucessivamente.

Considerando também a complexidade do case base (constante), podemos assim verificar que o algoritmo no pior caso apresenta uma complexidade temporal de O((n-1)!), no entanto considerando a potência de grau mais elevado a complexidade temporal será $O(n^{n-1})$.

Em termos de complexidade espacial, no pior caso, utilizamos N para o board uma vez que o mesmo apresenta um tamanho total de R * C e este contém N peças, N para o array de peças e 4N para o unordered_map. As restantes estruturas de dados não ultrapassam a complexidade de O(n). Deste modo, a complexidade espacial mais relevante será O(n).

5. Referências

Estrutura para utilizar unordered map com arrays como keys adaptada de [1] https://stackoverflow.com/questions/42701688/using-an-unordered-mapwith-arrays-as-keys

[2] https://www.cplusplus.com/

[3] https://en.cppreference.com/w/