Relatório do Problema de Programação 1

Equipa:

N.º Estudante: 2019216949 Nome: Duarte Emanuel Ramos Meneses
N.º Estudante: 2019213995 Nome: Patrícia Beatriz Silva Costa

1. Descrição do Algoritmo

Tendo as peças já nas estruturas que definimos, o nosso algoritmo trata de encaixar a primeira peça na primeira posição do tabuleiro, colocando essa mesma peça como utilizada. Decidimos que a nossa abordagem iria, para a peça que acabou de entrar no tabuleiro, procurar as que pudessem encaixar na posição seguinte. Caso o puzzle apenas tenha uma coluna, encaixamos a peça seguinte em baixo da já colocada. Caso contrário, encaixamos sempre à direita (percorrendo linha a linha, sempre da esquerda para a direita).

Como guardamos as peças como um conjunto de vetores (cada um representando um lado), foi-nos fácil reduzir o espetro de procura para cada posição. Se formos encaixar uma peça à direita de uma em que o seu vetor à direita é (2,3), a nova que entra terá de ter o vetor à esquerda (3,2) (ao contrário pois convencionamos ler sempre no sentido horário) e basta procurar entre as peças no vetor da posição (3,2) da matriz que definimos para o efeito. Tendo as peças com um determinado vetor guardadas e de fácil acesso (explicado na secção 2 deste relatório), é fácil fazer uma procura rápida e eficaz de potenciais peças a encaixar.

Ao ter o vetor dessas possibilidades, percorremo-lo chamando recursivamente a função *check*. Esta função vai encaixar a peça (pois as verificações de que encaixa já foram feitas na chamada recursiva anterior), calcular a posição em que vai encaixar a próxima peça (próxima coluna da mesma linha se ainda não chegamos ao fim dessa linha ou primeira coluna da próxima linha caso contrário), e realizar as verificações necessárias para ver as peças que podem encaixar nessa posição. Tendo um vetor de possibilidades, vai repetir todo o processo novamente.

No entanto, se para uma determinada posição não existirem possibilidades, o vetor devolvido vem vazio, pelo que não se percorre nada e a função retorna 0. Posto isto, onde tinha sido chamada, caso o vetor de possibilidades aí ainda não tiver sido percorrido até ao fim, vai tentar encaixar a próxima possibilidade. Caso contrário, esta função também retorna 0 e assim sucessivamente. Se chegar 0 à primeira chamada da função, é sinal que não se encontrou uma solução para o puzzle, sendo este impossível.

Para tornar o algoritmo mais eficiente, decidimos utilizar alguns truques para que este corresse mais rapidamente. Percebemos desde logo que a melhor maneira de o conseguir seria detetar os "*impossible puzzles*" o mais cedo possível. Para tal, recorremos a várias técnicas. Uma delas foi verificar, sempre que colocávamos uma peça antes da última linha, se existiam possibilidades que encaixassem na sua parte de baixo. Isto ajudou a encontrar puzzles impossíveis

mais rapidamente pois poupávamos o tempo de preencher 1 linha e chegar à conclusão de que naquela posição não havia nenhuma peça que encaixasse. Caso não existam peças que encaixem, a função retorna 0 e o raciocínio explicado anteriormente repete-se.

Isto leva-nos a concluir que as condições de rejeição são:

- Não encontrar peças que encaixem na posição a seguir à encaixada;
- Não encontrar peças que encaixem abaixo da encaixada.

O nosso caso base é o puzzle estar completo. Ao chegar à última coluna da última linha do tabuleiro com as peças todas encaixadas, retorna 1 (true) até chegar à primeira chamada recursiva. Acontecendo isto, a solução é impressa.

Já o nosso passo recursivo assenta em procurar a próxima peça que encaixe na próxima posição. Vai chamar a função de forma recursiva e volta para trás de acordo com o que é retornado.

Era-nos dito que, à exceção da peça inicial, todas as outras podiam rodar. Deste modo, para poupar tempo no algoritmo, decidimos representar essa rotação através de uma variável que nos indica que lado da peça está voltado para cima. Como definimos tudo no sentido horário, tendo o lado que está para cima, temos as posições dos outros lados por posição relativa.

Outra técnica que utilizamos para otimizar o código é, sempre que acabamos de ler o input de um caso de teste, calculamos o número de vezes que cada valor aparece. Após alguma análise, descobrimos que, tendo um contador para cada valor existente nas peças, para o puzzle ser possível de montar, tem de ter, no máximo, 4 em que esse contador seja ímpar (os cantos). Isto acontece, pois, os cantos do puzzle são as únicas posições em que os valores não têm de “encaixar” com outros. Ao terem de “encaixar”, os valores têm que ter logicamente um valor par. Deste modo, embora não saibamos que o puzzle é possível de montar, se tiver mais que 4 valores de contador ímpares, ficamos com a certeza de que é impossível.

2. Estruturas de dados

Decidimos que a melhor maneira de representar as peças seria através de uma estrutura composta por um vetor de vetores de par de inteiros (em que cada vetor representa um lado da peça, lado esse representado pelo par de inteiros), uma variável que nos indica que lado está para cima (para sabermos como a peça está rodada) e outra variável indicativa se a peça já foi usada ou não.

Ao ler as peças do input, colocamo-las num vetor. Aquando dessa leitura no input, vamos colocando cada lado da peça numa matriz (vetor de vetores de vetor de par de inteiros). Para cada lado da peça, por exemplo, (2, 3), adicionamos ao vetor da posição (2, 3) da matriz um par de inteiros em que a primeira posição indica a posição da peça no vetor correspondente e a segunda indica a posição do vetor correspondente ao lado da peça. Tendo isto, quando queremos encaixar uma peça através de um lado (3, 2), basta percorrer o vetor que está à posição (2, 3) desta matriz (coordenadas trocadas pois definimos sempre que iríamos implementar de acordo com o sentido horário).

Tal como já explicado acima, temos também um vetor de 1000 posições em que cada uma representa um valor possível nas peças (0 a 999). Ao ler as peças vamos incrementando a posição correspondente no vetor para depois verificar se o puzzle é à partida impossível (explicado na secção anterior).

Para finalizar, decidimos representar o tabuleiro como um vetor do tamanho do número de peças em que as inserimos acedendo à posição row*columns+col.

3. Correção

Pensamos que o nosso algoritmo está correto uma vez que tentamos encontrar puzzles impossíveis rapidamente na execução do código através dos mecanismos já explicados acima.

Também como a matriz em que constam as peças que contêm determinado lado nos ajuda a reduzir o espetro de procura, o nosso algoritmo demora pouco tempo a testar as possibilidades.

4. Análise do Algoritmo

Complexidade espacial: Sendo que no nosso algoritmo temos um vetor de tamanho N, outro de tamanho N em que em cada posição há 4 vetores de pares, outro de K e outro vetor de vetores de vetor de tamanho K por K (sendo K o máximo da gama de valores possíveis), a complexidade espacial geral é:

$$N + N * 2 * 4 + K * K => 9N + K * K => O(N + K^2)$$

A complexidade espacial é igual tanto no caso base como no passo recursivo.

Complexidade temporal: No nosso algoritmo percorremos 1 vez as peças todas na leitura e outra no output. Entre estes dois momentos, percorremos sempre, até ter todas as peças colocadas, uma parte (variável) da totalidade das peças que vamos designar de N/j (sendo j no pior caso 1). Deste modo, a complexidade temporal geral é dada por:

$$N + N + \left(\frac{N}{j}\right) * (N-2)! => 2N + N(N-2)! => 2N + \frac{N!}{N-1} => o\left(\frac{N!}{N-1}\right)$$

No caso base, a complexidade temporal é a mesma, embora tenha menos 1 ciclo de tamanho N (o de output). Já no passo recursivo, a complexidade é a mesma que no caso geral.

5. Referências

- https://stackoverflow.com/questions/2151084/map-a-2d-array-onto-a-1d-array [acedido em 07/03/2022]
- https://stackoverflow.com/questions/24333170/put-a-multidimensional-array-into-a-one-dimensional-array [acedido em 07/03/2022]
- https://sergioprado.org/otimizacao-de-codigo-em-linguagem-c-parte-2/ [acedido em 09/03/2022]
- https://www.geeksforgeeks.org/vector-in-cpp-stl/ [acedido em 10/03/2022]