Report for Programming Problem 1

Team: 2019216646_2019216792

Student ID: 2019216646 Name: António Pedro Correia

Student ID: 2019216792 Name: Mário Guilherme de A. Martins Sequeira Lemos

1. Algorithm description

-In order to make the program more efficient while reading the information on every piece we do a few operations. First, we decompose every piece into four (corresponding to each one of its' possible rotations) adding them to the vector of pieces explained in point 2. We then iterate trough all pieces in the pieces vector first checking if the new piece can be placed in the right or bottom of any already present piece; and secondly checking if any of the pieces in the vector can be placed in the right or bottom of the new piece. In case it can be placed, the index of the piece will be stored in the vector (right, bottom) of the given piece. Because of this step, later when we recursively try to construct the puzzle, we don't need to constantly check all the pieces to see if they can be placed, only needing to check the vector of given piece.

-Our base case happens when the total of pieces placed matched the result of the multiplication of the With and the Height of the board.

-We try to construct the puzzle by placing one piece at a time, starting from the top left, and going from left-to-right and top-to-bottom. Because of this approach, in the recursive step we must check where we are on the board, as the restraints for placing a piece might be different.

-First, we check if we are in the first row of the puzzle; in this case we only need to check if a piece can be placed on the right of the last placed piece; for this we access the vector **right** of the last placed piece where all pieces that can be placed in the current board position are listed.

-Secondly, we check if we are in the first column of the puzzle; in this case we only need to check if a piece can be placed on the bottom of the piece in the current row – 1; for this we access the vector **bottom** of the piece specified before, as it contains a list of all the possible pieces for the current board position.

-If the recursion doesn't enter the above cases, then it can move on to a more generic approach. For any piece that enters this case we need to check the **right** vector of the last placed piece (similarly to the first case) and the **bottom** vector of the current row – 1 (similarly to the second case). We now know that only a piece that appears in both vectors can be placed in the current position, so we determine the list of pieces possible.

- In the cases where the list of placeable pieces is empty or there are no pieces left to try to place, our function will end that recursive branch and return false, removing the last placed piece. This is our rejection case.

2. Data structures

-We started by creating a struct called **Piece** to store all the information we need about the pieces given. This struct is composed by 4 int's, a bool and two vectors of int's. The four int's represent the colors present in every piece. The bool is just for making the identification of the state of a piece easier (if it's already placed on the board or not). The two vectors are for pre-processing techniques, storing the indexes of the pieces (in the Pieces vector explained in front) that can be placed on the right and on the bottom of the piece we are storing the information of in each vector correspondingly.

-In the beginning of the program it's created a vector of Piece (struct presented above) to store all the information given by the input.

-It's also created a vector named **colors** to store the number of appearances of every color in every piece to make the identification of an Impossible Puzzle more efficient.

-To be able to recursively simulate the construction of the puzzle, we created an array of int's. This array is initialized with 0 and has a size corresponding to the total number of pieces in the puzzle.

3. Correctness

We managed to achieve a score of 200 in Mooshak mainly because:

- we have a small usage of memory overall, as we only create two vectors and one array. One of the vectors has a fixed size of 1000 (color range) for a technique that checks the validity of the puzzle before any recursion begins;

-we opted for using an array instead of a vector for the board as it was made multiple accesses to it in the recursive step. This decision lowered the execution time significantly and removed de Time Limit Exceeded problem;

-the decision to store the two vectors in each piece struct while reading the input made the code much more efficient because it meant that in every recursive step, we didn't have to go through all the pieces that weren't placed yet in order to select the correct one to place next;

-in the recursive step, when trying to place a piece that is not positioned in the first line or the first column (as these are special cases) by comparing the size of the right vector and the bottom vector and iterating by the smallest of the two, we also noticed a significant decrease in execution time;

-by storing the number of appearances of every color present in every piece, if there are more than 4 odd numbers, we can immediately reject the puzzle; since (by the rules of the Impossible Puzzle) only the four corners can have a color that appears an odd number of times.

4. Algorithm Analysis

-The space complexity of our algorithm is O(2N + R + B) where R and B are, respectively, the amount of connections that can be made on the right and bottom of every piece.

-The 2N comes from storing both the pieces received from input and the board where the those are placed.

-Since in the worst case we will have to iterate trough, every piece expect the one we are placing, the time complexity is given by:

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

$$T(n) = (n - 1) * ((n - 2) * T(n - 2) + C) + C$$

$$T(n) = (n - 1)(n - 2) * T(n - 2) + (n - 1)C + C$$

$$T(n) = (n - 1)(n - 2) * ((n - 3) * T(n - 3) + C) + (n - 1)C + C$$

$$T(n) = (n - 1)(n - 2)(n - 3) * T(n - 3) + (n - 2)(n - 1)C + (n - 1)C + \mathrm{C}$$

…

$$T(n) = (n-1)(n-2)\dots(n-n) * T(n-n) * \left[\dots + \frac{(n-2)(n-1)}{(n-1)!} + \frac{(n-1)}{(n-1)!}\right] c(n-1)!$$

$$T(n) = 0 * 0 + \left[\dots + \frac{1}{(n-3)!} + \frac{1}{(n-2)!} + \frac{1}{(n-1)!}\right] c(n-1)!$$

$$T(n) = \left[1 + \frac{1}{2!} + \frac{1}{3!} + \cdots + \frac{1}{(n-1)!}\right] c(n-1)!$$

$$T(n) = [\textstyle\sum_{k=1}^{n-1} \frac{(n-1)!}{(n-k)!}]\mathrm{c}$$

5. References

-C++ Cheat Sheets: https://hackingcpp.com/cpp/cheat_sheets.html