Report for Programming Problem 1

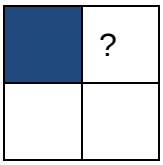
Team:

Student ID: 2019243695 Name: Tiago Filipe Santa Ventura

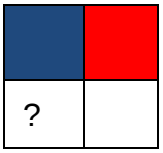
Student ID: 2019217590 Name: Sancho Amaral Simões

1. Algorithm description

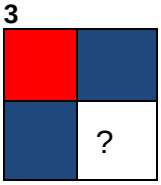
1



2



3



The algorithm starts by reading the input. When a piece is read, it is stored and pre-processed. This **pre-processing** involves the addition of the piece's **left side**, **top side** and **left-top side** (equivalent to the tuple (left side, top side)) in all four **rotations**, if absent, as keys to three maps: *left*, *up*, and *upLeft*. Whether those keys are absent or not, the piece in question is added to the list that is the value correspondent to that key. Then, for each corner number of the piece, the number of occurrences of that number in the current test case scope is incremented. Before calling the main solving function, a verification is run to test whether the given puzzle configuration is possible or not: this happens if there are more than four numbers with an odd number of occurrences. Every number must have a pair with exception of one number in all four corner pieces, because they will only connect with two other pieces. Since this is a problem that requires the application of the backtracking programming technique, a recursive function is then called to solve the puzzle for every piece, starting in the initial board position - *(0, 0)*. The current piece is marked as used and is put in the current board position. If we reach the final board cell – *(R – 1, C - 1)* - it means that a solution has been found and the function returns true – **base case**. Otherwise, the program will proceed to the retrieval of the possible candidates for the next position the recursive function will be called on. These candidates vary from the next cell position: if it is in the first row, we will have to fit a piece that has the left side equal to the current piece right side, so we will get the candidates via the map *left* (scenario **2**) with the current piece right side as key; if it is in the first column, we will have to fit a piece that has the top piece equal to the bottom side of the piece above, so we'll get the candidates via the map *up* (scenario **1**) with the above piece bottom side as key; otherwise, we will have to take into account the piece above and the previous piece, so we'll consult the map *upLeft* (scenario **3**) with key (*leftside*, *topside*). Having the candidates for the next board position, we will loop through them and call the recursive function for every unused candidate piece, in the next board position – **recursive step**. If it's possible to solve the puzzle with a certain candidate, the recursive calls will keep going until reaching the final board cell. If there are no candidates or they are all used (**rejection condition**), the current piece will be marked as unused and will be removed from the board. The function will then return false and this value will propagate upwards in the recursion stack, causing the algorithm to backtrack and to try with another piece. The main features of this algorithm that would speed up the solution acquisition are: the preprocessing with pieces sides and the odd number occurrences trick.

2. Data structures

The main data structures used were:

- *Piece* – a created class that contains the corner numbers of an original piece.
- *RotatedPiece* – a created class that contains the corner numbers of a rotated piece and the type of rotation (*0º, 90º, 180º* or *270º*).
- *PieceSide* – a created class that contains the numbers of a piece side.
- *PieceSides* – a created class that contains the numbers of a piece's two consecutive sides (in this case, the left and top sides).
- *HashMap* – contains all the **pre-processing** data referred in **1.**, providing fast lookup time (usually *O(1)*, *O(N)* worst case).
- *ArrayList* – serves as the value for the previously mentioned *HashMap*s.

3. Correctness

Even though our algorithm may be a bit heavy due to the extensive usage of *Java* classes, objects and *HashMap*'s, we were able to reach the 200 points with it because it did a very complete preprocessing, avoiding any kind of expensive computation while in the recursive function, e.g., by testing all possible combinations of pieces. This, plus the corner numbers odd occurrences trick, would suffice (we thought, and well), to avoid a *TLE*. Also, a *WA* was not in our roadmap, because we started by designing a solution that would include all the possible scenarios and then, after validating its universality, we proceeded to fine tune it in order enhance its execution time.

4. Algorithm Analysis

Time complexity: *O(N!)* worst case – even though this is a backtracking algorithm with some cuts, which speed up its execution time considerably, there may be a hard test case that requires to test of all combinations. In this extreme case, the first piece is put and then N-1 remain and then N-2 and so on until reaching the final board position… This is the rational behind the *O(N!)* time complexity. It is important to point out that this complexity is solely due to the **recursive step.** The **base case** time complexity is *O(1)* since it only implies an array access, to check whether we reached the final board cell.

Space complexity: *O(N)* – the space required is the one used to store the board pieces (*O(M * N) = O(N)*), the array of pieces (*O(N)*), the three pre-processing *HashMap*s (*O(N * 4 * 3) = O(12N) = O(N)* – four rotations), and the array that stores the numbers occurrences (*O(1000) = O(1)* - constant). So, in total, we have *O(N) + O(N) + O(N) + O(1) = 3O(N) + O(1) = O(N) + O(1) = O(N)*.

5. References

- Shirriff, Ken, *Ken Shirriff's blog*, *Solving edge-match puzzles with Arc and backtracking*, http://www.righto.com/2010/12/solving-edge-match-puzzles-with-arc-and.html, accessed in 15/03/2022.