Report for Programming Problem 1

Team:
Student ID: 2019216747 Name: João António da Silva Melo
Student ID: 2019216809 Name: Miguel António Gabriel de Almeida Faria

1. Algorithm description

The program starts by reading the input as it's stated in the Problem. There are 1000 possibilities for the colours of each corner in a piece, so 5 vectors are created with different purposes: countColorsAux, upCompAux, leftCompAux, board and pieces.

As each piece is read, the piece and all its configurations are stored so that it's easier to know if a piece can be inserted in a specific position.

Before using the algorithm, the vector countColorsAux[1] is checked to see if there are more than 4 colours that don't have an even number of occurrences. By doing this we can verify that the puzzle is impossible straight away. The board is also initialized so it isn't needed to calculate the next position every time the recursive function is called.

After this, the first piece is inserted in the board and the recursive algorithm to insert the other pieces starts.

According to the algorithm, the pieces are inserted row by row, from the left to the right. There are 3 possibilities for the insertion:

- The piece is inserted in the first row, so it must match with the piece to its left.
- The piece is inserted in the first column, so it must match with the piece above.
- The piece is inserted in the remaining positions, so it must match with the piece above it and the piece to its left.

To obtain the pieces and configuration that are compatible with the spot to insert on, we search on upCompAux and/or leftCompAux and iterate through them. If there are none, the puzzle is either impossible or there are other compatible pieces in a previous recursion.

When a piece is compatible, it is inserted in the board and the function to insert pieces is called recursively. When the recursion on that piece is over, it's verified if the puzzle is possible with the piece in the position; if it isn't, the piece is removed from the board.

The base case of the algorithm is on the start of the recursive function, where it's checked if the last position of the board has a piece inserted. If so, the puzzle is possible and the recursions ends.

For this approach we used some speed-up tricks. In the beginning of the program, we start by making pre-processing of all pieces and their configurations, so we can later search for pieces more efficiently. We also stop the search[2] for pieces that need to fit with the one above it and the one to its left; by having 2 vectors with all compatibility possibilities, when searching for the same piece, since the pieces are in order, if the index in one of them is greater than the other, it's impossible to find the piece.

2. Data structures

- BoardCell – it's a struct used for the cells of the board, that stores the row position, column position, the piece and if it is inserted.
- Piece – it's a struct used to store all the configurations of a piece and if it is used.
- countColorsAux – used to check the number of occurrences of each colour.
- upCompAux – tri-dimensional vector used to store the piece and configuration that match the indices, which are the upside of a piece in all configurations.
- leftCompAux – tri-dimensional vector used to store the piece and configuration that match the indices, which are the left side of a piece in all configurations.
- board – vector that simulates the board
- pieces – vector that stores all the pieces

3. Correctness

Our algorithm solves correctly and efficiently the problem stated. By using pre-processing of the data to solve the problem, we don't need to test with all the pieces and configurations each time we do the recursive step. We also check the cases that make the problem immediately impossible, as stated before, involving the vector countColorsAux[1]. Another optimization that we made was stopping the search for the same piece when trying to match with the piece above and to the left[2].

Although we were able to achieve the maximum score, our approach could be improved. As we mentioned before, we use 2 tri-dimensional vectors that allocate a lot of unused memory. This could be solved by using *hash maps*, where both colours of the side would be the key and the pieces' number and configuration the values. This way, we only store the used colours.

4. Algorithm Analysis

In the resolution of this problem, we used backtracking, so, to analyse the time complexity, we need to consider the number of recursive calls in the worst-case scenario. In some situations, there would be cases where our algorithm would be almost instantaneous, because some of these make the puzzle impossible or when there's little to no additional recursion. Let T_s denote the cost of stack operations at each recursive step and T(n) be the temporal cost of the recursion call. This way, unrolling the recurrence of our algorithm would look like the following:

$$
\begin{aligned}
T(n) &= (n-1) * T(n-1) + T_s \\
&= (n-1)*(n-2)*T(n-2) + (n-1)T_s + T_s \\
&= (n-1)*(n-2)*(n-3)*T(n-3) + (n-1)*(n-2)*T_s + (n-1)*T_s + T_s \\
&= (\dots) \\
&= \frac{(n-1)!}{(n-k)!} * T_s + \sum_{i=1}^{k} \frac{(n-1)!}{(n-i)!} * T_s \\
&= (\dots) \\
&= (n-n) * (\dots) * (n-1) * T(0) + \sum_{i=1}^{n} \frac{(n-1)!}{(n-i)!} * T_s \\
&= (n-1)! * T_s + (n-2)! * T_s + (\dots) + (n-1) * T_s + T_s \quad \in \mathbf{O((n-1)!)}
\end{aligned}
$$

In the first recursion, T(n-1), (n-1) represents the number of recursions that are possible to make, since, in the worst-case scenario, there are (n-1) pieces left to choose from.

When it comes to spatial complexity, our approach uses a relatively large amount of memory, since we use multiple data structures. The first one we use is a vector with 1000 positions where each position is an Integer. The following data structure is a vector with N of size; however, it is a vector of board cells which allocate 7 of space. To have all the pieces and configurations, we use a vector with N elements, each element being a piece with size 17. At last, there are 2 tri-dimensional vectors created, each of these has 999 rows and 999 columns and, in the all cases, will have to allocate a size of 2 * 4 * N-1. Additionally, there is a constant amount of memory allocated "m", used to store variables in the recursive step.

$$S(N) = 1000 + N*7 + N*17 + 2*999*999*2*4*N-1 \Rightarrow O(N)$$

5. References

[1] PowerPoints given in classes
[2] https://www.geeksforgeeks.org/vector-in-cpp-stl/