COMP2008 Coursework: 3-In-A-Row solver

My objective was to build a program that can either solve a puzzle by filling the rest of the board with coloured pebbles or deem it unsolvable. These are the rules that I had to follow:

- There are never 3 pebbles next to each other, horizontally or vertically, all of the same colour.
- Every row and every column has the same number of black and white pebbles.

First I looked at the potential problems of the inputs the program shouldn't allow. The input size of the grid must be more than 0 and be even because there must exist an equal number of different coloured pebbles in each row horizontally and vertically. Also if there are no inputs for any coloured pebble positions, the program will immediately complete the program with default values.

Secondly I developed an algorithm to check the status of the grid to alert the program of some form of event for the puzzle. The first event is triggered if any 3 coloured pebbles that are in line horizontally and vertically in the grid. The second event is triggered if any of the coloured pebbles has more than half the grid length because every row and column must have the same number of black and white pebbles. The last event is triggered if there were no rules broken and there are currently no spaces left in the grid, meaning the puzzle had been solved.

Lastly, I went onto develop the run function for the program which starts off with the status check of the grid. If there was a trigger event, prevent the function from continuing any further. Otherwise, follow on with the 3 different methods that will help solve the puzzle.

Method 1

_ X X = O X X
X X _ = X X O
X _ X = X O X

Method 2

X _ X O X _ = X O X O X O
O _ O _ _ O _O = O X O X X O XO

The program will initially run method 1 and 2 in a sequence. Method 1 detects any patterns found in any row and column, and then immediately completes the pattern. Method 2 checks if one coloured pebble has reached it quantity limit in the row or column and fills in the rest of the blanks tiles in that row or column with the opposite colour. They are run individually because it is easier identify the different algorithm but run together because they are considered safety inputs of the puzzle. The program will keep on iterating on these 2 methods until the program is fully solved.

Method 3

There may be a situation where the puzzle will encounter a run where there are no longer any safe inputs to make. In that case, the program would have to make a logical decision in placing the next coloured pebble because the next position could potentially render the whole puzzle unsolvable, even though there might have been a solution. The safest and simplest method would be to have the program make a random decision and have an undo feature in case that decision was wrong. So the program would have to memorise its trail in order to implement such feature. This means that it doesn't matter what colour is placed or where it is placed, only until a problem comes along forcing it to undo the mistake and try an alternative path. At first I was extremely reluctant in using a linked list due its complexity, but then I realised that as I was using a run function over and over again, I eventually figured out a nice simple solution in altering the program to use recursion.
My solution:

If method 1 and 2 did not alter the grid in any way, save the state of the grid, put a white pebble in the next available blank position and then recall the function to solve the new change. When the program returns back to that point, either the puzzle has been solved and so the program is just trying to unwrap itself from the recursion, or the white coloured pebble was misplaced and so the grid would have to revert back to its previous state and undo anything else in order to place a black pebble in position of where the white pebble was. When the program again returns back to that point, it will just unwrap itself to a previously saved state to solve from there or until there are no more recursion points stored in the stack and in which the program will finally declare the results of the puzzle.

As I was programming, I knew I was dealing with a very computive intensive program so I had to look at different ways I could optimise the program as much as possible. Although I have many loops to analyse the rows and columns for the puzzle, they are all in the order of O(n^2), and I avoided going into O(n^3) altogether to keep the program running in a reasonable time at reasonable inputs. I also made the program prioritise these first two safety methods before considering in using the last method involving saving states as last resort, simply because the safety methods get more of the job done efficiently. All this recursion can get extremely memory intensive with all this state saving of the grid involved, especially as the size of the grid gets larger so I did best to do this state saving where it was needed.