**Sudoku Solver – Summary**

Constructed by Yoav Mor

Background

Sudoku is a visual mathematical puzzle game. Using different algorithms, the user is given a partially-filled board (9x9 cells) and must solve the board such that each 9-cell row, 9-cell column, and 3x3 cubic region contains the numbers 1-9 once and once only. Each board has exactly one unique solution.

There have been many books and online solvers created for the solving of Sudoku puzzles. I’ve created my own solutions, based in Excel, using the methods I know (and some which I researched) and use to solve Sudoku by hand.

Solver algorithm

The solver is not advanced, yet it is meticulous in its coded formulation of “visual logic” normally used by players. As such, it breaks down the methods used by players into loops and functions.

1. The solver finds the available numbers for each blank cell. For each number 1-9, if a number is not contained in the row, column, or cube of the blank cell, it is an available solution for that cell. If there is only one available number in the set of solutions for that cell, we may fill in that cell and set its object property as ‘solved’. This is the most basic method, and it typically yields little result in the beginning of a puzzle, but is used much more later on once the board fills up.
2. The solver searches for ‘singlets’ – instances where a cell may have more than one number in its set of available solutions, yet one of those only appears once in that cell’s row, column, or cube. By the logic of the game, if a number is only available in one spot for one of the aforementioned regions, we may fill it in the specified cell, and call it solved.
3. If methods 1 and 2 yield results, the solver must re-populate the available number set (this is stored in an array for each cell object) based on more numbers available on the board. The solver then re-runs the loop until the board is complete, or no new solutions are found
4. If we still have an incomplete puzzle, we must take a guess – this is most easily achieved in a remaining blank cell that only has 2 available solutions (to reduce our possibility for error). Once this guess is made, we return to step one. Many times, one extra filled-in guessed cell is all that separates us from being stuck with an incomplete board to a complete solution. If we still cannot complete the board, we clear all the cells which we solved as a result of our guess, then we re-introduce the second available solution in the original “guess” cell, and try to solve once more.
5. If we have guessed on both available solutions and not reached a solution, then we do a double guess – 2 cells that have 2 possible solutions each. In effect, we must consider 4 permutations of guesses: [cell1option1,cell2option1], [cell1option1,cell2option2], [cell1option2,cell2option1], [cell1option2,cell2option2]. We try and solve the puzzle this way and see if any path leads to a solution.
6. If we still cannot find a solution, then our solver has run out of options. It can be coded for more guesses, or more advanced mathematical algorithms to better achieve a solution (ie, ‘doublets’). These are good ideas for expansion, yet are beyond the scope of the current solution project.

Results

Based on a sample of 10 puzzles from each difficulty level from [www.websudoku.com](http://www.websudoku.com):

* 100% of all Easy puzzles were solved without using brute force guessing
* 100% of all Medium puzzles were solved without using brute force guessing
* 100% of all Hard puzzles were solved – most with either no guessing or a single guess, yet a small percentage required a double guess
* 50% of all Evil (hardest difficulty) puzzles were solved even after reaching the double guess method.

Concepts used

* Named ranges
* Dynamic formatting
* ActiveX controls
* Functions and subroutines
* Class modules
* Range objects
* Loops
* Recursion
* Arrays
* Nested loops
* Nested logicals