Jeff's Sudoku Solver

Version 2012-12-29

This Sudoku solver is implemented in Python 2.7.3, using two different classes to represent the actual Sudoku grid, the ‘Node’ class and the ‘Sudoku_Matrix’ class.

The Node class represents each of the 81 elements within a Sudoku grid, and keeps track of that nodes current value, possibilities, and coordinates. All cells are identified with coordinate ranges from 0 through 8; the top left cell is (0,0), bottom right is (8,8).

The Sudoku_Matrix class is the heart of the Sudoku Solver. It contains 81 instances of the 'Node' class, along with helper functions to quickly get a list of all values within a Row, Column, and Box, and facilitates reducing the list of possibilities for each Node, given already-solved Nodes within the same Row, Column and Box. Using this framework, you can easily implement different methods of solving a Sudoku-solver.

The ‘Node’ Class

Attributes of the Node class

- **row** : unsigned integer, represents the row this node resides in; indexed from 0 to 8 from Top to Bottom
- **col** : unsigned integer, represents the column this node resides in; indexed from 0 to 8 from Left to Right
- **box** : unsigned integer, represents the box this node resides in; indexed from 0 to 8, from Top-Left to Bottom-Right. This value can be determined if given the nodes Row & Column
- **val** : unsigned integer, represents the answer for this node; if not solved this will be 0.
- **possible** : list of unsigned integers, represents the list of all possible integers this Node can be; when solved, will be an empty (0-length) list
- **cannot** : list of unsigned integers, represents the "opposite" of possible, a list of all values that this node CANNOT be. For a solved Node this will be the full range of integers from 0 through 9. Therefore, possible + cannot = full range of integers from 0 to 9 at any given time, so this attribute is redundant and ONLY used for assertion checks.

Functions of the Node class

- **coord()** : returns a string of this Node’s (x,y) coordinates within the Sudoku puzzle (NOT a 2-length list of integers!)
- **not_possible(val)** : passed an unsigned integer, removes 'val' from the list of possibilities for this Node. Returns ‘True’ if val was in the list of possibilities to indicate that we have done meaningful work by calling this function, otherwise returns False.
- **set(val)** : passed an unsigned integer, solves a given node; this fills in the 'val' attribute, and makes 'possible' a 0-element list. No return value, but will print the node and value so we can see our progress while the program runs.

The ‘Sudoku_Matrix’ Class

Attributes of the Sudoku_Matrix class

- **matrix** : a row-oriented 2D list of Node instances.

Functions of the Sudoku_Matrix class

Utility Functions

- **print_game(string)** : passed an optional string, prints a console-friendly representation of the current Sudoku matrix. The optional string is formatted as a title that will appear above the matrix. No return value.
- **visual_test(Sudoku_Matrix)** : passed a Sudoku_Matrix instance, prints out a console-friendly version of the current Sudoku_Matrix object along with additional diagnostic information, such as the possibilities for every node. This was mostly used for debugging the code.
- **output()** : returns a simple 2D list of integers – this is the expected output format for the Sudoku Solver.
- **sorted_possibilities()** : returns a list of all unsolved nodes in the game, sorted by number of possibilities (node with least number of possibilities at top of the list)
- **Done()** : Returns True if this puzzle has been completely solved, otherwise returns False if there are still unsolved Nodes
- **DoneRow(row)** : passed an unsigned integer (row index), returns True if all nodes in that row are answered, otherwise returns False.
- **DoneCol(col)** : same as DoneRow, but for columns
- **DoneBox(box)** : same as DoneRow, but for boxes

Get Functions

- **getRowVals(row)** : passed an unsigned integer (row index), returns a 9-length list of unsigned integers representing all the nodes in that row
- **getColVals(col)** : same as getRowVals, but for columns
- **getBoxVals(box)** : same as getRowVals, but for boxes. Take a look at the implementation, as getting values for a box is quite different than for a row or column!
- **getRowNodes(row)** : Similar to getRowVals, however instead of returning a list of unsigned integers this returns a 9-length list of Nodes instances, allowing full access to a Node's value, coordinates, possibilities, etc.
- **getColNodes(col)** : same as getRowNodes, but for columns
- **getBoxNodes(box)** : same as getRowNodes, but for boxes

Reduction Functions

- **reduceRow(row)** : passed an unsigned integer (a row index), will iterate through all nodes in that row (via getRowNodes) and remove each answered node's value from the rest of the nodes possibilities
- **reduceCol(col)** : same as reduceRow, but column-oriented
- **reduceBox(box)** : same as reduceRow, but box-oriented

- **reduce_Node(row, col)** : given a node's x-y coordinates, find all values it cannot be based on other nodes in the same row/col/box
- **reduce_Matrix()** : reduces all nodes in a matrix similar to reduce_Node, however will go through the ENTIRE matrix at once and thus does not require any parameters. Additionally, after an unsolved node is reduced, this function will check if there are no possibilities -- if this is the case, we have an over-constrained and therefore impossible game.
- **reduce_Dependancies(nodes)** : This is actually a helper function of solve_game(..) described below, and may be tricky to wrap your head around, and is best explained with an example. Given a 9-length list of Node instances (a full row, column, or box) if *n* nodes have the exact same *n*-length list of possibilities, then no other Nodes in that row/col/box can contain any of those same possibilities.
  Example: if in a row there are two unanswered nodes with the exact same possibilities, say [5, 6], then we know 5 and 6 MUST be in those two nodes. If a third node had [5, 6, 8] as its possibilities we can effectively reduce its possibilities down to only [8].

Solving Functions

- **solve_game(game, loop)** : given a Sudoku_Matrix instance and optional unsigned integer 'loop', this is the heart and soul of this assignment. The loop parameter is used internally in the function to keep track of how many recursion iterations we go through, with the default being 0. There are several helper functions defined within solve_game;
- **node_analysis(node)** : given a Node instance, if the Node is unsolved and there are 0 possibilities, then we have encountered an impossible game and function returns -1, but if there is only 1 possibility we have found the solution for that Node and function returns True; returns False otherwise.
- **solve_Nodes(nodes, guess)** : given a 9-length list of Node instances (complete row, box or column) and an unsigned integer representing one of the possible answers (1 through 9) checks if there is only a single Node that has the 'guess' value in its list of possibilities (if it hasn't already been solved for!)