Project Assignment #3: Sudoku Solver

In this project your goal is to design a solver for the Sudoku puzzle. Your goal is to fill out all the cells in a row, column, or a 3×3 subblock of the 9×9 grid with digits from 1 to 9. The algorithm you are going to use is the backtracking algorithm, which employs a depth-first search algorithm. Backtracking algorithm is a constraint satisfaction problem solver. In this puzzle, the constraints are row, column and subblock constraints.

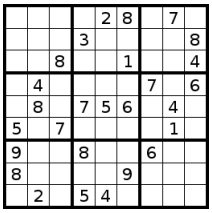

Figure 1: Sudoku puzzle example.

1 Preparation To-Dos

1. Make sure you have Python 2.7 installed
2. Download the source code (sudoku.tar.gz) from LMS, extract it to your local disk

2 Pre-Implementation Questions

(5 points)

- What is the number of constraints for each number (cell) in the grid, if you were to formulate this as a CSP, similar to the car-assembly problem?

3 Backtracking Algorithm

(95 points)
Backtracking algorithm is basically a depth-first search solver, which pops the most recently generated node from a stack. This is implemented in `DFSSolver.py`. The stack variable is named *frontier*. After a node is popped from the stack and if it is not

the goal node, new nodes are generated using the *getBranches()* function. This function returns a list of nodes, which are pushed to the stack. This operation is iterated until a solution is reached.
Your goal in this project is to implement the *getBranches()* function so that all the nodes returned in the list are ordered according to the most constrained variable criterion. The node list returned is a list of game grids that is created by assigning all possible values to a chosen cell in the gaming grid. You are supplied with a naive (but goood) implementation of this function in *getRandomBranches()*. In this functin first a cell in the grid is randomly chosen, and all possible values are assigned to the chosen cell, creating as many gaming grids as the number of possible digits that can be assigned to that cell. Each grid correspondes to a new generated branching node for the DFS solver. When you run your code, it should produce the solutions for an example set of games using *getRandomBranches()*, and print performance info such as the average number of visited nodes by the DFS solver. Implement the *getBranches()* function so that it returns a list of nodes which are generated by assigning values to the most constrained variable (cell).

4 Bonus

(20 points)
Design a heuristic criterion for the the *getBranches()* function so that DFS solver performes better, that is, finds a solution on a smaller number of total visited nodes. Use the *hard.txt* for the games.

Good luck!