Advanced Constraint Satisfaction on Sudoku Version: **BLUE**

Eckel, TJHSST AI1, Fall 2021

Background & Explanation:

We've been looking at constraint satisfaction problems. A famous example is Sudoku. This is big; plan carefully!

Important note: for most work in this class, the best strategy is to go sequentially through the difficulty levels – do BLUE, then do RED, then do BLACK. This assignment is different. There is a BLUE version that is worth two BLUE credits. There is a **different** RED version, which is worth two RED credits and also automatically gives credit for the two BLUE assignments **without separate BLUE submissions**. So: you should only be working from this document if you think you're probably going to get BLUE credit and stop there. If you plan to get RED or BLACK credit, you should use the RED version document.

You **can change your mind later!** But: you'll have to go back and rewrite some code to do so. So, you might want to look at the front page of both versions to compare them and make your choice.

I estimate that the RED version of this assignment is approximately **three times as much work** as the BLUE version. Either option will be successful preparation for Modeling Challenge III and the Unit 2 Extensions.

Puzzle Sizes: BLUE

BLUE version: your assignment will be to solve Sudoku puzzles of **standard size only**. You can hardcode the characteristics of the board – specifically, that boards are 9x9, and which indices represent each row, column, and block.

Algorithm: BLUE

On the N-Queens lab, we discussed simple backtracking and incremental repair. For Sudoku, we will need more sophisticated techniques.

BLUE version: You'll learn one advanced technique, **forward looking**, which keeps track of all the possible values that each variable can hold and updates them, returning a failure if any of them becomes empty.

Test Cases

Several test case files are provided for you on the website and will be referenced throughout the assignment. These test cases were generated in order to make the experience of working on this assignment smoother by several students in AI in 2020, specifically Om Duggineni (TJHSST '23), and Anika Karpurapu, Mikhail Mints, Akash Pamal, and Victoria Spencer (all TJHSST '22).

BLUE Part 1: Simple Backtracking on Sudoku

1) We'll need to quickly access each constraint set (ie, each row, column, and block). For instance, the constraint set of the first **row** is the set of indices **{0, 1, 2, 3, 4, 5, 6, 7, 8}**. The constraint set representing the second **column** would be **{1, 10, 19, 28, 37, 46, 55, 64, 73}** (each index going down starting from index 1). The third square sub-block (top right) would be **{6, 7, 8, 15, 16, 17, 24, 25, 26}**. (Can you verify that? Does that make sense?) You'll need a list of all 27 constraint sets. You can use loops to generate these or write them out by hand.
2) Hardcode a global list or dictionary that associates each square with the squares it constrains / is constrained by. Achieving efficient code will be impossible without this! You can do this by looping over the previously written sets in part 1, or by writing it all out by hand. For example, index 0 needs to associated with the set **{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 18, 19, 20, 27, 36, 45, 54, 63, 72}** because those are all the indices that can't hold the same value as index 0 holds. (Can you verify that? Does that make sense?)
3) Write code that opens a file of 9x9 Sudoku puzzles and reads them each in, one by one.
4) Write a function that displays a puzzle state as a board as we would write it. (Anything you can do here to make the formatting easier to read is encouraged.)
5) Write a function that takes a board state and displays how many instances there are of each symbol in `symbol_set` in that state. (We can use this on solved puzzles as a crude way to make sure our code is producing plausible solutions. It won't check for all possible errors, but it will provide a gut check.)
6) Finally, write a simple backtracking algorithm much like what we saw with N Queens. The next available variable is simply the first period in the string; the get sorted values method will use the dictionary / list of neighbors created in step 3 to find out which values are possible and return them in order. Begin testing your code on given text files. You should be able to handle the file `"puzzles_1_standard_easy.txt"` in well under a minute.

Specification for BLUE Credit for Sudoku Part 1

Submit a single Python script to the link on the course website.

This assignment is **complete** if:

- The "Name" field on the Dropbox submission form contains your **class period**, then your **last name**, then your **first name**, in that order.
- Your code does all of the following:
    - Accept one **command line argument** – the name of a file.
    - Read a puzzle off of each line of the file. These puzzles will be 9x9 puzzles of trivial difficulty.
    - Solve each puzzle, printing the solution *as a single line with no other information on that line*. If you print out beautiful grids, you will **not** meet the specification. If you print any extra stuff, you will **not** meet the specification. In particular, I do **not** want you to print timing information this time.
- Total runtime is less than 1 minute. (If you can do the given file in less than a minute, you should be fine.)

BLUE Part 2: Add Forward Looking

Try `"puzzles_3_standard_medium.txt"` in your previous code and it should be clear: we need more sophisticated techniques. It's just too inefficient to solve many sudoku puzzles in a reasonable amount of time. So: let's add forward looking!

Forward looking, in this case, means we will keep track of all the possibilities for any given index. We will remove possibilities when they are eliminated by other choices. For instance, if I place a "1" in a certain row, all the rest of the spaces in that row can't be "1". If any space has only one option, it is solved. If any space has zero options, we've made a mistake and need to backtrack.

You'll need some kind of data structure to store the possibilities at each location. The simplest is a dictionary or list of *strings*, so that we don't have to deep copy. (Deep copy is **slow** and should basically **never** be used.) If you're willing to write custom comprehensions to copy, you could have a dictionary of sets or something like that. This can be in addition to or in place of our previous board representation.

You'll also want a separate function to do the forward looking. Don't just add a bunch of code to your backtracking function directly. There are a *lot* of ways to do this! You can find your own if you like. Here is one method that is straightforward, if a little bit inefficient:

1. Make a list of all indices that have one possible solution (or, alternately, are solved).
2. For each index in this list, loop over all other indices in that index's set of neighbors, and remove the value at the solved index from each one. If any of these becomes solved, add them to the list of solved indices.
3. If any index becomes *empty*, then a bad choice has been made and the function needs to immediately return something that clearly indicates failure (like "None").
4. Continue until the list is empty.

Storing the board state as a list of strings also provides us another benefit: we can now select the most constrained square when we're choosing which variable to attempt next! Just look for the index with the **smallest** set of possible answers with length **greater** than 1.

In summary, your backtracking function with forward looking should now look like this:

```
csp_backtracking_with_forward_looking(board):
    if goal_test(board): return board
    var = get_most_constrained_var(board)
    for val in get_sorted_values(board, var):
        new_board = assign(board, var, val)
        checked_board = forward_looking(new_board)
        if checked_board is not None:
            result = csp_backtracking_with_forward_looking(checked_board)
            if result is not None:
                return result
    return None
```

As a final note, you should also call the forward looking function in a separate call once **before** you start the recursive backtracking algorithm. Often, forward looking can solve the whole puzzle without even needing to try anything.

This should be enough for you to solve everything in `"puzzles_3_standard_medium.txt"` in under a minute, and you might even find that `"puzzles_5_standard_hard.txt"` solves in a pretty reasonable amount of time. This is a solid algorithm to solve standard sudoku puzzles quickly!

Specification for BLUE Credit for Sudoku Part 2

Submit a single Python script to the link on the course website.

This assignment is **complete** if:

- The "Name" field on the Dropbox submission form contains your **class period**, then your **last name**, then your **first name**, in that order.
- Your code does all of the following:
    - Accept one **command line argument** – the name of a file.
    - Read a puzzle off of each line of the file. These puzzles will 9x9 puzzles of non-trivial difficulty.
    - Solve each puzzle, printing the solution *as a single line with no other information on that line*. If you print out beautiful grids, you will **not** meet the specification. If you print any extra stuff, you will **not** meet the specification. In particular, I do **not** want you to print timing information this time.
- Total runtime is less than 1 minute. (If you can do `"puzzles_3_standard_medium.txt"` in less than a minute, you're in good shape here.)

Consider Next Steps

If this came together way faster than you thought, it might be worth checking out the RED version and seeing if your code can be adapted! Otherwise, continue on to Modeling Challenge III.