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

Eckel, TJHSST AI1, Fall 2023

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. Only the RED option allows access to this unit's BLACK assignments.

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. We'll call these "neighbors", even though they aren't necessarily directly next to each other. 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's neighbor set would be the set of indices **{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:

- You follow the instructions on the submission form to format your submission properly.
- 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 is a standard technique when solving a constraint satisfaction problem. The idea is this: as your code makes each choice, it should calculate the consequences of that choice *now* (looking ahead) and see if that choice causes any problems for other variables *later*, before the backtracking algorithm actually reaches those variables. Specifically, in the case of Sudoku, we will modify our code to keep track of all the possible values for every index on the board at all times. Instead of just storing "." for all unsolved spaces, we'll store all the possible values that could still plausibly go there. As our algorithm progresses, every time we choose a certain value to place in a certain index we will remove that value from every neighbor index. For instance, if I place a "1" in a certain cell, all the rest of the spaces in that cell's row, column, and block all can't be "1".

Why does this count as forward looking? Well, in the simple backtracking version you coded earlier, it was possible to come across a "." that had no available choices – every possible option was ruled out by a previously placed character somewhere else in the puzzle. This meant we needed to backtrack. But if we keep track of all the possibilities everywhere as we go, then we can determine if a choice that we make *now* narrows down a *future* space to zero options and backtrack *now*, instead of waiting until our algorithm gets to that space later.

There's another benefit here: we can chain multiple forward looking passes together. Specifically, during the process of forward looking after making one choice, if we find that we've narrowed down a different cell's possibilities to only one option, then that index becomes solved as well and we can then forward look from that index in turn. Especially towards the end of solving a puzzle, we can often make a single choice that ends up placing several digits at once without needing further recursion.

Making this work requires a major change in how you store the board, though! One big string won't work anymore.

Specifically, you'll need some kind of data structure to keep track of all the possibilities at each location as the algorithm progresses. The board state is no longer just a string with some cells solved and some cells blank, it's a full accounting of every possible value of every cell on the board. The simplest way to store this is a dictionary or list of *multiple strings*, one per cell, 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, but remember that we discussed this earlier in the year; mutable data structures containing other mutable data structures are hard to work with. Additionally, if you find it useful, you can store *both* this data structure *and* a board string like before, but though many students each year initially find this option appealing, my strong recommendation is against it. Tracking two different data structures means twice as many ways to make mistakes, and this program is complex enough already! Just use a single data structure to represent the current board state, a list or dictionary where each index of the sudoku puzzle gets its own string of possibilities. If that string has length 1, the cell is solved. If it has length greater than 1, then it is unsolved with multiple currently plausible options. If any string has length 0, backtrack.

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 you've processed every solved index – the ones you started with & the ones you found along the way.

Important Note

A crucial comment here about copying. When we just used a string, we didn't have to worry about copying the board; strings are immutable. But now we're using a list or a dictionary, and that means that if we pass our board state into a function and modify it inside that function then the changes have also been made outside that function, as we've created a situation where there are multiple pointers to the same mutable data structure. To avoid this causing problems, we'll need to manually copy our board state when it's about to change.

Let's be specific - when is the right moment in this algorithm to copy the board dict/list? **Just before you assign a particular value to a particular variable.** Think about it this way. The current recursive call has received a board, and **that board should not change**, because any choice you make at this point might be wrong and if so the current recursive call will need to return to the original board it was passed, make a different choice, and continue. Note carefully where this appears in the pseudocode below!

As we think about the modified backtracking algorithm with forward looking incorporated, another benefit of storing the board state as a list of strings reveals itself: 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. This is a good idea because a sudoku puzzle only has one solution. If a cell right now looks like it might have 5 possible values, then we have a 1 in 5 chance of picking the right one. If a cell looks like it might have 2 possible values, then we have a 1 in 2 chance of picking the right one instead! Focusing on the most constrained index at each decision point can reduce the amount of backtracking by a considerable margin.

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)      # Note the change here!
    for val in get_sorted_values(board, var):
        new_board = board.copy()               # VERY IMPORTANT!
        assign(new_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:

- You follow the instructions on the submission form to format your submission properly.
- 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, you may wish to start the next unit early.