Advanced Constraint Satisfaction on Sudoku Version: **RED/BLACK**

Eckel, TJHSST AI1, Fall 2022

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: if you plan to get RED or BLACK credit, you should use this version of the assignment and skip the BLUE version entirely.

You **can change your mind later!** If it turns out this is too hard, aiming for RED at the beginning and then switching later won't give you any disadvantage in trying to accomplish the BLUE specification. It just means you will have spent some time on features the BLUE assignment won't need.

I estimate that the RED version of this assignment is approximately **three times as much work** as the BLUE version. If you finish the RED version, you also have access to this unit's BLACK assignments. You can either do a BLACK extension of Sudoku or try your hand at a related puzzle – KenKen.

Puzzle Sizes: RED/BLACK

RED or BLACK version: your assignment will be to solve Sudoku puzzles of **any size**. Traditional Sudoku puzzles are 9x9, but other sizes are possible, and you'll get test cases of all different sizes from 4x4 to 25x25. You will need code that can adapt based on the size of the puzzle it reads in, dynamically deciding which indices represent each row, column, and block. The easiest puzzles are N x N where N is a perfect square, leading to square sub-blocks (ie, a 16x16 puzzle has 4x4 sub-blocks). Really, though, any size N is workable as long as N is not prime. If N is not a perfect square, then the sub-blocks on the board aren't square either. In that case, sub-block width is the smallest factor of N *greater* than its square root, and the sub-block height is the greatest factor of N *less* than its square root. For example, a 12x12 Sudoku would have sub-blocks that are 4 spaces wide and 3 high. (See the course website for a link to a website to generate these.)

Obviously, if N > 9, we need more symbols than 1-9. We will use 1-9 and then letters starting at A and going up as high as we need. So, for example, a 16x16 puzzle would use 1-9 and A-G. We will also use periods to represent blank spaces, to aid in readability. Please note, this means we do *not* use *integers* 1-9. These are all *characters*.

My experience is that it **seems** easier to do 9x9 first, and adapt your code to any size once that's working, but this is deceptive. To aim for RED or BLACK credit, you should **start** by coding the general, any-size solution **from the beginning**.

Algorithm: RED/BLACK

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

RED version: You'll use two advanced techniques - **forward looking**, which keeps track of all the possible values that each variable can hold and updates them and returns a failure if any of them becomes empty, and **constraint propagation**, where you examine each constraint in turn and update the set of values when those constraints imply certain necessities. More detail is below.

BLACK version: You'll add in additional Sudoku logic to improve on the two above techniques. In order to facilitate this, you'll want to **begin** by coding your algorithm in a more sophisticated way (instead of modifying a RED solution).

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).

RED Part 1: Simple Backtracking on Sudoku

1) Write code that opens a file of Sudoku puzzles and reads them each in, one by one.
2) Refer to the "Puzzle Sizes: RED/BLACK" discussion above. We'll need to do some calculations based on the size of each puzzle we read. Specifically, for each puzzle, find the length of the input string, and use that to set values of global variables representing:
    a. `N`
    b. `subblock_height`
    c. `subblock_width`
    d. `symbol_set`
3) 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.)
4) Now, we want to set up 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! This comes in a couple of steps:
    a. Go through each separate constraint set (row, column, block) and find the indices of each square in that set. Store each constraint set. There should be N row constraint sets, N column constraint sets, and N sub-block constraint sets, so this could for example be a list of 3N different sets of integer indices. (Note: referring to this list of sets might be helpful later; you can consider making this a global variable as well, but it won't be required.)
    b. Go through each square, find all the constraint sets it belongs to, and add each other square in each of those constraint sets to a set of neighbors. Store all of these in a global dictionary or list, so we can retrieve a set of all neighbors of any given square while solving without having to regenerate it.

    Remember this code must be *general* – no hardcoding of board sizes – and refers only to *indices*. You should generate all of this information just referring to the board's *size.* None of this has anything to do with the values already placed in the puzzle yet.
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.
7) Test your code on `"puzzles_1_standard_easy.txt"` and `"puzzles_2_variety_easy.txt"`. You should get each file to solve in well under a minute.

Specification for RED Credit for Sudoku Part 1 (Automatically also gives BLUE credit)

Submit a single Python script to the RED link on the course website (you do NOT need to also submit to the BLUE link).

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 of trivial difficulty, but a ***variety of sizes*** as small as N = 4 and as large as N = 16.
    - 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 2 minutes. (If you can do both files in step 7 above in well under a minute, you should be good here.)

RED Part 2: Add Forward Looking

NOTE: *if you're aiming for BLACK credit, you might want to jump ahead to "Notes for Efficiency for BLACK Implementation" at the end of this document and plan ahead before you start this section.*

Try `"puzzles_3_standard_medium.txt"` or `"puzzles_4_variety_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, it's time to 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 the list is empty.

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 the two medium-difficulty puzzle files in well under a minute. If you have an especially good implementation, it might even get you a reasonable time on `"puzzles_5_standard_hard.txt"`! It turns out that forward looking alone is pretty great for solving standard sudoku puzzles. But `"puzzles_6_variety_hard.txt"` is going to cause you huge problems still; we need more.

RED Part 2, Continued: Add Constraint Propagation

Forward looking deals with the fact that each number *can only* appear once in each row. It is also true that each number *must* appear once in each row. These aren't the same thing. We can't check this from the perspective of one of the spaces, eliminating values from its neighbors; now, we have to look at each *constraint* independently to see if we can make progress. Specifically, we need another new function, this one for constraint propagation. Once again, there are a *lot* of ways to do this! You can find your own if you like. Here is a straightforward one that's a bit inefficient:

1) Look at each constraint set one by one.
2) Within each constraint set, loop over each possible value. Determine if only one space in the constraint set contains that value. If so, set that space to that value. If *no* space in a constraint set contains a value, return failure.
3) When finished, if any changes have been made, call the forward looking function again, since new spaces have been cleared. If that function returns a failure, return failure. Otherwise, return success.

I leave it up to you, the exact sequence of when forward looking is called and when constraint propagation is called and how many times (loop until changes stop happening, maybe?) Once you find a solution that works for you, implementing this should be enough for you to get both of the hard puzzle files, in their entirety, in well under 30 seconds; you should be amazed at how fast they complete.

Specification for RED Credit for Sudoku Part 2 (Automatically also gives BLUE credit)

Submit a single Python script to the RED link on the course website (you do NOT need to also submit to the BLUE link).

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 of non-trivial difficulty, but a variety of sizes as small as N = 4 and as large as N = 16.
    - 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 30 seconds. (If you can do both hard files in less than 30 seconds, you should be good here.)

BLACK Credit: Advanced Sudoku Logic

There are *so many ways* for you to keep going from here. For one thing, you can optimize the prior tasks like crazy. Some advice on this is below in the next section. As an alternative, there is another BLACK option on the course website you could try instead – KenKen. Take a look at both before you decide which to aim for.

Just as important, you can add as much additional Sudoku logic as you want:

- If two squares in the same constraint set have the *exact same pair of possible values*, then no other square in that constraint set could have either of those values. Find situations where this occurs, then remove those values from the other squares.
- If there are two values that only occur in two squares in a certain constraint set (and nowhere else), then *only* those values will occur in those squares. You may delete any other values that are theoretically available in those two squares.
- The last two both apply to situations with *three* squares and *three* values. Or four and four, five and five… Can you find a general implementation of this type of logic?
- Ok, follow me through this one. If you look at a block and a row that overlap on a 9x9 board, they overlap on three squares. If you loop over the block and find that a particular value *only appears in the overlapping three squares*, then on the corresponding row, you can *delete* that value from the *non*-overlapping section as well. Same in reverse – if you look at the row, and a particular value is only in the overlapping three squares, then it can be removed from the non-overlapping section of the block. And the same also applies to each overlapping pair of block and column. To help you visualize, I've used a 9x9 board; please note that this same logic applies on larger board sizes as well. This is *remarkably helpful* if you get a reasonably efficient implementation!
- Anything else you can find or think of! There's lots more where that came from.

You want to test your advanced logic on `"puzzles_7_bigger.txt"`. My current best implementation solves the whole file in about 7 minutes. I don't expect you to get that good; if you can knock it out in less than an hour, you're probably fine here.

Specification for BLACK Credit for Sudoku: Advanced Logic

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 of hard difficulty and they will be big, as large as potentially 25x25.
    - 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 five minutes. (If you can do the 7th file in less than an hour, you're good.)

Notes for Efficiency for BLACK Implementation

In the past, students who have succeeded in making super awesome sudoku implementations have thought a lot about efficiency. The crucial principle here is **avoiding redundant work**, and it's harder than you think in this assignment.

Let me give you an example. Consider the constraint propagation algorithm. For each row, column, and block; for each value in that block; we look to see if that value appears only once in an unsolved square. If so, we solve it. Think how much wasted work is happening here! If we make a single change to a Sudoku board, we're only altering the possibilities for *one digit* in *three constraint sets*. Then calling constraint propagation loops over *nine digits* in *twenty-seven constraint sets*, fully 81x as much work as necessary. This adds up! So: how do you prevent this?

(You can find your own answers to this question if you like! Feel free to stop here and go ponder.)

The best student submission I've ever gotten (which was faster than my code in my first year teaching the class) changed its approach entirely in a way that was very clever, and which I stole to make my best code so far.

Here's the key: instead of thinking of this as **placing** one value at a time, think of it as **removing** one value at a time. When you place a value in a certain square, visualize that as **removing** the **other** possibilities one by one until you're only left with the value you intend to place. Each time you remove a value, do Sudoku logic **only on the locations that might be affected by removing this value**. If there are any problems, backtrack.

This is dual recursion! We have a PLACE function, which calls REMOVE on each unwanted value; then, in REMOVE, we check our sudoku logic and if it turns out we have to place a certain value somewhere, that calls PLACE… which in turn calls REMOVE… which in turn calls PLACE… and either the recursion ends because all the consequences of our choice have been explored and they all work, or a failure is discovered somewhere and "None" is returned back down the chain very quickly, wasting no additional work.

How does this solve the constraint propagation problem I explained above? Well, let's say a cell currently has the possibilities "345" and we want to place a "3". Let's begin by removing a "4". From the perspective of constraint propagation, it is now possible that in either the row, column, or block, there USED TO BE two cells with a "4", and NOW there is only one cell with a "4". We are checking only the cells that might be affected by this specific choice! If we find that there's only a single "4" somewhere, we can then call place on "4" on that cell, and continue to explore the consequences of those actions. If that all comes back successfully, then we can remove "5" and keep going. If that works, we've successfully placed the "3". If not, we return None and backtrack.

This requires a pretty fundamental rethinking of how the entire algorithm works, but in my experience produced speed gains of something like a factor of 3 to 8, depending on how large the puzzle was, without any changes in logic. A massive improvement!