Advanced Constraint Satisfaction on Sudoku

Eckel, TJHSST AI1, Fall 2020

Background & Explanation:

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

We will want code that models a Sudoku puzzle of any size in a way that allows quick access to important information. Traditional Sudoku puzzles are 9x9, but can vary. The easiest options are if they 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.

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*. For your use in this assignment, there are several txt files of Sudoku puzzles following these conventions on the course website.

On the N-Queens lab, we discussed simple backtracking and incremental repair. For Sudoku, we will need more sophisticated techniques – **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, and **constraint propagation**, where we examine each constraint in turn and update the set of values when those constraints imply certain necessities. More detail is below.

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) 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. 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.
   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 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 don't need to read information from the puzzle to do this, just know its size.
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 puzzles 0 to 50 in the first text file in well under a minute.

Part 1, Continued: Build Additional Test Cases

For this assignment, I'm not giving you sufficient test cases to make sure your code covers every situation. This is intentional; an important skill when building algorithms is designing your own tests. You have a lot of files of 3x3 puzzles and a few 12x12 and 16x16 but I'll be testing you on more cases than just these. It's your job to make sure your code works before submitting it to me.

Simon Tatham's Portable Puzzle Collection contains "Solo", which is precisely Sudoku. In the custom settings you can specify any size and difficulty you like. (Keep in mind that, according to our convention, the subblocks are either square or *wider than they are tall*. A 10x10 Sudoku would have blocks 5 wide and 2 high. His puzzle lets you set them in any direction you like; be sure to follow our convention.) Build a variety of tests. For part 1, I will only test trivial puzzles, but they could be of any size. If the puzzles generated by the game are too hard, generate solved ones and delete spaces yourself.

(As a quick note, when you build your files, put some smaller puzzles *after* bigger ones. When I made a test file with smaller puzzles following larger ones last year, it messed up a surprising number of people's code.)

Specification for 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 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 puzzles 0 to 50 in the first text file in less than a minute you should be fine here.)

For **resubmission**:

- Complete the specification correctly.

Part 2: Add Forward Looking

Now, let's add more sophisticated techniques.

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 the first puzzle file in well under two minutes. Try the harder puzzles, though, and it's clear that this isn't sufficient.

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 all four of the first four puzzle files, in their entirety, in well under thirty seconds. If not, tweak your implementation until it's done.

Specification for 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 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 files 1 to 4 in less than 30 seconds, you're in good shape here.)

For **resubmission**:

- Complete the specification correctly.

Specification for Outstanding Work: Sudoku Optimization

There are *so many ways* for you to keep going from here. For one thing, you can optimize the required tasks like crazy. Choose different data structures, find ways to avoid repeated effort (ie, only check neighbors of *newly solved squares* in your forward looking function, or maybe differentiate between placing values and removing them), etc etc. You can also add as much 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!

This extension **receives outstanding work credit** if your code can solve 5 complex N=16 puzzles in under 5 minutes. The sample file of the first five puzzles in the fifth puzzle file are a good test. (The student record last year was just under a minute and a half!) To attempt this extension, submit your Python script to the link on the course website, following the usual criteria.