AI Sudoku Solver

Nicholas Stafford
Abhishek Pandit
Ean Dodge

Motivation

From the very beginning of our childhood, we have been looking at Sudoku puzzles. With pens and paper, on magazines and mobile apps. The 9x9 grid numbers cannot be repeated in a row, column or 3x3 subgrid. Using our brain power, we used to solve the sudoku taking a lot of time.

While studying Constraint Satisfaction chapter in our AI class we were introduced to the Sudoku problem. Through backtracking and constraint heuristics we can quickly solve sudoku puzzles, coloring map, and similar puzzles bound by rigid rules. After talking with group members and evaluating different projects, we thought Sudoku would be a good challenge. We also wanted to test various optimization techniques once finished. It would also be a good opportunity to learn python and its related environments and libraries. In addition to solving 9x9 grid size puzzle, we could also scale it up for larger grid sizes. Also, the project learnings can be used for broader implications such as scheduling and resource allocation.

Project Description

This project uses the constraint satisfaction problem (CSP) to solve any given Sudoku puzzle. Sudoku is a popular puzzle game that can be framed as a CSP where the variables (cells) have constraints on their values (numbers 1-9). The python programs we implemented solve this in intelligent ways, as opposed to brute force.

A Sudoku puzzle consists of a 9x9 grid where some cells are pre-filled with numbers, and others are empty. The goal is to fill the empty cells with numbers from 1 to 9 such that:

- Each number appears only once in each row,
- Each number appears only once in each column,
- Each number appears only once in each 3x3 subgrid.

We implemented three approaches to solve the puzzle

- brute_force.py - uses a naive brute force algorithm.
- sudoku_solver.py - uses CSP to solve algorithm.
- lookup_table.py - uses CSP and a lookup table for increased efficiency

Problems Faced

The problems faced in making the sudoku solver were mostly at the beginning. In search of a library to print the sudoku puzzles in a neat manner, we found the python-sudoku library on python's website. We tried to use that library to do our printing features, but we found out that it was only good for generating a sudoku puzzle and leaving the rest up to you to print it out nicely. We then found a library called sudoku-py. This library was incredibly good for printing the sudoku puzzle, but it was unable to generate the sudoku puzzle in the way we needed, so instead, we were able to use it to plug in a multi-dimensional list to then it converted it to a good looking sudoku puzzle. Lasty, in a search to find the right library for what we needed, we found what we should have found first. It seemed to be the best sudoku library yet, but we were unable to figure out the installation code for it. Since we are using python, we need to use pip to install libraries like this. We tried to put in pysudoku, sudokypy, and sudoku. All of these were not working, so we decided to instead try and use the libraries that we had to generate and print. It was not working great as the code started to get messy. We went for another search for what the library could be called. In attempt, we tried “pip install py-sudoku". As we were waiting for the error message, we instead got the message telling us that it found it and going to install it. With this new library, it generated a sudoku puzzle at a difficulty set by us. It had a

method that could show the puzzle generated but was unable to show a puzzle that was made with a multi-dimensional list.

The sudoku puzzle that was created needed to be able to be indexed. This was a problem as we kept getting errors every time, we would try to simply subscript the sudoku object. The object was not able to be indexed as it does not have an operator overload for that. Instead, we had to do some research on the new library for sudoku. We found out that it has an attribute called a board that allows us to index the puzzle. Once we were able to do this, we made our own list of lists to make our 2D board. We would use this later for our algorithm to solve the board.

We started out with an algorithm that only used backtracking. To backtrack, it would put in a valid number into the puzzle and move on. If it figured out in the future that it did not work with that number, it would recursively go back to change that number to a new number. It would then run through the numbers again to make sure each is valid. The problem with this algorithm is that it can be slow. With the worst-case running time, it could run at O(9^n*n). It must check if the number is valid for every number 1-9, in every row, column, and 3x3 grid. Once it has done this, it will move on to the next block, in hopes it will pick a good number in the block before it. If it did not, it would go back to the block before it and see if there are any other valid numbers it could choose. It will keep doing this until the puzzle is complete. Instead, we wanted it to run more efficiently. We added heuristics that will look for blocks of the board that have different degrees and will give blocks a value based on their degree of constraint. With the heuristics set into the program, it could run at O(9^n) instead, which is much faster than the backtracking algorithm.

CSP Background

Our project is based on the constraint satisfaction problem. This problem describes variables having constraints that make them only able to be one value. Like the in-class map coloring problem, every state on the map needs a color and cannot have the same color as a state bordering it. This puts constraints on every state around it. If a state has options, it may pick one, but it could be wrong in the future. This is where it could use backtracking to solve the problem. It will choose a value for the variable that it thinks is best, and it will choose the other colors around it accordingly. Once it gets to a state that has no more options for values, it would have to backtrack to solve the problem of the variable not having any values. This can be very inefficient as a huge problem could be backtracking the entire time and spending a lot of time during this process. To make this faster, the look ahead table was created for the algorithm to look ahead to see what decisions it could make. If it fails in the look-ahead table, it can backtrack in the table to keep looking or decide what not to do when it comes to that in the problem. While a look ahead problem can increase execution time, it comes with a tradeoff of using more memory.

There are also heuristics that can be used for the constraints too. It can look through the options to see which ones are most constrained, most constraining, and least constraining. All these heuristics can be used on the same problem as variables will end up running out of values to choose and it could be starved of values and make the problem fail. If most constrained is used, it could help the variables that are in desperate need of a value as they are running out. If a variable is constraining a lot of other variables, it would be smart to give this one a value first as all the other variables can work around this variable in the future. Lastly, the last heuristic is least constraining. This one explains that if a variable is not constraining a lot, to color that one as it will barely affect anything.

Our sudoku puzzle works with the constraint problem just like the map coloring problem. The sudoku puzzle has constraints on it all over the place. As explained in the project description, the sudoku puzzle's constraints are that a number can be duplicated in the 3x3 block, the row, or in a column. To solve this, we use multiple heuristics. We use the most constrained, most constraining, and least constraining.

Most constrained works with the cell that has the highest degree. If a cell has a high degree, then it should be looked at first. A high degree means that it has a high number of constraints on it. If you give a value to every variable around the constrained variable, then it could be hard for the constrained variable to have values left to be given. For example, our map coloring problem, if there is a state with multiple states bordering it, the number of states bordering it is the degree of that state. If we give all the bordering states a color,

we will run out of colors for the most constrained state. Instead, we want to color the state that is most constrained first because that will give more leverage to the states around it. States around the most constrained will have a less constrained degree since we worry about the highest degrees first.

Most constraining works on the cell that is constraining the greatest number of variables. If it is constraining variables, that means that it is limiting a variable to what values it can choose. If it is limiting the greatest number of variables, we will give that a value first. We do this so that the other variables that are being constrained have a chance to get a value as if we give a value to the most constraining variable first, the limit on all the variables is less by 1.

Least constraining works on the cell that is constraining the least number of variables. There are two reasons that this variable would need a value. The first being that it is not hurting much by giving it a color. Giving values that works are an accomplishment every time, so making sure that a variable gets a value is important. The second reason is that it does not run out of value. If it is the least constraining, that means that it is not constrained by many variables either, so it would be worked on last. If it is worked on it last, it would be risky as it could run out of variables, forcing it to backtrack. Instead of having to backtrack at the end, you can give the least constraining a value before it runs out, and if you need to fix in the middle vs the end, you are going to save a lot of time.

The cool thing about these heuristics is that they all work together to make the program run efficiently. The order they run most efficiently is running the most constrained variable first. This will find the variables that need a value first due to many variables that are limiting it. Then you want to run most constraining as it will look for variables limiting many other variables. Lastly, least constraining variable to make sure that no variable is getting starved at the end.

Implementation

The program leverages the modular approach using python and the sudoku-py library. The program has been split into five different files. The user is prompted to either input a custom 9x9 grid, or else, one is automatically generated for them.

sudoku_solver.py

Main function of the application. The board is sent to the solve_heuristics function which sequentially applies heuristics. Most constrained (least number of available numbers) is applied first, which provides a list of board positions. The list is sent to the most_constraining_variable to find the cells that impose the most constraints on their

neighbors. The cell position with the most constraining value is returned. In the case of a tie, a random cell is chosen from among them. The least_constraining_values function next gets the possible values of a cell which are ordered by their least constraining effect.

To avoid backtracking, all these heuristics check potential state transitions with the is_valid function. This determines if the rows and columns and the 3x3 subgrid numbers are valid or not. This prevents illegal moves and improves performance.

cli.py

Contains several cli utility functions used in other files. The get_puzzle function prompts the user for a difficulty level which determines how "filled in" the initial puzzle is, _get_custom_puzzle initializes a grid of numbers, where 0 represents the empty space to be solved. It also contains format_board_ascii function that takes in the list of lists and prints the board in an eye pleasing way.

lookup_table.py

An alternative implementation to sudoku_solver.py, but it uses several optimizations for better performance. The lookup table is used to reference what future state changes are possible and does not constrain neighboring nodes. The key improvement is that instead of dynamically searching the grid to see if a future state is possible, a lookup table is checked proactively containing all states for every cell. Every time a cell is added, the constraints are checked, and potential futures states are pruned from the table.

While the lookup table in this implementation does share a standard lookup table's functionality, the data structure used is tailored to this applications' needs. Originally, a grid was used to hold the program's state, and a similar lookup table was used to show future states. Each cell in the table had a list of integers, which correspond to cell values. This duplicate data structure was inelegant, confusing, and not very dry. Thus, the grid and lookup table were merged into a class called state. The lookup table shows potential states, and two sets of (int, int) tuples are used to track which cells have been set already.

brute_force.py

Uses a naive brute force algorithm. for solving the puzzle. O(n^5)

While we implemented dynamic programming, in the end it was not used. The overhead of hashing state and referencing the hash table turned out to be more expensive than normal iteration. This code was deleted and can only be viewed in the git history.

tests.py

The test program contains functions to test each algorithm in batches. Several command line options exist to customize the test parameters.

(aigroupproject-py3.12) ➜ aigroupproject git:(master) ✗ python tests.py -h

- tests.py [-h] [-lt] [-bf] [-v] [-p] [epochs]
- epochs Number of test boards to run (default is 1000)
- -h, --help show this help message and exit
- -lt, --lookup_table Run the tests with a lookup table
- -bf, --brute_force Run the tests with the brute force algorithm
- -v, --verbose Print extra information to the console.
- -p, --parallel Run the tests in parallel

If faster performance is required, parallel processing can be used. For all tests, a timer measures the performance of each test batch.

nxn_solver.py

Just for the extra challenge, it was not too difficult to make the code work on nxn soduku puzzles. Getting the console to print them cleanly was actually the most difficult pa