CSE 4633-01
Programming Assignment 2: Sudoku

Josh Hawkins, JKH360

Assigned 9/27

Due 10/21/2016

10/21/2016
CSE 4633-01
Josh Hawkins (JKH360)

About this submission

The python file submitted (*pa2_jkh360.py*) is the completed programming files. This file is in the configuration with *all heuristics considered*. This is not the optimal configuration for time efficiency. The file contains comments of code that can be uncommented to disable each heuristic, should you be interested to test this file in different configurations.

Data

Configuration	All Heuristics Considered	No Minimum Remaining Value	No Maximum Degree	No Least Constraining Value	No Heuristic
Average seconds per puzzle	2.2793098402	-	6.3266314292	1.73728096008	~75

Figure 1. Table of Raw Data Collected Sorted by Heuristic Disabled

Data Analysis

Heuristics were “disabled” by modifying their functions to return before performing any meaningful calculations. For instance, to disable the Minimum Remaining Values heuristic, the *mrv()* function was modified to return *[unassigned]* before sorting the array. To disable the Maximum Degree heuristic, the *max_degree()* function was modified to return *[tied]* before filtering the array. To disable the Least Constraining Value heuristic, the *order_values()* function was modified to return the domain of the cell referenced (*puzzle.cells[row][column].domain[:])* before sorting the values in the domain.

It would make sense that considering all three heuristics would be the most efficient manner to solve this problem. However, it was determined that disabling the Least Constraining Value heuristic proved to make the total solution quicker to obtain. I assume this is because each time the *order_values()* function is called, forward checking is performed on each value in the domain for the cell. This is very time consuming, especially when repeated so frequently.

The program did not solve any puzzles in 15 minutes when executing without the Minimum Remaining Values heuristic. From what I can tell, it seems that this is because the program will need to select more variables than if the heuristic were considered. Additionally, the resulting variables selected may have had significantly more values remaining in the domain the results from using the heuristic would have had. It appears that the program would eventually solve a puzzle, but it would take a prohibitively long time to solve 100 puzzles.

The program performed slowly when not considering the Maximum Degree heuristic. I assume this is because execution required backtracking significantly more than had it considered degree of variables.

Finally, when no heuristic was considered, the program completed 20 puzzles in roughly almost exactly 25 minutes. This an average of about 75 seconds per puzzle. This was by far the slowest configuration, of course. This seems very logical, given that the program can then only "brute-force" the solution by testing every value in every variable one-by-one until a contradiction is found, then backtrack until eventually a solution is determined.

Thus, the most time-efficient configuration of heuristics was disabling the Least Constraining Value heuristic. While this solution took less time, it likely examined more states than the configuration involving the heuristic. Thus, if states examined must be stored in memory or considered at all, it may be worthwhile to involve this heuristic. Similarly, if the domain for variables was larger than at-most 9 values, the heuristic would likely serve to reduce a greater number of failed states. In that case, the heuristic would make the program much faster. However, since Sudoku variables have a small domain, the heuristic took more time to consider than it saved.

Conclusion

While it does seem logical after further examination, I was surprised that considering the Least Constraining Value heuristic slowed the overall program execution. I believe this experiment would be beneficial to me even if only for this one reason, as it showed that some heuristics may be too computationally intensive for small gains. However, this experiment still managed to reinforce the value in an efficient heuristic for quickly solving tasks in a time- or memory-specific manner. Furthermore, this experiment was valuable for a number of reasons, particularly for understanding a few Python features and CSPs in general.

This program provided an opportunity for me to learn more about Python's underlying operations. For instance, I began solving the *mrv()* task by using a list comprehension to evaluate the length of each cell's domain from the *[unassigned]* list. Then, I used the *list.sort()* function to sort the array based on the second key by using *sized.sort(key=1)*. I eventually decided to reorder the list comprehension to eliminate the need for *key=1*, but this mistake introduced me to the *cmp*, *key*, and *reverse* optional parameters in the *list.sort()* function.

Similarly, this program reinforced the utility of splicing an entire list with the *list[:]* syntax. This program encouraged me to learn more about how it works. I was already aware of *list[start:end]* syntax, but I quickly discovered *list[start:end:step]* thanks to this research. These syntactical sugar features have continuously reinforced Python as my favorite language for programs focusing on algorithm implementation.

Finally, I was initially intimidated by programming a CSP for sudoku. I'd attempted to solve a sudoku game with a simple algorithm by hand about a month ago, and the backtracking became very overwhelming. However, this experiment demystified constraint satisfaction problems for me. This experiment showed that these problems are very easy to program solutions for, while still being powerful and easily understandable.