Colton Avila

CS531

12 May 2021

Sudoku CSP

Abstract:

In this paper we explore the constraint satisfaction problem named Sudoku. We constrained this puzzle using not only the all-diff constraint, but 6 other constraints specific to this problem. Each of these constraints were used to reduce the domain of each cell that was unfilled. Forward-checking was implemented alongside these constraints with the goal of producing a board whose unfilled cells had domains with only one value inside. We used two different methods to decide which cells to pick. A simple fixed order method – across then down the board – and a method that chose the most constrained variable – the cell with the smallest domain. Without constraints, or inferences, the only way that the algorithm could solve the puzzle was through backtracking. We found that the algorithm performed the worst without inferences. Limiting the calculation to only 1000 steps, only 9 of the 77 puzzles were solved for the fixed order cell selection method and 15 out of 77 puzzles for the most constrained variable cell selection method. The algorithm performed the best when the set of inferences, or constraint set, was maximized. In this case we used advanced Sudoku rules to constrain the problem: naked and hidden singles, pairs, and triples. For both cell selection methods, this algorithm found a solution to 71 out of the 77 problems when all rules were implemented. We found that the number of cells filled in the base board did not necessarily reflect the difficulty of the problem for the algorithm.

Introduction:

We sought to explore the constraints that can bound Sudoku and increase the speed it takes to solve the puzzle. Backtracking search was the baseline algorithm used to achieve a solution. This algorithm works by exploring different sets of values, and when the algorithm encounters the situation where there are no valid values, it returns to a previous state. In Sudoku we would try a value for a cell, then move to the next one until we found a cell that had no valid options, or we solved the puzzle. If we find no valid options in our search, we then would return to a previous state to try and another value for a cell.

There are a few constraints that were used in this paper to improve the speed of the sudoku solution.

1. Naked Singles: If there is one cell that contains a single value for its unit (row, column, or square) then the rest of the unit cannot have that value.
2. Hidden Singles: If any cell in the unit have a value that can only belong to that cell, then remove all other values from that cell’s domain.

3. Naked Pairs: If there are two cells with the same set of values – (1,2) and (1,2) for instance, then all other cells in that unit cannot have those values.
4. Hidden Pairs: If there are two cells that have the same subset of values and they are the only cells to have those values, then remove this subset of values for all other cells in the unit.
5. Naked triples: If there are three cells and they all share 3 values – not all cells need to have all three values necessarily – then remove those three values from all other cells in the unit
6. Hidden Triples: Like hidden pairs, if there are three cells that contain values from a three-value tuple, then remove the values in the tuple from all other cells.

Methods:

We analyzed 77 different Sudoku puzzles with 24 -36 cells already filled in. Then varying levels of constraint propagation was used to determine candidates for each empty cell. Choosing the next cell to be analyzed was determined either using a fixed-order method or choosing the most constrained variable. We calculated 4 levels of constraint propagation for each cell picking option: no inference, naked and hidden singles, naked and hidden singles and pairs, and naked and hidden singles, pairs, and triples.

Our algorithm followed the pseudocode below:

```
def function(starting cell):
For each cell choice method and constraint set →
    Check if there are no 0's on the board
        Return "True" if there are none
    Get location of cell to be analyzed →
        If most constrained cell choice method
            Choose a cell with the smallest domain size.
        If fixed order cell choice method
            Choose next cell in row / column.
    For values in the domain (v) of the cell chosen →
        If the program has run for 1000 steps, then return False.
        Check to see if (v) is valid to placed.
        If v is valid then place it
            Implement inferences depending on which constraint set is being used
            If implementing inferences Does Not result in a cell with an empty domain
                Run function again →
```

```
                    If this returns True return result
            If implementing inferences **Does** result in a cell with an empty domain
            Add back all domains removed from cells
            Return current cell value to previous one
    Return False (This allows backtracking)
```

Results:

The following table contains all information generate from this experiment. A list of acronyms for the following table:

- NI – FB: No Inference, Fixed Baseline
- NI – MCV: No Inference, Most Constrained Variable
- NHS – FB: Naked and Hidden Singles, Fixed Baseline
- NHS – MCV: Naked and Hidden Singles, Most Constrained Variable
- NHP – FB: Naked and Hidden Singles and Pairs, Fixed Baseline
- NHP – MCV: Naked and Hidden Singles and Pairs, Constrained Variable
- NHT – FB: Naked and Hidden Singles, Pairs, and Triples, Fixed Baseline
- NHT – MCV: Naked and Hidden Singles, Pairs, and Triples, Most Constrained Variable

The values in the cells will take the following form:

- [U] → "Unsolved"
- [S & #] → Solved and the number of backtracks.
- The "Problem #" cell has the number of the corresponding problem, the number of filled-in cells at the start, and the difficulty rank we decided on.

Problem #	NI-FB	NI-MCV	NHS-FB	NHS-MCV	NHP-FB	NHP-MCV	NHT-FB	NHT -MCV
1 / 33 / Easy	U	S 166	S 0	S0	S0	S0	S0	S0
2/ 31/ Easy	U	U	S 94	S0	S0	S0	S0	S0
3/ 25 /Med	U	U	U	S35	S0	S0	S0	S0
4 /26 /Evil	U	U	U	U	U	U	U	U
5/35/Easy	S 47	S0	S0	S0	S0	S0	S0	S0
6/36/Easy	U	S 171	S0	S0	S0	S0	S0	S0
7/34/Easy	U	S 55	S0	S0	S0	S0	S0	S0
8/28/Med	U	U	U	S126	S0	S0	S0	S0
9/30/Easy	U	U	S700	S0	S0	S0	S0	S0
10/30/Med	U	U	S90	S0	S0	S0	S0	S0
11/26/Evil	U	U	U	U	U	U	U	U
12/28/Easy	U	U	S826	S0	S0	S0	S0	S0
13/26/Med	U	U	U	S71	S0	S0	S0	S0
14/24/Evil	U	U	S849	S0	S0	S0	S0	S0
15/26/Evil	U	U	U	S176	S0	S0	S0	S0

16/30/Evil	U	U	U	S0	S0	S0	S0	S0
17/36/Easy	U	S5	S0	S0	S0	S0	S0	S0
18/35/Easy	S110	S0	S0	S0	S0	S0	S0	S0
19/36/Easy	U	S146	S0	S0	S0	S0	S0	S0
20/34/Easy	S131	S0	S0	S0	S0	S0	S0	S0
21/28/Med	U	U	S32	S0	S0	S0	S0	S0
22/32/Med	U	U	S400	S0	S0	S0	S0	S0
23/19/Med	U	U	U	S23	S0	S0	S0	S0
24/29/Med	U	U	U	S39	S0	S0	S0	S0
25/30/Easy	U	U	S700	S0	S0	S0	S0	S0
26/26/Med	U	U	U	S24	S0	S0	S0	S0
27/25/Med	U	U	U	S198	S0	S0	S0	S0
28/25/Easy	U	U	S245	S0	S0	S0	S0	S0
29/33/Easy	U	U	U	S6	S0	S0	S0	S0
30/35/Easy	U	U	S195	S0	S0	S0	S0	S0
31/29/Easy	U	U	S884	S0	S0	S0	S0	S0
32/27Med	U	U	U	S506	S0	S0	S0	S0
33/26/Med	U	U	U	S18	S0	S0	S0	S0
34/26/Med	U	U	U	S590	S0	S0	S0	S0
35/30/Easy	U	S93	S0	S0	S0	S0	S0	S0
36/32/Easy	U	U	S19	S0	S0	S0	S0	S0
37/25/Med	U	U	U	S122	S0	S0	S0	S0
38/28/Med	U	U	U	S282	S0	S0	S0	S0
39/26/Easy	U	U	S604	S0	S0	S0	S0	S0
40/24/Med	U	U	U	S156	S0	S0	S0	S0
41/35/Easy	U	S121	S0	S0	S0	S0	S0	S0
42/35/Easy	U	U	S15	S0	S0	S0	S0	S0
43/34/Med	U	U	U	S10	S0	S0	S0	S0
44/36/Easy	S279	S0	S0	S0	S0	S0	S0	S0
45/31/Easy	S158	S0	S0	S0	S0	S0	S0	S0
46/30/Easy	U	U	S419	S0	S0	S0	S0	S0
47/27/Med	U	U	U	S26	S0	S0	S0	S0
48/26/Med	U	U	U	S55	S0	S0	S0	S0
49/36/Easy	U	S60	S0	S0	S0	S0	S0	S0
50/34/Easy	S19	S0	S0	S0	S0	S0	S0	S0
51/24/Med	U	U	U	S1	S0	S0	S0	S0
52/26/Med	U	U	U	S234	S0	S0	S0	S0
53/27/Med	U	U	U	S102	S0	S0	S0	S0
54/29/Easy	U	U	S309	S0	S0	S0	S0	S0
55/34/Easy	U	S142	S0	S0	S0	S0	S0	S0
56/29/Med	U	U	U	U	U	U	U	U
57/32/Evil	U	U	U	U	U	U	U	U
58/27/Easy	U	U	S220	S0	S0	S0	S0	S0
59/27/Med	U	U	U	S112	S0	S0	S0	S0
60/24/Med	U	U	U	S211	S0	S0	S0	S0

61/36/Easy	U	S48	S0	S0	S0	S0	S0	S0
62/33/Easy	U	S161	S0	S0	S0	S0	S0	S0
63/30/Easy	U	U	S192	S0	S0	S0	S0	S0
64/28/Med	U	U	U	S25	S0	S0	S0	S0
65/28/Med	U	U	U	S0	S0	S0	S0	S0
66/25/Evil	U	U	U	U	U	U	U	U
67/34/Easy	S195	S0	S0	S0	S0	S0	S0	S0
68/34/Easy	S127	S0	S0	S0	S0	S0	S0	S0
69/34/Easy	S95	S0	S0	S0	S0	S0	S0	S0
70/36/Evil	U	U	U	U	U	U	U	U
71/28/Med	U	U	U	S71	S0	S0	S0	S0
72/25/Med	U	U	U	S509	S0	S0	S0	S0
73/25/Med	U	U	U	S0	S0	S0	S0	S0
74/25/Med	U	U	U	S519	S0	S0	S0	S0
75/25/Easy	U	U	S616	S0	S0	S0	S0	S0
76/25/Med	U	U	U	S100	S0	S0	S0	S0
77/25/Med	U	U	U	S407	S0	S0	S0	S0

Table 1. Results of Sudoku constraint satisfaction problem

Discussion:

There is a large difference in the efficiency of the algorithm as more constraints were placed on the problem. It is safe to say that the grading of a problem should be based on the rules needed to solve them without backtracking. If you need to implement hidden triples in your problem, then it is almost impossible to just brute force your way to a solution; you need to implement the complex rules like hidden or naked triples. We noticed that the more inference rules implement, the less backtracking there was. This makes sense as the forward chaining will reduce the size of the domain of future cells. It was surprising how little of a difference there was between the hidden and naked triples and the hidden and naked pairs. This may be due to inconsistences in the code, but it may also be that if one can find hidden pairs consistently, they may not need to find hidden triples.

The number of cells already filled in a problem did not seem to determine how difficult the problem would be to solve. We think that the configuration of the nodes is what matters when algorithmically solving the board. This relates to the effectiveness of the cell selection method. The fixed-order method performed significantly worse compared to the most constrained variable method. This is due to the amount of backtracking that must occur. Since there is no way to avoid having to deal with cells that have large domains, the fixed-order method is susceptible to failing when there are many open cells with large domains early in selection order.