Graduate Project - Solving Sudoku

Group 4
Raymond Chastain, John Wisnewski, Kaitlyn Zahn

December 16, 2021

Contents

1 Introduction

1.1 Problem Statement

Sudoku is a numerically based puzzle game, in which players must complete a board following a strict set of rules, given only a set of provided initial values. Sudoku boards vary in difficulty and the boards are traditionally 9x9 with a 3x3 subgrid, although they can be any size. The purpose of this report is to compare two known sudoku solving algorithms and to compare their efficiency.

6	7							
	2	5						
	9		5	6		2		
3				8		9		
						8		1
			4	7				
		8	6				9	
							1	
1		6		5			7	

Figure 1: Sample Board

6	7	*3*	*9*	*2*	*4*	*1*	*8*	*5*
4	2	5	*1*	*3*	*8*	*7*	*6*	*9*
8	9	*1*	5	6	*7*	2	*3*	*4*
3	*1*	*4*	*2*	8	*6*	9	*5*	*7*
2	*6*	*7*	*3*	*9*	*5*	8	*4*	1
5	*8*	*9*	4	7	*1*	*3*	*2*	*6*
7	*4*	8	6	*1*	*2*	*5*	9	*3*
9	*5*	*2*	*7*	*4*	*3*	*6*	1	*8*
1	*3*	6	*8*	5	*9*	*4*	7	*2*

Figure 2: What a solved board looks like, blue indicates user values

1.2 Defining Generic Terms

1.2.1 Backtracking

Backtracking acts as effectively a Depth First Search of the space, we will advance by making choices using our heuristics until be hit an unsolveable state, at which point we will revert to the previous state and evaluate it's other potential values. We will repeat this process until we find a solution or exhaust the search space.

1.2.2 Board

The board is the entire gridded play space, which has a subgrid of boxes. The board consists of blank spaces which may be populated from 1 to the size of the board. The size of the board and of the boxes is configurable with our solution, however the defaults are set to the assigned values of 9x9 for the board and 3x3 for the boxes. The board must be square.

1.2.3 Box

The box is a a subgrid within the board space in which all numbers must add up to the size of the Box. Boxes do not need to be square, a 6x6 game board could have a valid subgrid of 2x3 boxes for example.

1.2.4 Constraints

- Row Constraint - Every number from 1 to N, N being the size of the board, must be present only once per row.
- Column Constraint - Every number from 1 to N, N being the size of the board, must be present only once per column.
- Box Constraint - Every number from 1 to N, N being the size of the board, must be present only once per box.

1.2.5 Domain

We define the Domain to be the set of acceptable values for the position that do not violate any of the constraints. The final domain is the intersection of all row, column, and box domains for the position.

1.2.6 Minimum Remaining Value (MRV)

The Minimum Remaining Values is defined as the size of the domain for the position in question. For example if position (1,1) had a domain of 1,2,3 then it's MRV score would be 3.

1.2.7 Degree

To calculate degree we calculate the number of empty spaces relative to each constraint (Row, Column, Box) and sum them to get the total number of empty spaces that affect the domain of the position.

1.2.8 Clue

A clue in Sudoku is any value in the puzzle board that comes pre-provided. The number of clues does not seem to directly control the difficulty of the puzzle, however it clearly has an impact.

2 Algorithm 1 - Backtracking with MRV & Degree

2.1 A brief explanation on changes

This algorithm was updated to support any side board, box combination, whereas the prior implementation was limited ONLY to 9x9 sudoku boards with 3x3 boxes.

2.2 Calculating the next move

1. Calculate the MRV for each empty position on the board. We store this as a 2D array and refer to it as the ”MRV Board”.
2. Get the minimum value of the MRV Board
3. Find all positions in the MRV board containing that minimum value
4. If more than one position has the same MRV score then break the tie by calculating the Degree of each tied position.
5. Return the move

2.3 Forward Checking

Requires board, value, and position:

1. For the row containing the position ensure that using that value does not result in an empty domain for any position in that row - excluding the position being checked. If an empty domain occurs return False.
2. For the column containing the position ensure that using that value does not result in an empty domain for any position in that column - excluding the position being checked. If an empty domain occurs return False.
3. For the box containing the position ensure that using that value does not result in an empty domain for any position in that box - excluding the position being checked. If an empty domain occurs return False.
4. Return True

2.4 Solving the puzzle

1. Calculate the next move.
2. Calculate the domain of the move.
3. For each values in the domain for the move:
   (a) Pop one value from the domain
   (b) Perform forward checking use the move and the value from the domain.
   (c) If forward checking succeeds, set the value and repeat from 1. Otherwise, unset the value and attempt the next value in the domain.
4. If all values in the domain result in an unsolveable puzzle, the return False

3 Algorithm 2 - Knuth's Algorithm X

3.1 About the Algorithm

Algorithm X solves sudoku by reducing it to an Exact Cover problem, Exact Cover problems are a subset of Constraint Satisfaction Problems (CSPs). The problem statement is simple, given a list of a set of constraints X, and Y which is a list of moves mapped to sets of the constraints they satisfy, there should be a Y* which is a subset of Y that covers or satisfies every constraint in X. Algorithm X while efficient is a non-deterministic algorithm, which will be easily understood once the steps of the algorithm are explained below.

3.2 A brief aside on Dancing Links

A popular technique for implementing Algorithm X, and the technique suggested by Donald Knuth [2], is called Dancing Links. Dancing Links (often abbreviated DLX) utilizes a doubly linked list to efficiently implement backtracking algorithms. Although Knuth suggests the implementation he credit's Hiroshi Hitotsumatsu and Kōhei Noshita [1] with it's creation.

The implementation in this paper does not utilize Dancing Links, due to the garbage collector of python and the way references are handled it can be tricky and inefficient if not implemented exceptionally carefully. The implementation used instead utilizes pythons available Set, Dictionary, and List constructs which allow the same function to be achieved.

3.3 Preparing the Data

Preparing the data in algorithm X is a significant setup cost, however the cost savings of the algorithm are generally worth it, as will be shown in the results section. For an NxN Sudoku board, the algorithm generates X which will be of size $4xN^2$ where 4 is the number of constraints types we have (Box, Row, Column, Position). Generating Y will worst-case be $O(N^2)$, however on average it will be much less than this as provided clues will reduce the number of available moves.

1. Define X as a list of all constraints. The constraints needed are Position (every position must be filled), Row (every row must have every value in the domain), Column (every column must have every value in the domain, and Box (every Box must have every value in the domain). Presume a Sudoku board has dimensions NxN, this gives it N^2 positions, N rows, N cols, and B boxes of dimension HxW, where $B = (N/H)x(N/W)$. The algorithm represents these constraints as tuples where a unique string indicates the constraint type, r and c are placeholders for indexes into the board from 0 to N 1, v represents a value in the domain V, and b indicates an index into the boxes such that $b = 0$ is the top left box and $b = B - 1$ is the bottom right box. As such position constraints are ('p', r, c), row and column constraints are indicated as ('r', r, v) or ('c', c, v) respectively, and Box constraints are indicated as ('b', b, v)

2. Create Y such that is a mapping of all moves, to the constraints in X that said move satisfies.

3. Reformat X from a list of constraints, to a dictionary mapping of moves to sets, in which you map constraints to the moves that satisfy them.

3.4 Performing the algorithm

Algorithm X, being a backtracking algorithm is quite simple once the necessary data structures are implemented.

1. Pick the constraint with the smallest number of moves that satisfy it.
2. Pick any move that satisfies that constraint and add it to the solution.
3. "Cover" the constraint by removing any other moves that also satisfy that same constraint, store these such that they can be restored if backtracking is necessary.
4. Repeat from 1

If while executing the above the algorithm encounters an unsolvable board it will backtrack, restore deleted moves, and repeat from an earlier branching moment.

4 Similarities and Differences

There are strong similarities and subtle but key differences between the two algorithms. Both algorithms are a type of Constraint Satisfaction Problem (CSP), as Exact Cover is a subset of CSPs, and both are NP-Complete problems. Backtracking with MRV and Degree will be deterministic in any any case in which MRV and Degree are sufficient tie breakers, however Algorithm X is non-deterministic due to the fact that constraints are chosen according to a repeatable heuristic but moves are not. It is also noteworthy that while the heuristic we used here for Algorithm X is a logical choice, and the choice proposed by it's creator, it is not the only heuristic that can be used. It is also important to note that unlike backtracking with heuristic, algorithm X enumerates all solutions.

5 Results

Both algorithms are complete and are able to find a solution to every puzzle, this is to be expected as they are both backtracking algorithms, although backtracking should rarely be needed and acts primarily as a fail safe for harder puzzles. The results clearly indicate the overall efficiency improvements inherent in Algorithm X, and the advantages of problem reduction. For purposes of benchmarking we consider Algorithm X solved as soon as it has returned a single solution.

To get these results we utilized a set of 1 million sudoku puzzles from Kaggle, for each puzzle we gave each algorithm 5 attempts, and we plotted the average and min/max values of the range.

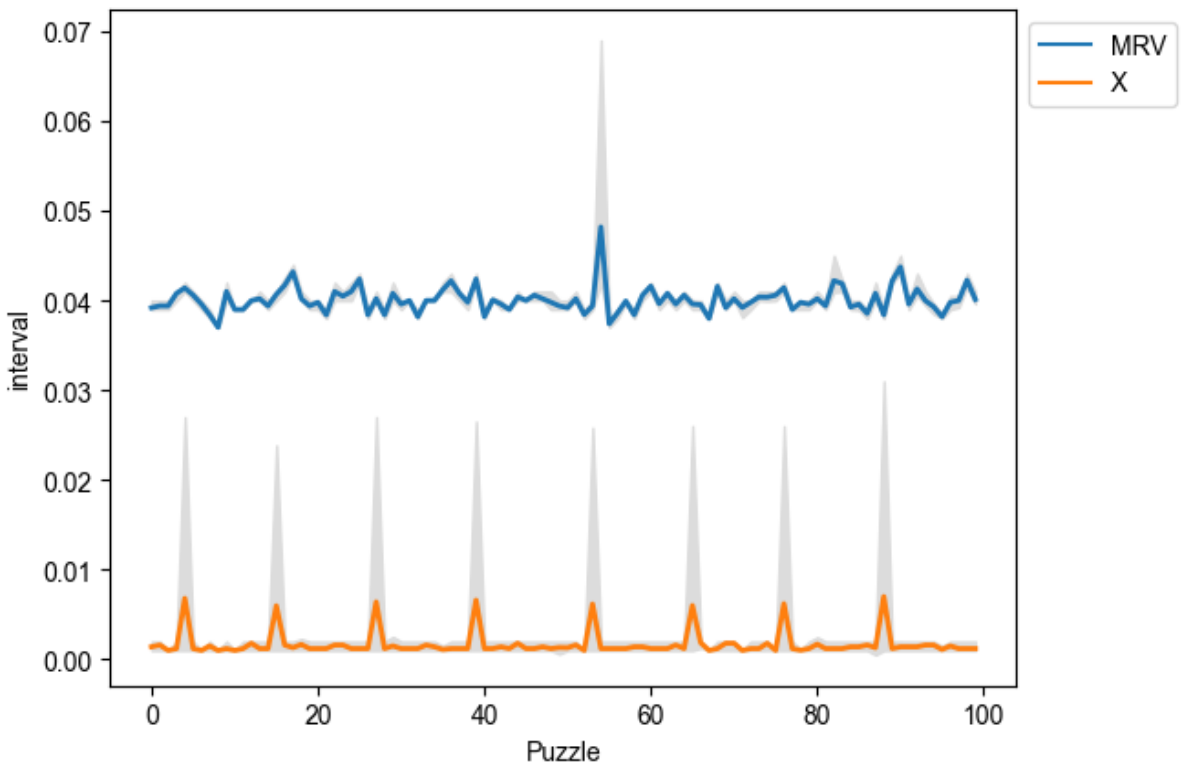

Figure 3: First 100 puzzles

The first 100 puzzles show the that Algorithm X has a bit more variance, this is to be expected due to the increased setup cost. The average run time, and even the worst case values are still well below MRV.

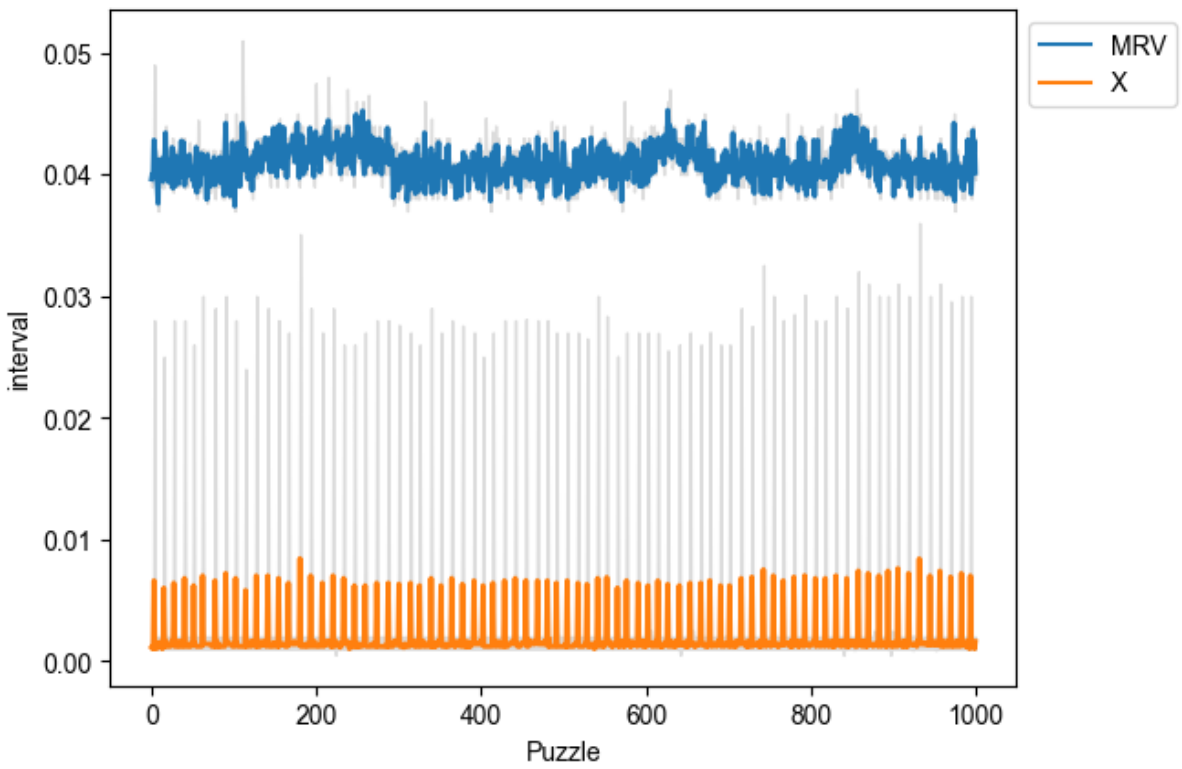

Figure 4: First 1,000 puzzles

The first 1,000 puzzles really showcases the average performance of the two algorithms. You can really see how the variance in Algorithm X plays a role.

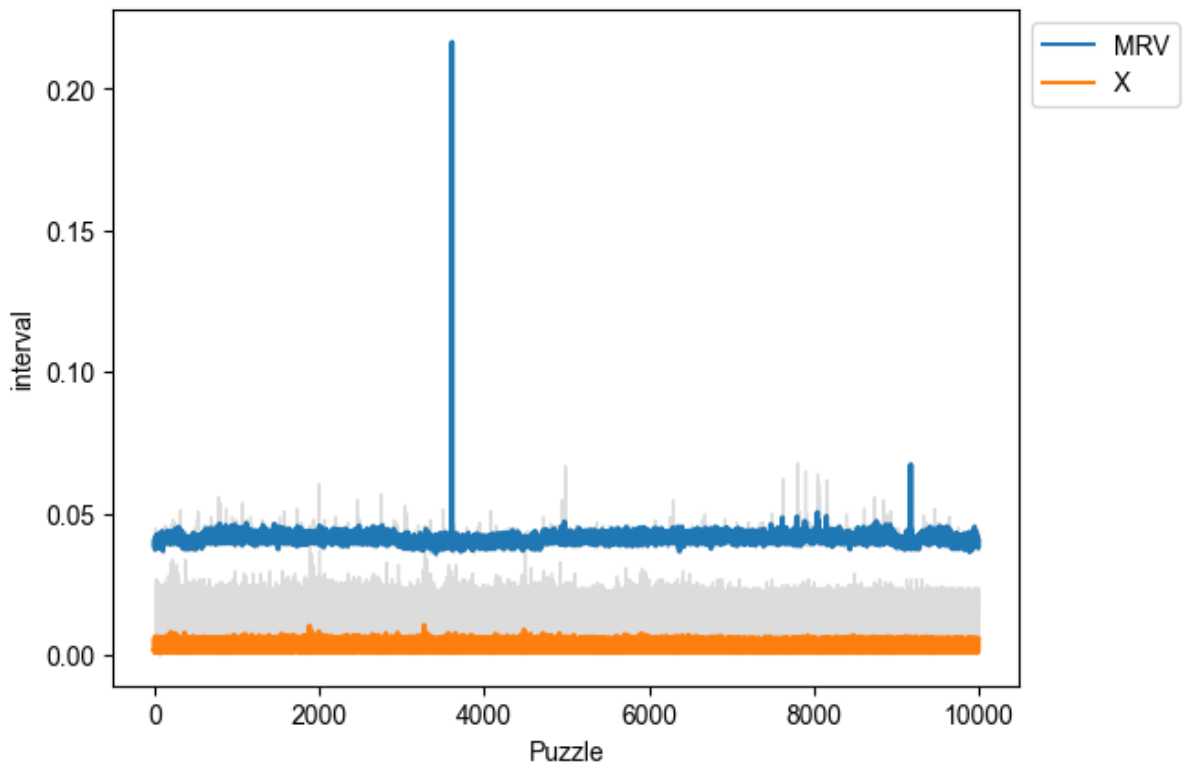

Figure 5: First 10,000 puzzles

The first 10,000 puzzles creates the most interesting graph yet, not only do you see a solid worst case for MRV in which a lot of backtracking was clearly necessary, but you also see moments around the 4500 puzzle count where the worst case Algorithm X is being out performed by the average case of MRV.

6 Conclusion

It is clear that problem reduction from a Constraint Satisfaction Problem to an Exact Coverage problem significantly helps us efficiency obtain solutions to Sudoku. The two algorithms both have unique performance characteristics and if different heuristics were used may behave quite differently. Nonetheless, this provides a good example of how the formulation of the problem can have a significant impact on the solutions derived.

Bibliography

[1] H. Hitotumatu and K. Noshita. A technique for implementing backtrack algorithms and its application. *Inf. Process. Lett.*, 8(4):174–175, 1979. URL `http://dblp.uni-trier.de/db/journals/ipl/ipl8.html`.

[2] D. E. Knuth. Dancing links. 2000.