Analyzing Backtracking and Rule Based Techniques For Solving 9x9 Sudoku Puzzles

Submitted by: Mann. S, Ng. E, Yuen.J

Simon Fraser University

For Intelligent Systems– CMPT417 D100

Submitted to: Prof. Hang Ma

20th December, 2021

Keywords: Sudoku Solver, Backtracking, Automating, Human Techniques, Python

Abstract

Sudoku is a very popular game which appears in almost every daily newspaper. Sudoku can also be found in different forms of media such as books, newspapers, or websites online that generate sudoku puzzles at a click of a button. There are numerous searching techniques used by humans who have mastered the art of sudoku, to solve basic and complex Sudoku problems. This paper will examine the methods used by humans to solve a Sudoku puzzle and compare it to solving Sudoku using backtracking.

Introduction

Sudoku is commonly represented by a 9x9 grid, which contains nine 3x3 subgrids. When the puzzles are generated some of the 81 boxes will be pre-filled in (which will be referred to as clues), and the goal of the player is to completely fill the entire 9x9 grid with numbers ranging from 1 to 9. There are a few rules however; each row should only have one of each number, each

column should only have one of each number, we can only have one number per box, and each 3x3 grid should contain one of each number between 1 to 9. In other words, each row, column and 3x3 subgrid should add up to 45, therefore is a set containing {1,2,3,4,5,6,7,8,9}. Generally another rule is that there is always a unique solution, however this relies on the fact that the puzzle was properly formatted. As Gary McGuire of University College Dublin discovered, for a Sudoku puzzle to be valid and have a unique solution, the original puzzle must have at least 17 clues (McGuire), and puzzles that are generated with 16 starting clues will not have unique solutions. For our project, we have focused on puzzles with unique solutions. Additionally, for comparing our two search techniques of Backtracking Rule-Based which uses constraints and logical reasoning, we will be using "easy" Sudoku puzzles. Easy Sudoku puzzles are defined as using simpler constraints and logic to solve the boards, while intermediate puzzles require techniques such as chaining or hidden trio. The Sudoku will be defined as a 2d array with 9 rows and 9 columns. For each index that is not a clue, there are a possible 9 values that could be used, this makes the runtime $O(9^n)$, as we have 9 possible values and n empty indexes. Despite this runtime there are a few common techniques when it comes to analyzing solving methods relating to Sudoku: backtracking and human techniques (otherwise known as rule based). Both of these have been implemented, to analyze how the algorithms perform in comparison to another.

Implementation

The Sudoku puzzles were generated by a Sudoku Generator which were then added as boards manually. This has been referenced in our reference section, as the author of the site is unknown. The Backtracking and Rule Based Techniques implementations were coded by the team.

Backtracking

Backtracking is a brute force search type of algorithm, which will try the numbers between for our empty indexes, and check for a valid move before it moves on. If it cannot find a valid move, the algorithm will mark the current index to the initial blank value, and move to the previous iteration and try different numbers, and if needed keep backtracking, until the algorithm can proceed further. In our case, the algorithm will first find the empty index by iterating through

each column of a row. After finding the first blank index, our Backtracking Algorithm will attempt to fill in the blank indexe with numbers between 1 to 9. After filling in a number, it'll check if the number entered is indeed valid as per the rules of Sudoku. If the number is valid we'll find the next empty index. If no empty index can be found then we have solved the sudoku. However, if the number is invalid, we'll try the next number and if we have tried all 9 numbers we'll backtrack to the previous number and try different possible combinations. Pseudocode of this has been provided below:

```
def next_empty(puzzle):
        for r in range(9):
                for c in range(9):
                        if puzzle[r][c] is not clue:
                                return row,col

def move_valid(puzzle, guess, row, col):
        row_values = puzzle[row]
        col_values = []
        for i in range(0,9):
                append col values to col_values
        if guess in row_values or guess in col_values:
                return False
        Get 3x3 subgrid row and col starting points
        if guess in 3x3 subgrid:
                return False
        return True

def backtracking(puzzle):
        row, col = next_empty(puzzle)
        if row and col are both None:
                return True
        for guess in range(1,10):
                if move_valid(puzzle, guess, row, col):
```

replace empty index with guess
if solve_sudoku[puzzle]:
return True
Since move not valid, reset current index to blank index and backtrack
return False

One key decision that was made was how we would look for the next empty index. We had the option to either look for empty values in row 1, followed by row 2 until we reached the end of row 9, or we could have looked for empty values in column 1, followed by column 2 until we reached the end of column 9. This minor change can improve the results of how long it takes for a puzzle to be solved by a factor of greater than 5 at times. This improvement and whether it is better to scan rows or columns is largely based on the puzzle input, as going through one way may produce less backtracking, therefore cutting down the time it takes to solve the puzzle. For consistency, our group chose to use the former and look for values in row, indexing by [row][col] and increasing col until we reach the last column before incrementing the row.

Rule Based (Human) Techniques

As a comparison to the brute force approach of Backtracking, the implementation of Rule Based techniques attempts to mimic the logical thought process of human players when trying to solve a Sudoku puzzle. There are a multitude of various techniques humans use to solve Sudoku puzzles, where more advanced techniques are required to solve more complex puzzles. The current implementation of the Rule Based Techniques is able to solve beginner level Sudoku puzzles, which requires techniques that find Singles and Hidden Singles. As a reference, the current implementation of the Rule Based Techniques is capable of solving the easy difficulty of New York Times' Sudoku puzzles. The general idea for the human approach is to create a list of potential candidates for each empty cell and use various techniques to eliminate candidates from the list until one is left to fill in the cell. As for the implementation of the Rule Based Techniques, the algorithm will first create a dictionary for all positions that are empty, with the key being a tuple of the position (row, col) and the value being a list of numbers between 1 to 9. This dictionary will be used as a reference for potential candidates for each empty cell. The algorithm will then find the first empty cell by iterating through each column of a row. After

finding the first blank index, the algorithm will then iterate through all possible guesses for the cell. Guesses that are not valid are removed from the dictionary. If the cell only has one candidate, the cell is filled. If the cell has multiple potential candidates, the algorithm will implement other methods to select amongst the candidates. If it successfully selects a candidate amongst the list, the cell is then filled, else the position is skipped and will be revisited after one full cycle of the board. The algorithm will then proceed to find the next empty cell and repeat the process. Pseudocode of the algorithm has been provided below:

```
def next_empty(puzzle, no_solution):
        for r in range(9):
                for c in range(9):
                        if puzzle[r][c] is empty:
                                if (r, c) is not in no_solution:
                                        return row,col

def valid_guess(puzzle, row, col, guess):
        if guess already exists in row, col, or subgroup
                return False
        return True

def select_possible_candidate(puzzle, row, col, candidates, candidates_dict):
        for guess in candidates:
                if hidden single:
                        puzzle[row][col] = guess
                        return True
                else if not check_other_candidates:
                        puzzle[row][col] = guess
                        return True
        return False

def human_solver_sudoku (puzzle):
```

```
solved = False
candidates_dict = initialize_candidate_dict(puzzle)
no_solution = []
while solved is False:
        cell_candidate = []
        row, col = next_empty(puzzle, no_solution)
        if row is None and col is None:
                if no_solution is empty:
                        solved = True
        else:
                empty no_solution
                row, col = next_empty(puzzle, no_solution)
        for guess in range(1,10):
                if valid_guess(puzzle, row, col, guess):
                        cell_candidate.append(guess)
        candidates_dict[(row, col)] = candidates
        if len of candidates = 1:
                puzzle[row][col] = guess
                delete position from candidates_dict
        else:
                if not select_possible_candidate:
                        append (row,col) to no_solution
```

As the current implementation only covers puzzles constructed using singles and hidden singles, the rule based solver is unable to complete boards constructed using more advanced techniques such as Naked Pairs/Triples/Quadruples, Hidden Pairs/Triples/Quadruples, Intersections, X-Wings and many more. More work is required to extend the human solver to be able to solve these difficult puzzles. The main difficulty of extending the solver to implement advanced techniques is being able to capture and quantify all the necessary conditions and rules that must occur for a technique to be applied.

Methodology

We will investigate whether the backtracking algorithm or Rule-Base, which uses logical reasoning and conditions is better to solve our Sudoku puzzles. We will be experimenting using a sample size of 25 Sudoku Boards which are classified as easy. The criteria defined for an Easy board involves puzzles that can be solved by using basic techniques such as singles and/or hidden singles. A sample size of 25 was chosen to investigate and patterns our solvers might develop, and to avoid coincidences where an algorithm performed better but overall that specific algorithm is worse.

Both our Rule Based algorithm and Backtracking algorithm highly depend on the difficulty of the puzzle. As currently our Rule Based algorithm can only solve easy puzzles, the algorithm fails when attempting to solve puzzles that require additional conditions implemented. Generally however, using advanced techniques such as chaining, X-Wing, and but not limited to Swordfish would allow the solver to solve more intermediate problems. Backtracking, however, solves any Sudoku puzzles as it uses a brute-force technique, as it attempts every number, before backtracking if needed, and repeats this process until the puzzle is solved or is determined to be unsolvable.

In addition to the difficulty of the puzzle, the number of pre-filled values will also affect both algorithms, but is more apparent for backtracking. As our pre-filled numbers decrease, our backtracking algorithm is more likely to have to backtrack and change its solutions as it considers a valid move if the number is not in the 3x4 subgrid, and doesn't appear in its designated row and column. However, since Rule-Based uses logical reasoning and conditions, this is avoided at times.

Experimental Setup

For consistency the experiment was ran on one machine. The experiment involved using Python 3.7 for crafting up the implementations (backtracking and rule-based "human techniques") and

test cases. As per the environment, our team utilized a 2017 Macbook Pro running on macOS Catalina Version 10.15.7. The Processor of this Macbook was a 2.3 GHz Dual-Core Intel Core i5 and had 8 GB 2133 MHz LPDDR3 memory. Additionally, for this report, the boards were not printed out to gather the computing times, as printing the boards would interfere with the actual solving. Therefore, the files human_solver.py and backtracking-solver.py were used.

Results

The time it took to solve each board was calculated, additionally the time it took to solve 5, 10, 15, 20 and 25 boards was also calculated. These results were then plotted into Figure 1, Figure 2, Figure 3, Figure 4 and Figure 5.

Figure 1 shows the time it took each board individually using Backtracking. The majority of the boards can be solved within ~0.001 to 0.03 seconds. However, as seen in Figure 1, board number 4, 6, 11 and 18 required more than 0.06 seconds to be solved, these boards can be classified as outliers.

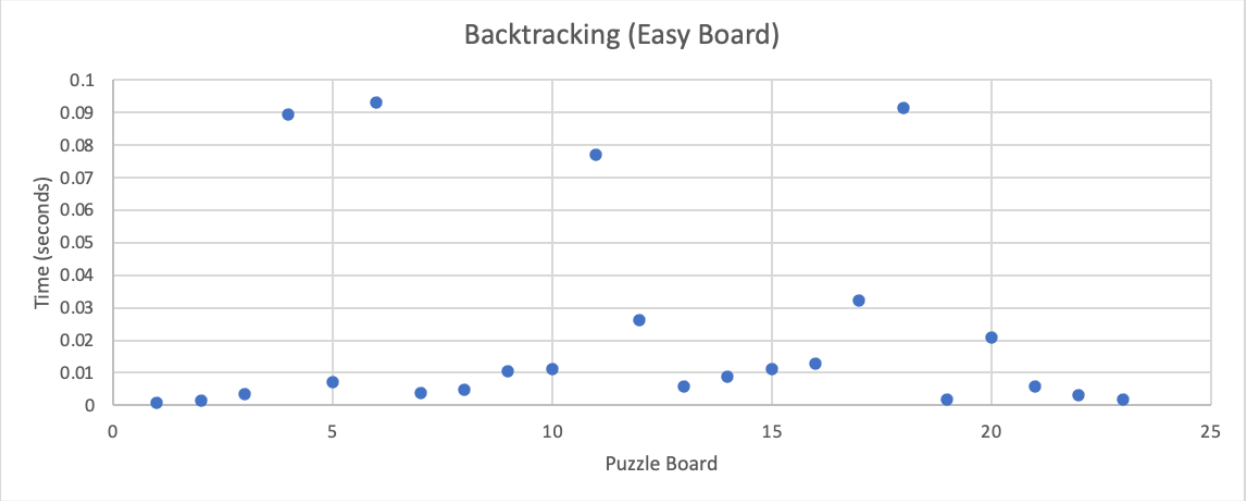

Figure 1: Solving Sudoku with Backtracking. Times taken per Board

Figure 2 showcases the total time it took our Backtracking Algorithm to solve all the boards in increments of 5. Backtracking is fairly consistent in solving the boards, as we notice a linear increase in the best of fit line. There is a slight non-linear increase between solving 15 boards and solving 20 boards, however this slope decreases again, between boards 20 and 25.

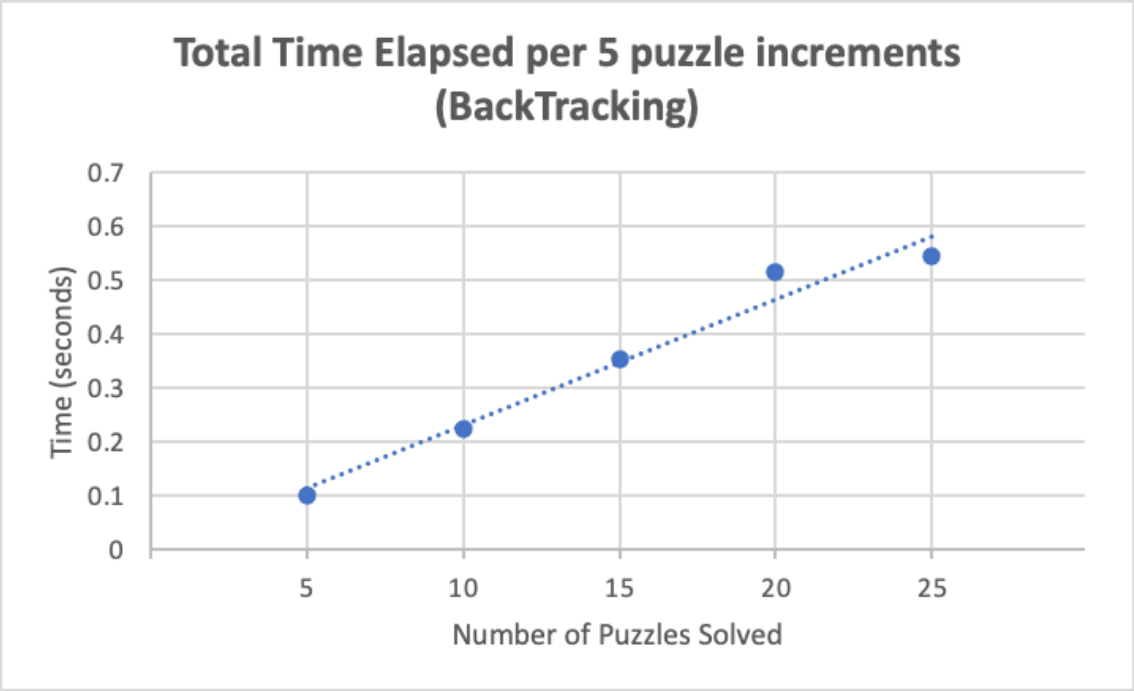

Figure 2: Solving Sudoku with Backtracking. Total Time Taken To Solve Boards

Figure 3 showcases the time it took to solve each board individually using our Rule Based Solver. Similarly to Backtracking, using Rule Based Techniques is quite consistent throughout as majority of the boards can be solved in 0.006 to 0.009 seconds. We have two boards which were solved rather quickly, those being boards 1 and 2.

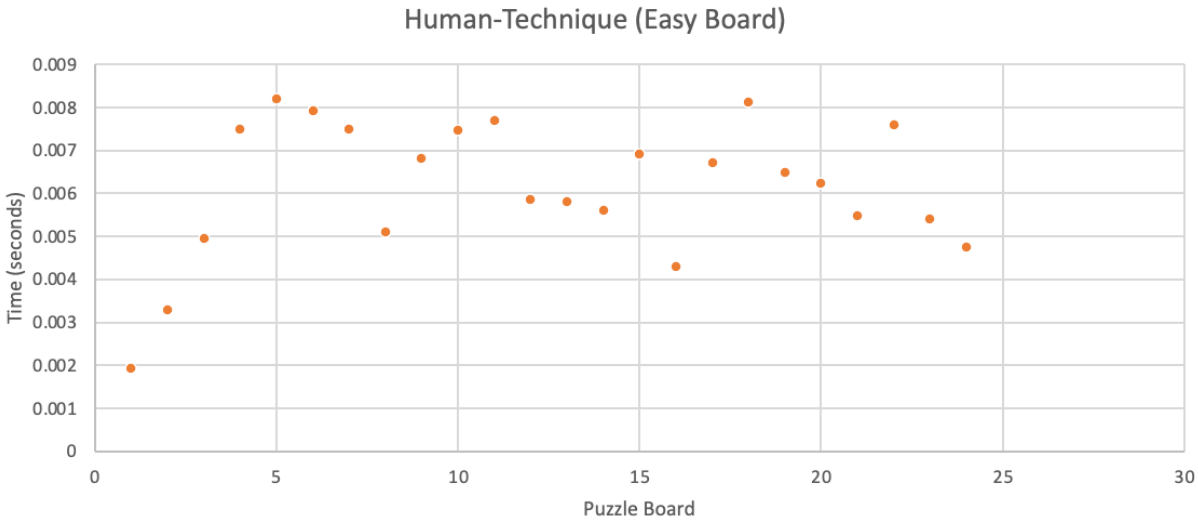

Figure 3: Solving Sudoku with Rule-Based (Human Technique). Times taken per Board

Figure 4 showcases the total time it took our Rule Based Algorithm to solve all the boards in increments of 5. Based off of our data, our algorithm does indeed take a Linear time to solve boards in increments of five, as all of the lines are on our best of fit line.

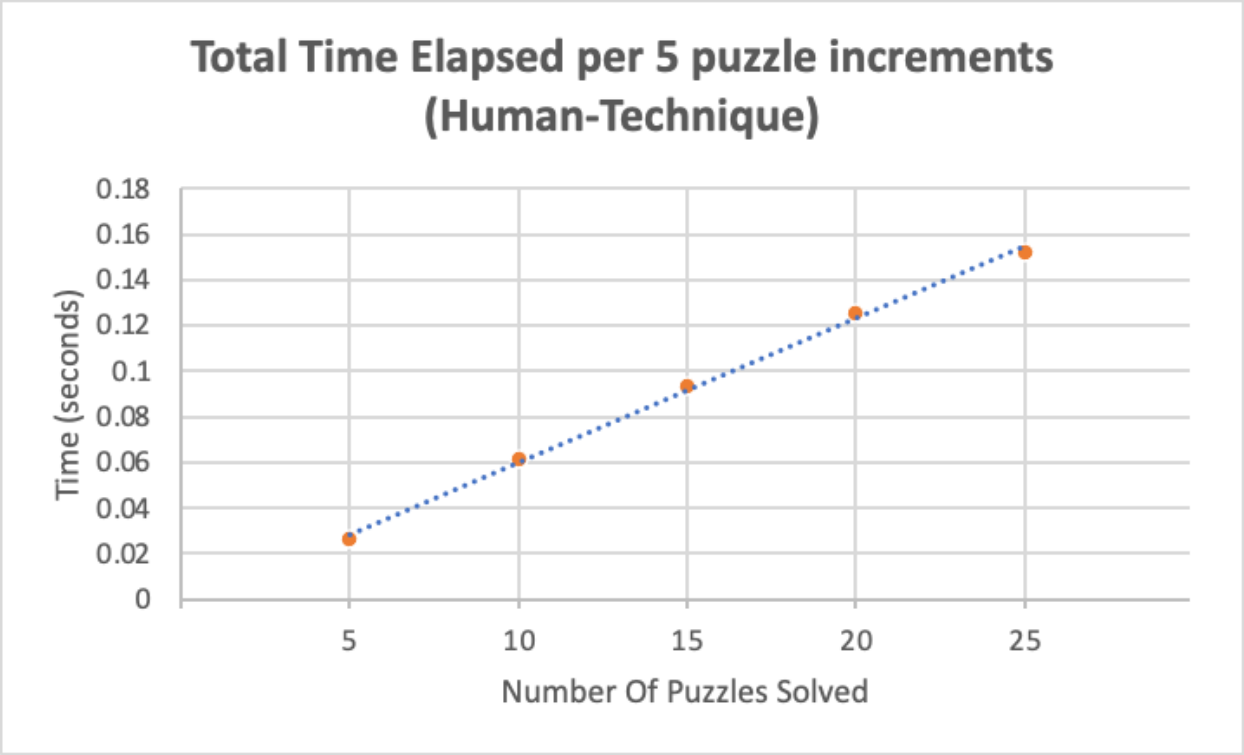

Figure 4: Solving Sudoku with Rule-Based Techniques. Total Time Taken To Solve Boards

Compare & Contrast

Upon running both Backtracking and Rule-Based “Human Technique " search algorithms, and plotting and graphing our data our simulated experiment data suggests that Rule-Based is more efficient than Backtracking.

Figure 5 plots the solving speeds of Backtracking vs a Rule Based Approach. The Rule-Based approach is considerably faster when solving each puzzle, as it stays below the 0.01 line, while Backtracking is consistently above this line. There are a few instances where Backtracking is faster, however, these instances are classified as outliers. Additionally, Table 1 showcases the times in numerical data it took to solve all of the boards in increments of five.

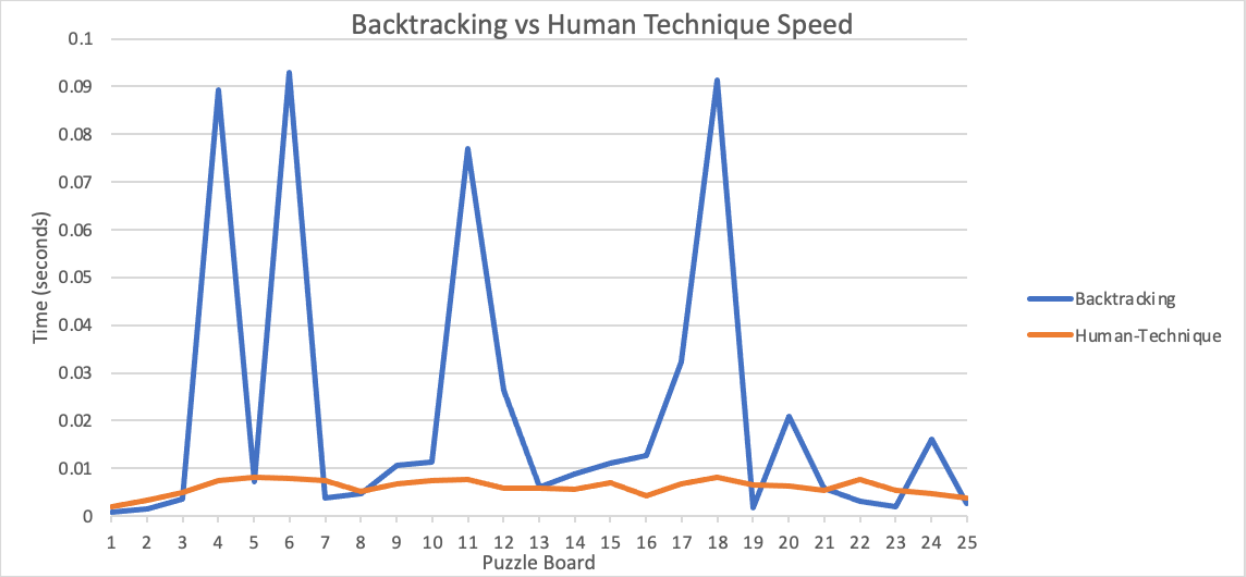

Figure 5: Backtracking vs Human Technique Solving Speed

	BackTracking	Human-Technique
TIME TAKEN TO SOLVE 5 BOARDS:	0.102604866	0.026113987
TIME TAKEN TO SOLVE 10 BOARDS:	0.226111174	0.061169863
TIME TAKEN TO SOLVE 15 BOARDS:	0.355428934	0.093236923
TIME TAKEN TO SOLVE 20 BOARDS:	0.514579058	0.125184059
TIME TAKEN TO SOLVE 25 BOARDS:	0.544414043	0.152349949

Table 1: Total Time Elapsed per 5 puzzle increments (Backtracking vs Human-Technique)

A plausible explanation for this phenomenon is due to Backtracking utilizing “random” uninformed search mechanisms. Because the approach is “random”, more time is required to traverse through all the options for every unknown box within the subgrid. Backtracking would be considered as a search problem with a large state-space. On the contrary, Rule-Based “human-technique” utilizes constraints and makes logical decisions, similar to how humans would play Sudoku. Because of these constraints, the state-space is shrunk and the amount of

work-load is significantly reduced. Theoretically, one can hypothesize that state-space is a crucial factor in determining efficiency - something that we would explore in additional next steps, if we were to build upon the project.

Furthemore, when looking at the median and mean for both Algorithms we notice that the Human Techniques values are much lower. As Figure 6 and Table 2 showcase, Backtracking takes approximately 3.6x longer on average to solve a Sudoku puzzle. Additionally, the median value roughly 3.5x higher

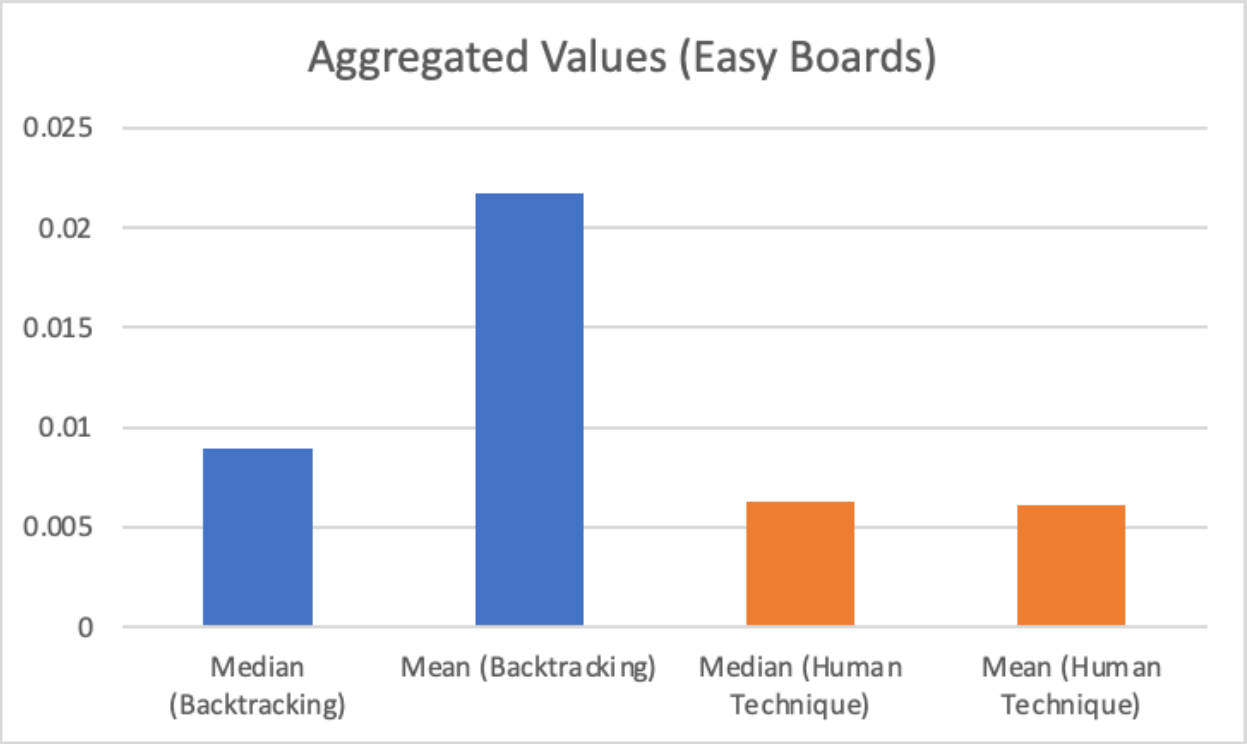

Figure 6: Aggregated Data (Raw

Median (Backtracking)	**Mean (Backtracking)**	**Median (Human Technique)**	**Mean (Human Technique)**
0.008886099	0.021750107	0.006235123	0.006058626

Table 2: Aggregated Data (Raw)

Additional Research

In this paper, we had a deep dive into the utilization of backtracking algorithms and how it compares to human technique. While backtracking with recursion is a popular method, for Artificial Intelligence (AI), to solve Sudoku puzzles, it is not the most efficient search algorithm. Beyond the scope of backtracking, there exist other search algorithms. One example, of a more *efficient* method, is Constraint Satisfaction Problem (CSP) Search, which relies on constraints being defined and satisfied. In the context of Sudoku, CSP involves 3 processes "constraints": (1) Elimination, (2) Only Choice, and (3) Naked Twin. Elimination is the process of removing already established values from the list of options from the row, column, and 3x3 subgrid (e.g., if the value 7 is present in a row, we can confidently remove 7 from the list of plausible values from other areas within that row, its column and its residing 3x3 box) - refer to Figure 7: Eliminate.

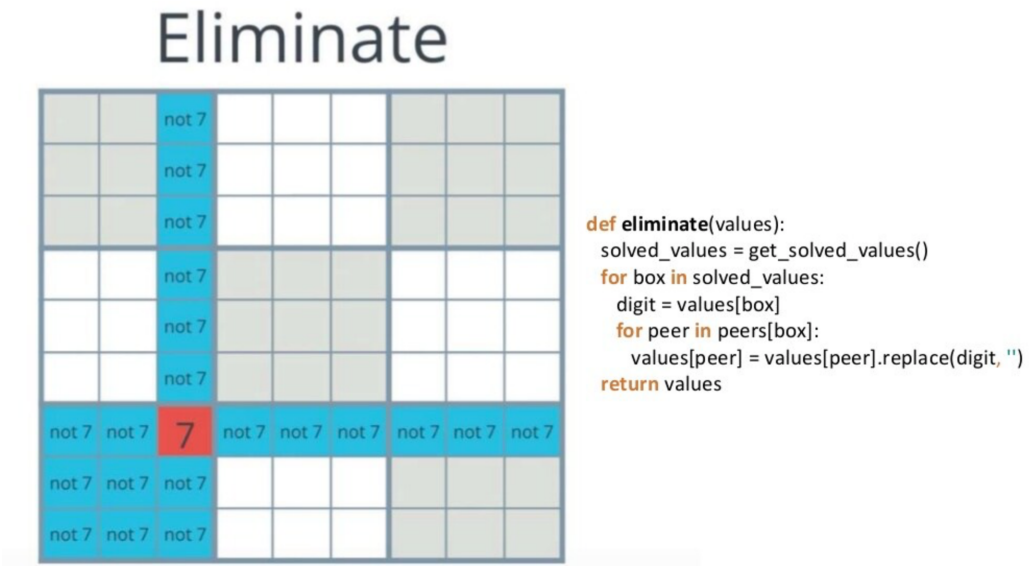

Figure 7: Eliminate (P. Denny, 2017)

Only choice is the process where a box can only contain one value, which would be specified by the user, it will be the only choice - refer to Figure 8: Only Choice.

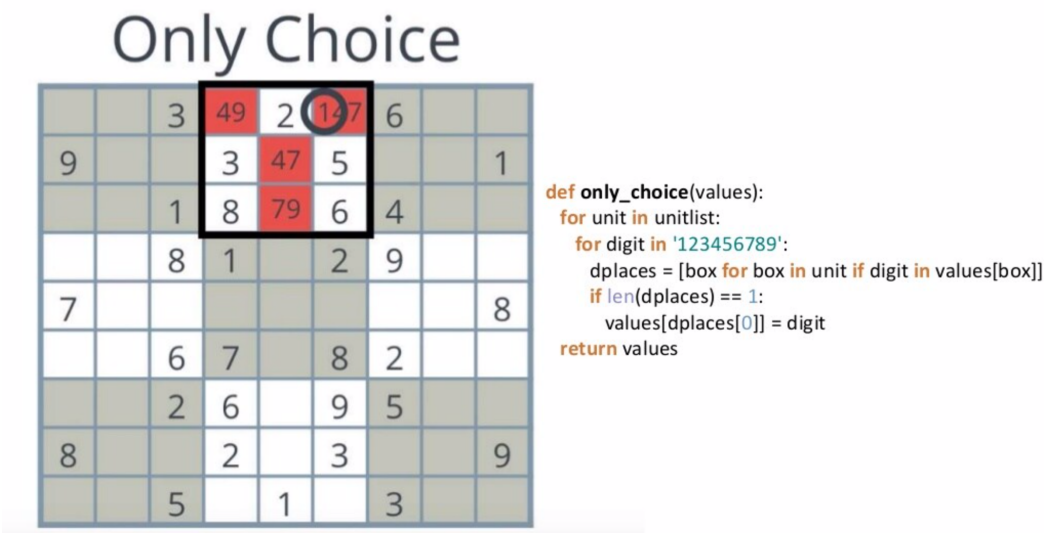

Figure 8: Only Choice (P. Denny, 2017)

Finally, naked twin, is when there are pairs of values that can fit within the same row or column; which means every other box in that row or column cannot be either pair's value - refer to Figure 9: Naked Twins.

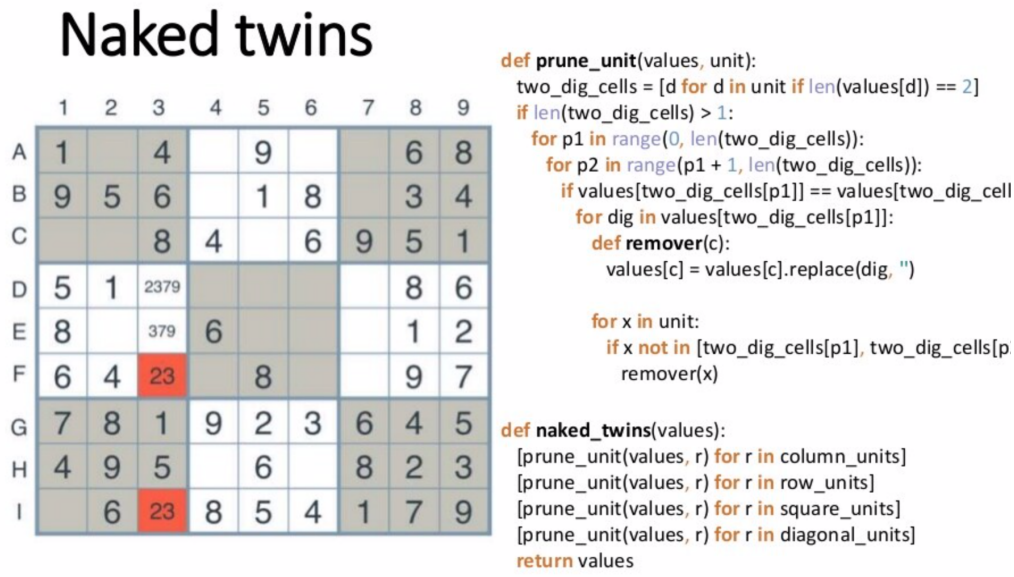

Figure 9: Naked Twins & Pruning (P. Denny, 2017)

Keep in mind, there is a repetitive iteration between both Elimination, Only Choice, and Naked Twin to drastically reduce the space-state. The pseudocode, for reducing puzzle size, has been provided below:

```
def reduce_puzzle (values):
    stalled = false
    while not stalled:
        number_of_solved_values_before = how many occupied boxes
        eliminate (values)
        only_choice (values)
        naked_twin (values)
        number_of_solved_values_after = new count of how many occupied boxes
        if number_of_solved_values_before = number_of_solved_values_after:
            return true
        if any box has in puzzle are not occupied:
            return false
return values
```

CSP differs from backtracking as it does not "brute-force" the entire state-space looking for plausible options (i.e, random trial-and-error of entries for each unknown box). Utilizing constraints in CSP, it significantly reduces the state-space and reduces the workload. In significantly complex puzzles, backtracking will take an extraneous amount of time due to *"trialing-and-erroring"*. Thus, backtracking is not the most efficient search algorithm.

Conclusion

From our experiment we concluded that, for the puzzle solutions that were solved by our Rule-Based solver and Back-Tracking solver, using a Rule-Based solver was a more efficient algorithm (refer to Results for more information). As predicted in our presentation, our rule based solver used logical decisions and constraints to find potential numbers, whereas Backtracking searches use brute force. This is what makes Backtracking less *time* efficient than

the Rule-Based solver as it does not use logical decisions or constraints to investigate individual unknown boxes. Backtracking is more superior in the context as it can solve any Sudoku puzzle given some time, whereas Rule-Based depends on the constraints and logics coded to make decisions. From additional research, a more efficient means for Sudoku solver would be to implement search algorithms that focus on reducing the state-space (fewer unknown slots in the puzzle will allow for less extraneous work by the agent). *However, that is beyond the scope of this paper.*

An additional next step in research would be to see how the algorithms handle other types of Sudoku such as Jigsaw Sudoku, Sandwich Sudoku and Jigsaw Sandwich Sudoku. Jigsaw sudoku differs from regular sudoku as there are no long 3x3 subgrids. Instead the subgrids are formed in puzzle type pieces consisting of 9 squares. In sandwich sudoku however, the subgrids are still 3x3 squares however, by each row and column there is a number listed ranging from 0 to 35 which indicate the sum of the digits between the 1 and the 9 in the specified row or column. Jigsaw Sandwich Sudoku is just a combination of both Jigsaw and Sandwich Sudoku. The algorithms would have to be modified accordingly to check whether the moves are valid, as the sum would need to be checked for Sandwich Sudoku and its subtypes, while the subgrid boundaries would first have to be found then checked in Jigsaw Sudoku and its subtypes. Another interesting thing would be to analyze if using a Sandwich Sudoku would allow fewer clues to be initially given, as there would be an additional constraint used.

Another additional step in research would be experminating with different puzzle sizes. Using different puzzle sizes would change the number of required minimum amount of clues for unique solutions. The process however for the algorithms would stay the same as described in the implementation sections, therefore we hypothesize that our results would be consistent with our findings found during this analysis. Finally, adjusting the Rule Based solver to use more techniques to handle more difficult puzzles would allow us to expand the comparisons between Backtracking and Human Techniques.

Appendix A - Puzzle Times

Puzzle #	Backtracking	Rule-Based
0	0.000734806	0.001925945
1	0.001527071	0.00329113
2	0.003535986	0.004940987
3	0.089360952	0.00750494
4	0.007312298	0.008188248
5	0.093001127	0.00793004
6	0.003834009	0.007498026
7	0.004822016	0.005104065
8	0.010547161	0.006813049
9	0.011209249	0.007467031
10	0.076969862	0.007688999
11	0.026273012	0.005865097
12	0.005986214	0.005814791
13	0.008886099	0.005602837
14	0.011096716	0.006916761
15	0.012792826	0.004299164
16	0.032252073	0.006718874
17	0.09125185	0.008112192
18	0.00168705	0.006486893
19	0.020906925	0.006235123
20	0.005787134	0.005482912
21	0.003194094	0.007601976
22	0.001950026	0.005406857
23	0.01615715	0.004746914
24	0.002676964	0.003822803

Appendix B - Time Taken To Solve Puzzles in intervals of 5

Puzzles Intervals	Backtracking	Rule-Based
0 - 4	0.102604866	0.026113987
0 - 9	0.226111174	0.061169863
0 - 14	0.355428934	0.093236923
0 - 19	0.514579058	0.125184059
0 - 24	0.544414043	0.152349949

References

Denny, P. (2018, February 6). *Constraint propagation*. Slideshare. Retrieved December 20, 2021, from https://www.slideshare.net/PeetDenny/constraint-propagation

Krazydad. “Jigsaw Sudoku by Krazydad.” *Krazydad*, https://krazydad.com/jigsawsudoku/.

McGuire, Gary. “There Is No 16-Clue Sudoku: Solving the Sudoku Minimum Number Of...” *ArXiv.Org*, 1 Jan. 2012, arxiv.org/abs/1201.0749.

“Sudoku Generator - Puzzle Maker - Printable Sudoku Puzzles.” *DIY Printable Generators*, 4 Oct. 2020, https://printablecreative.com/sudoku-generator.

Wealth Wizards Engineering. (2017). *Constraint Propagation for solving Sudoku puzzles*. *YouTube*. YouTube. Retrieved December 20, 2021, from https://www.youtube.com/watch?v=A_5Hh8xdLFQ.