A Survey of Sudoku Solving Algorithms

CS7IS2 Project (2019/2020)

Arun Bhargav Ammanamanchi, Lujain Dweikat, Donal Egan, Bharat Vyas

ammanama@tcd.ie, dweikatl@tcd.ie, doegan@tcd.ie, vyasb@tcd.ie

Abstract. The popular Sudoku puzzle is usually solved by humans using a range of inference strategies. While it can take us minutes or even hours to solve a Sudoku, computers can solve them in a fraction of the time using both inference and search methods. In this paper, we carry out a survey of different Sudoku solving algorithms. We test the performance of several algorithms on Sudoku puzzles with three different levels of difficulty - easy, medium, and hard. We analyse the algorithms' performance with respect to three criteria, namely the number of nodes expanded in the search tree, the time taken to find the solution, and the memory usage while searching for the solution. Of the algorithms that we test, we find that the best algorithms for solving a Sudoku puzzle are backtracking search with the minimum-remaining-values heuristic and Algorithm X.

Keywords: Sudoku, constraint-satisfaction, backtracking search, minimum-remaining-value, AC-3, breadth first search, Algorithm X

1 Introduction

Modern Sudoku, whose name derives from the Japanese translation of the phrase *the digits must be single*, first started gaining popularity in Japan in the 1980s. Since then, the puzzle has become extremely popular with humans all over the world who use different, and often quite complicated, inference strategies to solve them.

A Sudoku puzzle contains 81 squares (called cells) laid out in a 9×9 grid. This grid is subdivided into nine 3×3 sub-grids or boxes. Initially, some cells are filled with digits from 1 to 9. The aim of the game is to fill in the rest of the cells with digits from 1 to 9 so that no digit appears twice in any row, column or 3×3 sub-grid. A properly formulated Sudoku puzzle should have a exactly one solution.

In this paper we evaluate the performance of different algorithms when solving Sudoku puzzles. The chosen algorithms use both inference and search methods to solve the Sudoku. We evaluate the algorithms according to the number of nodes expanded in the search tree, the time taken to solve a puzzle, and the memory usage while solving a puzzle.

The rest of the paper is structured as follows: In section 2, we provide a review of the related research in the area. In section 3, we formalise our research question and provide detailed descriptions of the algorithms that will be evaluated. In section 4, we provide the methodology for our evaluation, the results of our evaluation, and a discussion of these results. Finally, in Section 5, we provide a brief summary of our work and conclude the paper.

2 Related Work

Sudoku are a classic example of a constraint satisfaction problem (csp). Approaches to solving them can include inference techniques, search methods or a combination of both.

In [2], the concept of *arc-consistency* is introduced. Two variables, X and Y, in a csp form an arc if they share a binary constraint. The variable X is said to be arc-consistent with respect to Y if for every value x in the domain of X there exists a value in the domain of Y that satisfies the constraint on the arc. A csp is arc-consistent if every variable in the csp is arc-consistent with every other variable. The AC-3 algorithm, also introduced in [2], can be used to achieve arc-consistency in a csp. We examine the ability of the AC-3 to solve a Sudoku by itself using only inference. We also use AC-3 as a pre-processing step to reduce the search space before applying search algorithms.

The third algorithm we chose is called Algorithm X. Algorithm X is an algorithm described by Donald Knuth in [7] that uses dancing links [7], a form of two-dimensional circular doubly linked lists, in order to solve an exact cover problem.

An exact cover problem is an NP-complete problem as described by [8]. Any NP-complete problem can be translated into an exact cover problem, and since a Sudoku puzzle is a form of an NP-complete problem, it can be translated into an exact cover problem solvable using Algorithm X in a straight-forward way.

3 Problem Definition and Algorithm

A Sudoku puzzle is an example of a constraint satisfaction problem (CSP). Each of the 81 cells in the Sudoku grid is called a *variable.* The *domain* of a variable is the set of values that it can take. Initially, empty cells have domain {1, 2, 3, 4, 5, 6, 7, 8, 9}, while pre-filled cells have a domain consisting of the single value entered in the cell. All constraints in a Sudoku puzzle can be expressed as binary constraints. For example, the requirement that all cells the first row contain different values can be expressed as

$$A1 \neq A2, A1 \neq A3, \ldots, A2 \neq A3, A2 \neq A4, \ldots, A7 \neq A8, A7 \neq A9, A8 \neq A9.$$

A cell's *neighbours* is the set of cells with which it shares a constraint (it does not include the cell itself). Each cell has 20 neighbours. An *assignment* is a partial or complete solution to a Sudoku.

For our research, We analyse the performance of several algorithms and heuristics for solving Sudoku. The following subsections describe the algorithms and heuristics we explore.

3.1 AC-3

The AC-3 algorithm introduced by [2] is an inference algorithm for ensuring arc-consistency in a constraint satisfaction problem (csp). The following pseudocode illustrates the algorithm. The function REVISE() which the algorithm uses it described first:

```
function REVISE($csp$, $X_i$, $X_j$):
    $revised \longleftarrow false$
    for each $x$ in the domain of $X_i$:
        if there is no $y$ in the domain of $X_j$ such that $x \neq y$:
            remove $x$ from the domain of $X_i$
            $revised \longleftarrow true$
    return $revised$

function AC-3($csp$):
    $queue \longleftarrow$ set of all arcs in $csp$
    while $queue$ is not empty:
        $(X_i, X_j) \longleftarrow$ next arc in queue
        if REVISE($X_i, X_j$):
            if domain of $X_i$ is empty:
                return $false$
            else:
                add arc $(X_k, X_i)$ to $queue$ for each neighbour $X_k$ of $X_i$
    return $true$
```

An arc is a pair of variables that share a constraint, i.e. two Sudoku cells that cannot have the same value. For example, the pair of cells $(A1, C3)$ is an arc. Initially, the *queue* consists of all arcs. The first arc, for example $(A1, C3)$, is removed from the queue. For each value x in the domain of $A1$, the algorithm checks if there exists some value in the domain of $C3$ which can satisfy the constraint, i.e. a value different to x. If no such value is found, x is removed from the domain of $A1$. If, after all values have been checked, the domain of $A1$ is unchanged the algorithm moves to the next arc. Otherwise, all arcs $(Y, A1)$, where Y is a neighbour of $A1$, are added to the queue. The algorithm runs until the earlier of the *queue* becomes empty or the domain of a variable becomes empty

If the domain of a variable becomes empty, the AC-3 algorithm returns *false* and the constraint satisfaction problem does not have a solution. If the AC-3 algorithm reduces the size of every cell's domain to one, then the Sudoku puzzle has been solved. Otherwise, we can pass the Sudoku puzzle with reduced domain sizes to the backtracking search algorihtm (described below).

We test the performance of AC-3 as a standalone algorithm for solving Sudoku and also as a preprocessing step to reduce domain sizes before using backtracking search (described below).

3.2 Backtracking search / Depth first search (DFS)

Depth first search Depth first traversal or Depth first Search is a recursive algorithm for searching all the vertices of a graph or tree data structure.A standard DFS implementation puts each vertex of the graph into one of two categories, Visited and Not Visited.The algorithm works as follows:

1. Start by putting any one of the graph's vertices on top of a stack.
2. Take the top item of the stack and add it to the visited list.
3. Create a list of that vertex's adjacent nodes and add the ones which aren't in the visited list to the top of stack.

4.Keep repeating steps 2 and 3 until the stack is empty.

```
function DFS (G, v):
    let S be a stack
     S.push ( v )
    while S i s not empty
        v ⟵ S . pop ( )
        if v is not labeled as discovered:
            label v as discovered
            for all edges from v to win G. adjacentEdges ( v ) do
                  S . push (w)
```

BackTracking The Backtracking is a modified DFS of a tree consisting of all of the possible sequences that can be constructed from the problem set. In Backtracking we perform a DFS traversal of this tree until we reach a node that is non-viable or non-promising, at which point we prune the subtree rooted at this node, and continue the DFS traversal of the tree.

```
function DFS(c):
    if reject(P, c ) then return
    if accept (P, c ) then output (P, c )
    S ⟵ first (P, c )
    while S NOT NULL do
            DFS( S )
            S ⟵ first (P, S )
```

The paper that I have choosen is Breadth-First Search, Depth-First Search and Backtracking Depth-First Search Algorithms applied on a Sudoku Puzzle Solver by[4], In which they have solved the Sudoku Puzzle using DFS and backtracking.

Minimum-remaining-values In the basic backtracking search algorithm described above, the next variable chosen to be assigned a value is just the next variable in the list of unassigned variables. However, there are other ways to choose this next variable. We test whether the *minimum-remaining-values* (MRV) heuristic, introduced by [3], improves the performance of backtracking search for solving Sudoku. The idea is to always choose the next variable to be the one that is most likely to cause the current assignment to fail as a solution soon. For example, if the unassigned cell $A1$'s domain is empty while all other unassigned cells' domains are non-empty, the MRV heuristic will choose $A1$ as the next cell to be assigned a value. The failure of the current assignment as a solution to the Sudoku will be immediately detected and time will not be spent expanding the current assignment further before the failure is detected.

3.3 Breadth first search

Breadth-First Search (BFS) is another graph search algorithm introduced by [6], which explores the graph layer by layer. It starts with the root node just like depth-first search (DFS). But unlike DFS, it explores all the neighbour nodes at present layer prior moving to nodes in the next layer. The function BFS() in the following pseudocode illustrates the algorithm.

```
function BFS(problem):
    node ⟵ INSERT(MAKE-NODE(INITIAL-STATE[problem])
    if GOAL-TEST(STATE[node]):
        return node
    frontier ⟵ a FIFO queue with node as only element
    explored ⟵ an empty set
    while frontier is not empty:
        node ⟵ POP (frontier)
        explored ⟵ PUSH (STATE[node])
        for each action in ACTIONS(problem, STATE[node]):
            child ⟵ CHILD-NODE(problem, node, action)
            if STATE[child] is not in explored or frontier:
                if GOAL-TEST(STATE[child]):
                    return child
                frontier ⟵ INSERTALL(child, frontier)
```

The search algorithms work by considering various possible action sequences, with solution being one of them. This algorithm starts with initial state as being the root node of graph. A*node* correspond to a state in the state space of problem graph, with *actions* as its branches.

The first step is to check if root node is the solution state i.e, if problem is already solved by using function GOAL-TEST(). Then, a FIFO(First In First Out) queue *frontier* is defined with just root node (initial state) in it. Now, root node is expanded into child nodes, by finding first empty cell and considering only domain values of that cell. For example, if A2 is the first empty ce ll and have 3 different domain values, then there are three possible action sequences in this layer. This means three child nodes are generated from root node after expansion. After each expansion, *frontier* is updated and every node or state is checked for solution.

Similarly, each node is further expanded into child nodes. The *explored* which is initially defined as an empty set, keep the record of all the explored nodes for solution. The newly generated nodes which matches the nodes in *explored* set are discarded and not added to *frontier.* So, the nodes in the *explored* set are not visited again for solution. When the function GOAL-TEST() returns true, the child node or solution node is returned.

3.4 Algorithm X

Algorithm X is used to solve exact cover problems on Dancing Links. A Sudoku puzzle is first turned into an exact cover problem, then it's solved using Algorithm X.

Dancing Links data structure was described by Donald Knuth in [7] as a two-dimensional, circular, doubly linked list. Each node in the list has four pointers pointing up, down, left and right. Each node is directly part of a column, and implicitly part of a row (there is no need for a "row pointer"). Columns are also nodes. The column nodes are connected to a header node, which is the root of the data structure. Moving into a single direction through a list from a node should eventually return the pointer to the start node (circularity).

In order for us to build a set of dancing links out of a Sudoku puzzle, we first need to determine number of columns we'll need. The number of columns is equal to the total number of constraints in the puzzle. In our chosen Sudoku size, each puzzle has $9 \times 9 = 81$ cells. Each cell is under four constraints Each cell should contain one digit only - 81×1 constraints Each row should contain nine distinct digits - 9×9 constraints Each column should contain nine distinct digits - 9×9 constraints Each 3×3 box should contain nine distinct digits - 9×9 constraints This leaves us with $81 + 81 + 81 + 81 = 324$ constraints that will be transformed into 324 columns. We implemented our code to add the constraints in the order they're presented above. The maximum number of rows for a completely empty Sudoku game is $9 \times 9 \times 9 = 729$ rows for nine possible answers per cell. However, due to the fact that we'll have some cells filled in, we'll end up with fewer rows.

Starting from a header node, columns will be added in order one by one until the 324th column is added. This is done for all puzzles similarly. Then, the specific puzzle grid is traversed one cell at a time, adding one row with four nodes into the links per pre-filled cell, carrying the filled digit, and nine per each empty cell on the grid, with the nine potential cell values that could be assigned.

After each cell in the grid has been visited, populating the dancing links is over.

```
function GENERATE DANCING LINKS (puzzle):
    $NumberOfColumns \longleftarrow nRows \times nColumns \times nConstarints$
    create header cell
    for column in range NumberOfColumns:
        create column
        add column to header
    for each cell in puzzle:
        if cell is empty:
            create rows for all values between 0 to 8
            link created rows to corresponding columns
        else:
            create rows for the value of the cell - 1
            link created rows to corresponding columns
```

Algorithm X starts by choosing a column from our dancing links, in our case, we chose the column with the least number of cells first. We remove or cover that column first. For each row in that column that has a node, we cover the containing column of that cell. When covering a column, it is removed from the list, but the removed column still retains pointers to its original location in the links. Therefore, if columns are covered then uncovered in the opposite order, we would have effectively and efficiently implemented backtracking. If the dancing links are empty by the end of this process, a solution has been reached. However, if covering a column is no longer possible, we start backtracking by uncovering the covered columns in reverse order.

```
function SEARCH ALGORITHM X (Links):
    while Links are not empty:
        choose the column with the least number of elements in it
        cover the chosen column
        for each row in the covered column:
            add row to solution
            cover the column of each cell in the row
        SEARCH ALGORITHM X (Links)
            remove row from solution
            uncover the column of each cell in the row
        uncover the chosen column
```

4 Experimental Results

4.1 Methodology

We test each algorithm on three data sets, each containing fifty Sudoku puzzles. Each of the three datasets has a different difficulty level, namely *easy*, *medium*, and *hard*.[1]

[1] We constructed the easy and medium datasets ourselves. The difficult data set was obtained from [5].

The performance of each algorithm is evaluated according to the following three criteria:

1. Number of nodes expanded in the search tree: a node is defined as a value being assigned to a variable in the search tree, i.e. a number being placed in a cell.
2. Time taken to solve a Sudoku.
3. Maximum memory used while solving a Sudoku.

For each criteria, we record its average value and its maximum value over each dataset for each algorithm.

All algorithms are implemented in Python and all of the relevant code is available in the GitHub repository associated with this paper.[2] All algorithms were run on a whatever computer with whatever specs.

4.2 Results

Tables 1, 2, and 3 display the results of our analysis. In each table, *mean* is the mean value recorded across all puzzles in the dataset while *max* is the maximum value recorded across all puzzles the dataset.

Number of Nodes in Search Tree						
	Easy		Medium		Difficult	
	mean	max	mean	max	mean	max
Backtracking / Depth first search (baseline)	134	3,994	111	2,027	503,831	25,158,597
Breadth first search (BFS)	39,246	270,649	81,816	511,645	-	-
Backtracking with AC-3	14,685	145,169	38,318	346,922	6,510,868	116,038,117
Backtracking with MRV	**53**	**59**	**57**	**59**	**62**	**64**
Algorithm X	85	113	110	228	416	2268

Table 1. Table illustrating number of nodes expanded in search trees for each algorithm

Memory Usage (in MBs)						
	Easy		Medium		Difficult	
	mean	max	mean	max	mean	max
Backtracking / Depth first search (baseline)	0.0146	0.156	0.0095	**0.0181**	0.014	0.1593
Breadth first search (BFS)	96.8672	107.0738	126.9245	164.526	-	-
Backtracking with AC-3	0.1674	0.2829	0.1659	0.2338	0.1659	0.2336
Backtracking with MRV	0.0093	0.1597	0.0109	0.1544	0.0142	0.1548
Algorithm X	**0.0054**	**0.0443**	**0.0054**	0.0448	**0.006**	**0.0448**

Table 2. Table illustrating memory usage for each algorithm

[2] https://github.com/doegan32/CS7IS2-AI-Group-Poject

Time Taken (in seconds)						
	Easy		Medium		Difficult	
	mean	max	mean	max	mean	max
Backtracking / Depth first search (baseline)	**0.0047**	0.1250	**0.0034**	0.0625	15.5403	776.0456
Breadth first search (BFS)	6.2722	43.1273	12.8609	79.6345	-	-
Backtracking with AC-3	0.2718	2.7125	0.643	5.6237	119.9591	2,121.9081
Backtracking with MRV	0.0353	0.0469	0.0387	0.0469	0.0437	0.0625
Algorithm X	0.0056	**0.0313**	0.0066	**0.0313**	**0.0148**	**0.0613**

Table 3. Table illustrating time taken for each algorithm

4.3 Discussion

Number of nodes: Backtracking with MRV was by far the best performing algorithm with respect to this criteria, significantly outperforming the baseline backtracking search across all three datasets. It required only 62 nodes on average for the difficult dataset. In contrast, the baseline required an average of 503,831 nodes for the difficult dataset. Algorithm X also significantly outperformed the baseline, requiring an average of 416 nodes on the difficult dataset. The AC-3 algorithm was able to completely solve some Sudoku puzzles in the easy dataset by itself, i.e. without any searching. Hence, AC-3 had a node count of zero for some puzzles. However, interestingly, when it was unable to fully solve a puzzle, AC-3 significantly diminished the performance of backtracking search. Adding AC-3 as a pre-processing step to backktracking search increased the average node count by almost 13 times for the difficult dataset. This was despite the variable domains being significantly reduced in size by AC-3 before searching began. Breadth first search performed very poorly, failing to solve any puzzles in the difficult dataset before running out of memory.

The difficulty of the Sudoku had a much smaller impact on the performance of backtracking with MRV than on the performance of all the other algorithms. Backtracking with MRV's average node count on the difficult dataset was only 1.17 times its average node count on the easy dataset. The corresponding figure for the baseline was 3,760. Similarly, backtracking with MRV's maximum node count on the difficult data set was 64, only 2 bigger than its average of 62. In contrast, the baseline's maximum node count on the difficult dataset was almost 50 times higher than its average.

Memory usage: Algorithm X was the best performing algorithm with respect to this criteria. Its average memory usage was less than half that of the baseline on the easy and difficult dataset. The baseline and backtracking with MRV performed similarly on the difficult dataset. The baseline outperformed backtracking with MRV on the medium dataset while the opposite was the case on the easy dataset. Once again, the AC-3 algorithm diminished the performance of the backtracking search algorithm. Unsurprisingly, breadth first search was the worst performing algorithm by a significant margin.

The difficulty of the puzzles only had a significant impact on memory usage for breadth first search. The baseline and backtracking with AC-3 actually had a lower average maximum memory usage on the difficult dataset than on the easy dataset, while Algorithm X's average maximum memory usage on the difficult dataset was only 1.1 times the corrsponding figure for the easy dataset.

Time taken: In terms of average time per puzzle, no algorithm outperformed the baseline on the easy and medium datasets. However, the baseline's performance was significantly worse on the difficult dataset and it was easily outperformed by backtracking with MRV and Algorithm X. Once again, breadth first search and backtracking with AC-3 were the worst performing algorithms.

Backtracking with MRV's performance was the least effected by the difficulty of the puzzles. Its average time on the difficult dataset was 1.2 times its average time on the easy dataset. Its maxmimum time on the difficult dataset was 1.4 times its average time on the difficult dataset. The corresponding figures for Algorithm X are 2.6 and 4.1. For the baseline, the figures are 3,306 and 49.9.

Taking all criteria into account and viewing performance on the difficult dataset as the most important, our analysis suggests that the best Sudoku solving algorithms are backtracking search with the minimum-remaining-values heuristic and Algorithm X. While backtracking with MRV easily wins in terms of node count, Algorithm X wins in terms of memory usage and time.

5 Conclusions

In this paper, we performed a survey of different algorithms that can be used to solve Sudoku. We analysed the performance of these algorithms on three datasets and according to three criteria - number of nodes in the search tree, time taken, and memory usage. Our results indicate that, of the algorithms tested, backtracking search with the minimum-remaining-values heuristic and Algorithm X are the best algorithms for solving Sudoku.

The only purely inference algorithm that we looked at was AC-3. This was only able to solve the simplest puzzles and surprisingly, it diminished the performance of backtracking search when used as a pre-processing step. It would be interesting to find out why this was the case. Another area for future work would be to look at the performance of other inference algorithms such as PC-3 which achieves path-consistency, a stronger condition than the arc-consistency achieved by AC-3.

References

1. Norvig, P., Russell, S.: Artificial Intelligence A Modern Approach Third Edition. Pearson Education Limited (2016)
2. Mackworth, A.K.,Consistency in Networks of Relations. In: Artificial Intelligence 1977 8(1):99-118. doi:10.1016/0004-3702(77)90007-8

3. Bitner, J.R., Reingold, E.M.: Backtrack Programming Techniques. In: Communications of the ACM, 18(11), 651-656. doi:10.1145/361219.361224
4. Breadth-First Search, Depth-First Search and Backtracking Depth-First Search Algorithms by Fernando Rodrigues Junior College of Science and Engineering University of Minnesota - Twin Cities
5. Solving Every Sudoku Puzzle. https://norvig.com/sudoku.html
6. Moore, Edward F. (1959). "The shortest path through a maze". Proceedings of the International Symposium on the Theory of Switching. Harvard University Press. pp. 285–292. As cited by Cormen, Leiserson, Rivest, and Stein
7. Donald E. Knuth: Dancing links
`https://arxiv.org/abs/cs/0011047`
8. Richard M. Karp. Reducibility among combinatorial problems. Complexity of computer computations, Proceedings of a Symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Center, Yorktown Heights, New York, edited by Raymond E. Miller and James W. Thatcher, Plenum Press, New York and London 1972, pp. 85–103