Shite Sudoku Solving Survey

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. Motivation.
What we do.
We test these algorithms on Sudoku puzzles with three different levels of difficulty - easy, medium, and hard. We analyse the algorithms' performance with respect to a number of 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 tested, we find that the best method for solving a Sudoku puzzle is to treat it as a constraint satisfaction problem and to use the backtracking search algorithm along with the minimum-remaining values heuristic.

Keywords: Sudoku, constraint-satisfaction problem, depth-first search, backtracking search, heuristics

1 Introduction

Motivation for work.

Describe what a Sudoku puzzle is.

Brief description of what we did.

Brief description of results.

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 of the algorithms, 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

In this section you will discuss possible approaches to solve the problem you are addressing, **justifying your choice** of the 3 you have selected to evaluate. Also, briefly introduce the approaches you are evaluating with a specific emphasis on differences and similarities to the proposed approach(es).

3 Problem Definition and Algorithm

A Sudoku puzzle can be treated as 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 ⟵ false
    for each x in the domain of X_i:
        if there is no y in the domain of X_j such that x ≠ y:
            remove x from the domain of X_i
            revised ⟵ true
    return revised

function AC-3(csp):
    queue ⟵ set of all arcs in csp
    while queue is not empty:
        (X_i, X_j) ⟵ 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 algorithm (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.

3.2 Backtracking search / Depth first search

3.3 Heuristics

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 the whether the *minimum-remaining-values*, or 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.

Least-constraining-value Similarly, once the next unassigned Sudoku cell has been chosen, the backtracking search algorithm needs to decide on order in which to check the values in the cell's domain. Basic backtracking search will just choose the next value in the chosen cell's domain. We test the whether the *least-constraining-value* (LCV) heuristic improves the performance of backtracking search for solving Sudoku. For example, suppose the only unassigned cells are the current chosen cell $A1$ with domain {5, 7} and the cell $A2$ with domain {5}. Basic backtracking search will first try setting $A1$'s value equal to 5. This will eliminate the last legal value from $A2$'s domain and the algorithm will have to backtrack to set $A1$'s value to 7. In contrast, the LCV heuristic will first try setting $A1$'s value to 7.

Maintaining arc consistency

3.4 Breadth first search

Breadth-First Search (BFS) is another graph search algorithm introduced by [5], 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 at 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.

4 Experimental Results

4.1 Methodology

We tested each algorithm on three data sets, each containing fifty Sudoku puzzles. Each data set contained puzzles with different levels of difficulty, namely, *easy*, *medium*, and *hard*. Need to find out what determines the level of difficulty?

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

1. Number of nodes
2. Average time[1]
3. Average maximum memory used

For the first criterion, a node is defined as . .. . . For each criteria, we record its average value and its maximum value over each data set for each algorithm.

All algorithms were 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 data set while *max* is the maximum value recorded across all puzzles the data set.

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	
Backtracking with MRV	**53**	**59**	**57**	**59**	**62**	**64**
Backtracking with LCV	134	3,994	111	2,027	503,831	25,158,597
Backtracking with MAC	000	000	000	000	000	000

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

present the results of the experimental evaluation. Graphical data and tables are two common ways to present the results. Also, a comparison with a baseline should be provided.

[1] Given the situation with the Covid-19 pandemic we had trouble running all pieces of code on the same machine and so figures relating to time may not be an accurate reflection

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

Memory Usage (in MBs)						
	Easy		Medium		Difficult	
	mean	max	mean	max	mean	max
Backtracking / Depth first search (baseline)	000	000	000	000	000	000
Breadth first search (BFS)	96.8672	107.0738	126.9245	164.526	-	-
Backtracking with AC-3	000	000	000	000	000	000
Backtracking with MRV	000	000	000	000	000	000
Backtracking with LCV	154	000	448	000	30026	000
Backtracking with MAC	000	000	000	000	000	000

Table 2. Table illustrating memory usage for each algorithm

Time Taken (in seconds)						
	Easy		Medium		Difficult	
	mean	max	mean	max	mean	max
Backtracking / Depth first search (baseline)	000	000	000	000	000	000
Breadth first search (BFS)	6.2722	43.1273	12.8609	79.6345	-	-
Backtracking with AC-3	000	000	000	000	000	000
Backtracking with MRV	000	000	000	000	000	000
Backtracking with LCV	154	000	448	000	30026	000
Backtracking with MAC	000	000	000	000	000	000

Table 3. Table illustrating time taken for each algorithm

4.3 Discussion

discuss the implication of the results of the proposed algorithms/models. What are the weakness/strengths of the method(s) compared with the other methods/baseline?

5 Conclusions

Provide a final discussion of the main results and conclusions of the report. Comment on the lesson learnt and possible improvements.

A standard and well formatted bibliography of papers cited in the report. For example:

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. Solving Every Sudoku Puzzle. https://norvig.com/sudoku.html

5. 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