Evaluating a CSP Solver for Sudoku

Group 28: Harsh Goel, Ritwik Jha, Chan Tjun Chuang Alex, and Shradheya Thakre

1 Problem Specification

The Sudoku puzzle is a 9×9 combinatorial number puzzle, further divided in nine 3×3 subgrids. The puzzle begins partially filled. The goal is to fill the remaining cells in the puzzle with numbers between 1 to 9, where each column, row, and 3×3 grid does not repeat the same number twice. This paper will explore the implementation of Sudoku as a CSP with the above 3 constraints.

1.1 Sudoku problem as a CSP

Sudoku can be set up as a CSP problem with 81 unassigned variables whose domains are integers $\{1,2...9\}$. For a $n^2 \times n^2$ puzzle with domain values from 1 to n^2 we have $3 \times n^2$ all different constraints for each row, column and sub-grid. We have converted this into Binary *alldiff* constraints which are set for each tuple of variables along a given row, column and the nine 3×3 subgrids. Thus, there are 20 *alldiff* constraints between a given cell/variable in the Sudoku and the corresponding cells/variables in its column or row or subgrid as we have 8 constraints for a given cell with the corresponding cells in the row and column and 4 constraints with the cells in the subgrid (we do not double count those in the same row or column). During the initialisation phase of a Sudoku puzzle, the domain of each assigned variable in the partially filled Sudoku is reduced to a single integer and domains of the remaining unassigned cells are subsequently suitably adjusted using forward checking.

2 CSP Algorithm and Heuristics

We implement a backtracking search algorithm for solving Sudoku. The backtracking search algorithm is a depth first search (DFS) strategy that builds candidate partial assignments for a given problem by selecting which variables (cells) to assign and which values to assign to them and prunes (backtracks) a given partial assignment if the given assignment is inferred to not fulfill the constraints of the problem. The search space of a Sudoku with V empty unassigned variables is 9^V.

We use the Minimum Remaining Values(MRV) heuristic to select a variable to assign a value as this helps reduce the branching in backtracking. For a given valid partial assignment, the MRV heuristic selects a variable with fewest legal values in its domain. Thus, this increases the probability of finding a valid assignment for that variable, and minimizes branching in the DFS tree. Furthermore, [2] shows that modifying forward checking with the Minimum Remaining Value (MRV) heuristic theoretically and empirically leads to better performance in terms of both number of backtracks and runtime as compared to the Degree heuristic. Ties are broken by selecting the variable with the smallest index.

We propose two variants for assigning values to selected variables. Our first heuristic uses the smallest value in the domain for a selected variable. The second heuristic is called Least Constraining Values (LCV) heuristic. It is based on ordering the domain values which are least constraining and cause fewest domain reductions for variables in the neighbourhood (in the constraint graph) of the selected variable.

We have used two inference algorithms for evaluating the feasibility of any given partial assignment: Forward Checking and Maintaining Arc Consistency.

Forward Checking During forward checking, the domain of a variable constrained along the row or column or a 3×3 subgrid of the assigned variable is reduced during a single iteration of the backtracking search. Each domain reduction recursively checks the neighborhood of the variable whose domain is reduced. If any variable is found to have its domain reduced to 0, the preceding assignment is pruned. This ensures that the domains of each variable consist of only valid assignments.

Maintaining Arc Consistency (MAC) MAC works by choosing a value at the highest variable in the DFS tree with non-singular domain after AC-3 has terminated. Constraint propagation is then repeated with this selected value. This is repeated iteratively proceeding down the DFS tree, until an empty domain is discovered, allowing for better pruning of domains. A primary inefficiency with MRV and forward checking is that after assigning a compatible value, all other variables whose domains have reduced to a single value are not checked for consistency. Moreover, for any two variables which have more than two remaining values in the domain, the *alldiff* constraint would always be satisfied. Thus, we use a modified MAC which adds only those variables which have single-value domains remaining to the queue of variables and checks whether the constraints are satisfied by propagating these assignments.

For the project, MRV was combined with forward checking, Maintaining Arc Consistency (MAC) and Least Constraining Values (LCV) heuristics in four variants to find the best configuration. Our primary considerations for analysing these 4 variants were total solution search time and number of illegal assignments (number of backtracks). The primary trade off was whether the additional overhead for evaluating the heuristics is compensated by the reduction in number of illegal assignments (number of backtracks), and thus, reduces overall time for finding a solution.

3 Experiments and Evaluation

3.1 Experimental Setup

The experiments ran backtracking with Maintaining Arc Consistency (MAC) and Forward Checking (FC), and with and without LCV—for a total of four variants. We wanted to check if the solving time is related to difficulty and if it might be the case that a heuristic might not be the best for certain problems. The database obtained from [1] was used as it orders test cases based on increasing difficulty. For each test case, we evaluate the number of backtracks and average total solution search time based on the average of 100 runtimes for each test case. We evaluate the trade-off between time taken to compute certain heuristics versus how well the heuristic reduces the search space of the DFS tree. The experimental results are shown in Table 1 and are illustrated in Fig 1. These experiments were run on a 16 GB 3.4 GHz Intel i7 Ubuntu Machine.

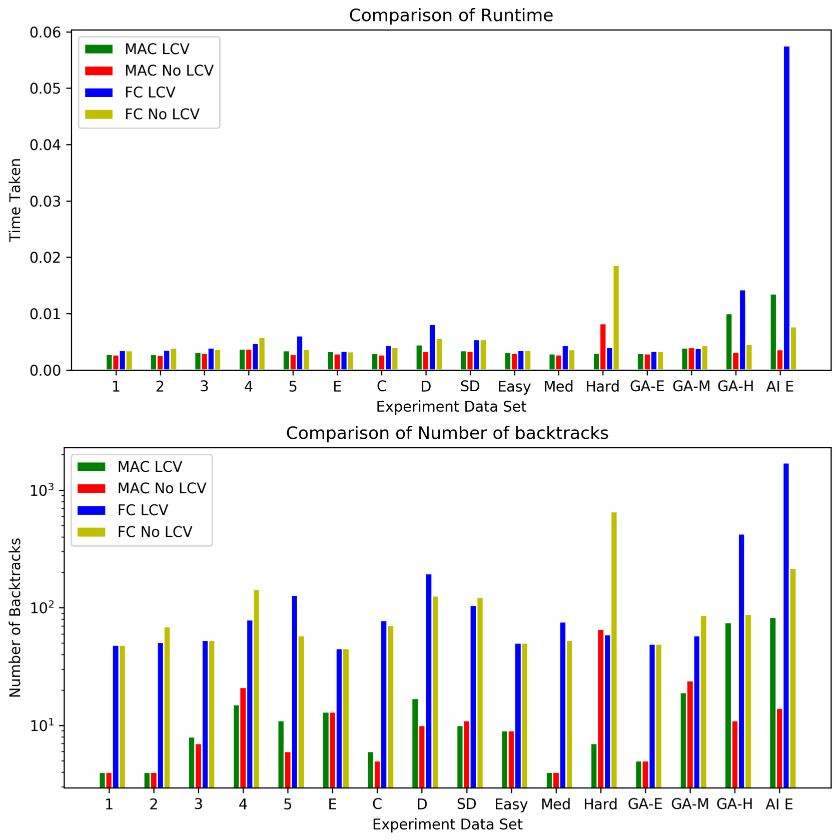

Fig. 1: Results of running Sudoku solver using MAC and FC algorithms with and without the LSV heuristic. Comparison of runtime based on 100 runs (top). Comparison of number of backtracks (bottom).

3.2 Results and Analysis

Based on the results of the experiment, it was observed that the best algorithm configuration was using MAC without LCV heuristic. We first analyse the benefits of using LCV vs no LCV for a given inference heuristic and then compare which algorithm had the best performance in the average test case.

The use of FC with LCV resulted in fewer number of backtracks across majority of test cases as compared to FC without LCV. Cases with greater backtracks for FC with LCV were 5, C, D, Med, GA-H and AI-E. The runtimes did not always correspond to fewer backtracks. Cases 3, 5, E, C, D, Med, GA-H and AI E had longer runtimes for FC with LCV. This was likely due to the LCV overhead outweighing the savings in time from fewer backtracks. The use of LCV with MAC returned mixed results. Test cases 3, 5, C, SD, Easy, Med and GA-E had fewer backtracks without LCV, while the other test cases had fewer backtracks with LCV. However, a larger number of backtracks did not always cause a longer runtime. Hence, the overhead due to LCV may not justify its use in the 9×9 Sudoku.

LCV inefficiencies with both MAC and FC are observed particularly amongst the ‘harder’ puzzles where each cell has a relatively large domain at the start. By virtue of MAC, each assignment is propagated and checked for consistency if the domains of neighbouring cells have been reduced to a single digit. However, using LCV prevents this propagation from taking place. LCV may hence be a worse strategy in eliminating ‘dead-end’ explorations early when used in conjunction with MAC. This seems to be the case in the two ‘hardest’ puzzle among our test cases: GA-H and AI E, which had 75 and 83 backtracks when solved using MAC/FC with LCV compared to 11 and 14 backtracks when using MAC without LCV.

Overall, our solution when run on the experimental test cases was able to complete and give the right results within 10ms. The number of backtracks also substantially decreased with our efficient heuristics like MAC and MRV. We observed that in general for difficult puzzles there is an increase in time/backtracks but this is not a hard and fast rule. Certain heuristics are more efficient for certain problem states. Our solution which is a combination of MAC and MRV is optimal since it ensures to find the solution node in the DFS tree at the quickest as seen above through heuristics and experimentally.

4 Conclusion

In this project, we implemented a Sudoku solver using MAC and FC algorithms. Variable (cell) selection for value assignment was performed using the MRV heuristic in both algorithms. The two algorithms were tested with and without the LCV heuristic. LCV reduced the number of backtracks with FC for all test cases, but did not reduce the number of backtracks in all cases with MAC. According to the results of the tests, the best configuration which had the fewest number of backtracks in the average case and balanced trade off between runtime overhead and number of backtracks was MRV without the LCV domain ordering heuristic and with the MAC inference heuristic.

References

1. T. Mantere and J. Koljonen, “Sudoku page,” University of Vaasa, 17-Jun-2008. http://lipas.uwasa.fi/ timan/Sudoku/ (accessed May 07, 2020).
2. Bacchus, Fahiem, and Paul Van Run. ”Dynamic variable ordering in CSPs.” In International Conference on Principles and Practice of Constraint Programming, pp. 258-275. Springer, Berlin, Heidelberg, 1995.

Appendix A

Table 1: Comparison of MAC and FC constraint propagation method, with and without the LCV heuristic. All runtime are in seconds. Test cases are taken from [1] where more details about these puzzles can be found.

	Maintaining Arc Consistency				Forward Checking			
	LCV		No LCV		LCV		No LCV	
Testset	Backtracks	Runtime	Backtracks	Runtime	Backtracks	Runtime	Backtracks	Runtime
1	4	0.00279	4	0.00269	48	0.00351	48	0.00341
2	4	0.00277	4	0.00265	51	0.00354	69	0.00393
3	8	0.00315	7	0.00293	53	0.00394	53	0.00365
4	15	0.00371	21	0.00374	79	0.00473	143	0.00584
5	11	0.00343	6	0.00277	128	0.00605	58	0.00368
E	13	0.00333	13	0.00288	45	0.00335	45	0.00322
C	6	0.00295	5	0.00266	78	0.00436	71	0.00403
D	17	0.00448	10	0.00332	194	0.00810	126	0.00565
SD	10	0.00344	11	0.00338	105	0.00542	123	0.00539
Easy	9	0.00313	9	0.00299	50	0.00351	50	0.00351
Medium	4	0.00288	4	0.00267	76	0.00436	53	0.00361
Hard	7	0.00301	66	0.00820	59	0.00401	654	0.01859
GA-E	5	0.00294	5	0.00285	49	0.00337	49	0.00332
GA-M	19	0.00393	24	0.00400	58	0.00387	86	0.00436
GA-H	75	0.01001	11	0.00320	425	0.01424	88	0.00456
AI E	83	0.01350	14	0.00361	1702	0.05751	217	0.00766