Project 2: Constraint Satisfaction Problems

Jordan White

April 6, 2021

1 Project Overview

1.1 Constraint Satisfaction Problems

Constraint Satisfaction Problems (CSPs) are search problems defined in terms of three sets: the variables V, the domain(s) D (typically each variable has its own domain), and the constraints C. Each state represents a different set of assignments for the variables in V. A goal state is reached when each variable in V is assigned a value from its domain that does not violate any constraints in C.

The domains in a CSP can be finite or infinite, as well as discrete or continuous. The constraints in a CSP can be unary (constraints involving one variable), binary (involving two variables), or higher order (involving three or more variables).

1.2 Problem Domains

CSPs are a useful concept because of the wide scope of problems that can be classified as CSPs (n-queens, Sudoku, map-coloring, class scheduling, etc.), and because of the general purpose algorithms that can be applied to every CSP.

1.2.1 Map Coloring Problem

The first type of CSP I will be solving in this project is the map coloring problem. This is a type of CSP in which each variable represents a region on a two-dimensional map. The values within the domain of each region represent the possible colors that region could be colored. Each region's constraints represent the other regions that share a border with it. The task of the map coloring problem is to color each region such that it is a different color from all of its bordering regions. As it turns out, every possible map coloring problem can be solved using only four colors in the entire map, so the domain of each region will start with four colors.

1.2.2 Sudoku

The second type of CSP I will be solving is the game of Sudoku. In Sudoku, there is a nine by nine grid of boxes, some of which start out containing values numbering one through nine. The task of Sudoku is to fill each box with a

number such that the same number does not exist within that box's row, column, or surrounding 3x3 grid. The variables in this CSP each represent one of the 81 boxes, the domain of each box will be the possible values that box could contain, and the constraints on each box will be all the other boxes within its row, column, and/or 3x3 grid.

1.3 Project Requirements

In this project, I was tasked with programming CSP solver in Python and analyzing the performance of three types of CSP search algorithms: Depth First Search (DFS) with backtracking-only, DFS with backtracking and forward checking, and DFS with backtracking and the Arc Consistency Algorithm #3 (AC3). I was also tasked with analyzing the effect on performance of three variable ordering heuristics: random variable ordering, minimum remaining value (MRV) ordering, and MRV with degree ordering.

In this report, I will analyze the relative advantages and disadvantages of using these different algorithms and heuristics to solve the map coloring and Sudoku constraint satisfaction problems.

2 Program Implementation

My depth-first search algorithm (shown in Figure 1), written in Python 3, is a general purpose algorithm which can solve (or show to be unsolvable) any finite, discrete CSP containing only binary-type constraints. My algorithm also contains the ability to selectively run one of two different domain-restricting algorithms (Forward Checking or AC3) if desired, as well as any of three different variable selection heuristics (random, MRV, and MRV with degree ordering).

2.1 Data Structures

I employ three primary data structures to represent a CSP. The first is a list called 'Assignments' of length n, where n is the number of variables in the CSP. For example, if my algorithm is attempting to color a map with 50 nodes, 'Assignments' contains 50 elements. At any given time during execution, the value at the 0th index represents the current value of the first variable, the value at the 1st index represents the current value of the second

variable, on to the value of the (n-1)th index, representing the current value of the last variable.

The second data structure I employ in my program is a list of lists called 'Domains.' At each index i of the mega-list, the inner-list represents the domain-values for variable i. Depending on the type of algorithm running, the domains of each variable may be altered to decrease total search time needed to solve the CSP.

The third data structure I employ is a list of lists called 'Constraints.' Each index i of this mega-list contains an inner-list of values, $v_1 ... v_j$. Each value v_k within the inner-list represents a binary 'not-equals' constraint between variable i and variable v_k. In the case of map-coloring, this constraint represents an edge between i and v_k; In the case of Sudoku, the constraint represents the fact that i and v_k share the same column, row, or grid. This Constraints mega-list and all of the sub-lists are immutable, because CSP constraints do not change at any time during a state search.

2.2 Depth First Search with Backtracking

DFS with backtracking-only is the simplest algorithm in this project. In my program, the entire CSP (Assignments, Domains, and Constraints) is passed to a recursive function in which some variable v is selected, and a value from the domain of v is assigned to v. If the Constraints on v permit this assignment, the function will recurse, selecting a new variable for assignment. If all assignments within the lower levels of recursion succeed, then the function returns 'True'. If the assignments within the lower levels of recursion fail, then the variable v is reassigned a different value from its domain and recursion is retried. If all values within the domain of variable v have been tried and failed, then the function returns 'False'.

This program will always find some set of assignments that satisfies the constraints, assuming a goal state exists. The performance of DFS with backtracking-only is analyzed in section 3.1.1.

2.3 DFS with Backtracking and Forward Checking

The second type of algorithm in my project is DFS with backtracking and forward checking. This is a variation of simple DFS with backtracking-only. To add forward checking, I added functionality to the original DFS algorithm such that, each time a value is assigned to a variable v, v's domain is reduced

to contain only that value, and the domains of the neighbors of v are altered to exclude that value. The algorithm then checks if any of the altered domains are empty, and if so, the algorithm fails that particular assignment to v and resets the domains.

The benefit of this algorithm is that, every time an assignment is made to a variable, the domains of other variables may be reduced. This means the DFS algorithm has fewer possible values to try when assigning future variables, which means a reduction in the average branching factor for each variable, and a corresponding reduction in the execution time.

The performance of DFS with backtracking and Forward Checking is analyzed in section 3.1.2.

2.4 DFS with Backtracking and Arc Consistency

The final type of algorithm within my project is DFS with backtracking and the Arc Consistency Algorithm #3 (AC3). Just like with Forward Checking, AC3 modifies the domains of variables in order to decrease execution time. Unlike Forward Checking, AC3 is more thorough in eliminating inconsistencies, because it restricts as many domains as is logically possible in the entire CSP, instead of restricting only the domains of the neighboring variables of a newly assigned variable. This makes AC3 theoretically faster than Forward Checking.

The way AC3 accomplishes this task is by creating a queue of arcs (constraints), typically all of the arcs in the CSP. AC3 then pops the arcs off the queue one at a time, and deletes any inconsistent values from the domain of the primary variable in the arc. If a value was deleted from a variable v's domain, and the length of v's domain is now 0, AC3 returns 'False', indicating that an inconsistency was found in the CSP. If v's domain still has values however, AC3 adds all of v's arcs to the queue, and repeats the process until the queue is empty. Therefore, AC3 has the potential to reduce the domain of every variable in the CSP in one run.

Instead of standard AC3, my program utilizes the 'Maintaining Arc Consistency' Algorithm (MAC3), in which, after a variable v is assigned a value, all of its arcs are added to the queue, and then AC3 is run on those arcs.

The performance of DFS with backtracking and MAC3 is analyzed in section 3.1.3.

2.5 Variable Ordering Heuristics

To assign values to every variable, my program needs some way to choose the order in which variables are assigned. The first and most obvious method is sequential variable selection, in which variable 1 is assigned first, then variable 2, then 3, on to the final one. Listed below are three different variable ordering heuristics I implemented within my program.

2.5.1 Random

When my DFS algorithm uses the random variable selection heuristic, it uses a pseudo-random algorithm to select the next variable to assign. The selection pool is all unassigned variables (all variables with a value of 0 in Assignments). This causes performance of my DFS search traversal to vary widely, depending on whether the order of variable selection is conducive to early search-tree pruning or not.

2.5.2 Minimum Remaining Value

When my DFS algorithm uses the MRV variable selection heuristic, it selects the variable with the fewest legal values left inside its domain. If there is a tie for the variable with the fewest legal values, the first of the tied variables is selected.

The intention of this heuristic is to reduce execution time. The manner in which MRV improves performance is by selecting variables sooner which are more likely to cause a failure; Thus, MRV can be said to prune the search tree.

2.5.3 Minimum Remaining Value with Degree Ordering

When my DFS algorithm uses the MRV-degree variable selection heuristic, it selects the variable with the fewest legal values left inside its domain. If there is a tie for fewest legal values, it selects from among the tied variables the variable with the greatest number of unassigned neighbors. If there is another tie, it selects the first variable from among these.

This heuristic also reduces execution time in the same way MRV does, but MRV with degree ordering has the additional advantage that it reduces the branching factor of future choices by selecting the variable first that has the greatest number of branches/constraints.

3 Results

In this section, I will most heavily analyze my algorithm's performance on the map-coloring problem, specifically with regards problem instances of 50 regions/nodes. This will allow me to compare the relative performance of each algorithm and heuristic when running on exactly the same problem domain. I will also provide some data for the 10 and 100 node cases, to analyze the effect the state space size has on the complexity of depth first search traversal.

Afterwords, I will provide statistics for the performance of the CSP solver on the Sudoku domain for problem instances of size = 20 empty squares, as well as for size = 35 empty squares. I will then briefly analyze the similarities/differences between my algorithm's performance on the map-coloring domain versus its performance on the Sudoku domain.

3.1 Results on the Map Coloring Problem

The solution to the sample input given to us at the start of the project is shown in Figure 3. My program found this solution using the forward-checking algorithm with the MRV heuristic.

The average execution time of each algorithm and heuristic, when run on a map coloring problem of size = 50 regions, is shown in Figure 2. Each value was calculated by averaging execution time over 12 different problem instances of 50 nodes each. Notice that the time data for the 'random' heuristic is given as a range, because of the high variance of its performance.

Example terminal output from my program is shown in Figure 4 to demonstrate how my data was collected.

Described below is a more detailed report of each algorithm's performance on the map coloring problem.

3.1.1 DFS with Backtracking

As shown in Figure 2, the average performance of DFS with backtracking-only on map coloring instances of 50 nodes, using the random variable selection heuristic, was between 0.5 and 60 seconds. After switching to the MRV variable selection heuristic, performance averaged 3.6 seconds. For the MRV with degree ordering heuristic, performance was 3.8 seconds.

On only 10 nodes, execution averaged 0.00007 seconds using the sequential heuristic. Execution time on 100 nodes exceeded 5 minutes no matter which variable heuristic was used.

3.1.2 DFS with Backtracking and Forward Checking

As shown in Figure 2, the average performance of DFS with Backtracking and Forward Checking on map coloring instances of 50 nodes, using the random variable selection heuristic, was between 0.9 and 30 seconds. After switching to the MRV variable selection heuristic, performance improved drastically to 0.0011 seconds. For the MRV with degree ordering heuristic, performance was 0.0019 seconds.

Using the MRV heuristic, average performance was 0.00009 seconds on 10 nodes and 0.0031 seconds on 100 nodes.

3.1.3 DFS with Backtracking and MAC3

As shown in Figure 2, the average performance of DFS with Backtracking and MAC3 on map coloring instances of 50 nodes, using the random variable selection heuristic, was between 0.005 and 0.5 seconds. After switching to the MRV variable selection heuristic, performance drastically improved to 0.0025 seconds. For the MRV with degree ordering heuristic, performance averaged 0.0035 seconds.

Using the MRV heuristic, performance averaged 0.00017 seconds on 10 nodes and 0.006 seconds on 100 nodes.

3.2 Results on Sudoku

The time, in seconds, for each algorithm/heuristic to solve a Sudoku table with 20 empty squares is shown in Figure 5. The time for each algorithm/heuristic to solve a Sudoku table with 35 empty squares is shown in Figure 6. These Sudoku performance statistics were averaged over exactly 100 problem instances.

In Figure 7, you can see an example of my program's output after solving a Sudoku CSP. This particular output is the solution to the sample Sudoku input given to us at the start of the project.

3.3 Analysis of Results

3.3.1 Performance Comparisons

The performance of my algorithms/heuristics on Sudoku problems is comparable to their performance on map coloring problems. The only stark difference is in the effect of the domain reducing heuristics (MRV and MRV-degree), which will be analyzed in section 3.3.2.

Evidently, problems containing fewer variables took much less time to solve. For example, an uncolored map with 10 regions is solved by DFS with backtracking-only almost instantly, while a map with 100 regions may take the same algorithm over 5 minutes to solve. This makes sense, since the number of complete assignments in a CSP rises exponentially with the number of variables.

The fastest algorithm used in conjunction with the sequential and random variable ordering heuristics is, by far, AC3. This makes sense, because AC3 performs the most pruning out of all three algorithms. When the variable ordering heuristic is MRV or MRV-degree, however, forward checking performs the fastest by a slight margin.

The fastest variable heuristic used in conjunction with DFS with backtracking-only is the sequential heuristic. The random heuristic sometimes performs better, but often results in extremely long searches. The MRV and MRV-degree heuristics actually make DFS with backtracking-only slower. This is because both of those heuristics rely on domain reductions to improve speed, but normal DFS with backtracking-only does not reduce the domains of any variables.

DFS with forward checking performs rather poorly with sequential and random variable ordering, but it becomes much faster with MRV ordering. MRV-degree also improves its speed drastically, but not quite as much as base MRV.

DFS with AC3 also performs better with MRV than any other heuristic; Its performance with MRV-degree follows close behind.

The fastest of my algorithm-heuristic combinations is forward-checking with the minimum remaining value heuristic.

3.3.2 Seemingly Analogous Data

There is some seemingly analogous data that requires explanation. For example, the performance of any algorithm using the random heuristic varies

widely; This is probably because the variable order that the random heuristic chooses is sometimes 'lucky', leading to tree pruning, and sometimes 'unlucky', leading to traversals of the majority of the state space.

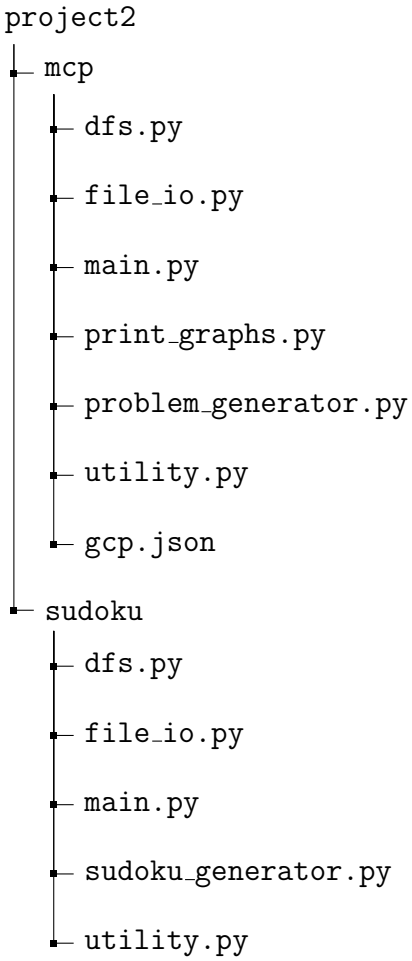
Additionally, MAC3 should theoretically be faster than forward-checking, even with the MRV/MRV-degree heuristics. In practice, forward checking is slightly faster. This may be because my particular implementation of MAC3, after each domain reduction, requires an additional O(N) traversal of the constraints of some variable in order to determine which of its neighbors are unassigned. Since MAC3 often performs several domain reductions at each level of recursion, these additional traversals add up, causing MAC3 to be slightly slower than base forward-checking. I may be able to solve this problem by updating the Constraints data structure to contain information about which neighbors are still unassigned; This will render an additional traversal unnecessary. However, this solution would cause my CSP solver to be less general and more domain dependent.

Also, MRV with degree ordering should be faster than base MRV. In practice, however, the fastest of the variable selection heuristics was base MRV by a small amount. One possible reason could be the additional traversals that MRV-degree carries out to determine which variable has the highest degree. This problem could be circumvented by creating a data structure that is global to all levels of recursion and contains all degree information; This would render additional traversals unnecessary.

Another strange result I discovered is that both the MRV and MRV-degree heuristics added to the average runtime of the CSP solver on all Sudoku problems. I believe the reason for this is because the Sudoku board generator creates Sudoku boards with many possible solutions instead of just one. This fact reduces the necessity of domain reductions for finding an answer, because whatever values are assigned at the beginning of the state search will probably be accepted, and pruning thus becomes less necessary. Another possible reason is because the size of the variable domains in Sudoku CSPs is much larger (nine) than it is for map coloring CSPs (four). The same is true for the number of constraints per variable in Sudoku CSPs when compared to that of map-coloring CSPs. This means the domain and neighbor traversals that MRV and MRV-degree require take much more time for Sudoku CSPs than for map-coloring CSPs.

3.4 Running My Program

The file structure for my program is as shown below:



The 'dfs.py' file in both folders is exactly the same. To run my program and see the output, simply navigate to either the 'mcp' or 'sudoku' folder in the command line and run: *python main.py*. When solving Sudoku CSPs, my program will create a new Sudoku board with 20 empty squares, solve it, then print the unsolved and solved boards in ASCII-art format to the terminal. If you wish to create a Sudoku problem with a different number of empty squares, or to change the algorithm/heuristic running, edit the driver code at the bottom of main.py. When solving map-coloring CSPs, my program will solve the problem instance 'gcp.json' in the 'mcp' folder. To create a new problem instance, use the problem_generator.py file.

4 Figures

```
def backtrack(assignments, domains, constraints, variable, var_selection_method, inference_method):
    if(not 0 in assignments):
        return True
    for poss_val in domains[variable]:
        if(_val_consistent(poss_val, assignments, constraints[variable])):

            assignments[variable] = poss_val

            changes = _alter_domain(variable, poss_val, assignments, domains, constraints,inference_method)

            if(not changes == [-1]):#if inferences succeeds
                next_var = _choose_next_var(assignments, domains, constraints, var_selection_method)

                result = backtrack(assignments, domains, constraints, next_var, var_selection_method, inference_method)

                if(result):
                    return True

                _undo_changes(changes, domains)

            assignments[variable] = 0

    return False
```

Figure 1: My general purpose backtracking algorithm.

Map Coloring, 50 nodes				
	Sequential	Random	MRV	MRV with degree
DFS with backtracking	1.8 s	0.5 - 60 s	3.6 s	3.8 s
DFS with forward checking	5.4 s	0.9-30 s	0.0011 s	0.0019 s
DFS with AC3	0.008 s	0.005-0.5s	0.0025 s	0.0035 s

Figure 2: Run-time statistics (in seconds) for my CSP solver on map-coloring problem instances of size = 50 nodes.

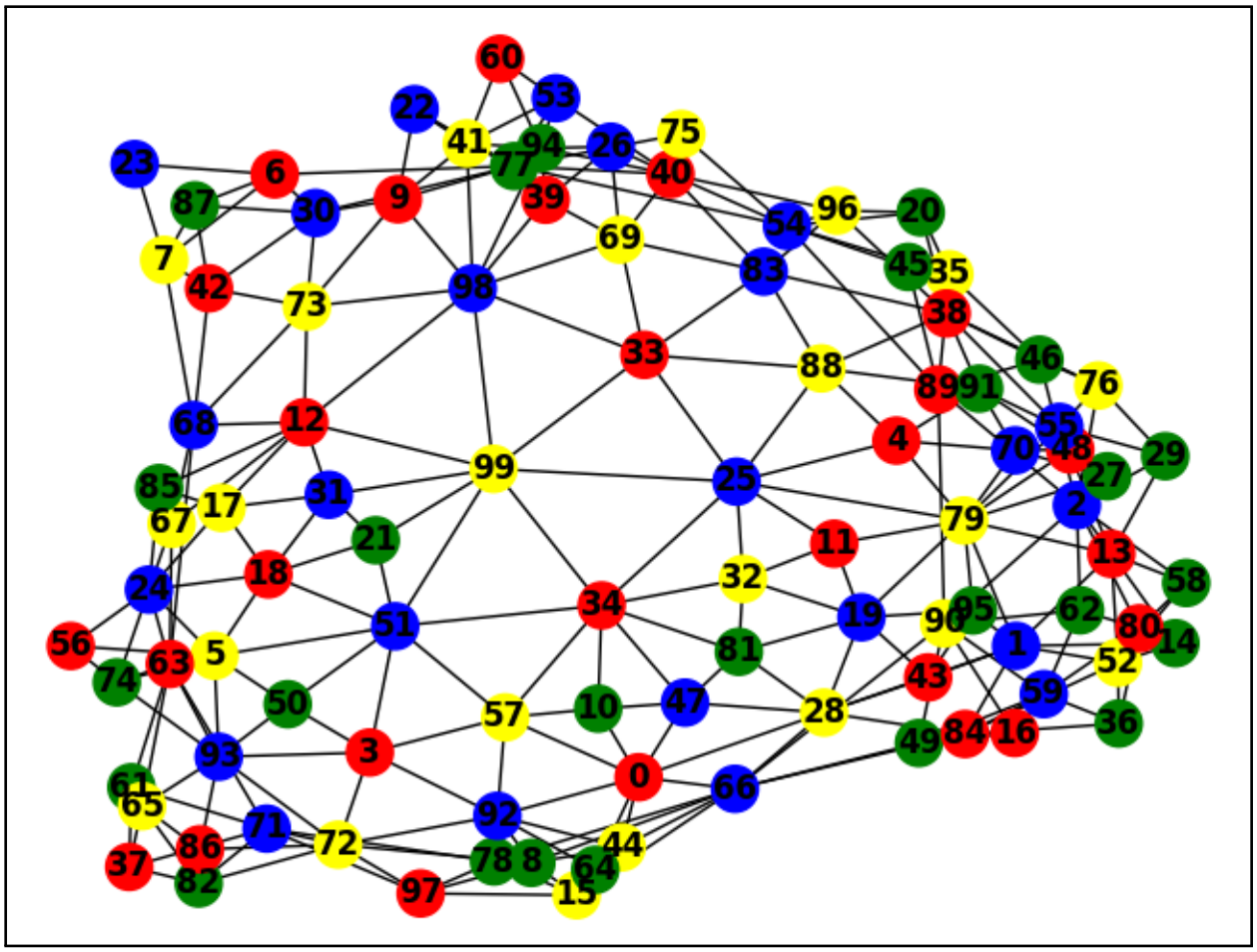


Figure 3: Solution for the map-coloring sample input given to us at the start of the project.

```
MAP SUCCESSFULLY COLORED
Algorithm type:                  * DFS with backtracking and AC3 *
Variable selection heuristic:    * mrv-degree *
Number of nodes:                 * 100 *
-----------------------------------------------------
Time in seconds:                 * 0.005305300000000068 *
```

Figure 4: Example terminal output of my program after solving a map-coloring CSP.

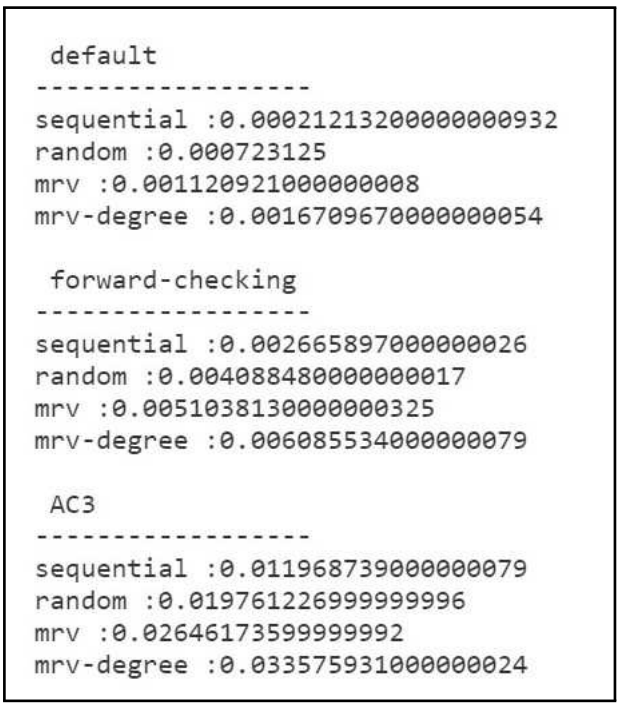


Figure 5: Run-time statistics (in seconds) for my CSP solver on Sudoku problem instances of size = 20 empty squares.

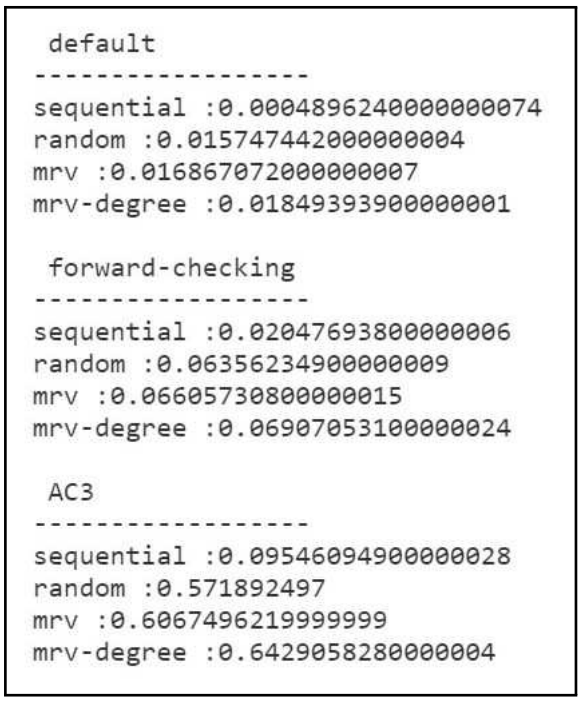


Figure 6: Run-time statistics (in seconds) for my CSP solver on Sudoku problem instances of size = 35 empty squares.

```
Sudoku board before DFS:

-------------------------
| 7 6 9 | 5 0 2 | 4 8 0 |
| 0 2 0 | 6 0 8 | 9 5 0 |
| 8 4 5 | 7 0 9 | 1 2 0 |
-------------------------
| 9 1 3 | 4 8 7 | 5 6 2 |
| 0 7 0 | 0 2 5 | 0 9 1 |
| 2 5 0 | 1 9 0 | 7 3 4 |
-------------------------
| 5 8 6 | 2 7 0 | 3 0 9 |
| 0 0 2 | 9 5 4 | 0 7 8 |
| 4 9 7 | 8 6 3 | 2 1 0 |
-------------------------

Sudoku board after DFS:

-------------------------
| 7 6 9 | 5 1 2 | 4 8 3 |
| 3 2 1 | 6 4 8 | 9 5 7 |
| 8 4 5 | 7 3 9 | 1 2 6 |
-------------------------
| 9 1 3 | 4 8 7 | 5 6 2 |
| 6 7 4 | 3 2 5 | 8 9 1 |
| 2 5 8 | 1 9 6 | 7 3 4 |
-------------------------
| 5 8 6 | 2 7 1 | 3 4 9 |
| 1 3 2 | 9 5 4 | 6 7 8 |
| 4 9 7 | 8 6 3 | 2 1 5 |
-------------------------
SOLVED SUCCESSFULLY
Algorithm type:                 * DFS with backtracking only *
Variable selection heuristic:   * sequential *
Number of empty squares:        * 20 *
Algorithm time in seconds:      * 0.00018960000000001198 *
```

Figure 7: Example terminal output of my program after solving a Sudoku CSP.