Homework Project 2
Sudoku CSP Solver
Ali Sultan Sikandar

Program Design:

The Program is mainly run by changing the values in the configuration file. This configuration file is of format YAML and is titled **config.yaml.** The user can choose between Vanilla, Triple, and Killer Sudoku for the type of Sudoku that he wants to be solved. There are 10 different encoded puzzles for every category which increase in difficulty as follows:

1-4	Easy
5-7	Medium
8-10	Hard

Once the values from the configuration dictionary are fetched, the methods belonging to the classes **Vanilla.py, Triple.py, and Killer.py** are invoked. The solved sudoku, the time it took to solve it, and the number of steps are documented in the **Output.txt** file.

User Manual

The entire code for the project is in **Python programming language**. **Python 3.7** or higher needs to be installed on the user system in order to run the program. The following are the options for the configuration file:

SODUKO	Vanilla, Triple, Killer
VERSION	1 - 10
FOWARD_CHECKING	True, False
AC3	True, False

There are three different input files in the running directory by the names of **Vanilla_Soduko.txt, Triple_Soduko.txt, and Killer_Soduko.txt.**

Once these are updated in the **config.yaml** file, the program's main could be run using the following command on the terminal:

python my_program.py

Alternatively, if using a code editor such as VScode, the run button could be pressed as well to obtain the same results.

The results of the solved sudoku can be viewed in the **Output.txt** file which is generated in the running directory after the program is done executing.

Design Description:

There are three different types of sudoku that we are solving using the Constraint Satisfaction Problem. All of these are encoded in different ways due to their different constraints. An example of each one of them is as follows:

6	1			5		8	2	7
	5		3	8	1			9
9			7		2	5		
	7		1	4				2
5		1		9		3		
	2	9				1		
8				1	7	4		
					6		1	
	9	6	8	2	4	7		

Fig 1. Vanilla Sudoku

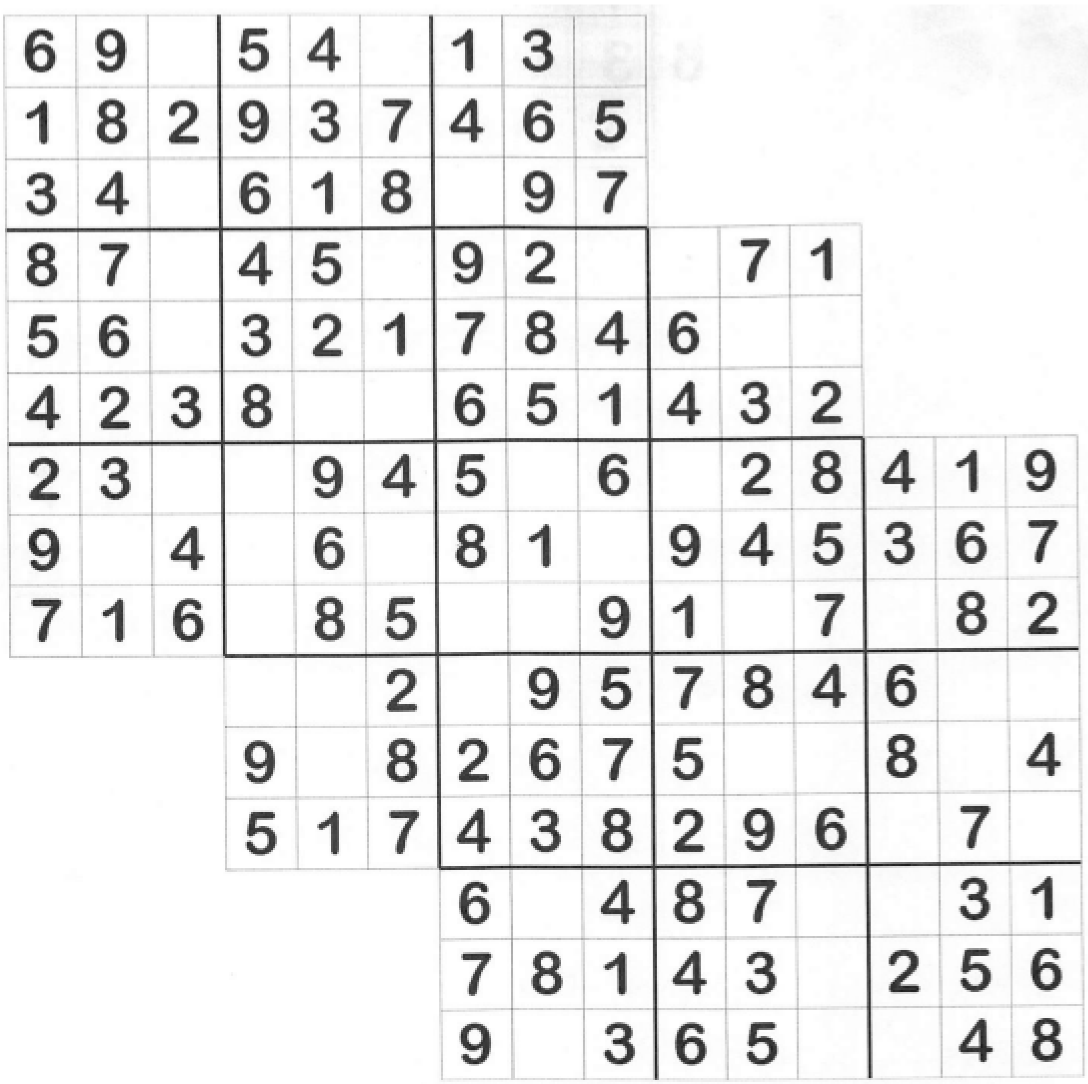


Fig 2. Triple Sudoku

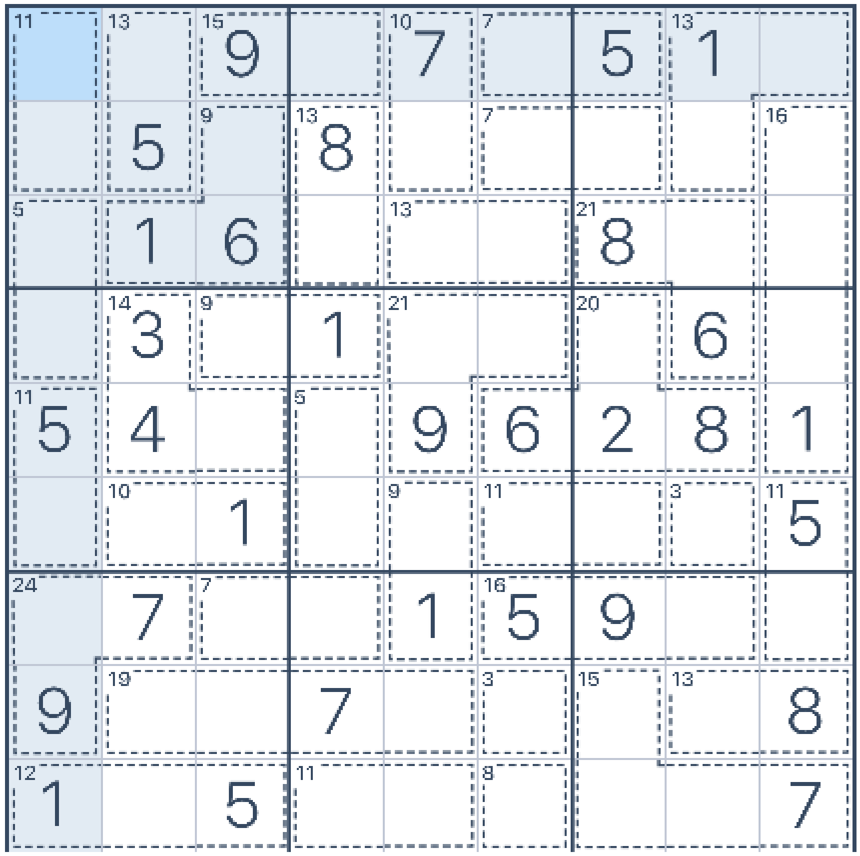


Fig 3. Killer Sudoku

Every cell is treated as a variable having different domains. If a cell is unassigned, its initial domain is for example A1: (1,2,3,4,5,6,7,8,9). On the other hand, an assigned variable would have the domain as for example A2: (2).

Since Killer and Vanilla versions of the sudoku have the same number of cells (81), they are encoded in a similar fashion. Killer sudoku has more cells (171), and therefore needs a different encoding.

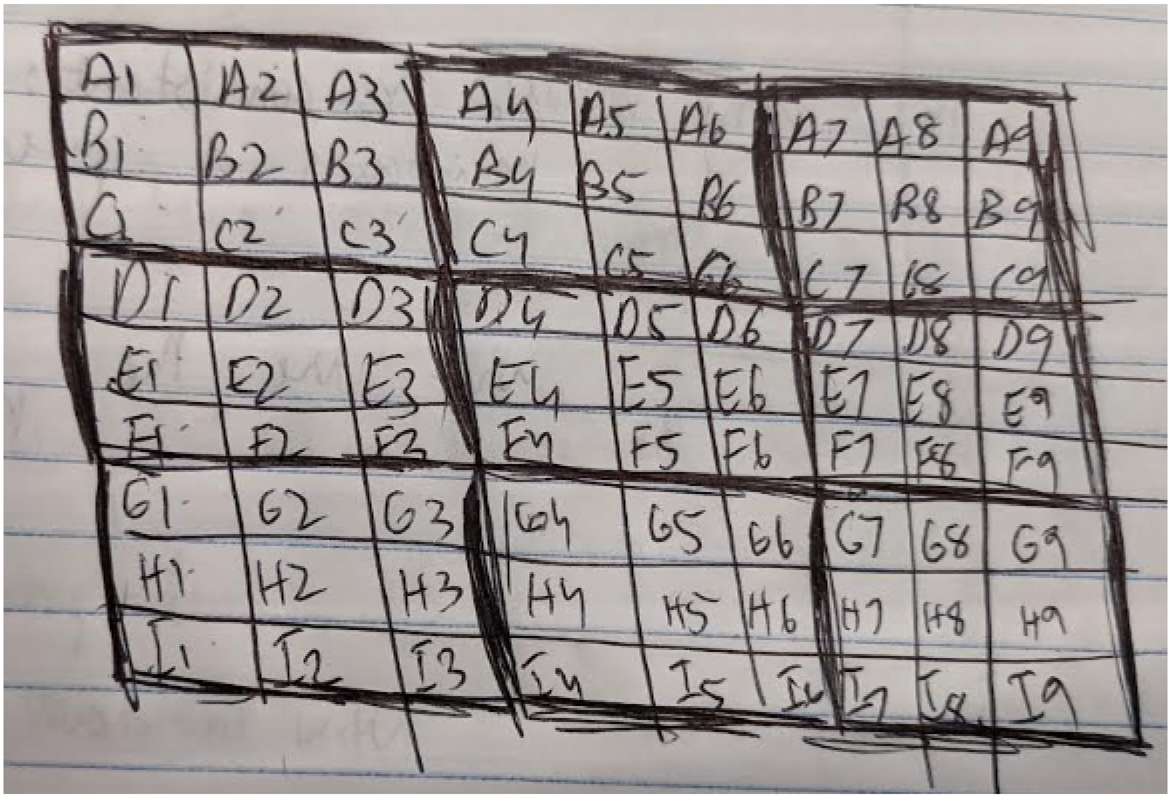


Fig 4. Vanilla Sudoku and Killer Sudoku Encoding

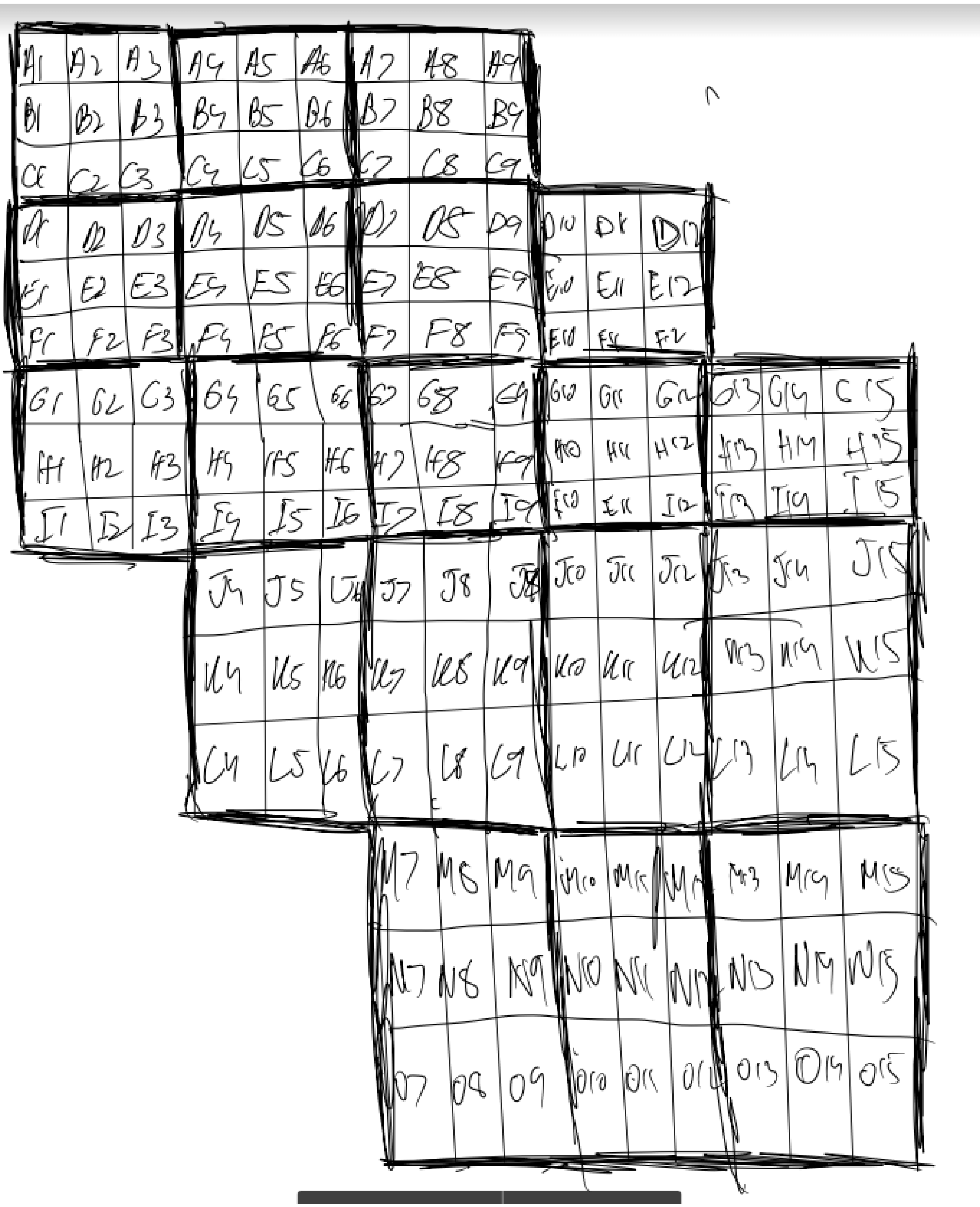


Fig 5. Triple Sudoku Encoding

Every variable needs to be consistent with all the other neighboring variables. An AC-3 pre-processing algorithm goes through every variable and reduces its domains if it is consistent with the other neighboring variables. An AC3 algorithm is also able to indicate if a particular sudoku instance is unsolvable. With our program, we aim to solve the sudoku puzzles with and without AC3 pre-processing to see how much difference there is in the performance.

function AC-3(*csp*) **returns** false if an inconsistency is found and true otherwise
  queue ← a queue of arcs, initially all the arcs in *csp*

  while *queue* is not empty **do**
    $(X_i, X_j) \leftarrow$ POP(*queue*)
    if REVISE(*csp*, X_i, X_j) **then**
      if size of $D_i = 0$ **then return** *false*
      for each X_k **in** X_i.NEIGHBORS - $\{X_j\}$ **do**
        add (X_k, X_i) to *queue*
  return *true*

function REVISE(*csp*, X_i, X_j) **returns** true iff we revise the domain of X_i
  revised ← *false*
  for each x **in** D_i **do**
    if no value y in D_j allows (x,y) to satisfy the constraint between X_i and X_j **then**
      delete x from D_i
      revised ← *true*
  return *revised*

Fig 6. AC3 Algorithm

The main algorithm that solves the sudoku is the **Backtracking Algorithm.** The backtracking algorithm uses **MRV (minimum-remaining-value)** to choose the next variable to be assigned: the variable with the smallest domain hence pruning the search tree. **Least-constraining value** heuristic is also used to help the algorithm pick a value from the domain so that the neighbors of the variable have maximum flexibility.There is an option to trigger **Forward Checking** on assigned variables to further decrease the domains of the neighboring variables. We also see how much of an effect Forward Checking has on the time it takes to solve various sudoku puzzles of differing difficulties.

```
function BACKTRACKING-SEARCH(csp) returns a solution or failure
  return BACKTRACK(csp, { })

function BACKTRACK(csp, assignment) returns a solution or failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(csp, assignment)
  for each value in ORDER-DOMAIN-VALUES(csp, var, assignment) do
     if value is consistent with assignment then
        add {var = value} to assignment
        inferences ← INFERENCE(csp, var, assignment)
        if inferences ≠ failure then
           add inferences to csp
           result ← BACKTRACK(csp, assignment)
           if result ≠ failure then return result
           remove inferences from csp
        remove {var = value} from assignment
  return failure
```

Fig 7. Back-Tracking Algorithm

Agent Performance

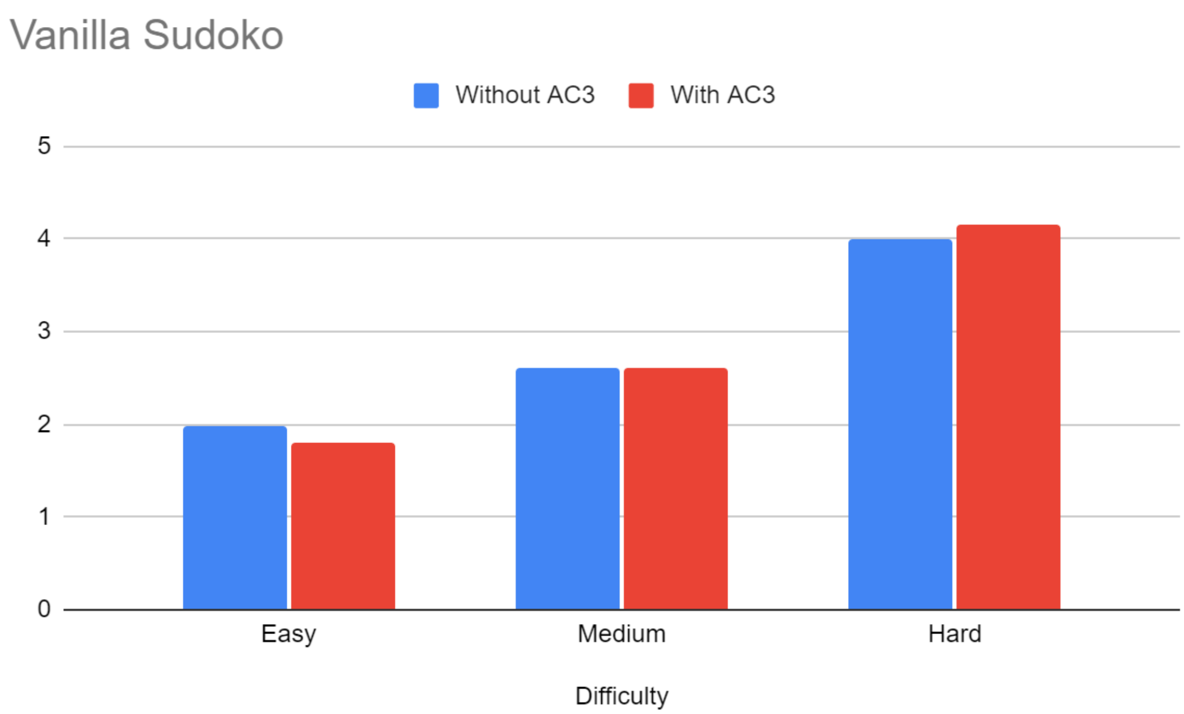


Fig 8. Time for Vanilla: AC3

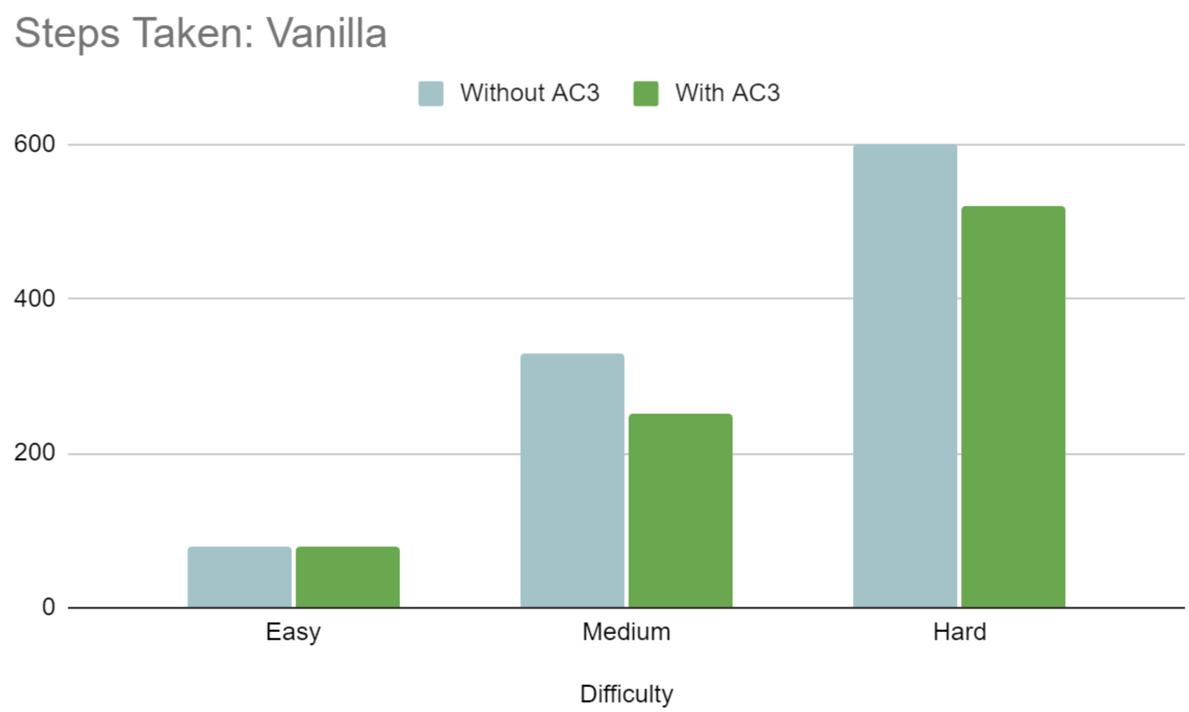


Fig 9. Steps for Vanilla: AC3

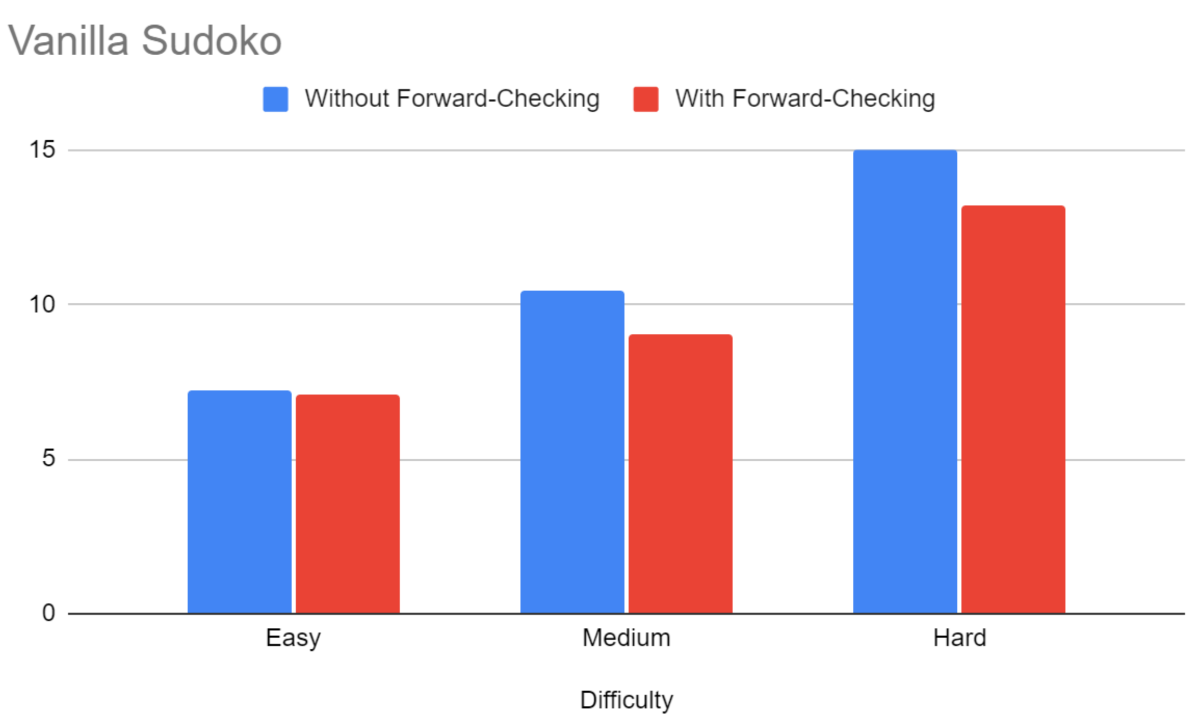


Fig 10. Time for Vanilla: Forward Checking

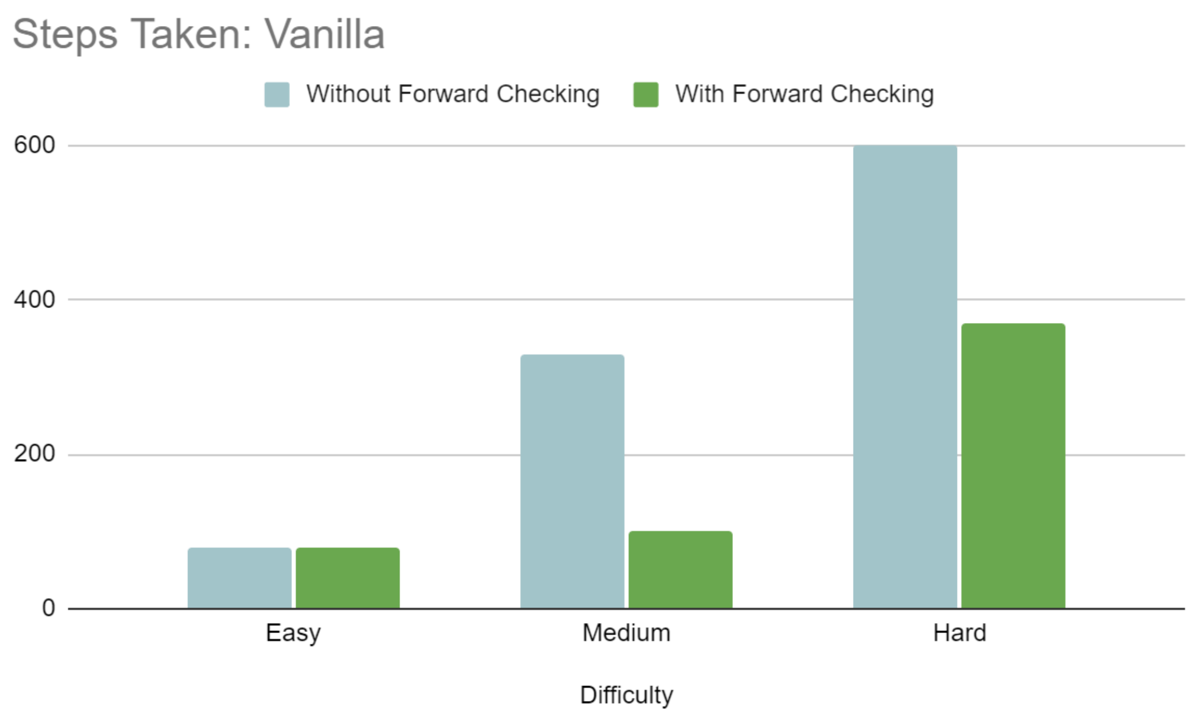


Fig 11. Steps for Vanilla: Forward Checking

The AC3 preprocessing does have a positive effect on the time performance of the CSP agent. We can clearly see from the graph above that while it is not significant, the time it takes to solve the puzzle decreases across all puzzle categories: easy, medium, and hard is lower than without the preprocessing. The same is the case when we look at the number of steps taken for the search tree to be parsed.

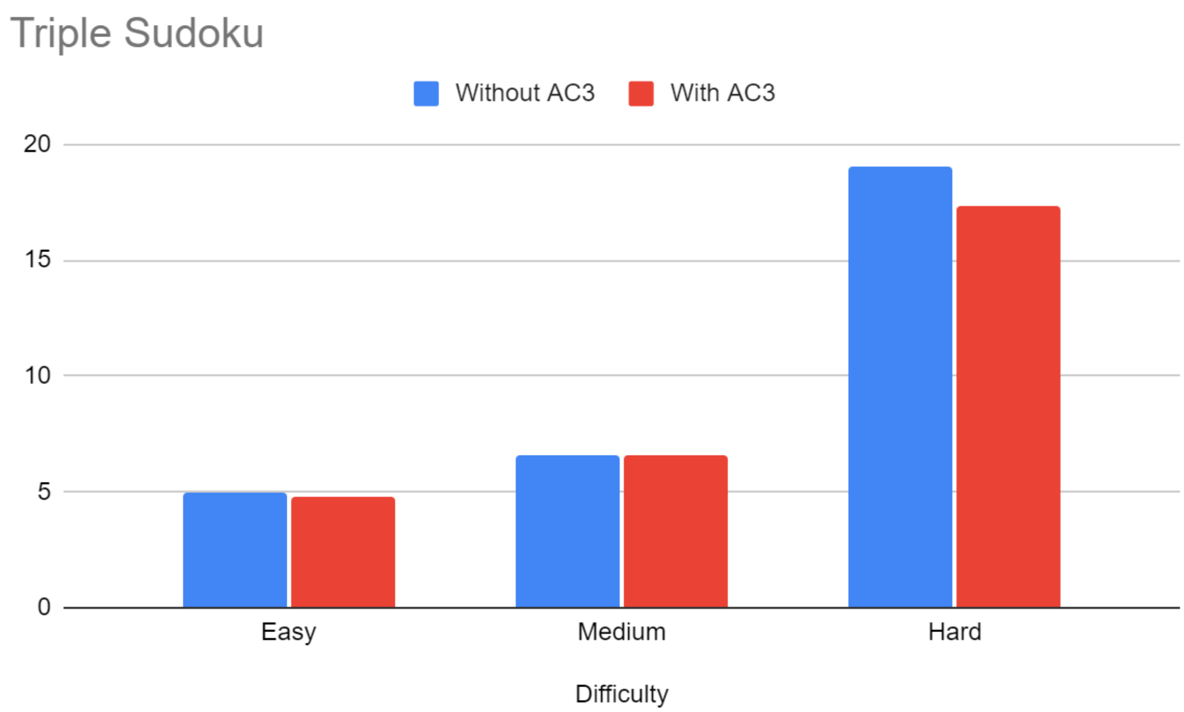
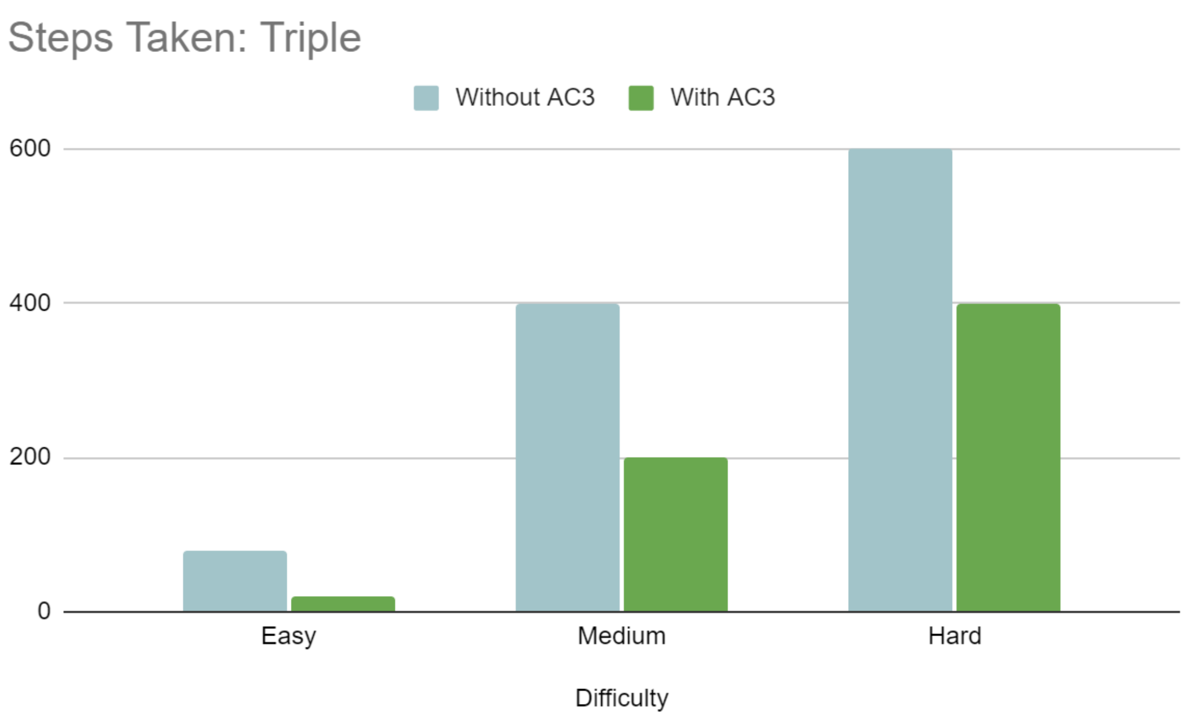


Fig 12. Time for Triple: AC3



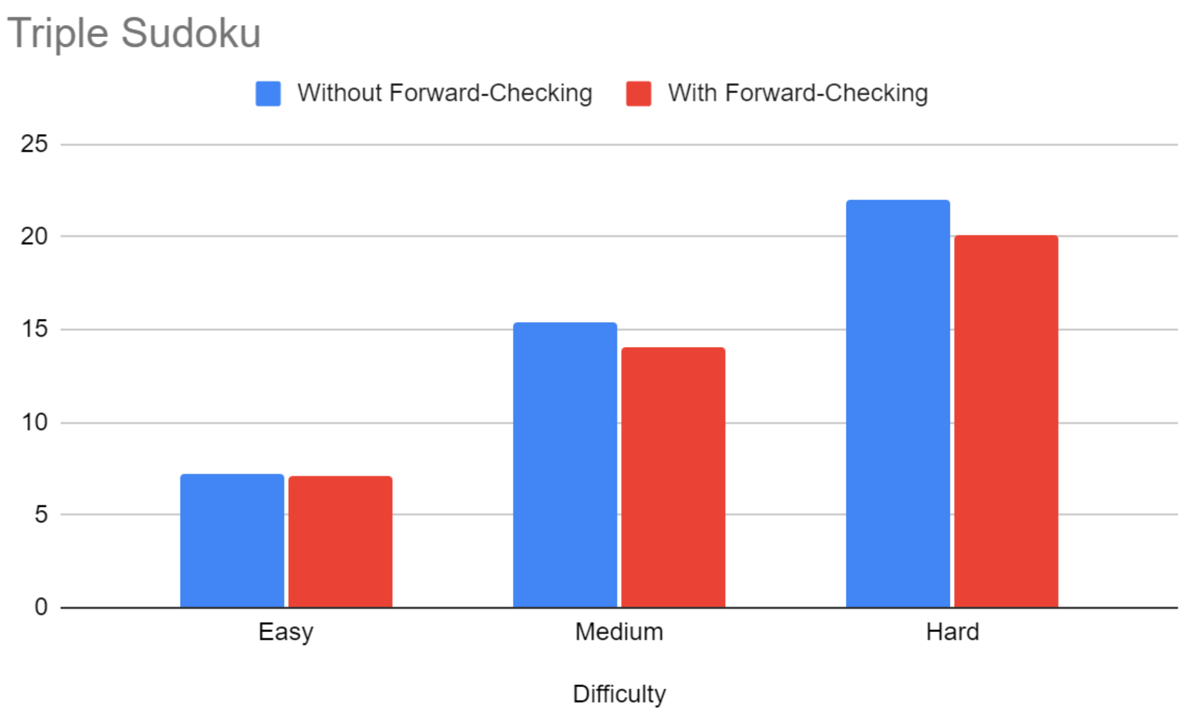


Fig 13. Time for Triple: Forward Checking

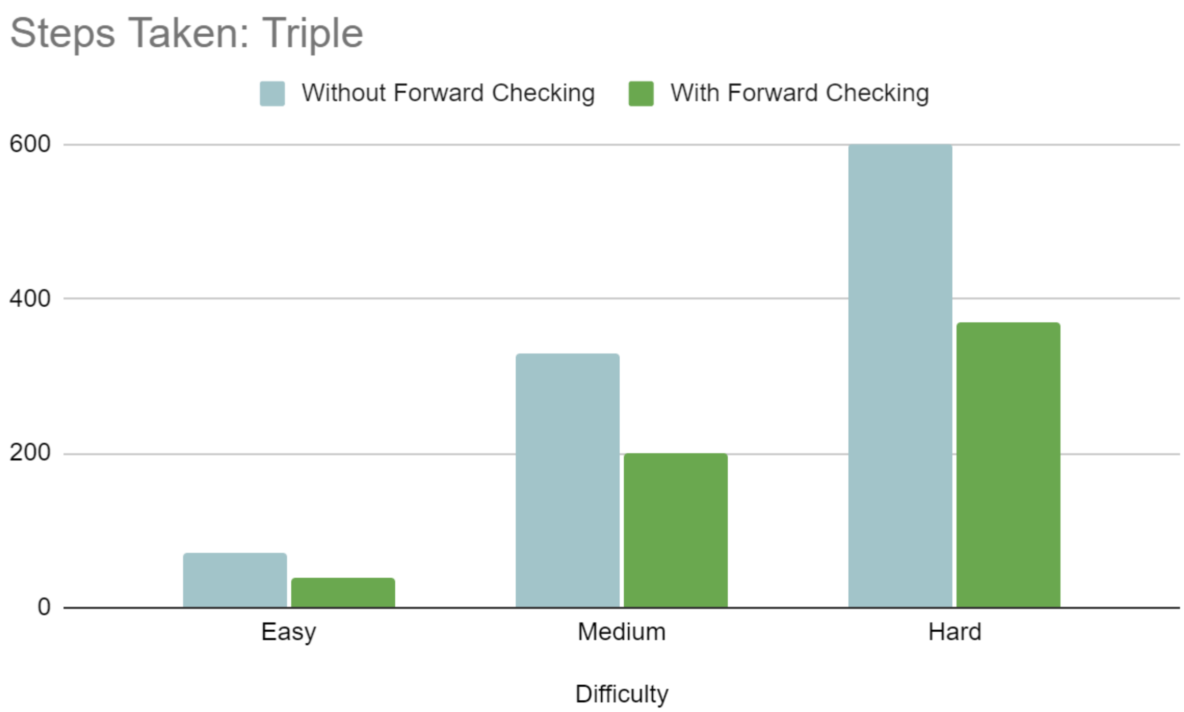


Fig 14. Steps for Triple: Forward Checking

Although the time it takes and the number of steps increases significantly in Triple Sudoku compared to the Vanilla, we can see that the story here is consistent as well. AC3 preprocessing and Forward Checking in the backtracking algorithm do help minimize the number of steps of the CSP search tree as well as computing time for solving the puzzle.

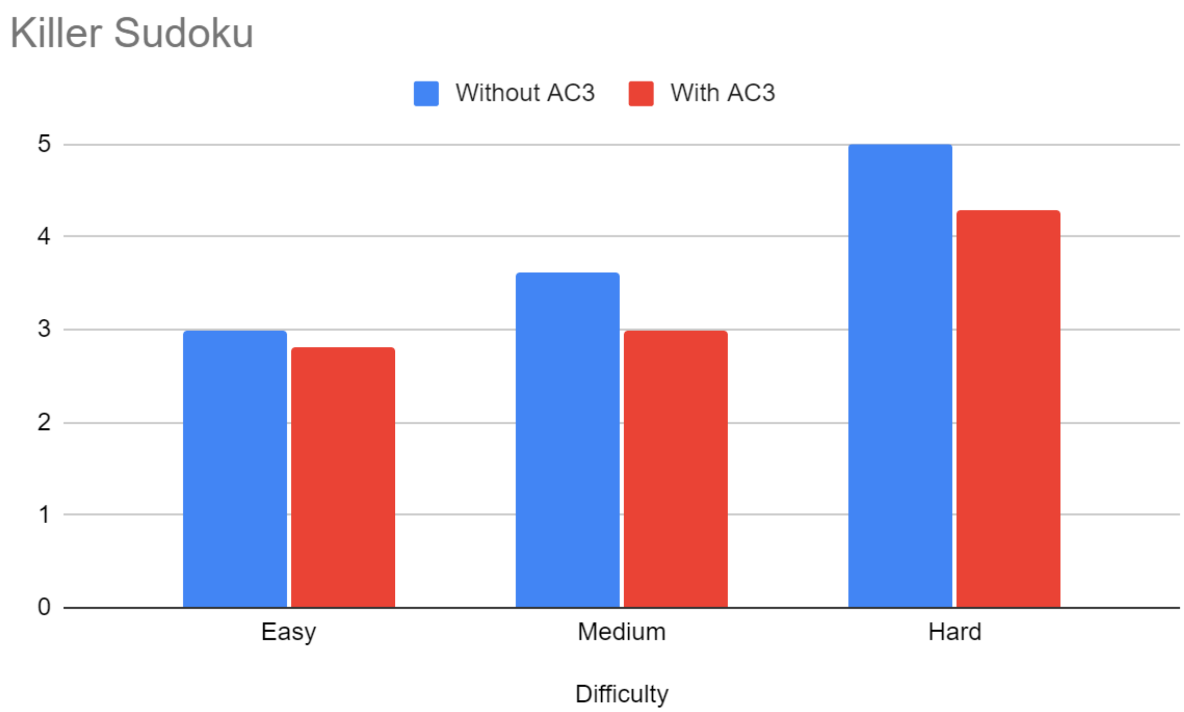


Fig 15. Time for Killer: AC3

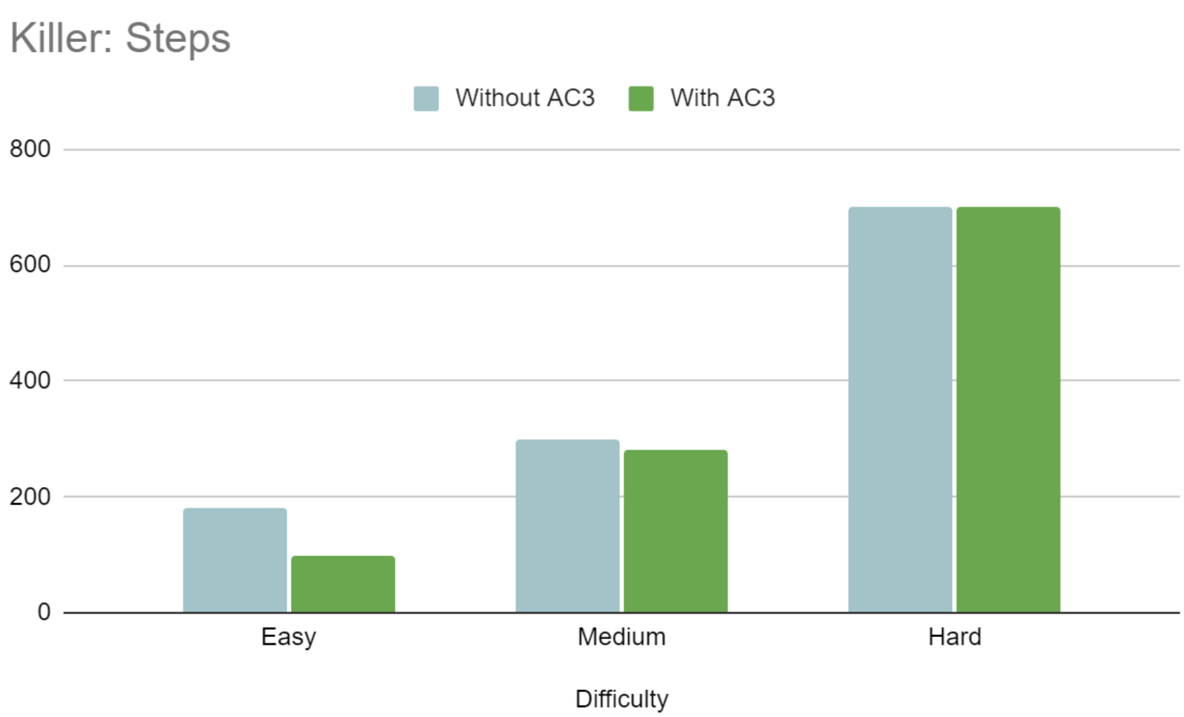


Fig 16. Steps for Killer: AC3

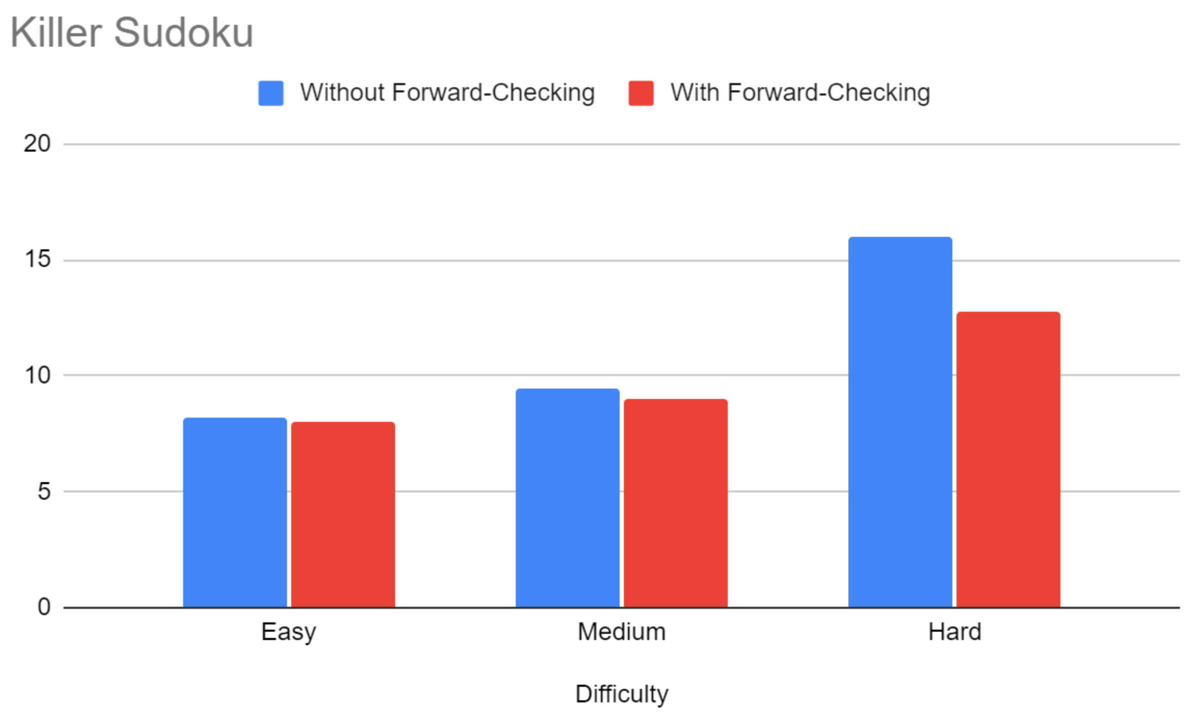


Fig 17. Time for Killer: Forward Checking

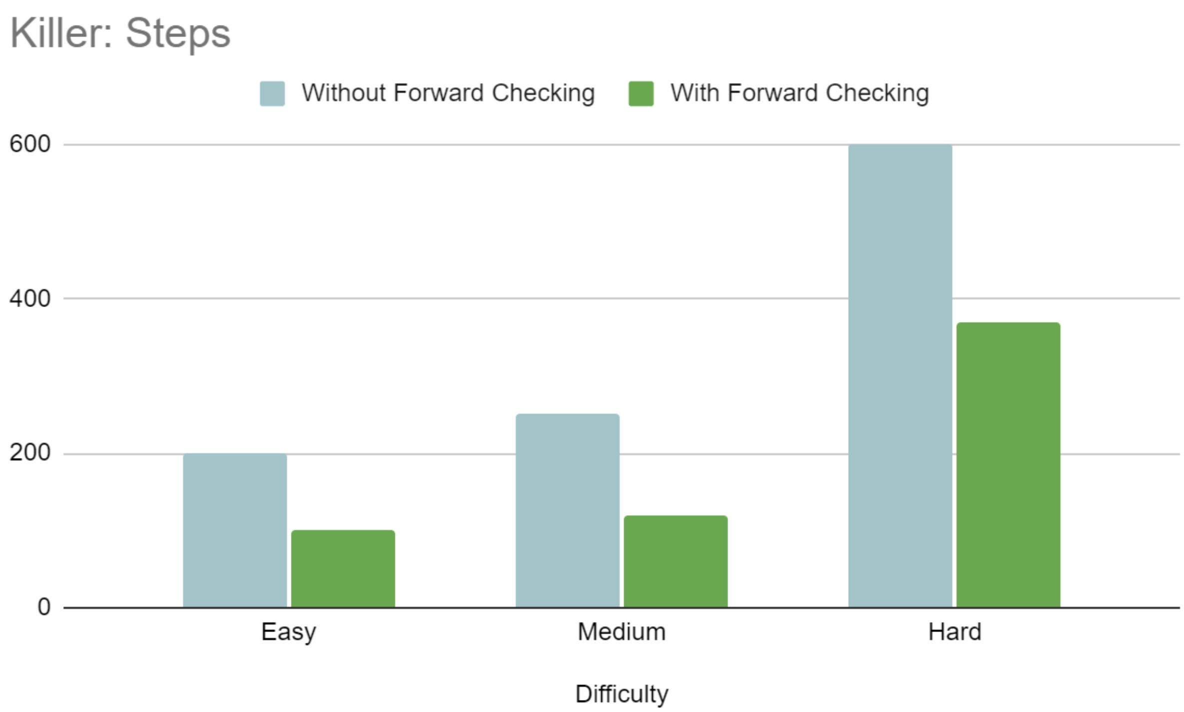


Fig 18. Steps for Killer: Forward Checking

Same story here: the time and the steps for the search tree decreases.

Conclusion:

We prove in this report that **AC3** preprocessing and **Forward checking** techniques help in making a sudoku puzzle less complex. AC3 alone might sometimes be sufficient to solve a constraint-satisfaction-problem. The graphs above reinforce the notion that these techniques are useful for arriving at a solution quicker and therefore must be used as complementary functions to the **CSP Backtracking** solver algorithm to decrease the overall size of the problem.