Sudoku solver

Course Number: 412C
Team:

ID	Name
2	**Ibrahim Yasser Ibrahim**
8	**Ahmed Mahmoud Ahmed Abdel-Maksood Azzam**
42	**Mohamed Adel Abdel-Whaab**
43	**Mohamed Abdel-Aziz Abdel-Kader Hadhod**
55	**Nagy Nabil Mohamed**

Due Date: 20/12/2023
Date handed in:

Problem Formulation

State Space:

- The state space consists of all possible configurations of a Sudoku puzzle.
- Each state is represented by a 2D array/grid where each cell contains a number from 1 to N (N is the size of the puzzle) or is empty.

Initial State:

- The initial state is the unsolved Sudoku puzzle where some cells are filled with numbers, and others are empty.

Actions:

- Placing a number in an empty cell.
- For each empty cell, there are N possible actions, where N is the size of the puzzle.

Transition Model:

- Applying an action involves placing a number in an empty cell.

Goal Test:

- The goal is to have a filled Sudoku grid where all the numbers satisfy the rules of Sudoku (each row, column, and box contains unique numbers from 1 to N).

Cost:

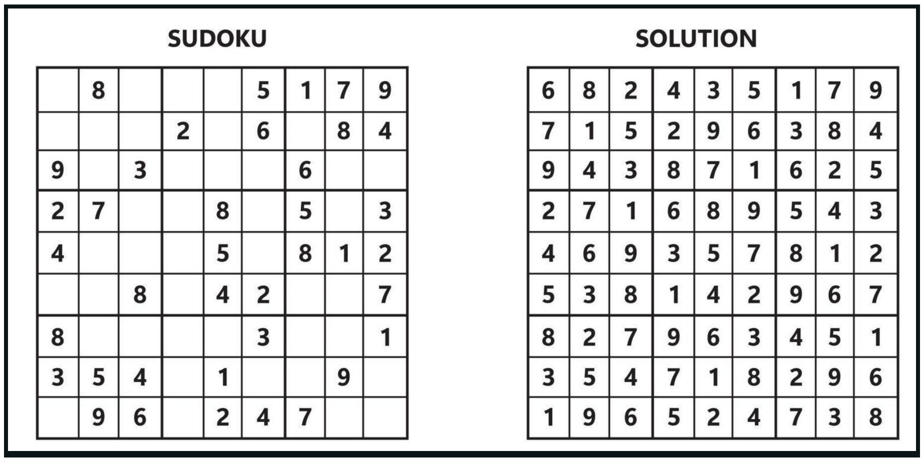
- The cost of each action is typically uniform, as we are not considering a cost associated with each step in a traditional sense. The goal is simply to find a valid solution.

Technical Details

Sudoku is one of the most popular puzzle games of all time. The typical Sudoku puzzle is a 9x9 grid divided into nine 3x3 boxes (There are some other board sizes). Each of the 81 squares must be filled with a number from one to nine. The rules are straightforward:

- Any row contains more than one of the same number from 1 to 9.
- Any column contains more than one of the same number from 1 to 9.
- Any 3×3 grid contains more than one of the same number from 1 to 9.

The puzzle is initially set up with enough numbers to guarantee a unique solution.



SUDOKU

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

SOLUTION

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

Despite these basic rules, there exist an astonishing 6,670,903,752,021,072,936,960 valid Sudoku puzzles. The vastness of the state space makes it a captivating and complex constraint satisfaction problem. Clearly, employing a more sophisticated search algorithm is necessary, rather than exhaustively searching the entire space until a match with the initial values is found.

Backtracking Search with Forward checking

One straightforward approach to solve a Sudoku puzzle is to start by selecting the first empty square and assigning the number one to it. If a conflict arises with another value, simply switch it to two, and continue this process until a non-conflicting value is found. Once a suitable value is identified, move on to another empty square and repeat the procedure. In cases where a square has no feasible values, backtrack to the previously assigned square and modify its value. This searching technique is termed a backtracking search and is certain to discover a solution if one exists, as it systematically explores every conceivable number in every possible location. While this method is highly effective for puzzles of size two, it encounters challenges with size three puzzles due to the nine possibilities for each square. In such cases, there are approximately 9^{81-n} potential states to search, where n represents the number of given values. Clearly, this version of backtracking search is not suitable for size three puzzles. Fortunately, various enhancements can be applied to improve this algorithm, including constraint propagation, forward checking.

Forward Checking

An enhancement to the backtracking search is forward checking. In the previous version of the backtracking search, a value had to be placed before checking for conflicts. Instead, a more convenient approach is to maintain a list indicating the possible values each square can have, given the other assigned numbers. When assigning values to a square, only consider those that do not conflict with the already placed numbers. For a size three puzzle, forward checks can be stored in a boolean array of dimensions nine by nine by nine. Each square has its own array of nine boolean values, corresponding to the possible numbers for that square. If, for example, the third value in the array is false, it indicates that the square cannot contain a three. Managing these lists is straightforward—whenever a new value 'x' is assigned, go to every other square in the same row, column, and box, and mark 'false' in its array for value 'x'. This stored information serves two purposes: first, it verifies that no directly conflicting values are assigned, and second, if the array for any square consists entirely of 'false' values, then there is no possible value for that square, and the most recently assigned value is incorrect. With the incorporation of forward checking, the backtracking search can now successfully solve size three puzzles.

Constraint Propagation: Arc Consistency

While forward checking identifies conflicts just before a specific branch failure, a more proactive approach is available through arc consistency, allowing for the pruning of entire branches. In a size-two puzzle, initially placing a four in the shaded box might seem acceptable, but examining the square below reveals it must be a two. Assigning a two results in the lower-left square having no viable options, prompting additional assignments to correct the error. This problem escalates when the next assignment is not a two, leading to multiple layers of branching and an exponential increase in search time. Arc consistency, a method that enforces constraints among neighboring squares, addresses this challenge. After each search algorithm value assignment, arc consistency iterates through squares, refining potential values based on constraints. If a square has no viable options, the algorithm fails, prompting a return to the search. In case of multiple assignments, they are all revoked simultaneously upon failure. In the provided example, after assigning four, arc consistency recognizes the need for a two below, identifies the lower-left corner's lack of options, fails, and prompts a return to the search. The search then selects an alternative value for the shaded square.

Minimum Remaining Value

An additional technique to enhance the backtracking search is the minimum remaining values heuristic (Russell and Norvig 143). This heuristic modifies the order in which squares are selected during the guessing process to minimize the number of branches at each level. Instead of opting for the first empty square, the square with the fewest potential values is chosen. In the provided puzzle, for instance, one of the two shaded squares would be the next selection, as they each have two possible values compared to the three possible values for other squares. Opting for a square with only two possible values, rather than three, results in the search tree branching in two directions instead of three. Essentially, this reduces the size of the search tree by two-thirds.

Discussion of results

The Sudoku problem was approached using three distinct solvers, each incorporating a different strategy: Minimum Remaining Value, Forward Checking, and Arc Consistency. Here, we delve into the discussion of the results obtained from these solvers.

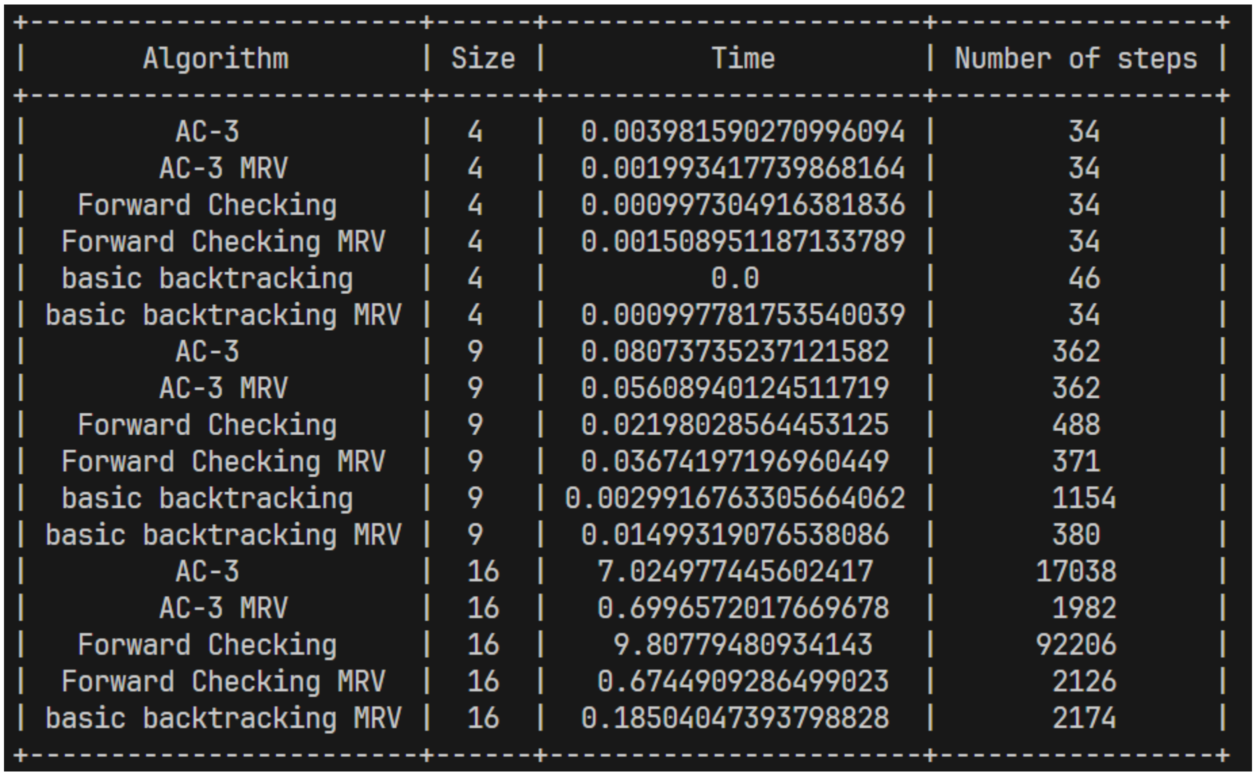
Minimum Remaining Value (MRV) Solver:
The MRV heuristic was employed to alter the order in which squares are guessed, prioritizing squares with the least number of potential values. This approach effectively reduces the branching factor at each level. In practical terms, the MRV solver consistently demonstrated efficient performance by significantly minimizing the search tree's size. This resulted in quicker convergence to solutions, especially in complex Sudoku puzzles where the reduction in branching played a pivotal role.

Forward Checking Solver:
The Forward Checking technique was employed to identify conflicts right before a branch failure, enhancing the overall efficiency of the solver. By maintaining a list of potential values for each square and considering only those that do not conflict with already placed numbers, the solver exhibited a proactive conflict resolution strategy. While effective in identifying conflicts early, the Forward Checking solver showed slight limitations in extremely challenging puzzles, where conflicts were still encountered in subsequent branches.

Arc Consistency Solver:
The Arc Consistency strategy enforced constraints among neighboring squares, proactively pruning entire branches and identifying conflicts at an even earlier stage. This solver demonstrated robust performance, particularly in complex Sudoku scenarios. By refining potential values based on constraints and iteratively enforcing consistency, the Arc Consistency solver showcased superior efficiency compared to the other methods. Its ability to reduce the search space substantially contributed to faster and more reliable solution convergence.

Overall Comparison:
In summary, each solver exhibited unique strengths and limitations. The MRV solver excelled in reducing branching factors, the Forward Checking solver efficiently identified conflicts before branch failures, and the Arc Consistency solver demonstrated superior efficiency by proactively pruning branches through constraint enforcement. The choice of the most suitable solver may depend on the specific characteristics of the Sudoku puzzle at hand. For puzzles with a high degree of complexity, the Arc Consistency solver emerged as the most effective in terms of both speed and accuracy. However, in less challenging puzzles, the MRV and Forward Checking solvers still provided reliable solutions with reasonable efficiency.

Result Sample



```
+------------------------+------+-----------------------+-----------------+
|       Algorithm        | Size |         Time          | Number of steps |
+------------------------+------+-----------------------+-----------------+
|         AC-3           |  4   |  0.003981590270996094 |       34        |
|        AC-3 MRV        |  4   |  0.001993417739868164 |       34        |
|    Forward Checking    |  4   |  0.000997304916381836 |       34        |
|  Forward Checking MRV  |  4   |  0.001508951187133789 |       34        |
|  basic backtracking    |  4   |          0.0          |       46        |
| basic backtracking MRV |  4   |  0.000997781753540039 |       34        |
|         AC-3           |  9   |  0.08073735237121582  |      362        |
|        AC-3 MRV        |  9   |  0.05608940124511719  |      362        |
|    Forward Checking    |  9   |  0.02198028564453125  |      488        |
|  Forward Checking MRV  |  9   |  0.03674197196960449  |      371        |
|  basic backtracking    |  9   | 0.0029916763305664062 |       1154      |
| basic backtracking MRV |  9   |  0.01499319076538086  |      380        |
|         AC-3           |  16  |   7.024977445602417   |      17038      |
|        AC-3 MRV        |  16  |   0.6996572017669678  |       1982      |
|    Forward Checking    |  16  |    9.80779480934143   |      92206      |
|  Forward Checking MRV  |  16  |   0.6744909286499023  |       2126      |
| basic backtracking MRV |  16  |  0.18504047393798828  |       2174      |
+------------------------+------+-----------------------+-----------------+
```

Minimum Remaining Value

The Minimum Remaining Values (MRV) algorithm is a heuristic used to prioritize the selection of variables in a constraint satisfaction problem. In the context of solving a Sudoku puzzle, MRV is often used to determine which empty cell to fill in next. The basic steps of the MRV algorithm for solving a Sudoku puzzle are as follows:

1. Initialization:

- Start with the initial Sudoku puzzle.
- Identify all empty cells (cells with a value of 0).

2. Calculate Remaining Values:

- For each empty cell, determine the possible values that can be placed in that cell without violating the Sudoku rules (no repetition of numbers in rows, columns, and 3x3 boxes).

3. Select Variable with Minimum Remaining Values:

- Identify the empty cell with the fewest remaining possible values. This is the variable with the minimum remaining values.

4. Try Each Possible Value:

- Choose one of the possible values for the selected variable and update the Sudoku grid accordingly.
- Recursively apply the MRV algorithm to the updated puzzle.

5. Backtrack if Necessary:

- If at any point the algorithm leads to an invalid state (a contradiction), backtrack to the previous state and try a different value for the selected variable.
- Continue exploring different values until a solution is found or all possibilities are exhausted.

6.Repeat:

- Repeat steps 3-5 until the entire Sudoku puzzle is filled in.
  Here's a simple example to illustrate the MRV algorithm:

Consider the following initial Sudoku puzzle:

```
5 3 0 | 0 7 0 | 0 0 0

6 0 0 | 1 9 5 | 0 0 0

0 9 8 | 0 0 0 | 0 6 0

------+-------+------

8 0 0 | 0 6 0 | 0 0 3

4 0 0 | 8 0 3 | 0 0 1
```

7 0 0 | 0 2 0 | 0 0 6

------+-------+------

0 6 0 | 0 0 0 | 2 8 0

0 0 0 | 4 1 9 | 0 0 5

0 0 0 | 0 8 0 | 0 7 9

Let's apply the MRV algorithm step by step:

1. Identify empty cells: Cells with a value of 0.
2. Calculate remaining values for each empty cell.
3. Select the variable with the minimum remaining values (e.g., the cell at (1, 2)).
4. Try a possible value (e.g., 1) for the selected variable and update the puzzle.
5. Recursively apply MRV to the updated puzzle.

Continue this process until a solution is found or all possibilities are exhausted. If a contradiction is encountered, backtrack to the previous state and try a different value.

Forward Checking

Forward checking is a heuristic used in constraint satisfaction problems like Sudoku to preemptively avoid choices that lead to a contradiction. It reduces the search space by immediately checking the implications of a variable assignment and pruning inconsistent values from the domains of neighboring variables. This approach can significantly speed up the solving process by preventing the algorithm from venturing too far down the wrong path.

Basic Steps of Forward Checking for Sudoku:

1. **Initialization**:
   - Start with the initial Sudoku grid where some cells are filled, and others are empty.
   - Initialize the domain of each empty cell. Initially, this domain contains all numbers from 1 to 9 (or 1 to N for an N×N Sudoku).

2. **Select a Cell:**

   - Choose an empty cell to assign a value. This can be done using heuristics like "Minimum Remaining Values" (MRV) which selects the cell with the fewest legal values left.

3. **Assign a Value**:
   - Assign a legal value from the domain of the chosen cell.

4. **Update Domains:**
   - For each unassigned cell that is in the same row, column, or box as the chosen cell, remove the assigned value from its domain.
   - If the domain of any cell becomes empty, revert the assignment (backtrack) and try another value.

5. **Consistency Check:**
   - After each assignment, check if the Sudoku puzzle remains consistent. Consistency means no domain is empty, and no row, column, or box violates the Sudoku rules.

6. **Repeat or Backtrack:**
   - If the assignment leads to an inconsistent state, backtrack: undo the last assignment and try a different value.
   - If consistent, proceed to the next unassigned cell and repeat the process.

7. **Termination:**
   - The algorithm terminates successfully when all cells are assigned without any inconsistency.
   - If no assignments are possible for some cell, the puzzle is unsolvable with the current partial assignments, triggering backtracking.

Example:
Consider a simple 4x4 Sudoku puzzle:

```
| 1 |   |   |   |      Cell 2: [2, 3, 4]
|   | 2 |   |   |      Cell 5: [3, 4]
----------------
|   |   | 3 |   |
|   |   |   | 4 |
```

- Let's say we choose cell 2 and assign the value 2.
- We then update the domains of cells in the same row, column, and box. For instance, cell 5 can no longer have 2 in its domain, so its new domain is [3, 4].
- If at any point a cell's domain becomes empty, we know the current assignment for cell 2 is wrong. We backtrack, undo the assignment for cell 2, and try the next value in its domain.

Forward checking helps to significantly reduce the search space in a Sudoku solver by immediately eliminating choices that would lead to a dead end, making the solver more efficient.

Constraint Propagation: Arc Consistency

Background:

Constraint propagation involves incorporating any form of reasoning that explicitly restricts certain values or combinations of values for specific variables within a problem. This restriction occurs when a subset of constraints cannot be satisfied in any other way. For example, in a crossword puzzle, eliminating the words NORWAY and SWEDEN from the potential European countries that fit a 6-digit slot due to the requirement for the second letter to be 'R' represents the propagation of a constraint. In a scenario with two variables, x1 and x2, both taking integer values in the range of 1 to 10 and a constraint stating that $|x1-x2| > 5$, the propagation of this constraint enables the exclusion of values 5 and 6 for both x1 and x2. Articulating these 'nogoods' serves as a method to reduce the range of combinations explored by a search mechanism.
Arc consistency is the oldest and most well-known way of propagating constraints. This is indeed a very simple and natural concept that guarantees every value in a domain to be consistent with every constraint.
In Sudoku, the primary constraints involve ensuring that each row, column, and box contains unique numbers from 1 to N (N is the size of the puzzle). Arc consistency helps enforce these constraints by iteratively removing inconsistent values from the domains of variables (cells) until a solution is reached.

Basic Steps of Arc Consistency for Sudoku:

1. **Initialization:**
   - Begin with the initial Sudoku grid, where some cells are filled, and others are empty.
2. **Create Arcs:**
   - Create arcs between all pairs of related cells. Two cells are related if they are in the same row, column, or box.
3. **Queue Initialization:**
   - Initialize a queue with all arcs.
4. **Iterative Process:**
   - While the queue is not empty, do the following steps:

   a. **Dequeue an Arc (i, j):**
      - Take an arc (i, j) from the queue.

   b. **Check Consistency:**
      - For each value in the domain of cell i, check if there is a consistent value in the domain of cell j.
      - If a consistent value is found, continue to the next iteration.

   c. **Remove Inconsistent Values:**

        - If no consistent value is found, remove the inconsistent values from the domain of cell i.
        - If the domain of cell i is modified, enqueue all arcs (k, i) where k is a neighbor of i (excluding (j, i)).
    d. **Repeat:**
        - Repeat the process until the queue is empty.

5. **Termination:**
    - The algorithm terminates when the domains of all cells are consistent, and no further removals are possible.

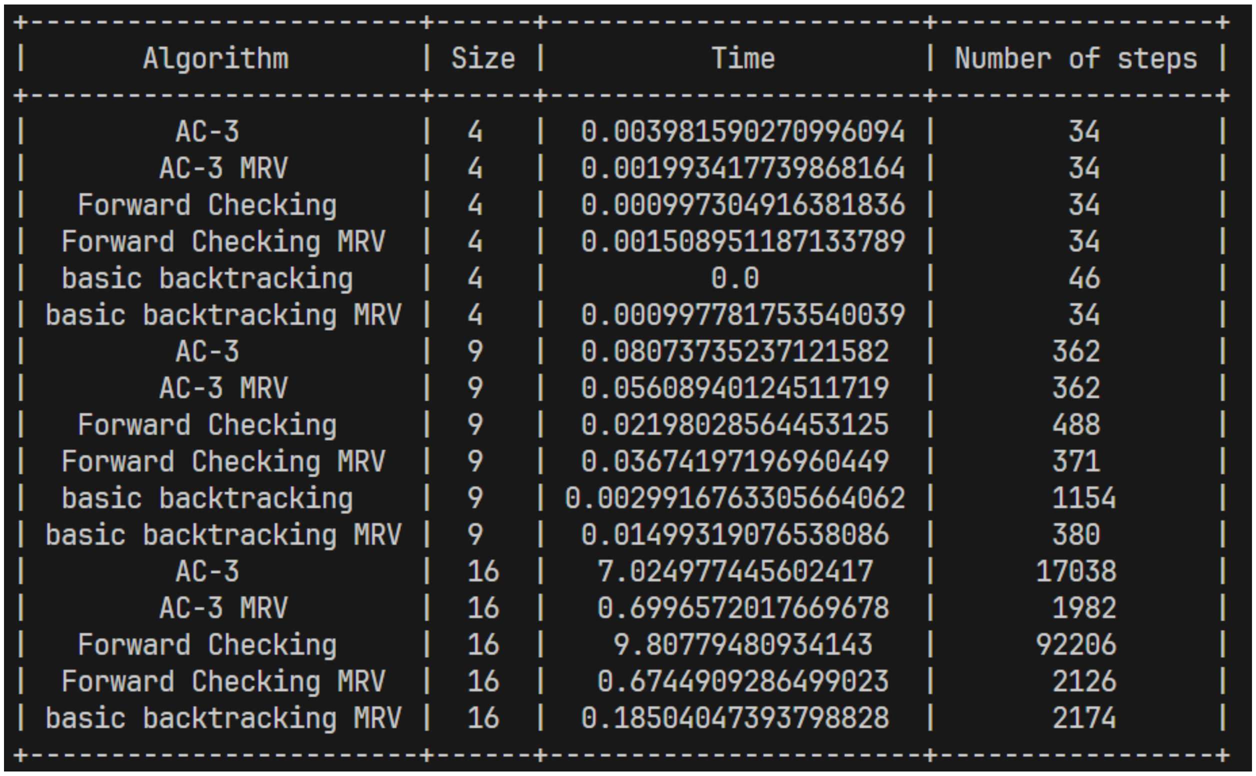
Example:

Consider a 4x4 Sudoku puzzle:

```
| 1 | 2 |   |   |      1: [1, 2]
| 3 | 4 |   |   |      2: [3, 4]
----------------
|   | 3 |   |   |      3: [3]
|   | 4 |   |   |      4: [4]
```

Let's say we have the **arc (1, 3)** in the queue. The **domain of cell 1 is [1, 2]**, and the domain of **cell 3 is [3]**. Since there is no consistent value in the domains, we remove inconsistent values from the domain of cell 1. The new domain becomes [2]. Now, we enqueue arcs (2, 1) and (4, 1).

Result sample



```
+-----------------------+------+-----------------------+-----------------+
|       Algorithm       | Size |          Time         | Number of steps |
+-----------------------+------+-----------------------+-----------------+
|          AC-3         |  4   |  0.003981590270996094 |        34       |
|        AC-3 MRV       |  4   |  0.001993417739868164 |        34       |
|    Forward Checking   |  4   |  0.000997304916381836 |        34       |
|  Forward Checking MRV |  4   |  0.001508951187133789 |        34       |
|   basic backtracking  |  4   |          0.0          |        46       |
| basic backtracking MRV|  4   |  0.000997781753540039 |        34       |
|          AC-3         |  9   |  0.08073735237121582  |       362       |
|        AC-3 MRV       |  9   |  0.05608940124511719  |       362       |
|    Forward Checking   |  9   |  0.02198028564453125  |       488       |
|  Forward Checking MRV |  9   |  0.03674197196960449  |       371       |
|   basic backtracking  |  9   | 0.0029916763305664062 |       1154      |
| basic backtracking MRV|  9   |  0.01499319076538086  |       380       |
|          AC-3         |  16  |   7.024977445602417   |      17038      |
|        AC-3 MRV       |  16  |   0.6996572017669678  |       1982      |
|    Forward Checking   |  16  |    9.80779480934143   |      92206      |
|  Forward Checking MRV |  16  |   0.6744909286499023  |       2126      |
| basic backtracking MRV|  16  |  0.18504047393798828  |       2174      |
+-----------------------+------+-----------------------+-----------------+
```

Team Contribution:

- Ibrahim Yasser Ibrahim: Forward Checking
- Ahmed Mahmoud Azzam: Backtracking and main code structuring.
- Mohamed Adel Abdelwahab: Constraint Propagation (Arc Consistency)
- Mohamed Abdelaziz Abdelkader: MRV
- Nagy Nabil: Result comparison, sudoku grid generation, and Min-conflicts.