**INFO 550**
**Ani Chitransh**

**Link to the github repo: https://github.com/Anichitra30/CSP-AI-Project**

## Introduction

Sudoku is a famous logic-based puzzle game in which players use numbers from 1 to 9 to fill in a 9x9 grid. Due to its straightforward rules and difficult gameplay, the game has amassed a large following globally, making it a great platform for testing various Artificial Intelligence (AI) strategies. The Constraint Satisfaction Problem (CSP), a popular strategy for tackling complicated issues involving finding solutions to a set of constraints, is one such approach.
The goal of this project and report is to provide a comparative examination of the use of CSP in solving Sudoku puzzles. Let us understand the general framework of CSP and how it can be used to solve Sudoku puzzles and then investigate several methods and techniques for optimizing the search process and increasing the efficiency of CSP-based Sudoku solutions.

We also compare the performance of our CSP-based Sudoku solver with that of other well-known methods like brute-force search and backtracking. We test the solver on a collection of Sudoku puzzles ranging in difficulty, and we compare the results in terms of runtime and solution quality.

## What is CSP? How is it used to solve puzzles?

Constraint Satisfaction Algorithm works on finding a set of constraints that must be satisfied for a solution to be found. The constraints are basically a set of variables, each with a range of potential values, and a set of constraints defining the relationships between the variables and their values. The objective is to identify a consistent variable assignment that satisfies each requirement.
While solving Sudoku puzzles, CSP can be used to represent an issue as a set of variables and constraints. Each cell in the grid is presented as a variable, with the digits from 1 to 9 representing the set of potential values for each variable. The rules of the game define the constraints, which require that every row, column, and 3x3 subgrid of the puzzle contain all digits from 1 to 9 exactly once.When utilizing CSP to solve a Sudoku puzzle, we begin with a puzzle that may have some cells with digits filled in and some cells with blank spaces. After that, we use a CSP solver algorithm to repeatedly fill in the blank cells with numbers that adhere to the puzzle's rules. The algorithm selects an empty cell, selects a value from its range of potential values, and then determines if this value satisfies the puzzle's restrictions. If it does, the algorithm moves on to the following empty cell after assigning the value to the variable that represents the cell. If not, until a reliable value assignment is made, the process goes back and tries a different value for the previous variable. In CSP-based Sudoku solvers, heuristics can be used to accelerate the search process and reduce the number of possible solutions. For instance, a common heuristic is to assign a value to the cell with the fewest potential values as the next variable because this

will probably require fewer backtracking steps. Other heuristics might comprise selecting values more likely to result in a solution or trimming the search space by removing conflicting values at the beginning of the search.

## Backtracking

The backtracking algorithm repeatedly tries several possible values for empty Sudoku grid cells to find one optimal solution. It begins by picking a grid cell that is empty and attempting a value from that cell's range of potential values. The algorithm repeats the process in the next vacant cell if the value does not contradict any of the Sudoku rules. When a value violates a constraint, the algorithm goes back to the previous cell and tries a different value until all of the grid's cells have consistent values assigned to them.

**Code snippet:**

```
# If all cells are filled, the puzzle is solved
if row is None:
    return grid

# Try all possible values for the empty cell
for value in range(1, 10):
    if is_valid(grid, row, col, value):
        # If the value is valid, add it to the grid
        BTIteration+=1
        grid[row][col] = value

        # Recursively try to solve the puzzle
        if solve_sudoku(grid):
            return grid

        # If the puzzle can't be solved with the current value, backtrack
        BTIteration+=1
        grid[row][col] = 0
```

**Explanation:** The backtracking approach can solve Sudoku puzzles quickly as it doesn't need to take into account all possible values for every cell in the grid. Instead, it does a systematic search of the space of potential answers and filters out inconsistent solutions at the beginning of the search. The above function works by finding the next empty cell in the grid and trying all possible values for that cell. It then

recursively tries to solve the puzzle using the chosen value. If the puzzle can't be solved with the current value, it backtracks and tries the next value. A sample solution is listed below using the algo.

For Easy Solution

[1, 9, 8, 5, 4, 3, 7, 2, 6]
[6, 4, 3, 2, 7, 8, 5, 9, 1]
[5, 2, 7, 6, 1, 9, 8, 4, 3]
[9, 1, 4, 7, 3, 5, 2, 6, 8]
[8, 7, 6, 1, 9, 2, 4, 3, 5]
[2, 3, 5, 4, 8, 6, 1, 7, 9]
[4, 6, 2, 3, 5, 1, 9, 8, 7]
[3, 8, 1, 9, 2, 7, 6, 5, 4]
[7, 5, 9, 8, 6, 4, 3, 1, 2]
Time taken: 1.375503 seconds
The number of Iterations for BackTracking= 220546

For Hard Solution

[5, 4, 3, 8, 2, 9, 7, 6, 1]
[1, 2, 8, 7, 6, 5, 9, 4, 3]
[9, 6, 7, 1, 3, 4, 2, 8, 5]
[2, 3, 4, 9, 8, 1, 5, 7, 6]
[6, 5, 1, 4, 7, 2, 3, 9, 8]
[8, 7, 9, 6, 5, 3, 4, 1, 2]
[4, 8, 2, 5, 1, 7, 6, 3, 9]
[7, 1, 5, 3, 9, 6, 8, 2, 4]
[3, 9, 6, 2, 4, 8, 1, 5, 7]
Time taken: 3.321877 seconds
The number of Iterations for BackTracking= 877111

**Drawbacks:** Backtracking can be computationally expensive since it requires a lot of recursive operations, especially for puzzles with many empty cells. Heuristics and optimizations are frequently utilized as a result to lower the number of potential solutions that need to be investigated and increase algorithm efficiency.

## Forward Jumping with Backtracking

In order to eliminate conflicting values as soon as feasible, forward jumping requires moving constraint

information forward in the search space. When a value is assigned to a variable that has a constraint connection with another variable in a sudoku problem, the variable's range of potential values is updated. For instance, the potential values for other vacant cells in that row or column can be modified to exclude the assigned value whenever a value is assigned to a cell in a row or column. We use backtracking here just to ensure that in case the algorithm does not work we can always backtrack to find the next possible value.

**Code snippet:**

```
def forward_jumping():
    # Find the next empty cell with the fewest possible values
    min_values = 10
    new_val=0
    for row in range(9):
        for col in range(9):
            if puzzle[row][col] == 0:
                values = get_possible_values(row, col)
                if min_values > len(values):
                    min_values = len(values)
                    min_row, min_col,new_val= row, col,values
    # If there are no empty cells, the puzzle is solved
    if min_values == 10:
        return True
    # Try each possible value for the cell

    for value in new_val:
        global ForwardIterartion
        ForwardIterartion+=1
        #print(ForwardIterartion)
        puzzle[min_row][min_col] = value

        # Check if the puzzle can be solved from here
        if forward_jumping():
            return True
        # If not, backtrack and try the next value
        ForwardIterartion+=1
        puzzle[min_row][min_col] = 0
```

**Explanation:** The function **'forward_jumping()'** finds the next empty cell with the fewest possible values, and tries each possible value for that cell. It then checks if the puzzle can be solved from that point using recursion. If it can't, it backtracks and tries the next possible value until a solution is found or all

possibilities have been exhausted. It also uses a global variable ForwardIteration to count the number of iterations of the algorithm. Sample solution:

For Easy Solution

[4, 5, 9, 1, 2, 7, 8, 6, 3]
[7, 3, 6, 4, 8, 9, 2, 1, 5]
[8, 2, 1, 5, 3, 6, 4, 7, 9]
[1, 8, 4, 7, 9, 5, 3, 2, 6]
[9, 7, 2, 3, 6, 1, 5, 8, 4]
[3, 6, 5, 8, 4, 2, 7, 9, 1]
[6, 9, 7, 2, 5, 4, 1, 3, 8]
[2, 4, 8, 6, 1, 3, 9, 5, 7]
[5, 1, 3, 9, 7, 8, 6, 4, 2]
Time taken: 2.6e-05 seconds
The number of Iterations for BackTracking= 8771119

For Hard Solution

[5, 8, 3, 9, 4, 2, 6, 7, 1]
[2, 1, 4, 6, 3, 7, 8, 5, 9]
[6, 9, 7, 8, 5, 1, 4, 2, 3]
[7, 4, 8, 2, 1, 6, 3, 9, 5]
[9, 3, 5, 4, 7, 8, 2, 1, 6]
[1, 2, 6, 3, 9, 5, 7, 4, 8]
[3, 6, 1, 7, 2, 9, 5, 8, 4]
[4, 5, 2, 1, 8, 3, 9, 6, 7]
[8, 7, 9, 5, 6, 4, 1, 3, 2]
Time taken: 2.1e-05 seconds
The number of Iterations for BackTracking= 7671103

**Drawbacks:** Forward jumping effectively prunes the search space by lowering the number of potential solutions that must be investigated. The success of this algorithm is, however, dependent on the caliber of the constraint propagation rules and the heuristics employed to choose the subsequent variable to be given a value.

## Arc Consistency (AC3) [with recursive backtracking]

When all potential values of a variable in a CSP are consistent with at least one value of the variable's neighbors, the variable is said to have arc consistency. This means that in Sudoku puzzles, each empty cell's potential values must match at least one value in the same row, column, and 3x3 subgrid.

This algorithm(AC1) uses arc consistency by looking at pairs of variables in the CSP and removing any inconsistent values from the set of potential values for the first variable. The repetition of this procedure for each pair of variables continues until no more values may be discarded. Whereas, AC3 is just a more effective extension of AC1 by only taking into account pairs of variables that may be inconsistent. To do this, arcs (variable pairs) that can become inconsistent are kept in a queue and only examined when arc consistency is being checked. By enforcing arc consistency with recursive backtracking, we can reduce the number of recursive calls needed to find a solution, and avoid considering many invalid assignments.

**Code snippet:**

```
def arc_consistency(CSP, revise = revise_sudoku):
    queue = [(Xi, Xj) for Xi in CSP.get("variables") for Xj in CSP.get("neighbors").get(Xi)]

    while len(queue) != 0:
        Xi, Xj = queue.pop(0)

        revise_state, new_CSP = revise(CSP, Xi, Xj)
        if revise_state: # the domain of Xi has been changed
            if len(new_CSP.get("domains").get(Xi)) == 0:
                return False, new_CSP

            for xk in new_CSP.get("neighbors").get(Xi):
                if xk != Xj:
                    queue.append((xk, Xi))

            CSP = new_CSP

    return True, CSP


def solve(grid_string, multipleSolutions = False):
    CSP = generate_CSP(grid_string)
    assignment = generate_assignment(CSP)
    solutions = []

    is_arc_consistent, new_csp = arc_consistency(CSP)

    if (is_arc_consistent):
        recursive_backtracking(assignment, new_csp, solutions, multipleSolutions)

    return solutions, new_csp
```

**Explanation:** Arc consistency on the CSP is done via the arc_consistency function using the revise_sudoku function. The CSP iterates over pairs of variables, removing values from the first variable's domain that are in conflict with the second variable. The function adds all of the first variable's neighbors (except from the second variable) to the queue for further inspection if the first variable's domain is narrowed. The function returns False if a variable's domain is made empty. As a result, the solution function here invokes the recursive_backtracking function, which tries to assign values to the remaining variables iteratively. The function backtracks and tries another value for the previous variable if an assignment results in a contradiction (i.e., no values can be assigned to a variable).

For Easy puzzle

1 9 8 5 4 3 7 2 6
6 4 3 2 7 8 5 9 1
5 2 7 6 1 9 8 4 3
9 1 4 7 3 5 2 6 8
8 7 6 1 9 2 4 3 5
2 3 5 4 8 6 1 7 9
4 6 2 3 5 1 9 8 7
3 8 1 9 2 7 6 5 4
7 5 9 8 6 4 3 1 2
Time taken: 1.583664

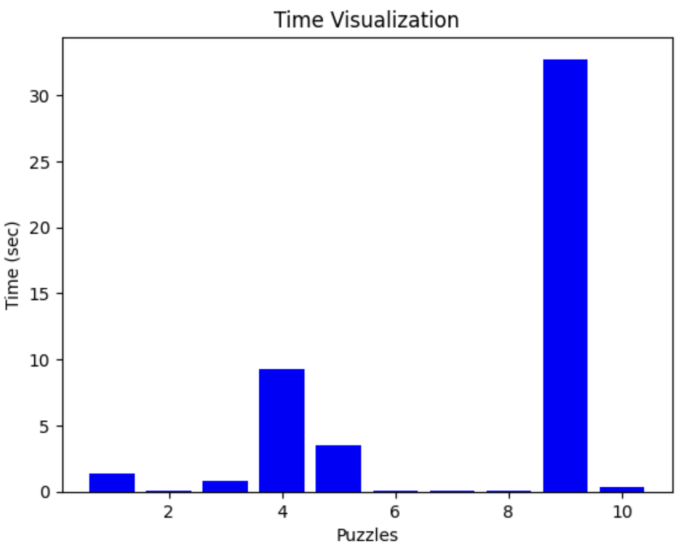
For Hard puzzle

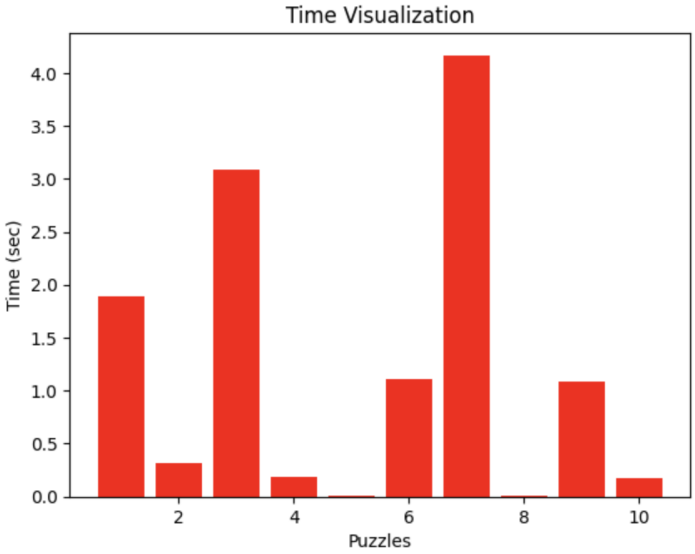
4 5 1 2 6 8 7 9 3
6 3 9 4 7 5 1 2 8
2 8 7 3 9 1 4 5 6
9 7 6 8 4 2 3 1 5
1 2 8 6 5 3 9 4 7
3 4 5 7 1 9 6 8 2
7 1 2 5 3 4 8 6 9
8 9 3 1 2 6 5 7 4
5 6 4 9 8 7 2 3 1
Time taken: 3.784045

**Drawbacks:** This method is considered to be the best optimal solution finder among all. However, this algorithm may not be effective for all types of Sudoku puzzles and may require additional heuristics and optimizations to be used effectively. Also, more than one solution might exist as the function might iterate for more than once.

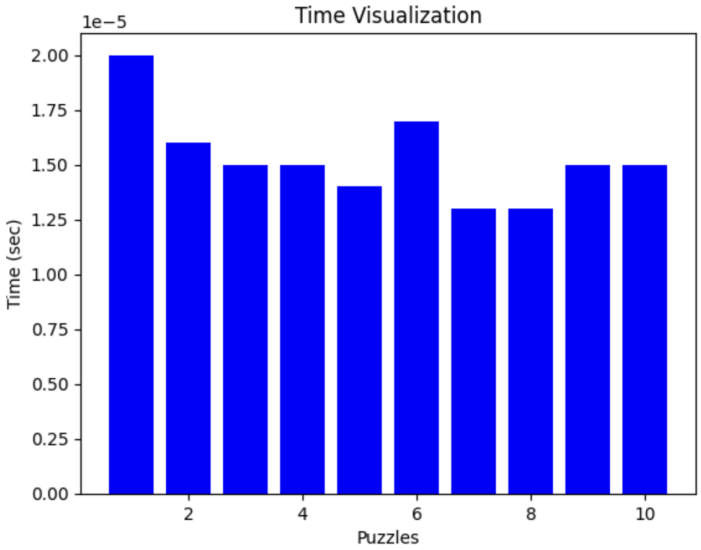
## Comparative Analysis: Time Visualization (Blue for Easy and Red for Hard)

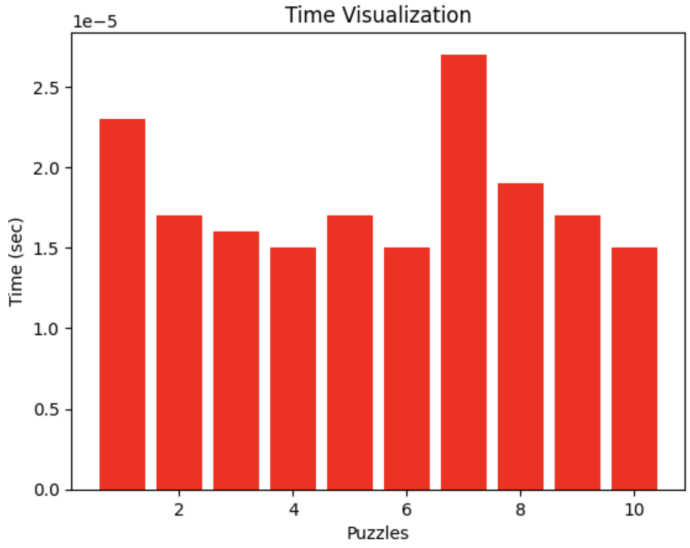
### Backtracking



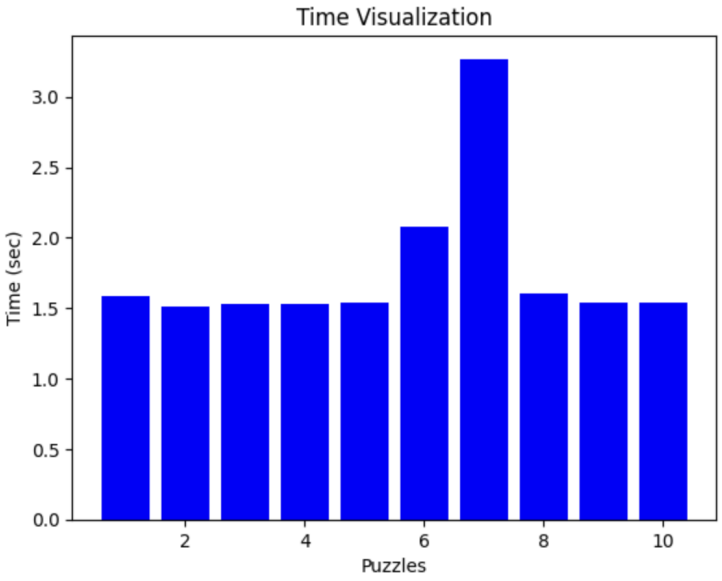



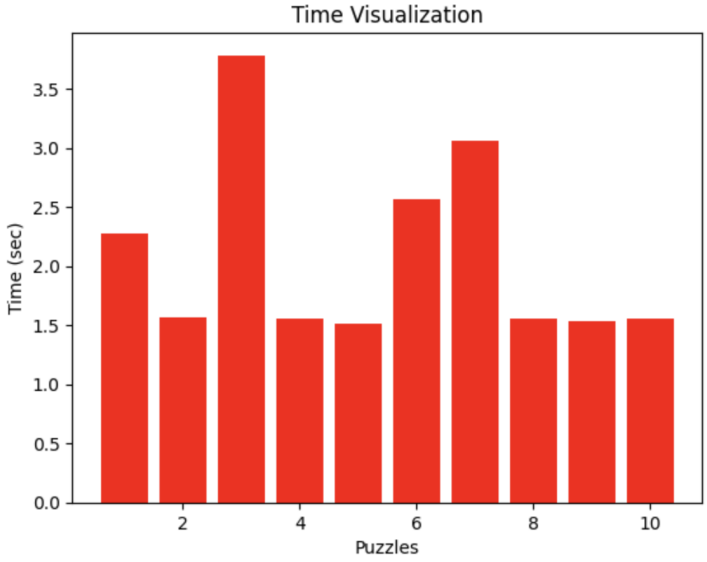

### Backtracking with Forward Jumping



**Arc Consistency with recursive Backtracking**




## Analysis Discussion & Conclusion

Let us look at the graphs above where time taken to find the solutions is compared for each algorithm and visualized using blue (easy) and red (hard) color. Points to be noted are:

1. Backtracking alone takes too long (up to 30 sec) to find even simple solutions and is not consistent at all.
2. We see a more consistent pattern in backtracking with forward jumping and the time taken is reduced to a few microseconds for calculating both easy and hard solutions.
3. Arc consistency is consistent too but has comparatively taken more time to calculate solutions for both kinds of puzzles. One thing to notice is that there is not much of a difference between both solutions for different kinds.

**Conclusion:** When the answer is discovered early in the search, backtracking is quite effective, but as the search space increases and the constraints become strict, it looks to be very slow. However, backtracking with forward jumping is found to be more effective than straight backtracking because it shrinks the

search space by removing options that lead to dead ends. Arc consistency combined with recursive backtracking is far more effective and consistent as it reduces the search space by eliminating choices that are guaranteed to be invalid and produces a solution quickly. However it becomes a little complex to implement. If we consider choosing an algorithm to be used in the industry for such kinds of sudoku puzzles, I think the best bet would be to use backtracking with forward jumping as it is simple to make , implement and if we look at the time, it performs amazing. Arc consistency is also an option but due to its complexity and multiple solutions we can think of a better way of using a simple backtracking coupled with forward jumping that will be easy to understand and learn.

## References


- https://github.com/nickgerold/ConstraintSatisfaction/blob/master/ConstraintSatisfaction.py
- https://www.geeksforgeeks.org/sudoku-backtracking-7/
- https://stackoverflow.com/questions/48905127/importing-py-files-in-google-colab
- https://www.kaggle.com/datasets/radcliffe/3-million-sudoku-puzzles-with-ratings/code?resource=download
- https://github.com/Amirhossein-Rajabpour/Constraint-Satisfaction-Problems/blob/main/Propagation.py