Introduction to Artificial Intelligence

Fall 2023

Sudoku Solver Using Backtracking & Arc Consistency

Farah Alaa El Din Khattab - 7487
Mariam Khaled Abdel Hakim - 7528

1. Introduction

This report outlines the development of a Sudoku solver utilizing backtracking and arc consistency. Sudoku, a popular number-placement puzzle, presents a challenge that demands logical deduction and systematic reasoning. The rules of Sudoku require filling a 9x9 grid with digits from 1 to 9, ensuring each row, column and 3x3 sub grid contains unique numbers.
The aim of this project is to create an efficient solver that navigates the puzzle space using backtracking, a systematic search algorithm, while employing arc consistency to refine constraints and optimize computational efficiency. By combining these methodologies, the solver aims to provide accurate and swift solutions to Sudoku puzzles of varying complexities.

2. Game Description

The game has 3 modes:

- **MODE 1:**
  In mode 1 the program generates a solvable sudoku puzzle and solves it.
- **MODE 2:**
  In mode 2 the user inserts the puzzle himself for the program to solve.
- **MODE 3:**
  In mode 3 the program generates a solvable sudoku puzzle for the user to solve or the user can make the program solve it.

3. Algorithms Used

3.1. Backtracking

```
function BACKTRACKING_SEARCH(csp) returns a solution, or failure
    return BACKTRACK({}, csp)

function BACKTRACK(assignment, csp) returns a solution, or failure
    if assignment is complete then return assignment
    var = SELECT_UNASSIGNED-VARIABLE(csp)
    for each value in ORDER_DOMAIN_VALUES (var, assignment, csp)
        if value is consistent with assignment then
            add {var = value} to assignment
            result = BACKTRACK(assignment, csp)
            if result ≠ failure then return result
        remove {var = value} from assignment
    return failure
```

3.2. Arc consistency

function AC-3(csp)
returns False if an inconsistency is found, True otherwise
inputs: csp, a binary CSP with components (X, D, C)
local variables: queue, a queue of arcs, initially all the arcs in csp
while queue is not empty **do**
    (X_i, X_j) = REMOVE-FIRST(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

4. Data Structures Used

The following part will explain any data structures used within the code:

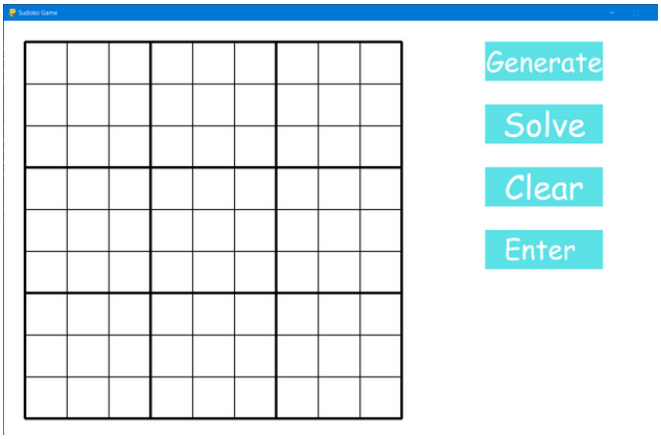
- **The Game Grid**: The game grid itself is represented as a **2D NumPy array** of size (9x9), where each element corresponds to a cell in the sudoku grid.
- The **values in the 2D array** belong to [0,9]
  - **0**: for empty spaces
  - **[1,9]**: for the puzzle that will be solved
- **Constants**: All constants in the game are present in a separate “Constants.py” file, these constants include:
  - **Colors**: specified as RGB values as (R, G, B) each in hexadecimal
  - **Fonts**: Font names are declared
  - **Sizes** and Dimensions: Screen, window, and shape dimensions are declared

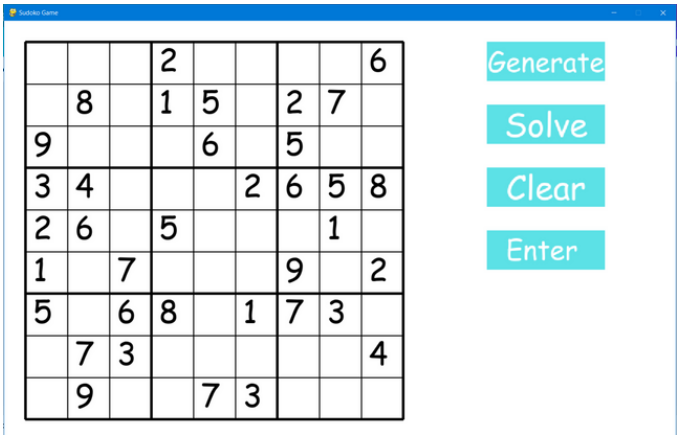
  “Constants.py” file also includes some static functions that are used in most files.
- **GUI Objects**: They are all PyGame shapes
  - **Buttons**: A Button Class is created, clicking and hovering mouse events are handled in the class, and each Button is a PyGame rect.
  - **Input**: The input is taken from as a keyboard event in PyGame.
  - **Grid**: Board is drawn using black lines from PyGame
- **Grid Cells**: A Class "Cell" is created to be used in arc consistency. The class has
  - **row:** the row that the cell belongs to
  - **col:** the column that the cell belongs to
  - **domain:** cell domain (all possible values for the cell)
  - **box:** the box that the cell belongs to

5. Game GUI

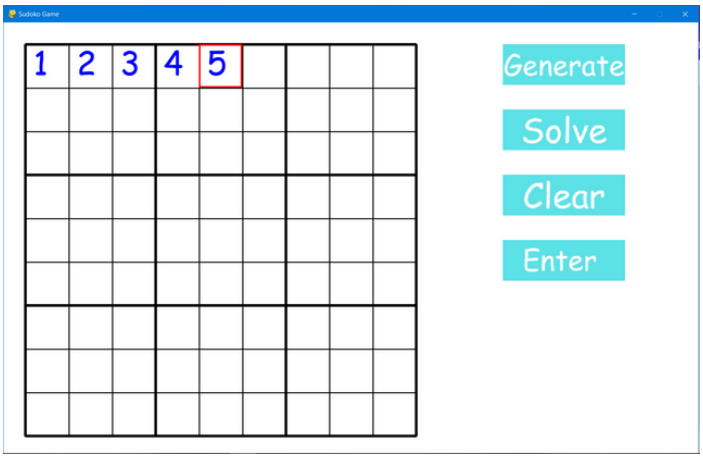
Our game consists of one window

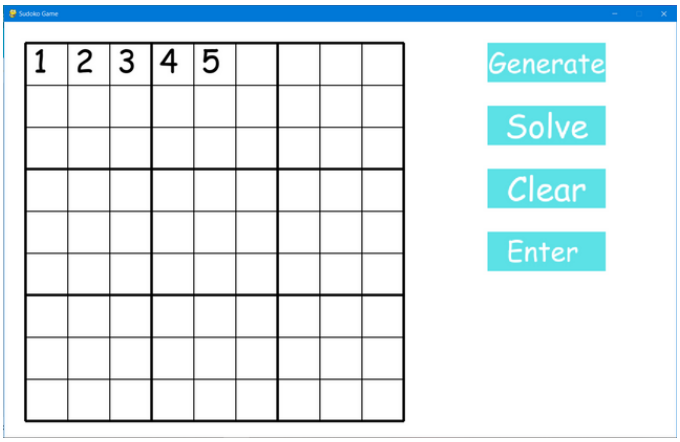
The grid starts empty until the user press generate button





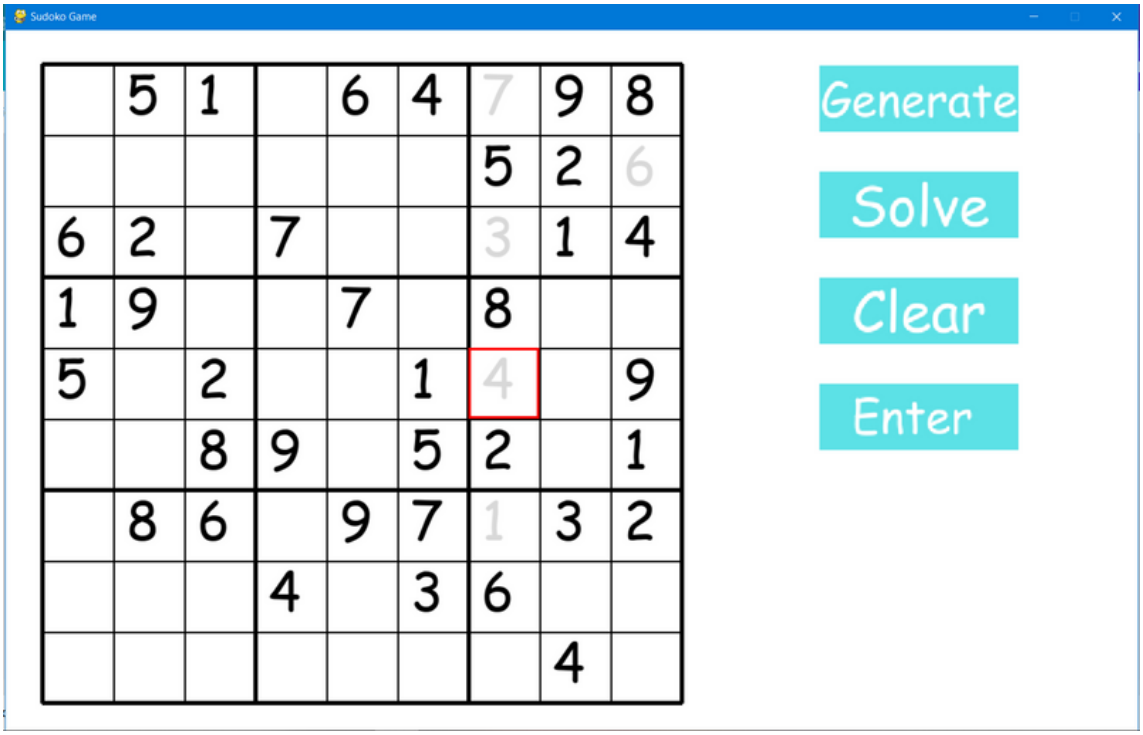
or the user enters the game grid and press enter.



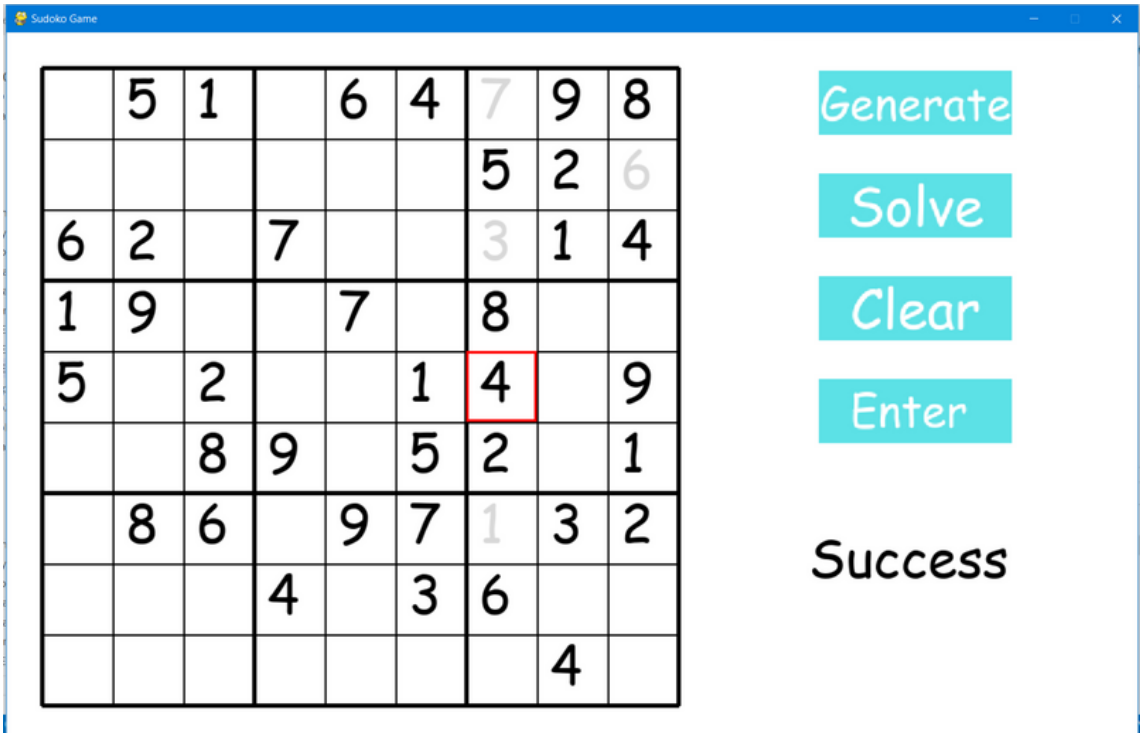


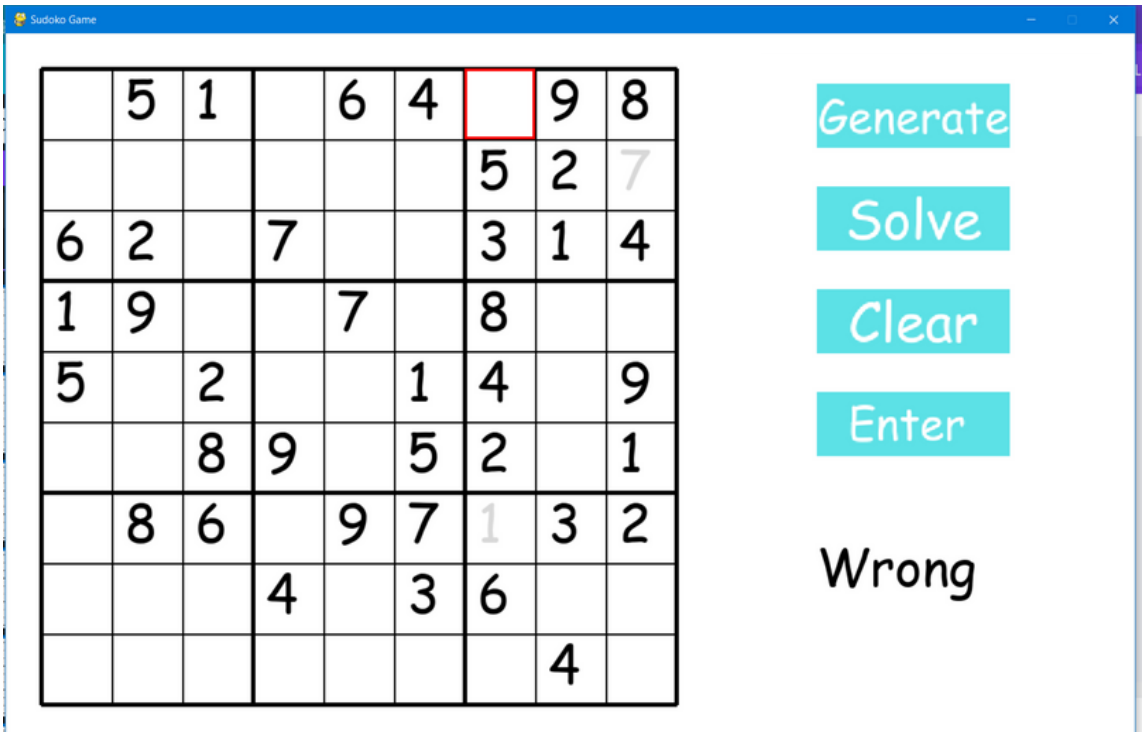
If the user presses clear before enter the grid will be empty.

The user can write temporary numbers in the grid while solving. The user can press clear to delete all temporary values.

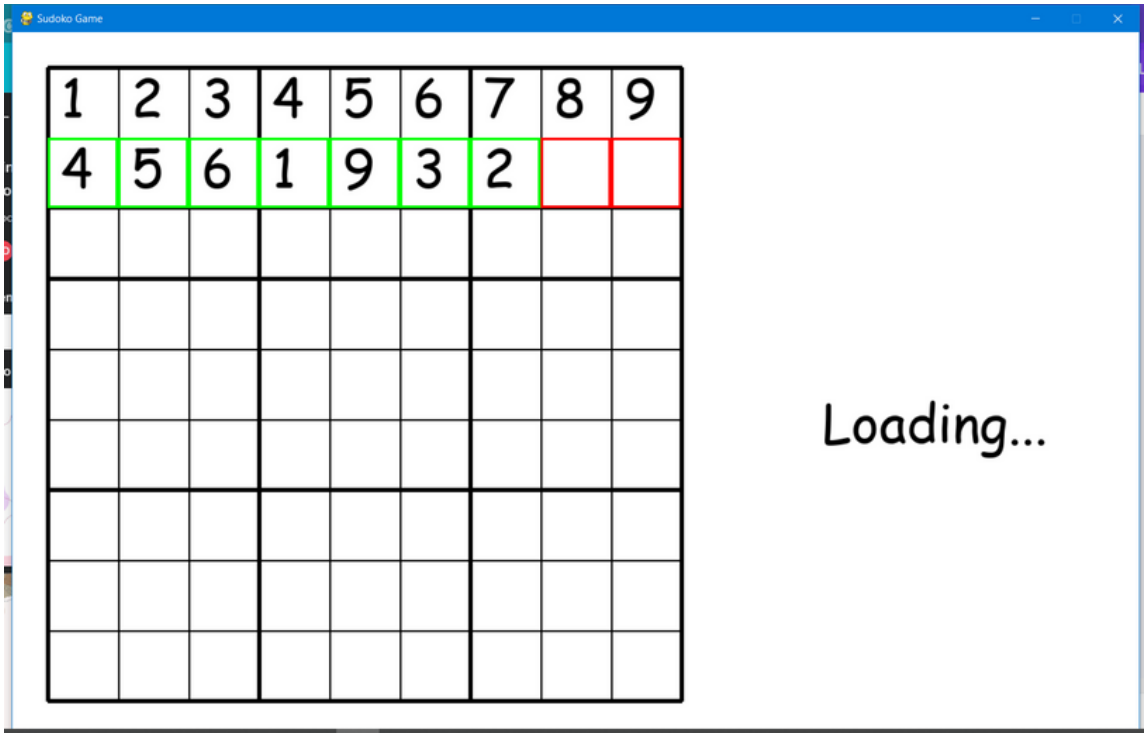


when the user press enter while a cell is selected and has a temporary value the program shows whether it's correct or not and clears the cell if it's wrong.





and the user could press solve for the program to solve the puzzle.



6. Assumptions

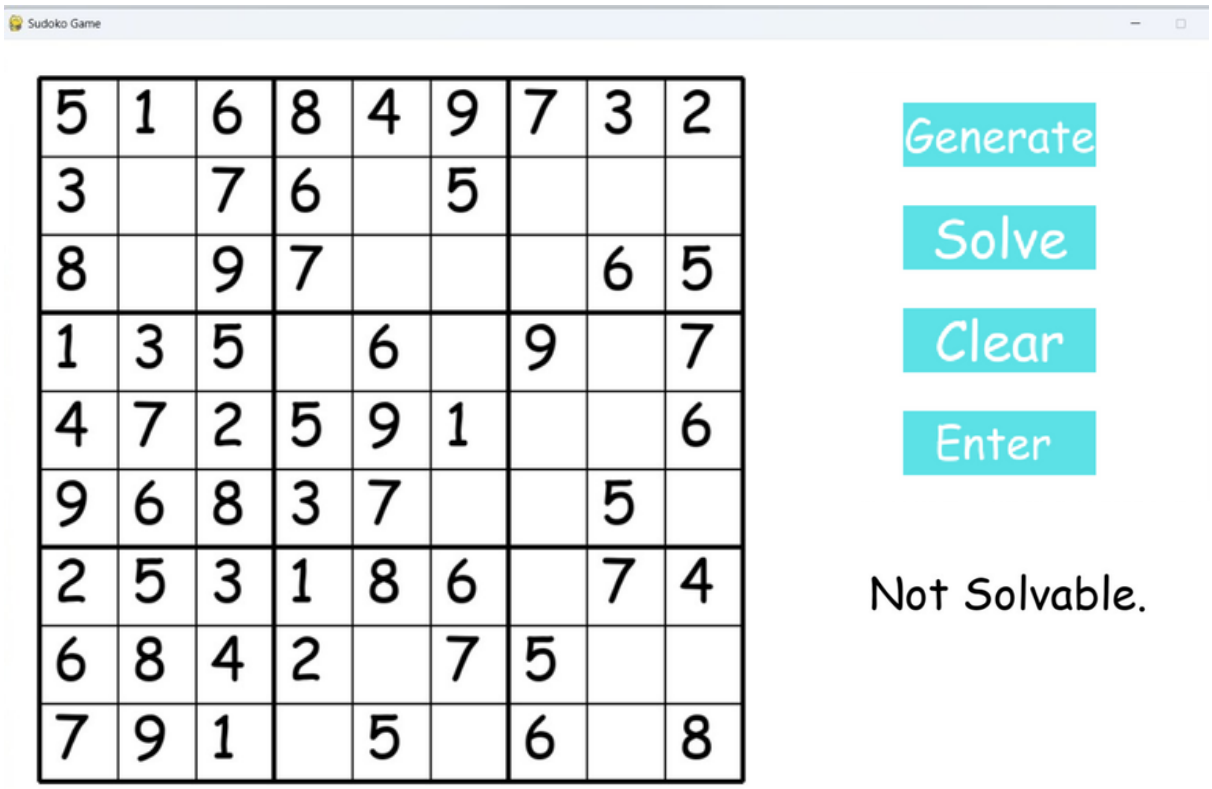
Here are the assumptions made throughout the code:

- **Grid Dimensions**: The grid has 9 rows and 9 columns (9x9 Board).
- **Screen Size**: The screens have a fixed size of (1280x800) pixels.
- **Consistency in GUI Display**: The GUI accurately reflects the state of the game and shows backtracking while the program solves the puzzle.

9. Sample Runs

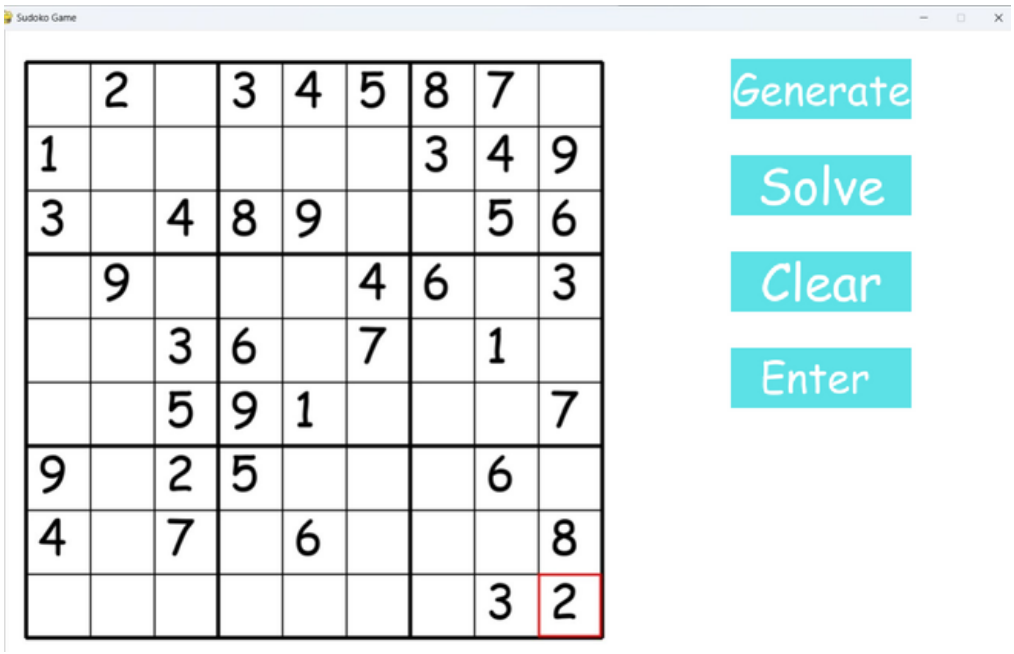
SAMPLE RUN #1

Unsolvable

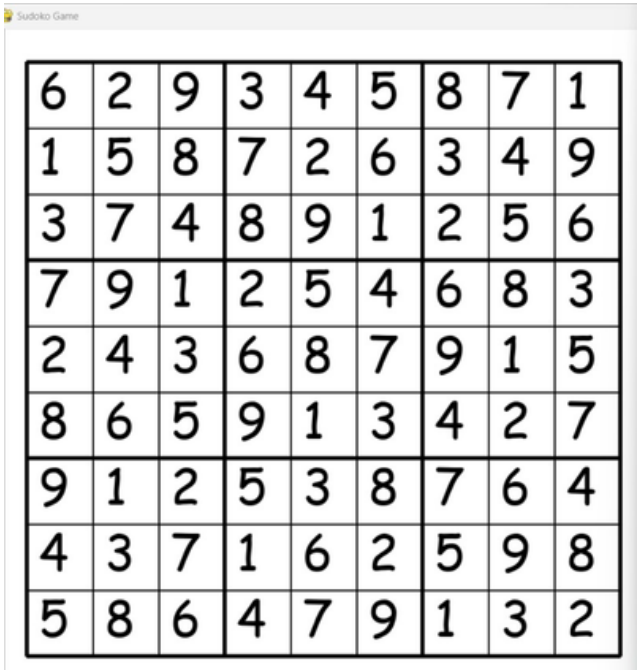


SAMPLE RUN #2

Easy



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



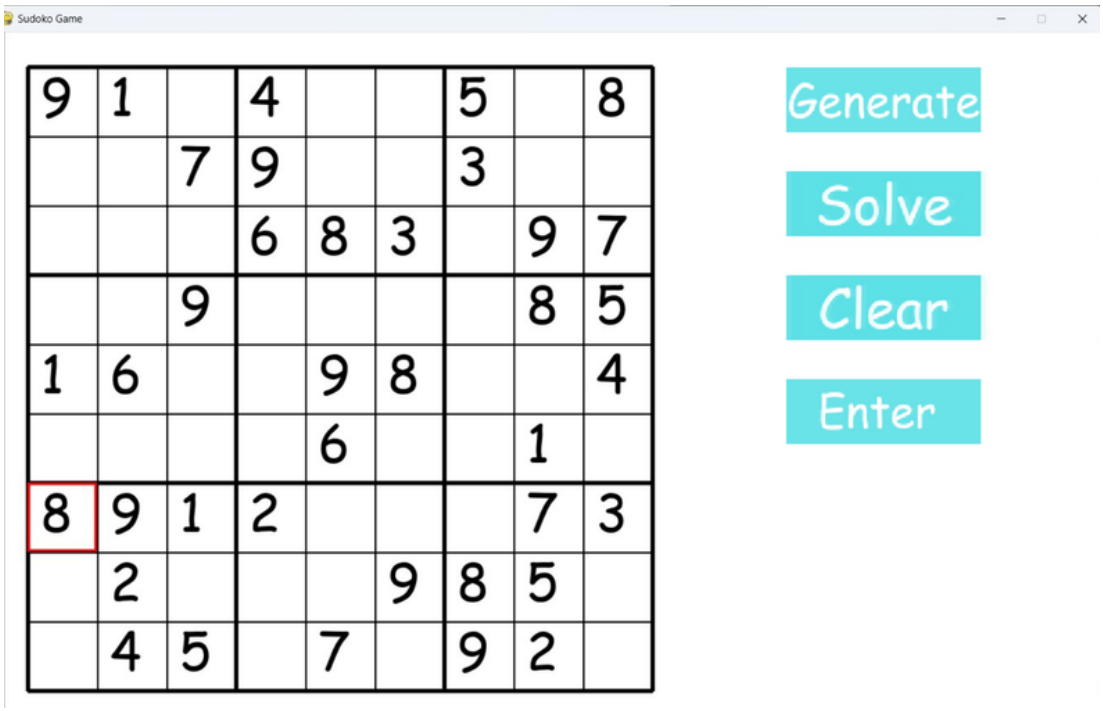
6	2	9	3	4	5	8	7	1
1	5	8	7	2	6	3	4	9
3	7	4	8	9	1	2	5	6
7	9	1	2	5	4	6	8	3
2	4	3	6	8	7	9	1	5
8	6	5	9	1	3	4	2	7
9	1	2	5	3	8	7	6	4
4	3	7	1	6	2	5	9	8
5	8	6	4	7	9	1	3	2

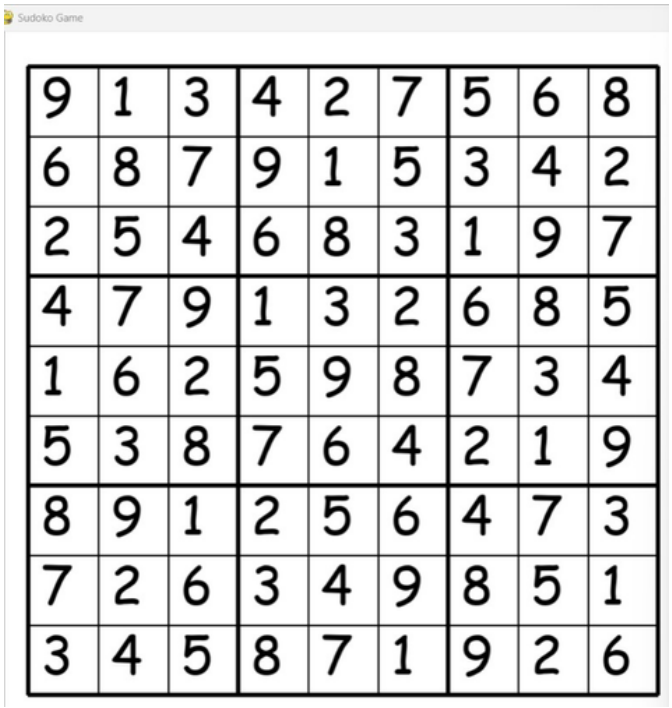
```
Step 344:
 Queue Size: 1622
 Arc: (4 , 6) -> (7 , 6)
 Domains: [5, 9] -> [5]
 Remove (5) From (4 , 6)
 New Domain: [9]

Elapsed time: 1.0351307392120361 seconds
```

SAMPLE RUN #3

Medium



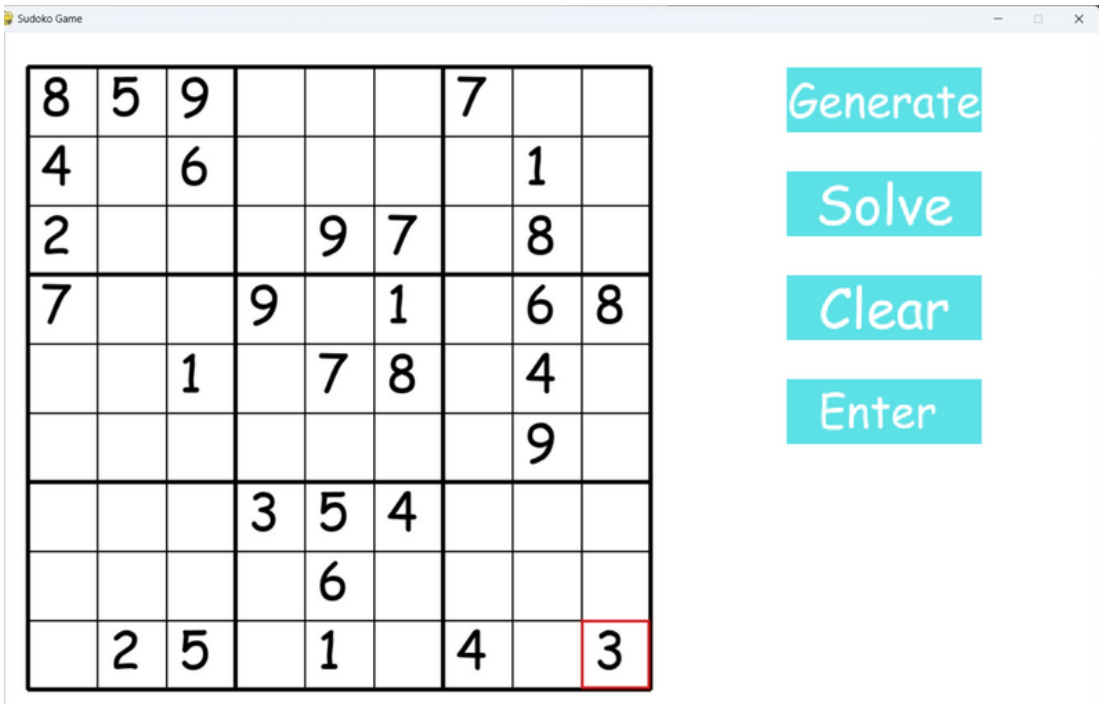


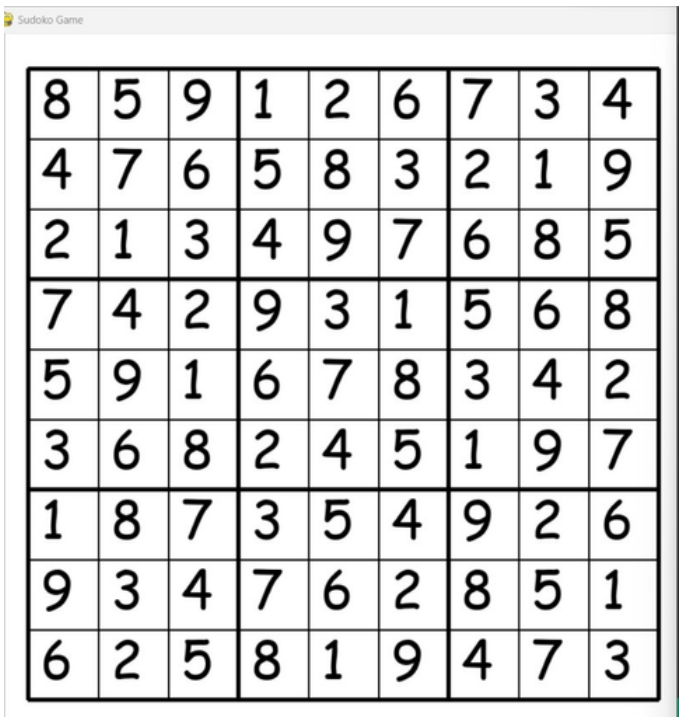

```
Step 344:
 Queue Size: 653
 Arc: (3 , 3) -> (5 , 3)
 Domains: [1, 7] -> [7]
 Remove (7) From (3 , 3)
 New Domain: [1]

Elapsed time: 1.1442701816558838 seconds
```

SAMPLE RUN #4

Hard






```
Step 342:
 Queue Size: 400
 Arc: (2 , 3) -> (0 , 5)
 Domains: [4, 5, 6] -> [6]
 Remove (6) From (2 , 3)
 New Domain: [4, 5]

Elapsed time: 2.017552137374878 seconds
```

10. Conclusion and Results Break Down

In conclusion, the implementation of a Sudoku solver using backtracking and arc consistency has proven to be an effective method in tackling the complexity of solving Sudoku puzzles. By employing backtracking, the algorithm systematically explores possible solutions, efficiently backtracking when encountering dead-ends, ensuring an optimal solution is reached. Arc consistency further enhances the solver's performance by reducing the search space, enabling faster elimination of inconsistent values. This combination of techniques not only provides a reliable means to solve Sudoku puzzles but also showcases the importance of algorithmic strategies in addressing challenging computational problems. The solver's ability to efficiently handle varying difficulty levels underscores its adaptability and robustness. Through this project, a deeper understanding of constraint satisfaction problems and their applications in problem-solving has been gained, highlighting the significance of leveraging different algorithms to optimize computational solutions. The successful implementation of this solver stands as a testament to the power of combining backtracking and arc consistency, offering a valuable contribution to the field of constraint-based problem-solving techniques.