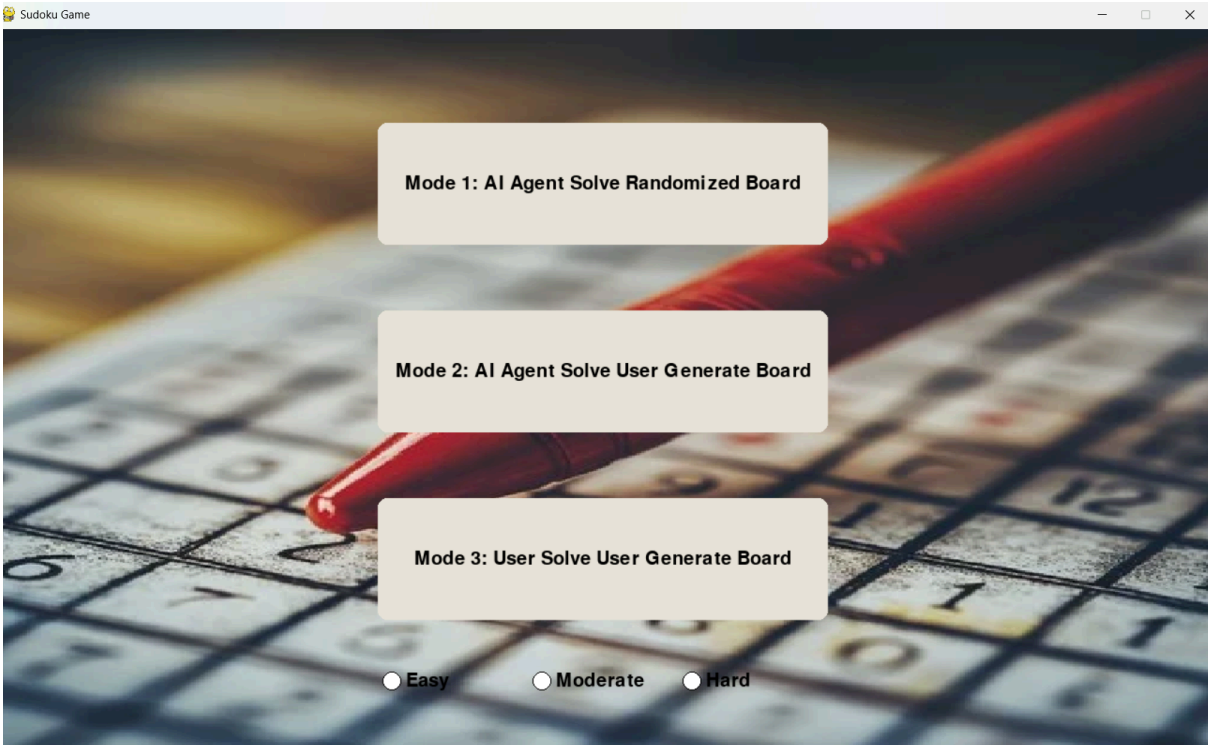
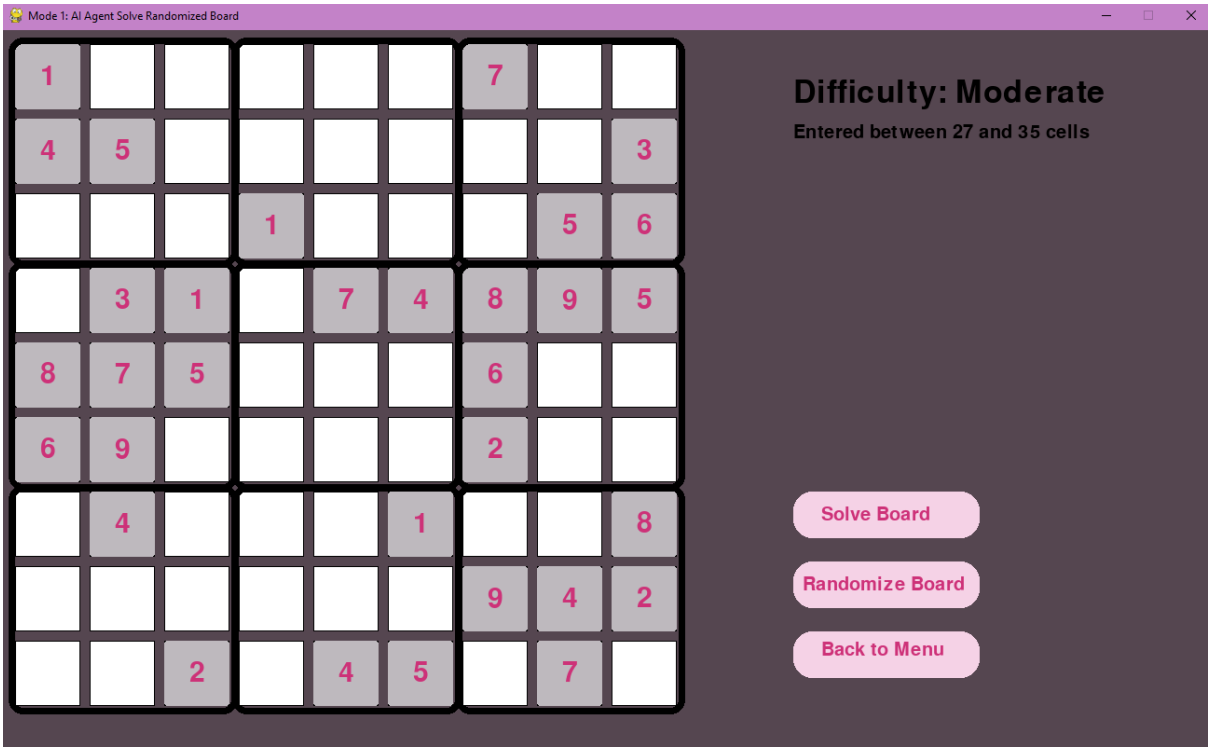
AI Lab (3)
CSP to solve Sudoku

Name	ID
Ranime Ahmed Elsayed Shehata	21010531
Rowan Gamal Ahmed Elkhouly	21010539
Karene Antoine Nassif	21010969





Data Structures Used:

1. **Board (2D list):**
    a. **Prototype:** `board = [[int for _ in range(9)] for _ in range(9)]`
    b. **Description:** Represents the Sudoku board. Each cell is an integer (`0` for empty cells, `1-9` for filled cells).
2. **Domains (2D list of lists):**
    a. **Prototype:** `domains = [[list(int) for _ in range(9)] for _ in range(9)]`
    b. **Description:** Stores the list of possible values (`1-9`) for each cell.
3. **Queue (List of tuples):**
    a. **Prototype:** `queue = [(int, int)]`
    b. **Description:** Stores coordinates of cells to process during the arc consistency algorithm.
4. **Domain Values (List):**
    a. **Prototype:** `domain_values = [int]`
    b. **Description:** Stores potential values for a specific cell (e.g., during MRV selection).
5. **Removed Values (List):**

a. **Prototype:** `removed_values = [int]`
b. **Description:** Stores values removed from a cell's domain during the arc consistency check.

How does the Algorithm work?

The Algorithm of our agent is split between 2 parts:

1. Randomization:

```
def generate_random_puzzle(difficulty):
    def remove_cells(board, num_to_remove):
        removed = 0
        while removed < num_to_remove:
            row, col = np.random.randint(0, 9, size=2)
            if board[row][col] != 0:
                board[row][col] = 0
                removed += 1
      You, 2 days ago • feat: game gui + AC algorithm applied
    filled_range = get_filled_cells_range(difficulty)
    num_filled = np.random.randint(*filled_range)

    board = np.zeros((9, 9), dtype=int)
    if not backtracking(board):
        print("Failed to generate a complete board.")
        logging.info("Failed to generate a complete board.")
        return None

    # Remove cells to match difficulty
    num_to_remove = 81 - num_filled
    remove_cells(board, num_to_remove)

    return board
```

This function generates a randomized Sudoku puzzle based on the specified difficulty level. The randomization involves the following steps:

1. Determine the Range of Pre-Filled Cells

```
filled_range = get_filled_cells_range(difficulty)
num_filled = np.random.randint(*filled_range)
```

- The function `get_filled_cells_range(difficulty)` returns a tuple `(min_filled, max_filled)` based on the difficulty level:
  - **Easy:** Between 36–45 pre-filled cells.
  - **Moderate:** Between 27–35 pre-filled cells.
  - **Hard:** Between 17–26 pre-filled cells.

```
def get_filled_cells_range(difficulty):
    if difficulty == "Easy":
        return 36, 45
    elif difficulty == "Moderate":
        return 27, 35
    elif difficulty == "Hard":
        return 17, 26
    else:      You, 2 days ago • feat: game g
        return 0, 81
```

- `np.random.randint(*filled_range)` selects a random number of cells to fill within the specified range.

2. Generate a Fully Solved Sudoku Board

```
board = np.zeros((9, 9), dtype=int)
if not backtracking(board):
    print("Failed to generate a complete board.")
    logging.info("Failed to generate a complete board.")
    return None
```

- A 9x9 empty board (all zeros) is initialized.
- The **backtracking** algorithm fills the board with valid values to create a complete (solved) Sudoku board. The randomization here comes from the backtracking logic itself:
  - **MRV (Minimum Remaining Values):** The cell with the fewest valid options is chosen.
  - **LCV (Least Constraining Value):** The algorithm tries numbers in the order that creates the least constraints for other cells.
  - Recursive backtracking ensures the board remains valid as it is filled.

3. Remove Cells to Match the Difficulty

```
num_to_remove = 81 - num_filled
remove_cells(board, num_to_remove)
```

- The number of cells to remove (`num_to_remove`) is calculated as the total cells (81) minus the number of pre-filled cells (`num_filled`).

The `remove_cells` function uses randomization:

```
def remove_cells(board, num_to_remove):
    removed = 0
    while removed < num_to_remove:
        row, col = np.random.randint(0, 9, size=2)
        if board[row][col] != 0:
            board[row][col] = 0
            removed += 1
```

  - Randomly selects a row and column using `np.random.randint(0, 9, size=2)`.
  - If the selected cell is already empty (`board[row][col] == 0`), it skips removing it and continues until the desired number of cells is removed.
- This random removal ensures that each generated puzzle is unique while adhering to the desired difficulty level.

<u>Supporting Randomization in the Backtracking:</u>

- **MRV and LCV Logic:**
  - For each unfilled cell, the backtracking algorithm sorts possible values by the "least constraining value." This ensures that even if multiple puzzles are generated with the same difficulty, the paths taken during backtracking differ due to varying constraints.
- **Recursive Nature:**
  - The depth-first search (DFS) approach of backtracking inherently introduces variability in the order of exploration.

Randomization Key Points:
Randomized Pre-Fill Count: The number of cells to pre-fill varies based on difficulty.

- **Randomized Cell Selection for Removal:** Cells to be emptied are randomly selected.
- **Randomized Backtracking Path:** The LCV and MRV heuristics ensure unique solutions even for identical inputs.

This combination of strategies ensures the generated puzzles are **unique** and adhere to the desired difficulty level.

---

2. Backtracking itself:

- First, we start by:

```
def backtracking(board, domains=None):
```

- It takes a board which is a must parameter, and an optional one for Domain.
- At the start of the algorithm, domains = None is checked **to make arc consistency right away.**

```
if domains is None:
        domains, steps = apply_arc_consistency(board)
```

- We then get the MRV cell (**variable with the minimum remaining values**)

```
mrv_cell = is_empty_cell(board, domains)
```

- We also **choose the least constraining value of its domain**, sorting by the number of constraints violated by each one and choosing them ordered ascendingly.

```
domain_values.sort(key=lambda num:
count_constrained_values(board, row, col, num))
```

- Now we have **the MRV variable** and its domain values ordered, we start:
    1. Looping over the domain values, and checking if `is_valid_move(board, row, col, num):`

a. If that returns False then we check the next domain value
b. If it returns True → We apply it first, then we check for Arc consistency, which returns the Domain or None (if not consistent)
    i. If consistent:

```
# Recursive call to fill the next cell --> DFS way
                if backtracking(board, new_domains):
                    return True
```

    ii. Not consistent: skip this and continue to the next iteration

2. If none of the inner calls of DFS returned True, then this board is unsolvable so, we return False

```
 print(f"No valid value found for cell ({row}, {col}).
Backtracking...")
    logging.info(f"No valid value found for cell ({row},
{col}). Backtracking...")
    return False
```

- As for **Arc Consistency Part:**
    1. We loop over the board and add any assigned element, because we want to check the consistency of arcs going to them, we add them to our queue:

```
if board[i][j] != 0:
                domains[i][j] = [board[i][j]]
                queue.append((i, j))
```

    2. We pop these cells out of the queue, one by one, then we check:
        a. Arcs between it and any cell sharing constraint with it in the same row
        b. Arcs between it and any cell sharing constraint with it in the same column
        c. Arcs between it and any cell sharing constraint with it in the same 3x3 grid
    3. Checking arc consistency: (looking for any value of the other cell's domain that doesn't violate this one's domain)

```
if isinstance(domains[xj[0]][xj[1]], list) and len(domains[xj[0]][xj[1]]) ==
```

```
1 and value in domains[xj[0]][xj[1]] :
```

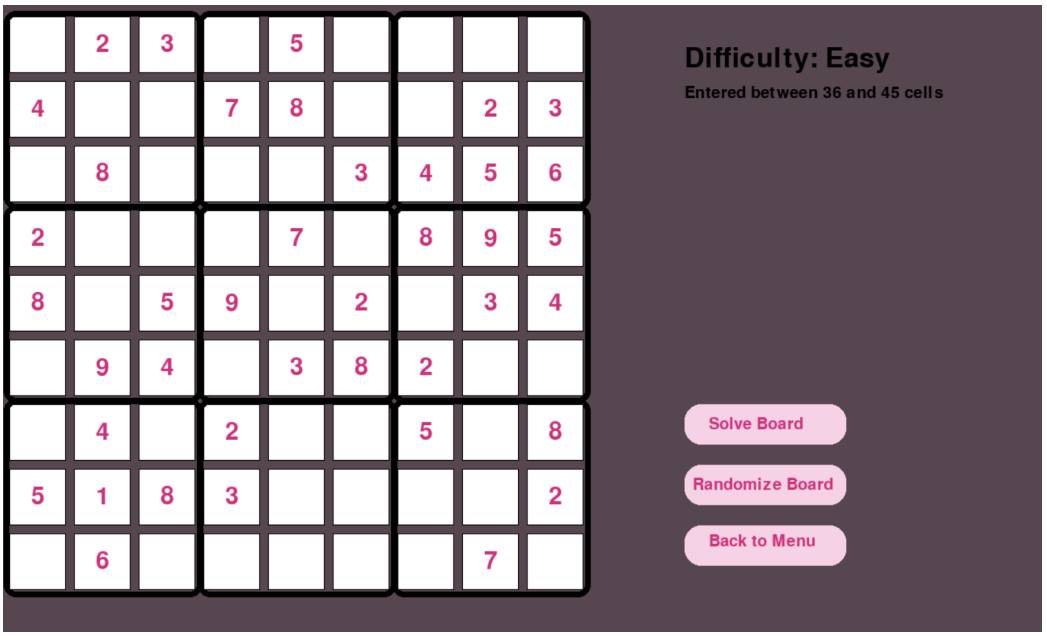
4. For any of these cases, We check **if any domain reduction happens to the cell:**
    a. We verify, that if the domain of any cell == 0, then we terminate because we reached an invalid/unsolvable state.
    b. Otherwise, We make sure **to add this cell to our queue**, to check all arcs going to it. (because its domain changed)

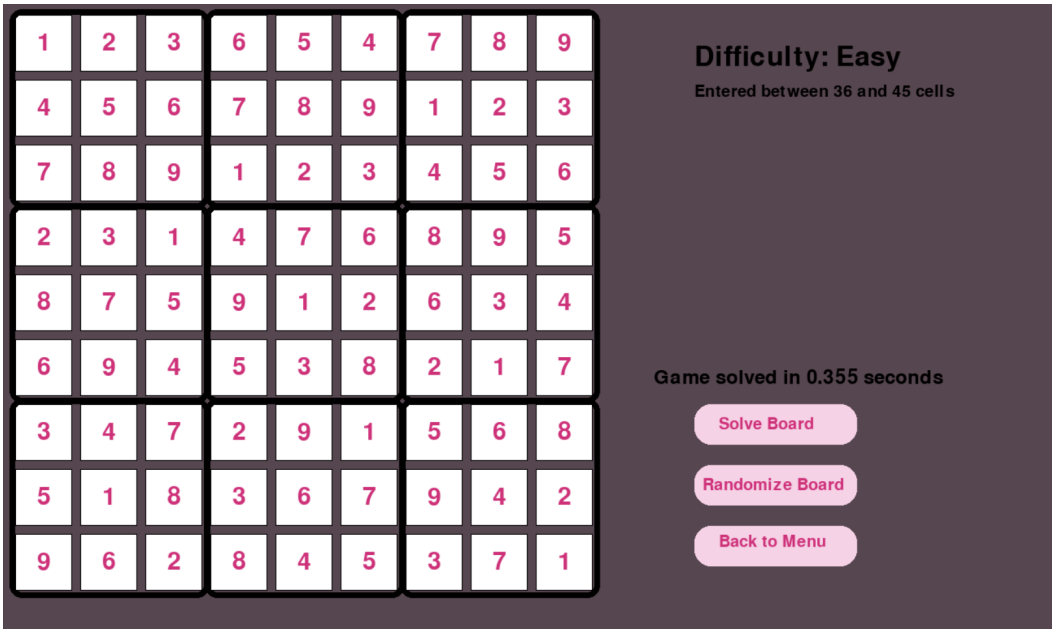
Time Analysis:

- **Easy Level:**

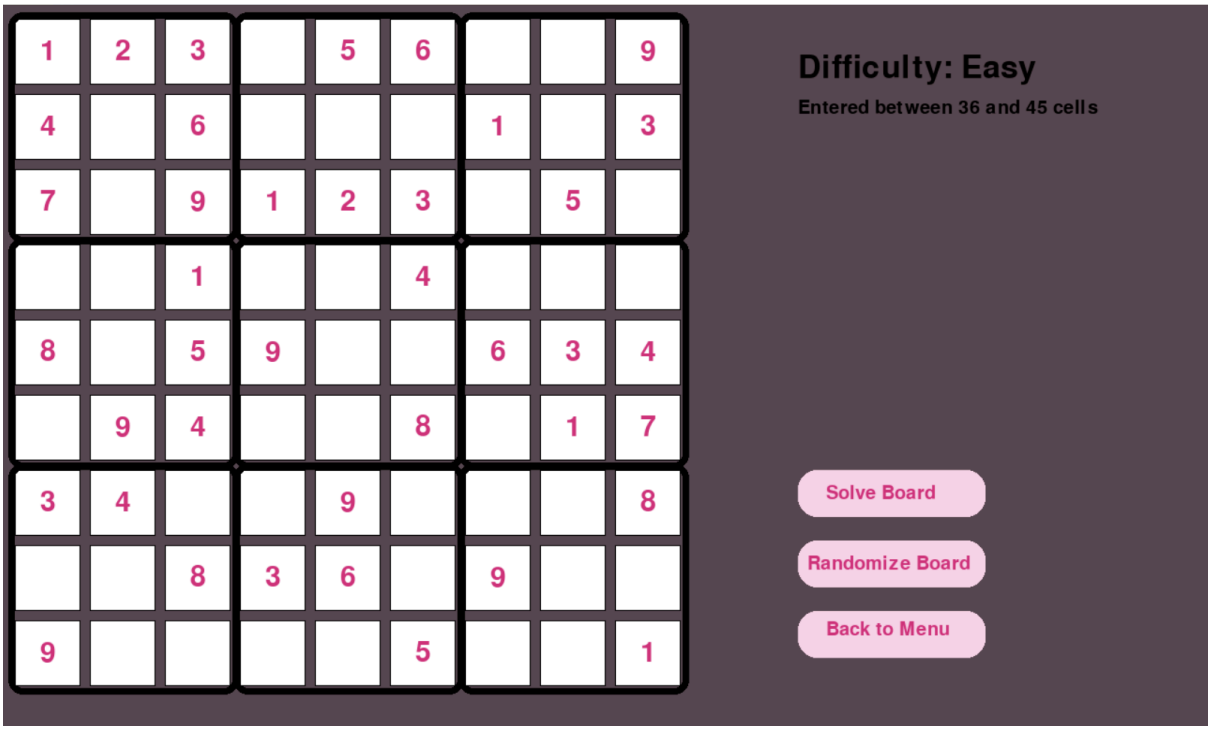
	Time Taken (Secs)
Sample 1	0.355
Sample 2	0.35
Sample 3	0.361

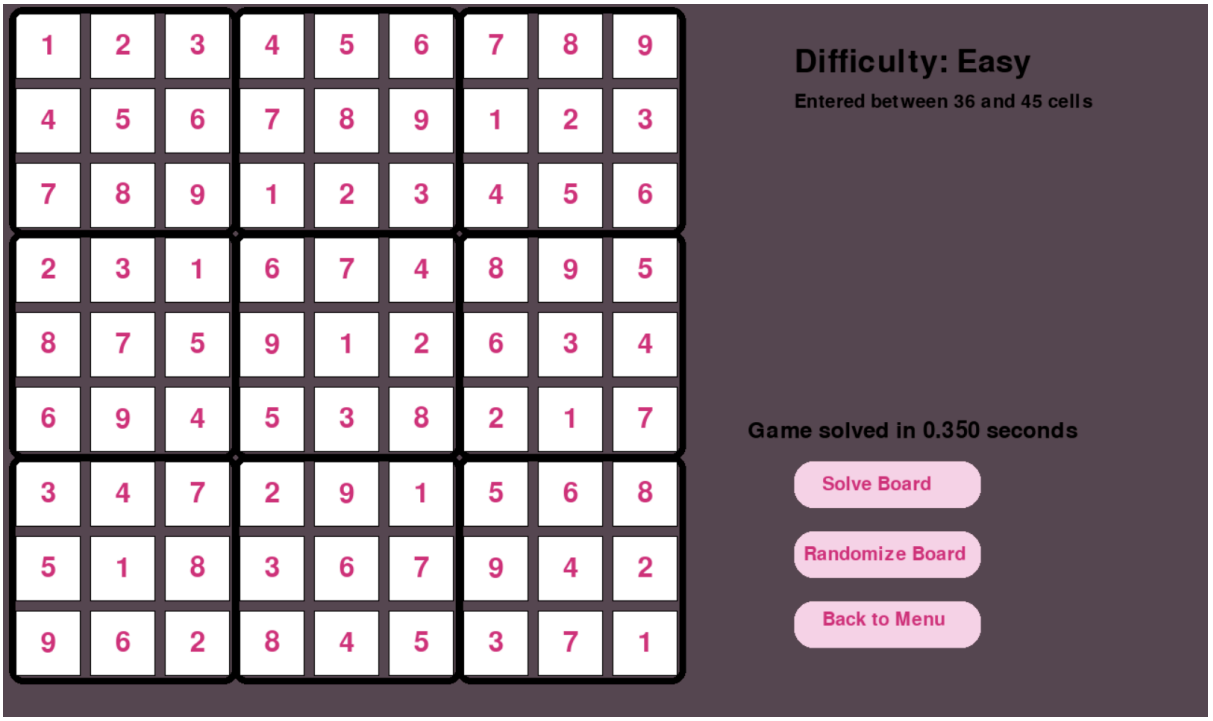
➔ **Sample 1:**



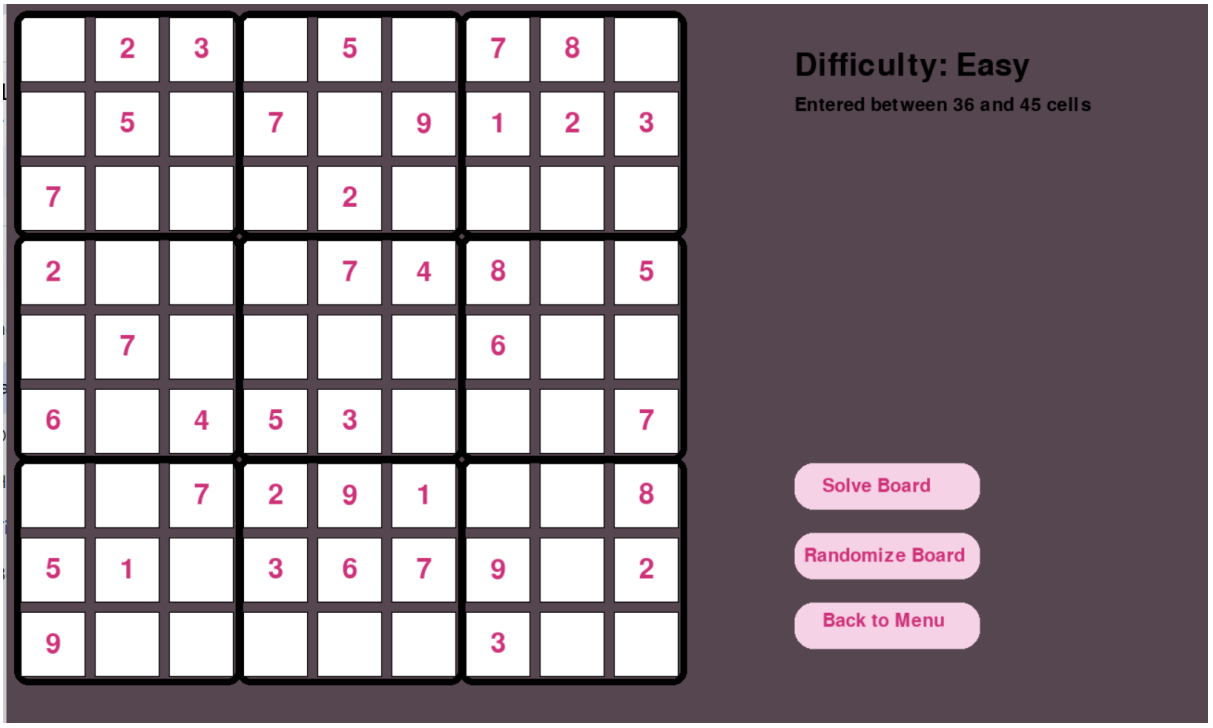


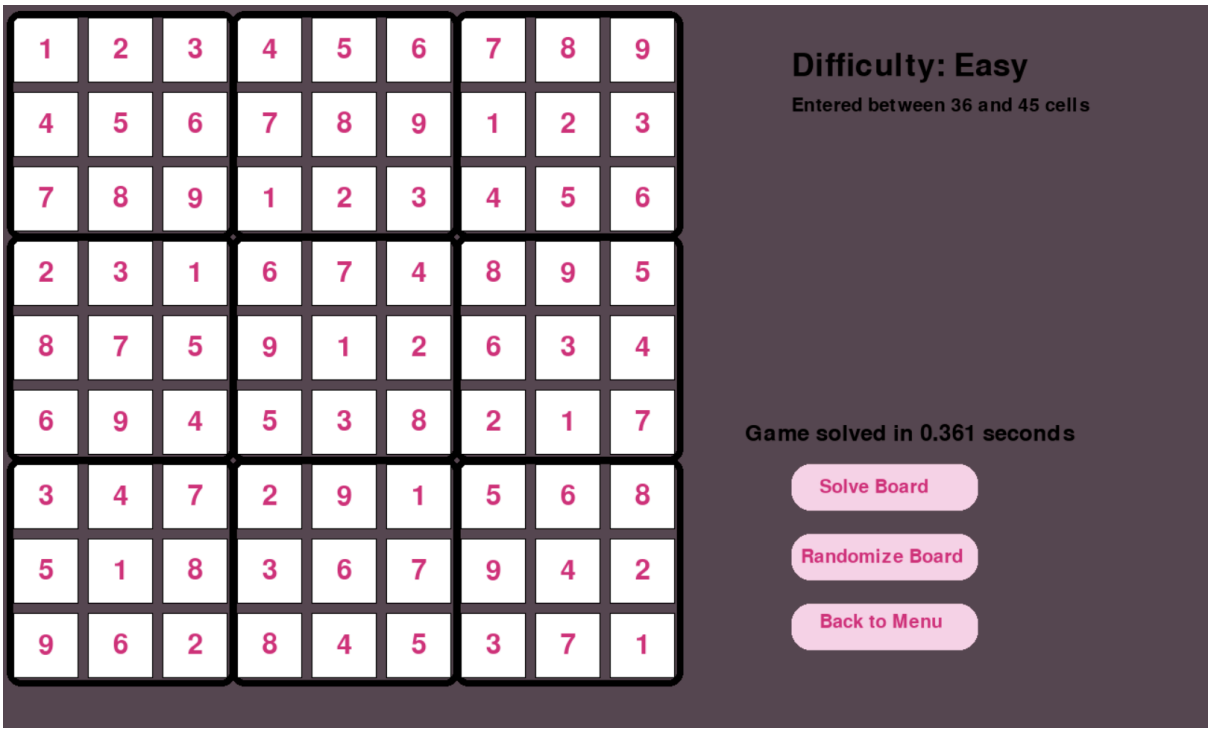
➔ **Sample 2:**


1 2 3 5 6 9
4 6 1 3
7 9 1 2 3 5
1 4
8 5 9 6 3 4
9 4 8 1 7
3 4 9 8
8 3 6 9
9 5 1
Difficulty: Easy
Entered between 36 and 45 cells
Solve Board
Randomize Board
Back to Menu


1 2 3 4 5 6 7 8 9
4 5 6 7 8 9 1 2 3
7 8 9 1 2 3 4 5 6
2 3 1 6 7 4 8 9 5
8 7 5 9 1 2 6 3 4
6 9 4 5 3 8 2 1 7
3 4 7 2 9 1 5 6 8
5 1 8 3 6 7 9 4 2
9 6 2 8 4 5 3 7 1
Difficulty: Easy
Entered between 36 and 45 cells
Game solved in 0.350 seconds
Solve Board
Randomize Board
Back to Menu

➔ **Sample 3:**

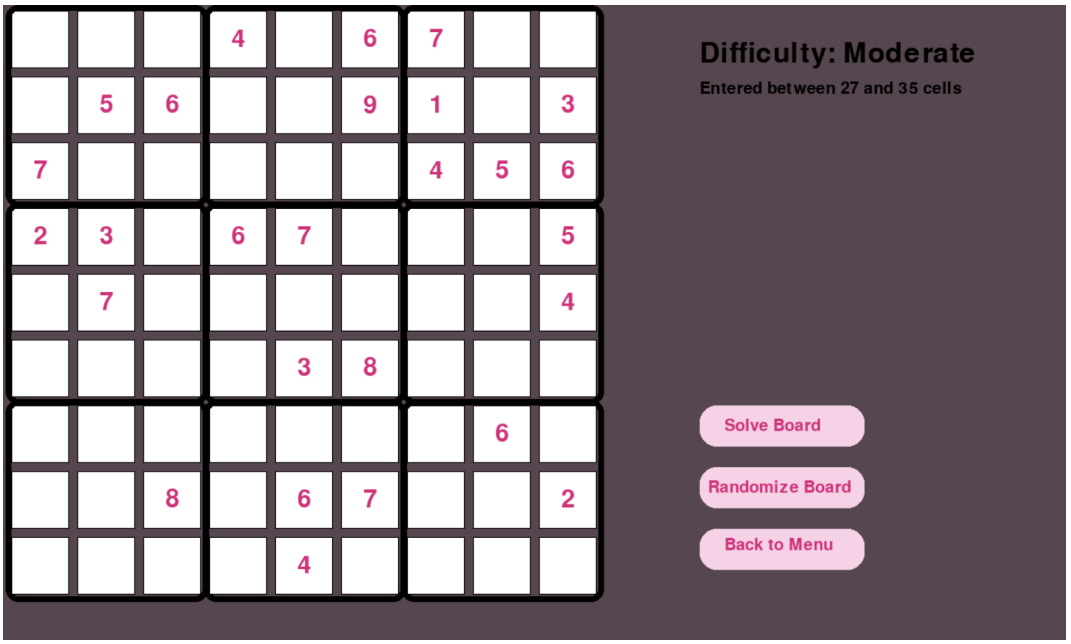

Difficulty: Easy
Entered between 36 and 45 cells
Solve Board
Randomize Board
Back to Menu

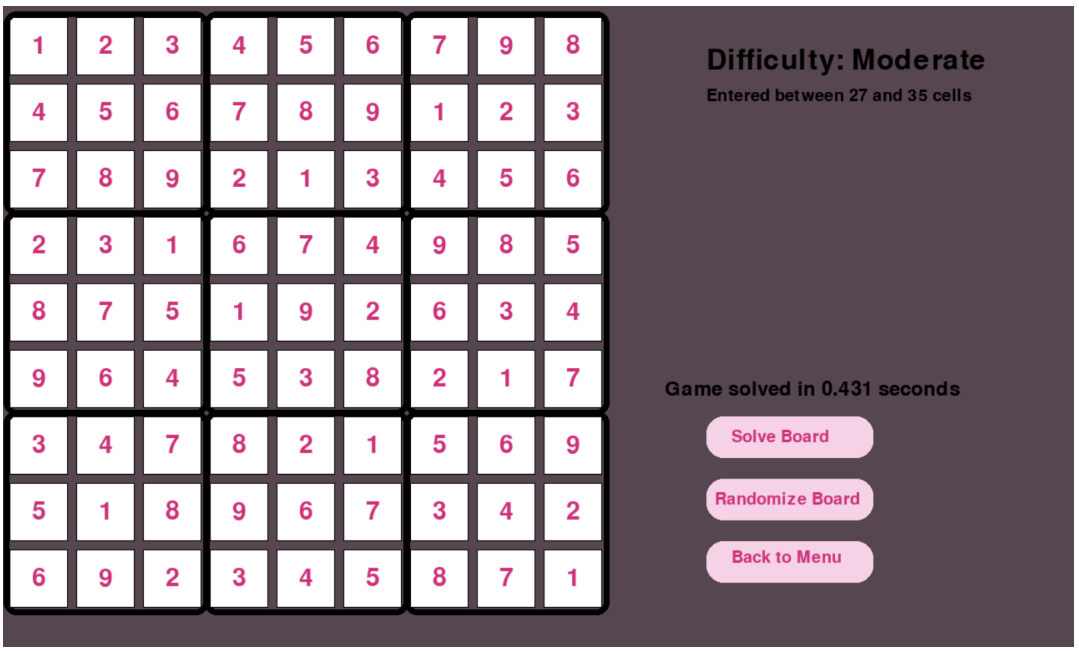

Difficulty: Easy
Entered between 36 and 45 cells
Game solved in 0.361 seconds
Solve Board
Randomize Board
Back to Menu

- **Moderate Level:**

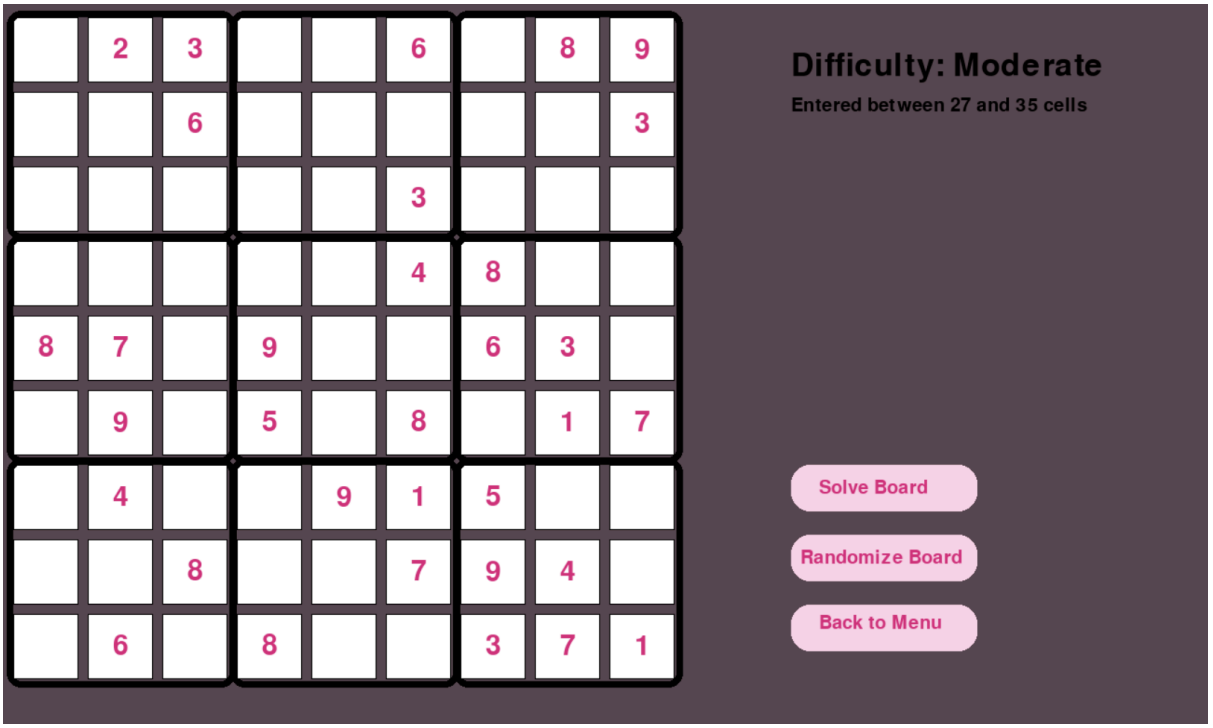
	Time Taken (Secs)
Sample 1	0.431
Sample 2	0.413
Sample 3	0.434

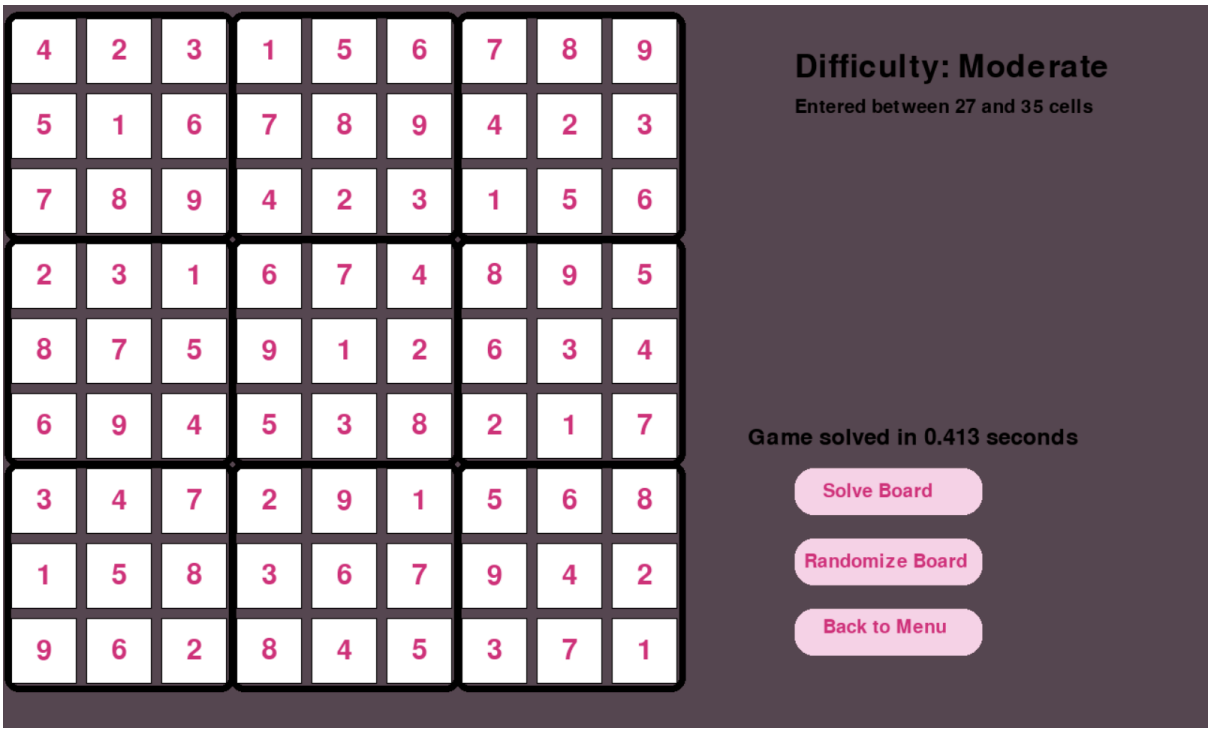
➔ **Sample 1:**



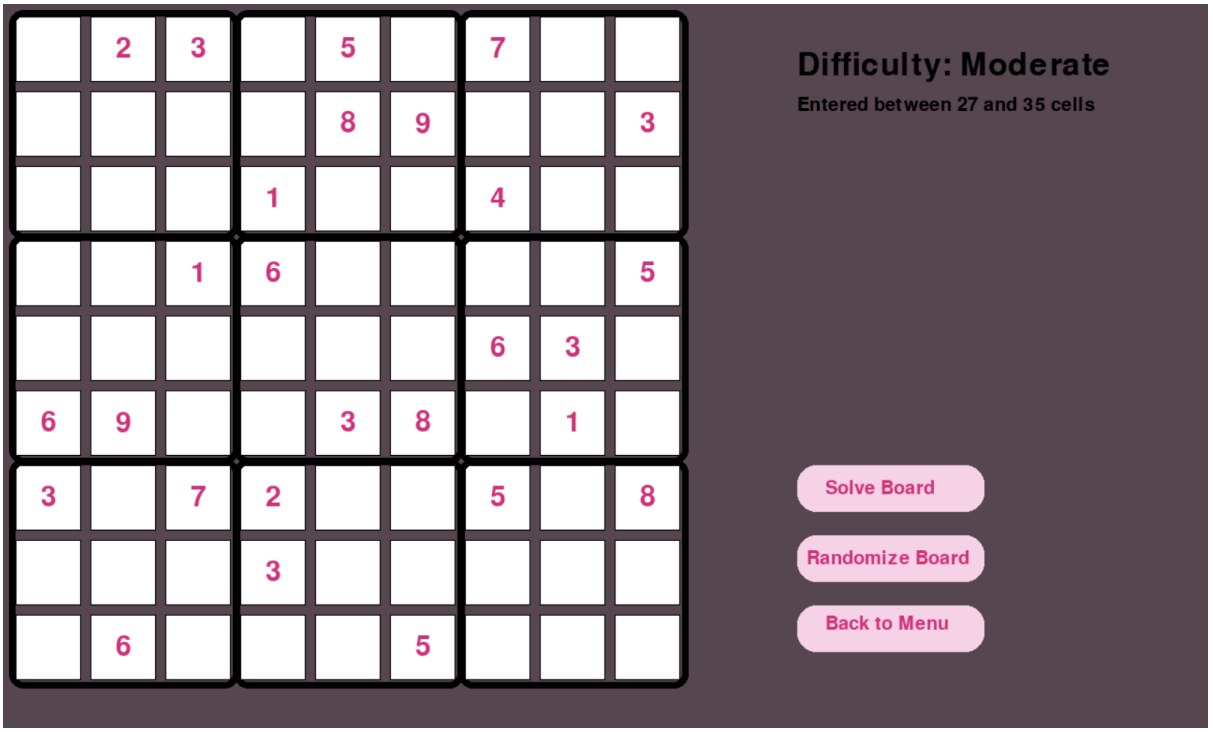


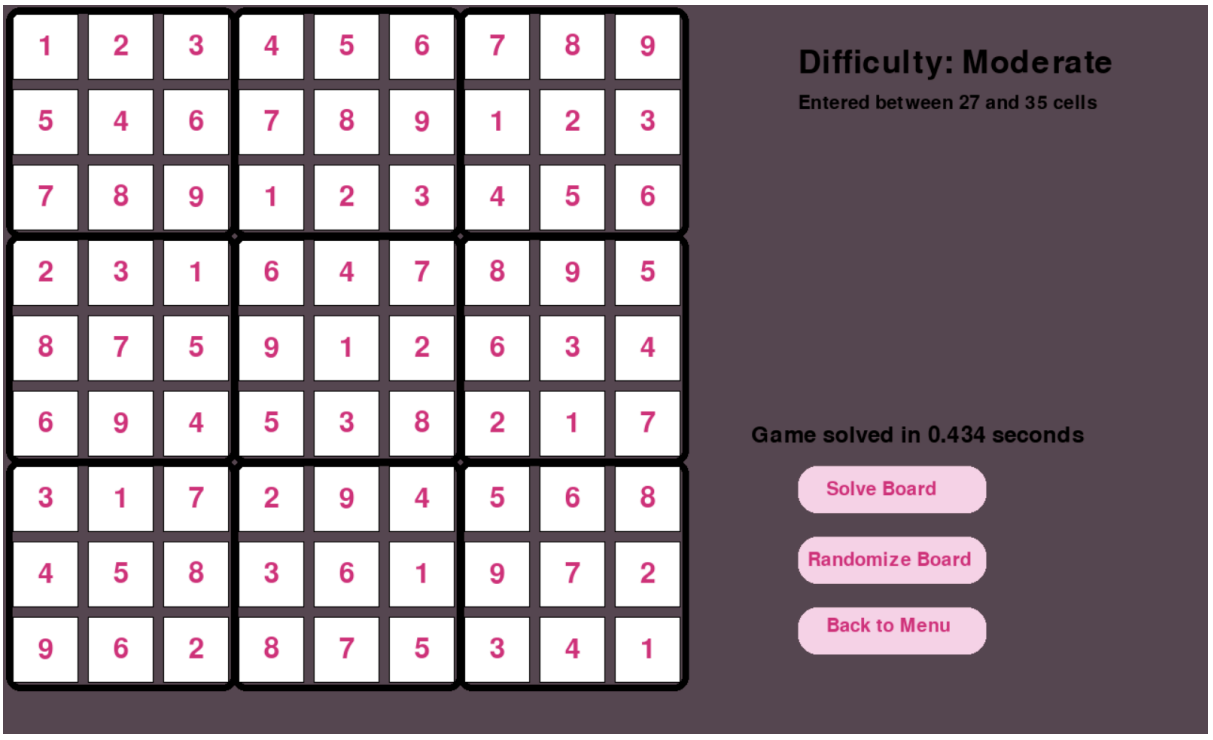
➔ **Sample 2:**


Difficulty: Moderate
Entered between 27 and 35 cells
Solve Board
Randomize Board
Back to Menu


Difficulty: Moderate
Entered between 27 and 35 cells
Game solved in 0.413 seconds
Solve Board
Randomize Board
Back to Menu

➔ **Sample 3:**

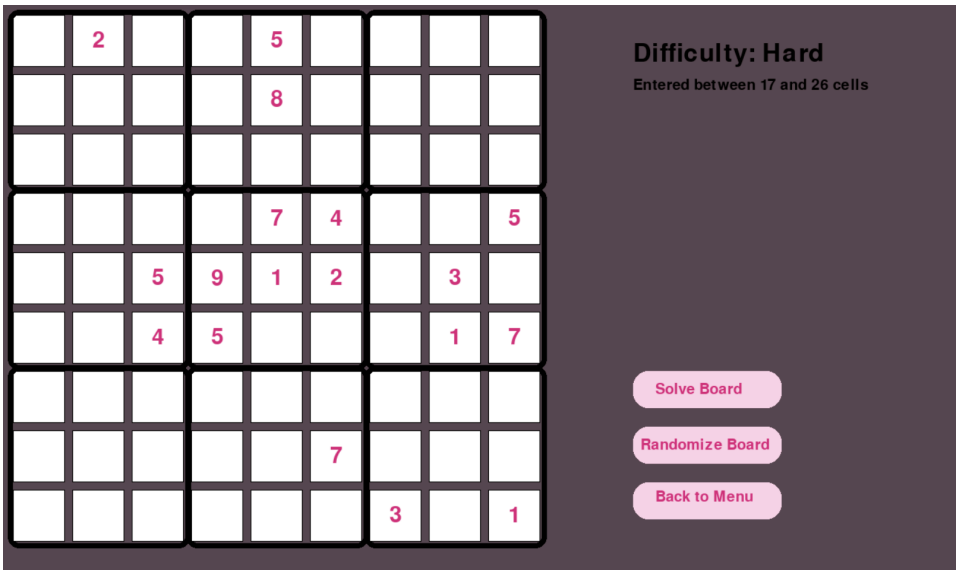

2 3 5 7
8 9 3
1 4
1 6 5
6 3
6 9 3 8 1
3 7 2 5 8
3
6 5
Difficulty: Moderate
Entered between 27 and 35 cells
Solve Board
Randomize Board
Back to Menu

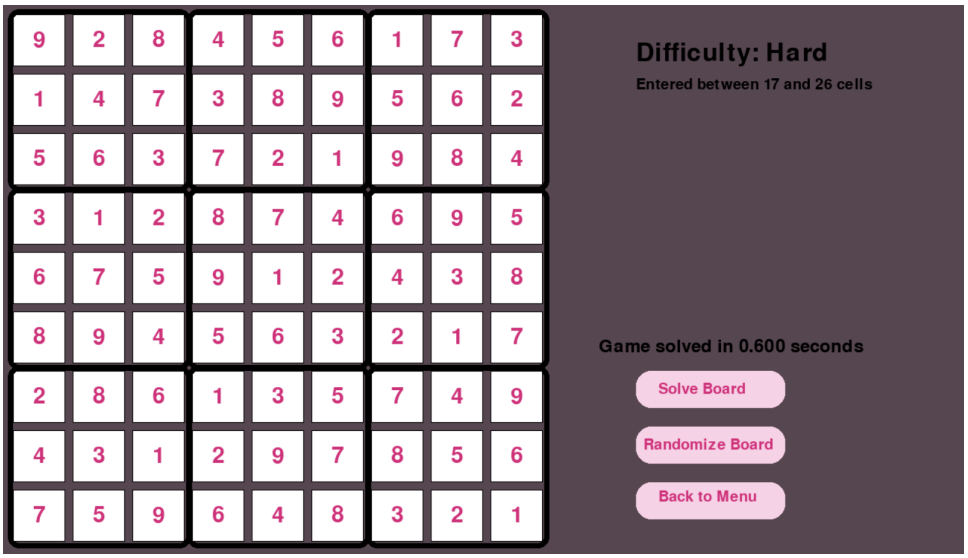

1 2 3 4 5 6 7 8 9
5 4 6 7 8 9 1 2 3
7 8 9 1 2 3 4 5 6
2 3 1 6 4 7 8 9 5
8 7 5 9 1 2 6 3 4
6 9 4 5 3 8 2 1 7
3 1 7 2 9 4 5 6 8
4 5 8 3 6 1 9 7 2
9 6 2 8 7 5 3 4 1
Difficulty: Moderate
Entered between 27 and 35 cells
Game solved in 0.434 seconds
Solve Board
Randomize Board
Back to Menu

- **Hard Level:**

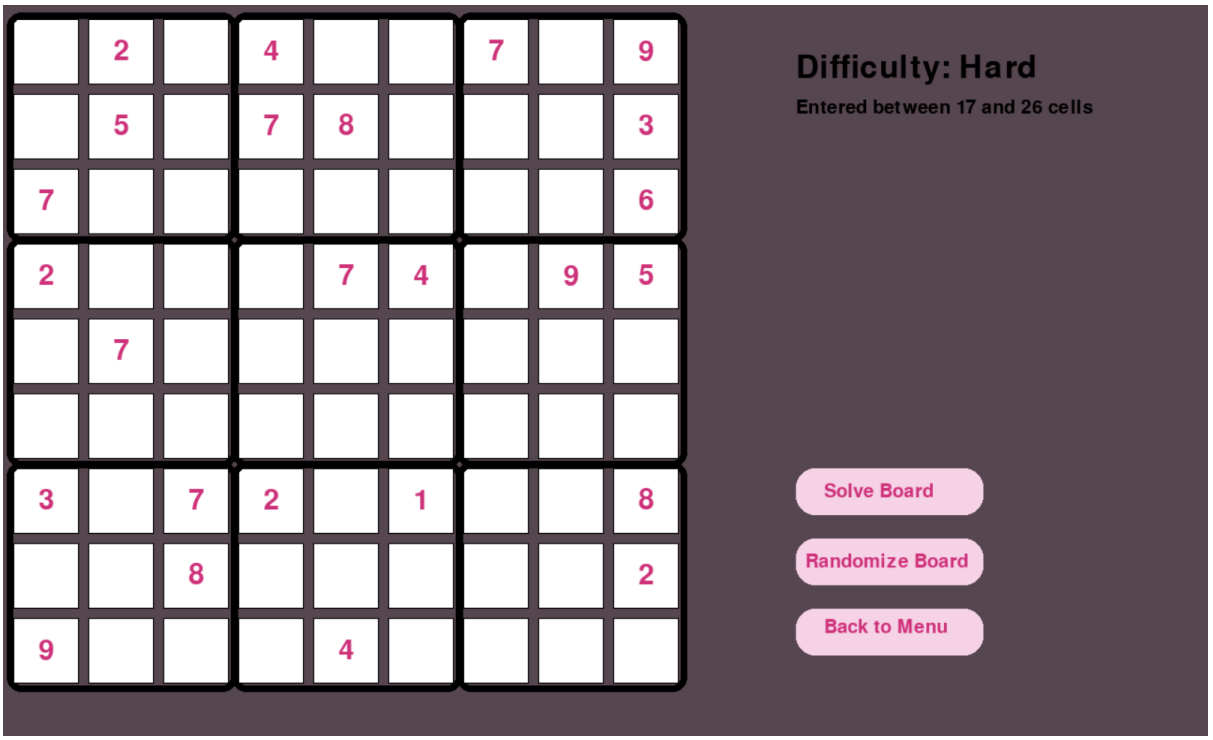
	Time Taken (Secs)
Sample 1	0.6
Sample 2	0.596
Sample 3	0.617

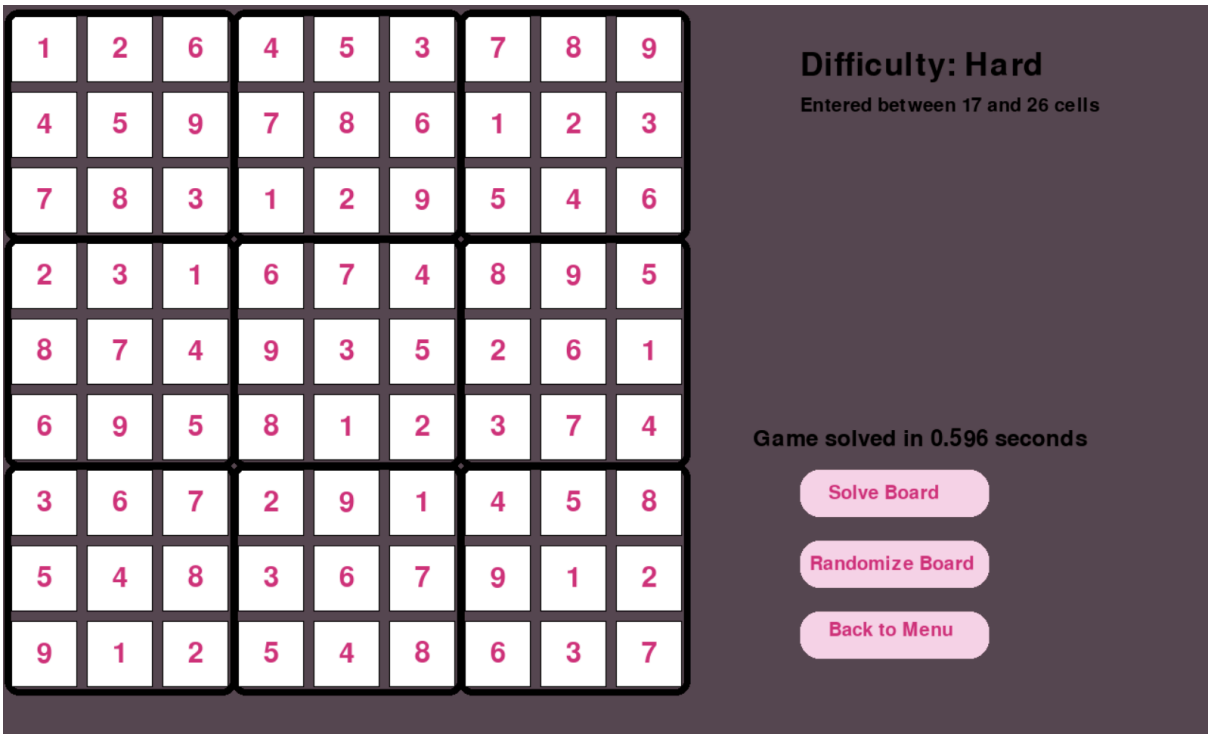
➔ **Sample 1:**



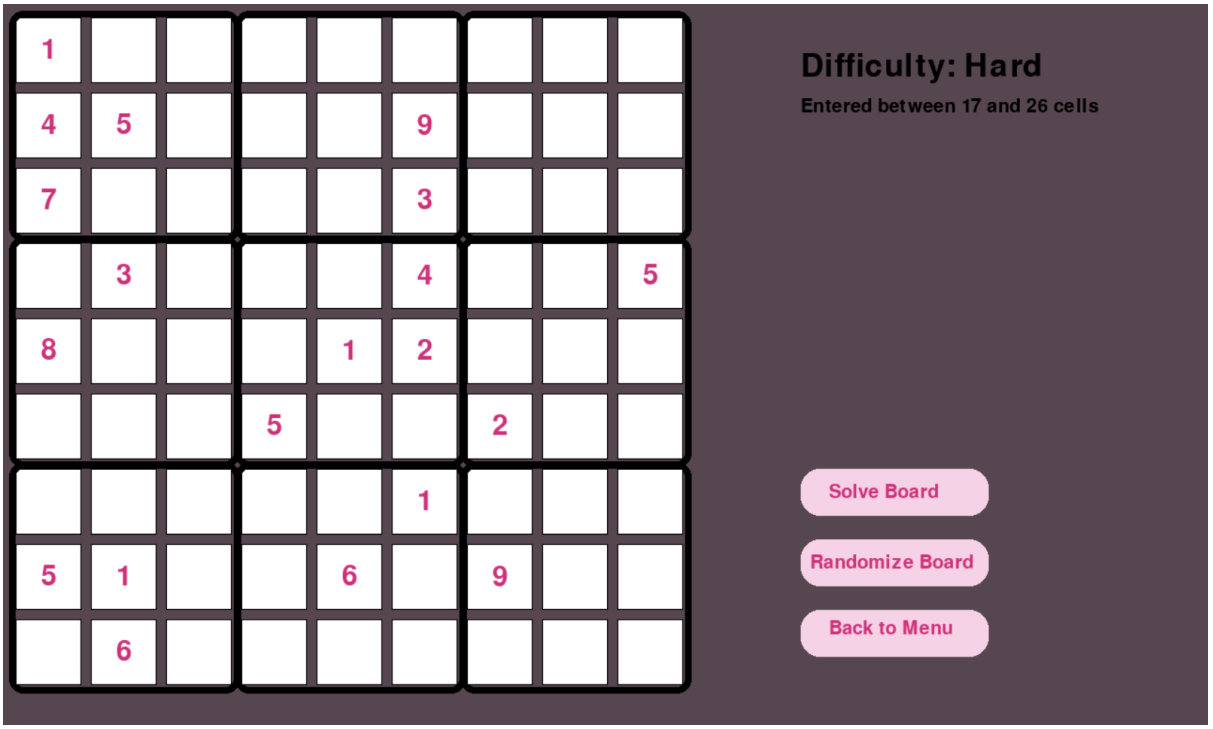


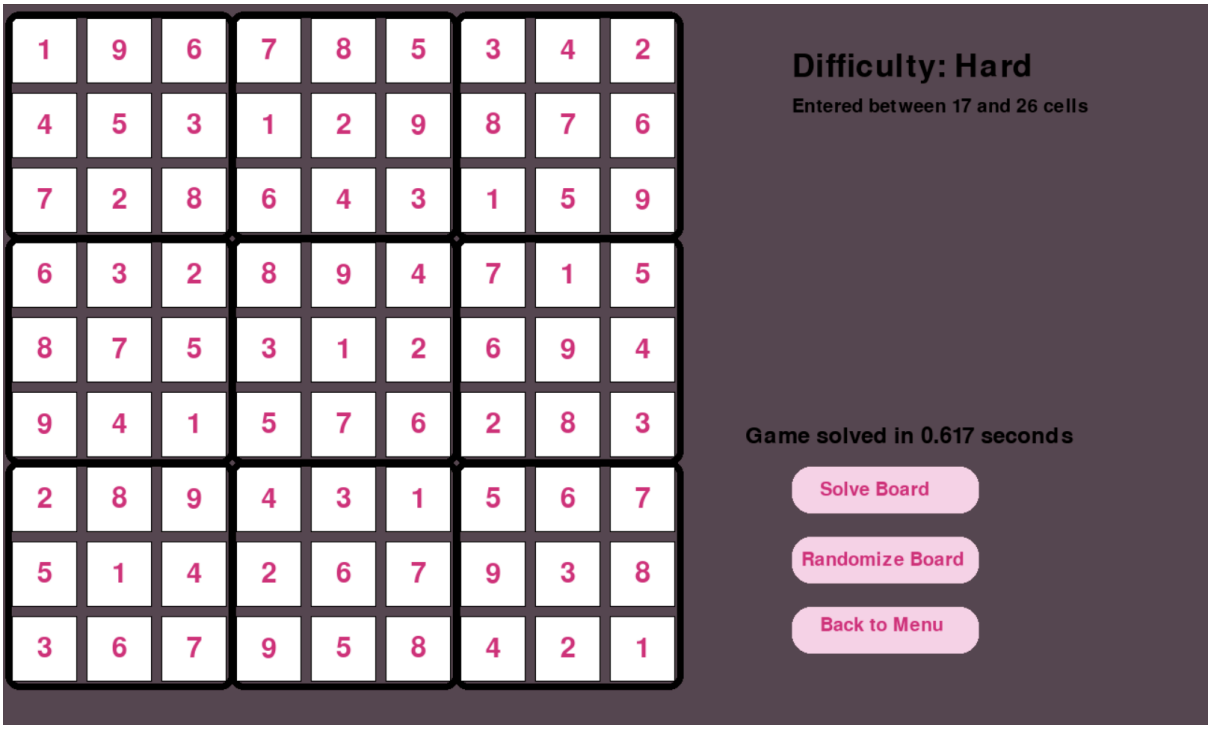
➔ **Sample 2:**


Difficulty: Hard
Entered between 17 and 26 cells
Solve Board
Randomize Board
Back to Menu

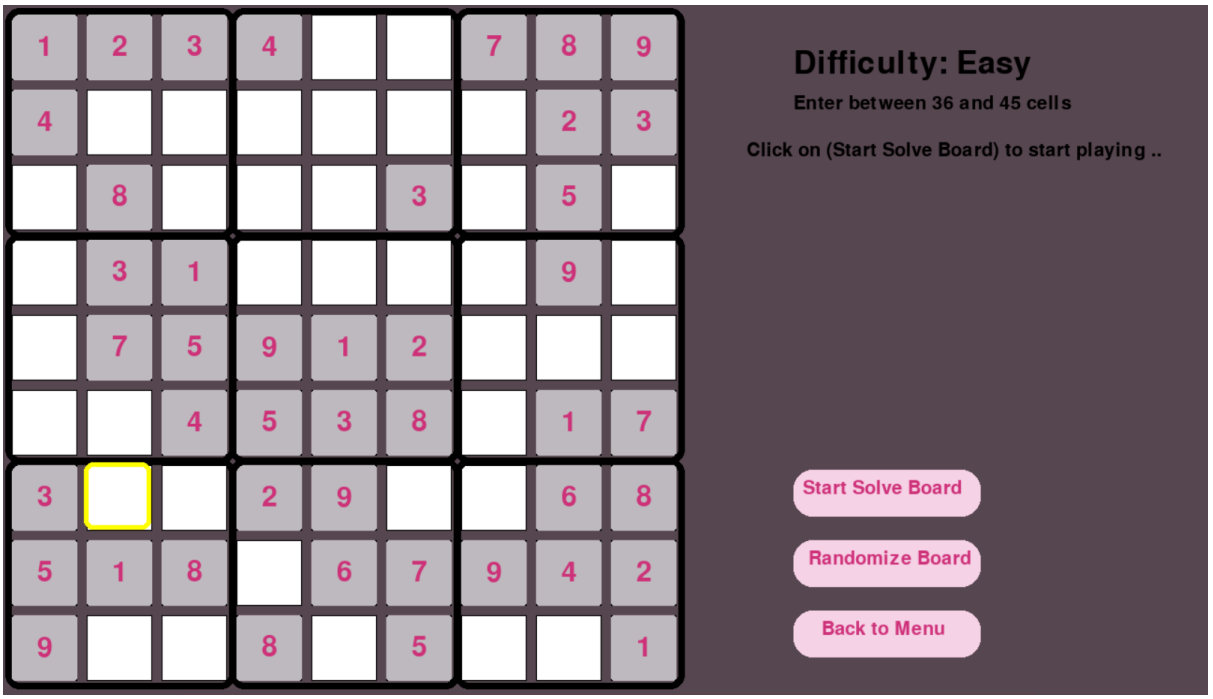

Difficulty: Hard
Entered between 17 and 26 cells
Game solved in 0.596 seconds
Solve Board
Randomize Board
Back to Menu

➔ **Sample 3:**

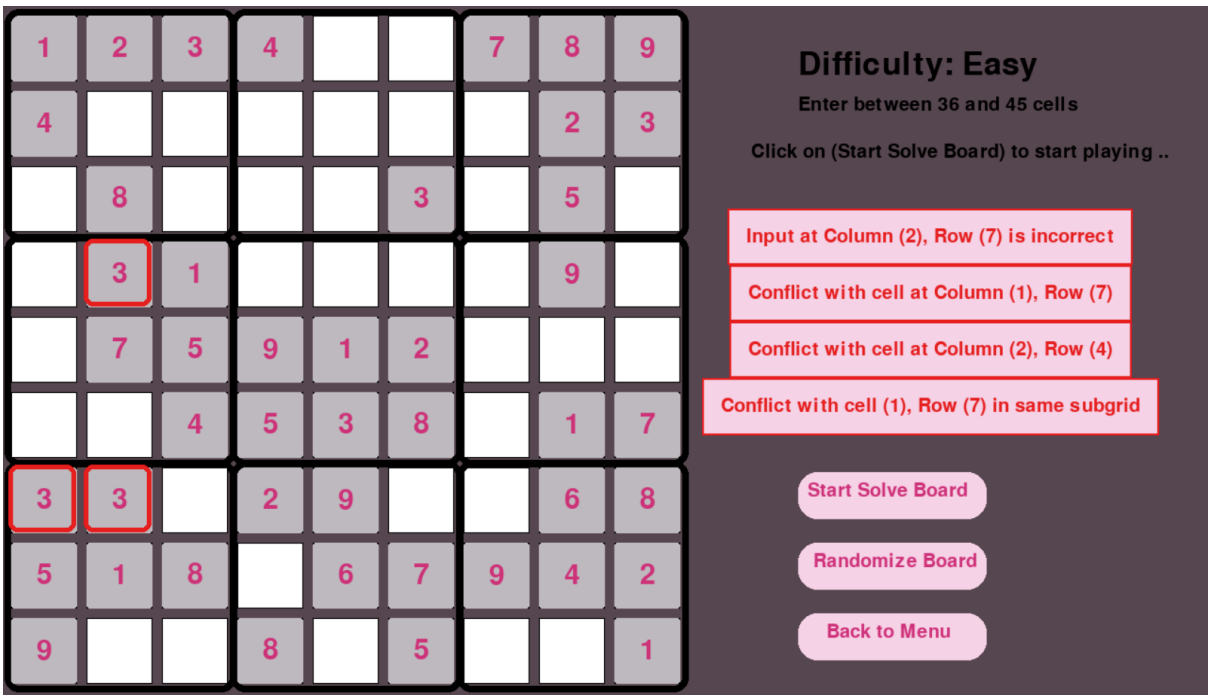

1
4 5 9
7 3
3 4 5
8 1 2
5 2
1
5 1 6 9
6
Difficulty: Hard
Entered between 17 and 26 cells
Solve Board
Randomize Board
Back to Menu

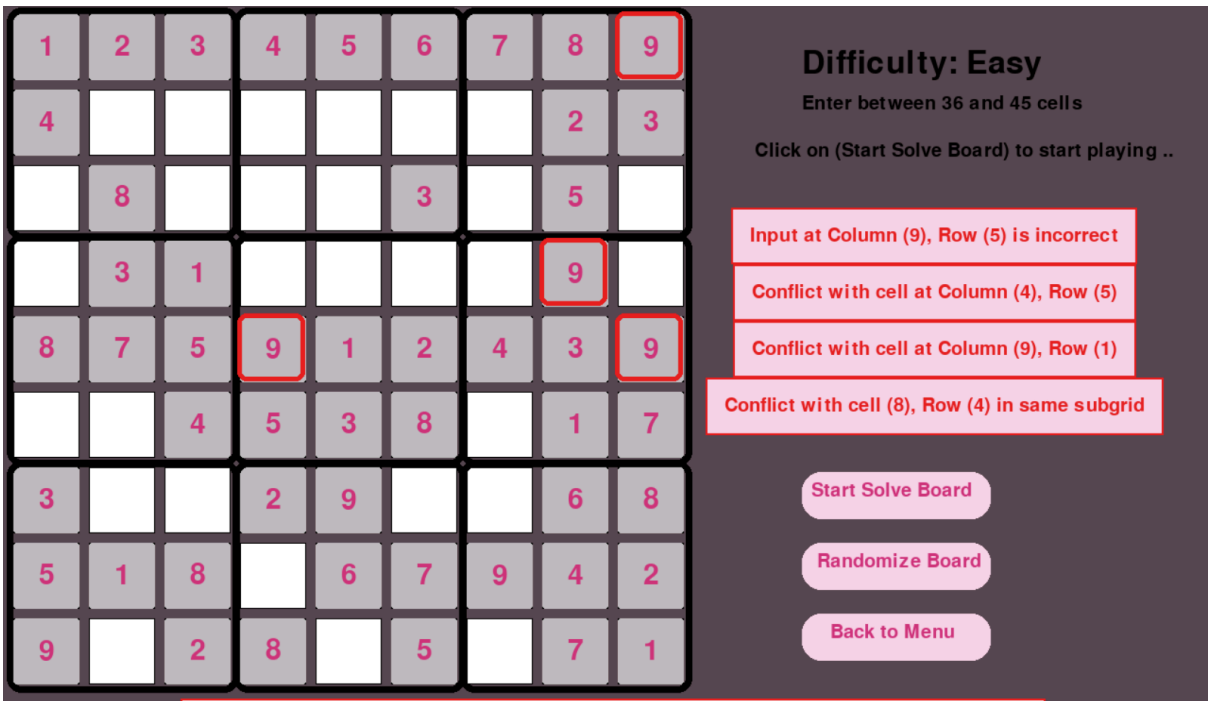

1 9 6 7 8 5 3 4 2
4 5 3 1 2 9 8 7 6
7 2 8 6 4 3 1 5 9
6 3 2 8 9 4 7 1 5
8 7 5 3 1 2 6 9 4
9 4 1 5 7 6 2 8 3
2 8 9 4 3 1 5 6 7
5 1 4 2 6 7 9 3 8
3 6 7 9 5 8 4 2 1
Difficulty: Hard
Entered between 17 and 26 cells
Game solved in 0.617 seconds
Solve Board
Randomize Board
Back to Menu

Bonus:

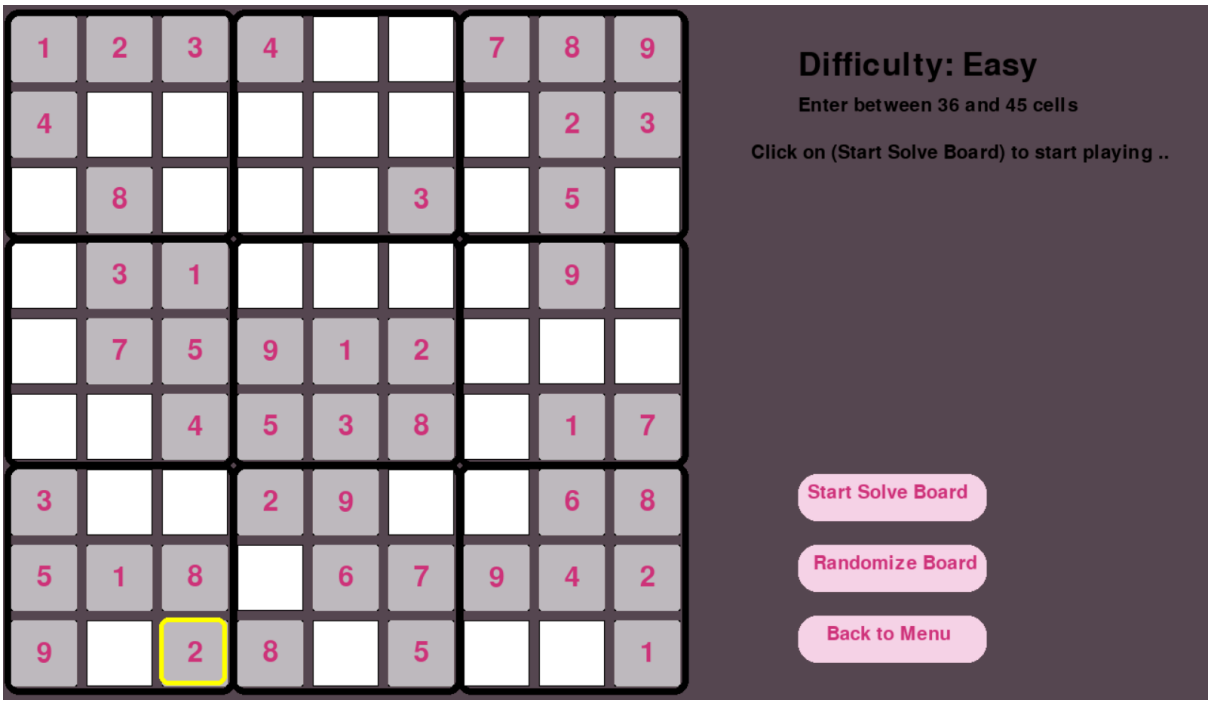


➔ **When wrong input is entered:**





➔ **Continues normally when right input is entered:**




1 2 3 4 5 6 7 8 9
4 5 6 7 8 9 1 2 3
7 8 9 1 2 3 6 5 4
2 3 1 6 7 4 8 9 5
8
6
3 4 7 2 9 1 5 6 8
5 1 8 3 6 7 9 4 2
9 6 2 8 4 5 3 7 1
Difficulty: Easy
Enter between 36 and 45 cells
Click on (Start Solve Board) to start playing ..
Congratulations! You have successfully solved the puzzle.
Start Solve Board
Randomize Board
Back to Menu