Kriti Arora
23BBS0050

# Assignment 1

CBS3004 Artificial Intelligence

# Assignment 1

## 1) Tic Tac Toe Game in Python using GUI

## Exercise Date: 15-07-2025

**Aim:** To Implement the classic Tic Tac Toe game using Python's Tkinter GUI, supporting both two-player (human vs. human) and single-player (human vs. computer with AI) modes.

## Procedure:

1. Design a 3x3 grid layout using Tkinter buttons.
2. Allow players to take alternate turns placing 'X' and 'O'
3. For Human vs Computer mode, use the Minimax algorithm for optimal computer moves.
4. Check for win or draw conditions after each move.
5. Display results and reset options via GUI prompts.

## Code:

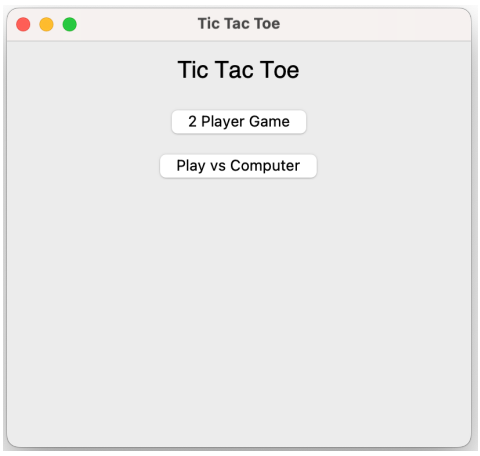
```
import tkinter as tk
from tkinter import messagebox
import random

EMPTY = ' '
PLAYER_X = 'X'
PLAYER_O = 'O'

class TicTacToeGUI:
    def __init__(self, root):
        self.root = root
        self.root.title("Tic Tac Toe")
        self.root.geometry("400x350")
        self.board = [EMPTY] * 9
        self.buttons = []
        self.current_player = PLAYER_X
        self.play_with_computer = False
        self.human_player = PLAYER_X
        self.ai_player = PLAYER_O
        self.create_start_screen()

    def create_start_screen(self):
        self.clear_window()
        tk.Label(self.root, text="Tic Tac Toe", font=('Helvetica', 20)).pack(pady=10)
        tk.Button(self.root, text="2 Player Game",
command=self.start_two_player).pack(pady=5)
        tk.Button(self.root, text="Play vs Computer",
command=self.start_vs_computer).pack(pady=5)
```

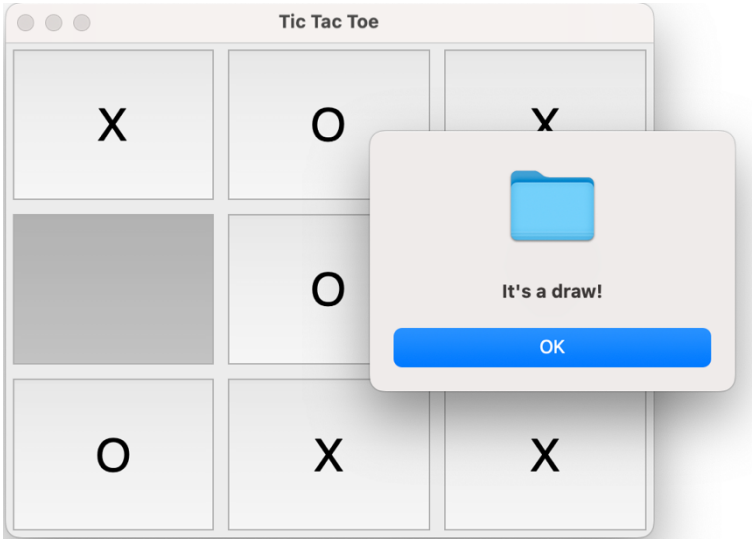
```
    def start_two_player(self):
        self.play_with_computer = False
        self.reset_board()
        self.create_board()

    def start_vs_computer(self):
        self.play_with_computer = True
        self.choose_symbol_screen()

    def choose_symbol_screen(self):
        self.clear_window()
        tk.Label(self.root, text="Choose your symbol:", font=('Helvetica',
14)).pack(pady=10)
        tk.Button(self.root, text="X (You go first)", command=lambda:
self.set_symbol(PLAYER_X)).pack(pady=5)
        tk.Button(self.root, text="O (Computer goes first)", command=lambda:
self.set_symbol(PLAYER_O)).pack(pady=5)

    def set_symbol(self, symbol):
        self.human_player = symbol
        self.ai_player = PLAYER_O if symbol == PLAYER_X else PLAYER_X
        self.current_player = PLAYER_X
        self.reset_board()
        self.create_board()
        if self.ai_player == PLAYER_X:
            self.root.after(500, self.computer_move)

    def create_board(self):
        self.clear_window()
        self.buttons = []

        # Configure the 3x3 grid to expand with window size
        for i in range(3):
            self.root.grid_rowconfigure(i, weight=1)
            self.root.grid_columnconfigure(i, weight=1)

        for i in range(9):
            btn = tk.Button(
                self.root,
                text='',
                font=('Helvetica', 32),  # Larger font scales better
                command=lambda i=i: self.on_button_click(i)
            )
            btn.grid(row=i//3, column=i%3, sticky="nsew", padx=2, pady=2)
            self.buttons.append(btn)


    def on_button_click(self, index):
        if self.board[index] != EMPTY:
            return
        if self.play_with_computer and self.current_player != self.human_player:
            return

        self.make_move(index, self.current_player)
```

```
        winner = self.check_winner()
        if winner:
            self.game_over(f"{winner} wins!")
        elif self.check_draw():
            self.game_over("It's a draw!")
        else:
            self.switch_player()
            if self.play_with_computer and self.current_player == self.ai_player:
                self.root.after(500, self.computer_move)

    def make_move(self, index, player):
        self.board[index] = player
        self.buttons[index].config(text=player)

    def computer_move(self):
        best_score = -float('inf')
        best_move = None
        for i in range(9):
            if self.board[i] == EMPTY:
                self.board[i] = self.ai_player
                score = self.minimax(0, False)
                self.board[i] = EMPTY
                if score > best_score:
                    best_score = score
                    best_move = i

        if best_move is not None:
            self.make_move(best_move, self.ai_player)

            winner = self.check_winner()
            if winner:
                self.game_over(f"{winner} wins!")
            elif self.check_draw():
                self.game_over("It's a draw!")
            else:
                self.switch_player()

    def minimax(self, depth, is_maximizing):
        winner = self.check_winner()
        if winner == self.ai_player:
            return 1
        elif winner == self.human_player:
            return -1
        elif self.check_draw():
            return 0

        if is_maximizing:
            best_score = -float('inf')
            for i in range(9):
                if self.board[i] == EMPTY:
                    self.board[i] = self.ai_player
                    score = self.minimax(depth + 1, False)
                    self.board[i] = EMPTY
                    best_score = max(score, best_score)
            return best_score
        else:
```

```
            best_score = float('inf')
            for i in range(9):
                if self.board[i] == EMPTY:
                    self.board[i] = self.human_player
                    score = self.minimax(depth + 1, True)
                    self.board[i] = EMPTY
                    best_score = min(score, best_score)
            return best_score

    def switch_player(self):
        self.current_player = PLAYER_O if self.current_player == PLAYER_X else
PLAYER_X

    def check_winner(self):
        for a, b, c in
[(0,1,2),(3,4,5),(6,7,8),(0,3,6),(1,4,7),(2,5,8),(0,4,8),(2,4,6)]:
            if self.board[a] == self.board[b] == self.board[c] != EMPTY:
                return self.board[a]
        return None

    def check_draw(self):
        return EMPTY not in self.board and not self.check_winner()

    def game_over(self, message):
        messagebox.showinfo("Game Over", message)
        self.create_start_screen()

    def reset_board(self):
        self.board = [EMPTY] * 9

    def clear_window(self):
        for widget in self.root.winfo_children():
            widget.destroy()

if __name__ == '__main__':
    root = tk.Tk()
    game = TicTacToeGUI(root)
    root.mainloop()
```

## Output:



## Result:

The Tic Tac Toe game was successfully implemented with both Human vs Human and Human vs AI modes in an interactive GUI.

## 2) Water Jug Problem using GUI and BFS

## Exercise Date: 22-07-2025

**Aim:** To create a program that solves the Water Jug Problem using Breadth-First Search (BFS), and show each step of the solution in an interactive way using Tkinter.

## Procedure:

1. Accept jug capacities and the target volume as input.
2. Use BFS to find a series of valid operations to reach the target volume.
3. Display the jugs graphically using Canvas.
4. Show each step only when the user clicks "Next Step", preventing auto-play.
5. Clearly update jug water levels and action description on each step.

## Code:

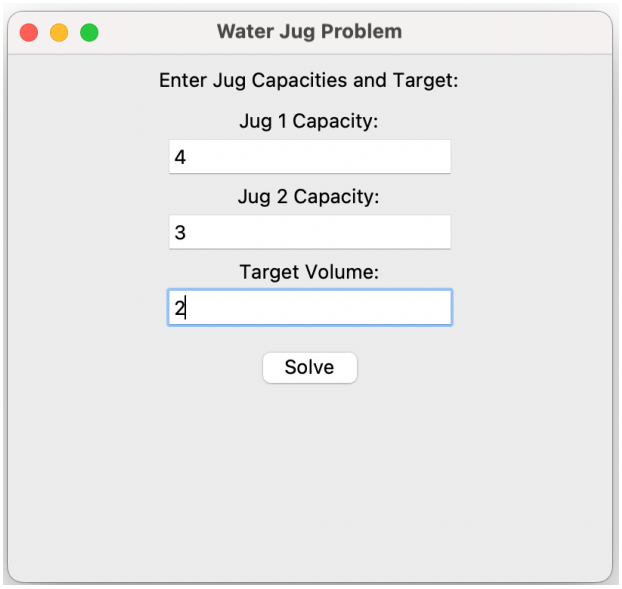
```
import tkinter as tk
from tkinter import messagebox
from collections import deque

# ----- Water Jug Problem Implementation -----
def water_jug_solver(jug1, jug2, target):
    visited = set()
    queue = deque()
    actions = {}

    queue.append((0, 0, []))

    while queue:
        a, b, steps = queue.popleft()

        if a == target or b == target:
            steps.append(((a, b), "Goal Reached"))
            return steps

        if (a, b) in visited:
            continue

        visited.add((a, b))
        next_states = []
```

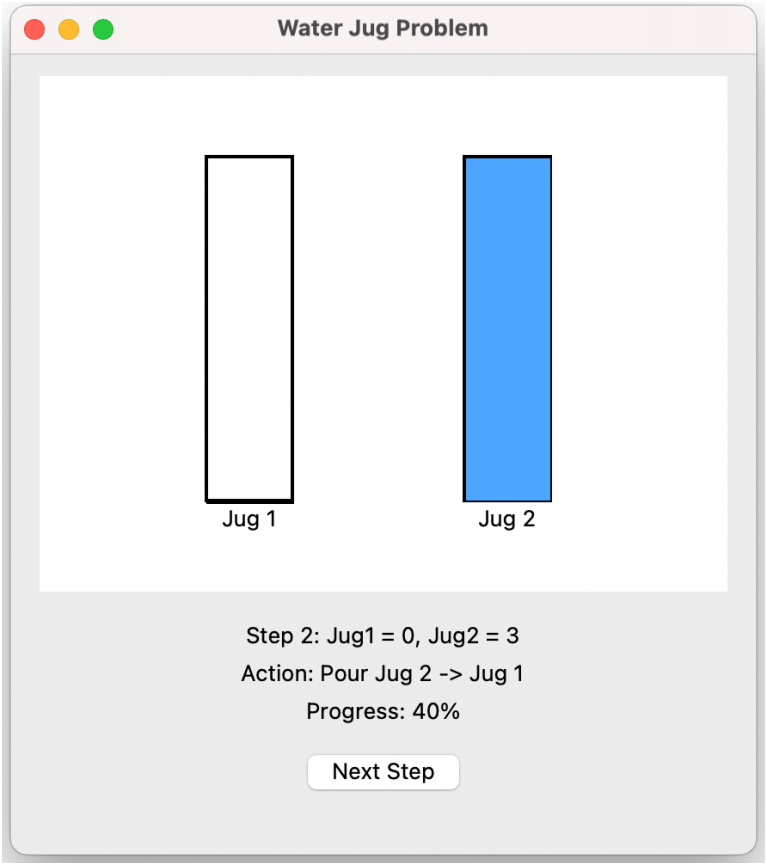
```
        next_states.append(((jug1, b), "Fill Jug 1"))
        next_states.append(((a, jug2), "Fill Jug 2"))
        next_states.append(((0, b), "Empty Jug 1"))
        next_states.append(((a, 0), "Empty Jug 2"))

        transfer = min(a, jug2 - b)
        next_states.append(((a - transfer, b + transfer), "Pour Jug 1 -> Jug 2"))
        transfer = min(b, jug1 - a)
        next_states.append(((a + transfer, b - transfer), "Pour Jug 2 -> Jug 1"))

        for (state, action) in next_states:
            if state not in visited:
                queue.append((state[0], state[1], steps + [((a, b), action)]))

    return None

# ----- GUI with Click-to-Advance Animation -----
class WaterJugGUI:
    def __init__(self, root):
        self.root = root
        self.root.title("Water Jug Problem")
        self.jug1_capacity = 0
        self.jug2_capacity = 0
        self.steps = []
        self.canvas = None
        self.step_index = 0
        self.progress = None
        self.create_input_screen()

    def create_input_screen(self):
        self.clear_window()
        tk.Label(self.root, text="Enter Jug Capacities and Target:").pack(pady=5)

        tk.Label(self.root, text="Jug 1 Capacity:").pack()
        self.jug1_entry = tk.Entry(self.root)
        self.jug1_entry.pack()

        tk.Label(self.root, text="Jug 2 Capacity:").pack()
        self.jug2_entry = tk.Entry(self.root)
        self.jug2_entry.pack()

        tk.Label(self.root, text="Target Volume:").pack()
        self.target_entry = tk.Entry(self.root)
        self.target_entry.pack()

        tk.Button(self.root, text="Solve",
command=self.prepare_animation).pack(pady=10)

    def prepare_animation(self):
        try:
            jug1 = int(self.jug1_entry.get())
            jug2 = int(self.jug2_entry.get())
            target = int(self.target_entry.get())

            self.jug1_capacity = jug1
```

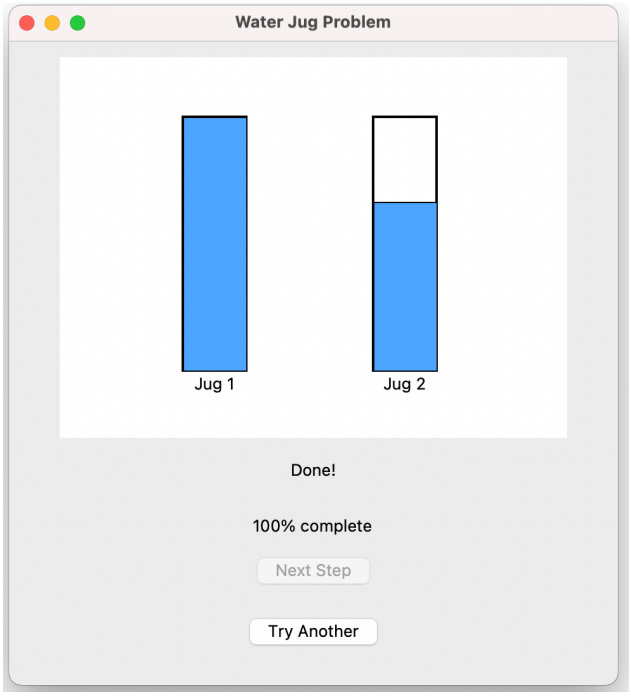
```
            self.jug2_capacity = jug2

            self.steps = water_jug_solver(jug1, jug2, target)
            self.clear_window()

            if not self.steps:
                tk.Label(self.root, text="No solution found.").pack()
                tk.Button(self.root, text="Try Another",
command=self.create_input_screen).pack(pady=10)
                return

            self.canvas = tk.Canvas(self.root, width=400, height=300, bg="white")
            self.canvas.pack(pady=10)

            # Jug outlines
            self.jug1_rect = self.canvas.create_rectangle(100, 50, 150, 250,
outline="black", width=2)
            self.jug2_rect = self.canvas.create_rectangle(250, 50, 300, 250,
outline="black", width=2)
            # Jug fills
            self.jug1_fill = self.canvas.create_rectangle(100, 250, 150, 250,
fill="#4DA6FF")
            self.jug2_fill = self.canvas.create_rectangle(250, 250, 300, 250,
fill="#4DA6FF")

            # Labels
            self.canvas.create_text(125, 260, text="Jug 1")
            self.canvas.create_text(275, 260, text="Jug 2")

            # Step text and progress
            self.step_label = tk.Label(self.root, text="")
            self.step_label.pack()
            self.action_label = tk.Label(self.root, text="")
            self.action_label.pack()
            self.progress = tk.Label(self.root, text="")
            self.progress.pack()

            self.next_button = tk.Button(self.root, text="Next Step",
command=self.next_step)
            self.next_button.pack(pady=10)

            self.step_index = 0
            self.update_visuals()

        except ValueError:
            messagebox.showerror("Invalid Input", "Please enter valid integers.")

    def next_step(self):
        if self.step_index < len(self.steps):
            self.update_visuals()
            self.step_index += 1
        else:
            self.step_label.config(text="Done!")
            self.action_label.config(text="")
            self.progress.config(text="100% complete")
            self.next_button.config(state="disabled")
```

```
            tk.Button(self.root, text="Try Another",
command=self.create_input_screen).pack(pady=10)

    def update_visuals(self):
        (a, b), action = self.steps[self.step_index]

        self.step_label.config(text=f"Step {self.step_index + 1}: Jug1 = {a}, Jug2 =
{b}")
        self.action_label.config(text=f"Action: {action}")
        self.progress.config(text=f"Progress: {int((self.step_index +
1)/len(self.steps)*100)}%")

        jug1_height = 200 * (a / self.jug1_capacity) if self.jug1_capacity else 0
        jug2_height = 200 * (b / self.jug2_capacity) if self.jug2_capacity else 0

        self.canvas.coords(self.jug1_fill, 100, 250 - jug1_height, 150, 250)
        self.canvas.coords(self.jug2_fill, 250, 250 - jug2_height, 300, 250)

    def clear_window(self):
        for widget in self.root.winfo_children():
            widget.destroy()

if __name__ == '__main__':
    root = tk.Tk()
    app = WaterJugGUI(root)
    root.mainloop()
```

# Output:






## Result:

The Water Jug Problem was successfully implemented and visualized step-by-step, enabling clear understanding of the process to reach the goal.

## 3) N-Queens Problem Visualizer using Tkinter

## Exercise Date: 29-07-2025

**Aim:** To implement the N-Queens problem using backtracking and visualize all valid queen placements on an N×N chessboard.

## Procedure:

1. Take input N for board size.
2. Use backtracking to compute all valid queen placements.
3. For each solution, draw the board using Canvas with alternating cell colours.
4. Replace default markers with an image of a queen (`queen.png`) for realism.
5. Allow the user to click “Next Solution” to see all valid combinations one by one.

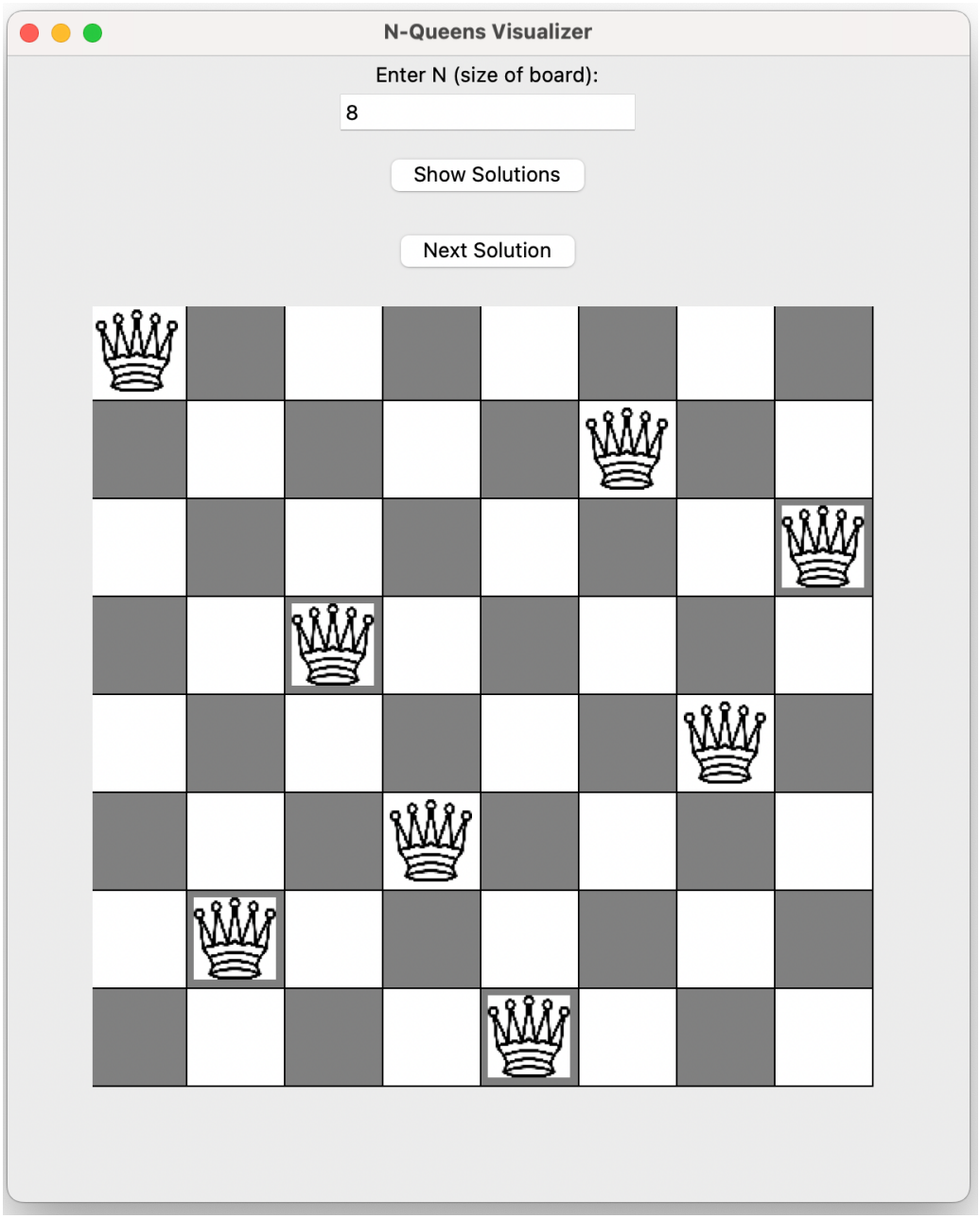
## Code:

```
import tkinter as tk
from tkinter import messagebox
from PIL import Image, ImageTk, ImageOps
```

```
class NQueensGUI:
    def __init__(self, root):
        self.root = root
        self.root.title("N-Queens Visualizer")
        self.size = 8
        self.solutions = []
        self.current = 0
        self.board_canvas = None
        self.queen_img = None
        self.next_button = None

        self.setup_ui()

    def setup_ui(self):
        tk.Label(self.root, text="Enter N (size of board):").pack()
        self.n_entry = tk.Entry(self.root)
        self.n_entry.insert(0, "8")
        self.n_entry.pack()

        tk.Button(self.root, text="Show Solutions",
command=self.solve_and_display).pack(pady=10)

    def solve_and_display(self):
        try:
            self.size = int(self.n_entry.get())
            self.solutions = []
            self.current = 0
            self.solve_n_queens()
            if not self.solutions:
                messagebox.showinfo("No Solutions", f"No solutions found for
N={self.size}.")
                return
            self.display_solution()
            if not self.next_button:
                self.next_button = tk.Button(self.root, text="Next Solution",
command=self.next_solution)
                self.next_button.pack(pady=10)
        except ValueError:
            messagebox.showerror("Invalid Input", "Please enter a valid number.")

    def solve_n_queens(self):
        def is_safe(queens, row, col):
            for r, c in enumerate(queens):
                if c == col or abs(r - row) == abs(c - col):
                    return False
            return True

        def backtrack(row=0, queens=[]):
            if row == self.size:
                self.solutions.append(queens[:])
                return
            for col in range(self.size):
                if is_safe(queens, row, col):
                    queens.append(col)
                    backtrack(row + 1, queens)
                    queens.pop()
```

```
        backtrack()

    def next_solution(self):
        self.current = (self.current + 1) % len(self.solutions)
        self.display_solution()

    def display_solution(self):
        if self.board_canvas:
            self.board_canvas.destroy()

        self.board_canvas = tk.Canvas(self.root, width=500, height=500)
        self.board_canvas.pack(pady=10)

        cell_size = 500 // self.size
        board = self.solutions[self.current]

        # Load the queen image (once)
        if not self.queen_img:
            try:
                img = Image.open("./Downloads/queen.png").resize((cell_size - 10,
cell_size - 10), Image.Resampling.LANCZOS)
                self.queen_img = ImageTk.PhotoImage(img)
            except Exception as e:
                messagebox.showerror("Image Load Error", f"Could not load queen
image: {e}")
                return

        for row in range(self.size):
            for col in range(self.size):
                x1 = col * cell_size
                y1 = row * cell_size
                x2 = x1 + cell_size
                y2 = y1 + cell_size
                fill = "white" if (row + col) % 2 == 0 else "gray"
                self.board_canvas.create_rectangle(x1, y1, x2, y2, fill=fill)
                if board[row] == col and self.queen_img:
                    self.board_canvas.create_image(x1 + 5, y1 + 5, anchor='nw',
image=self.queen_img)

if __name__ == '__main__':
    root = tk.Tk()
    app = NQueensGUI(root)
    root.mainloop()
```

## Output:




## Result:

The N-Queens problem was successfully implemented with dynamic board size, solution generation, and visual representation using queen icons.