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():

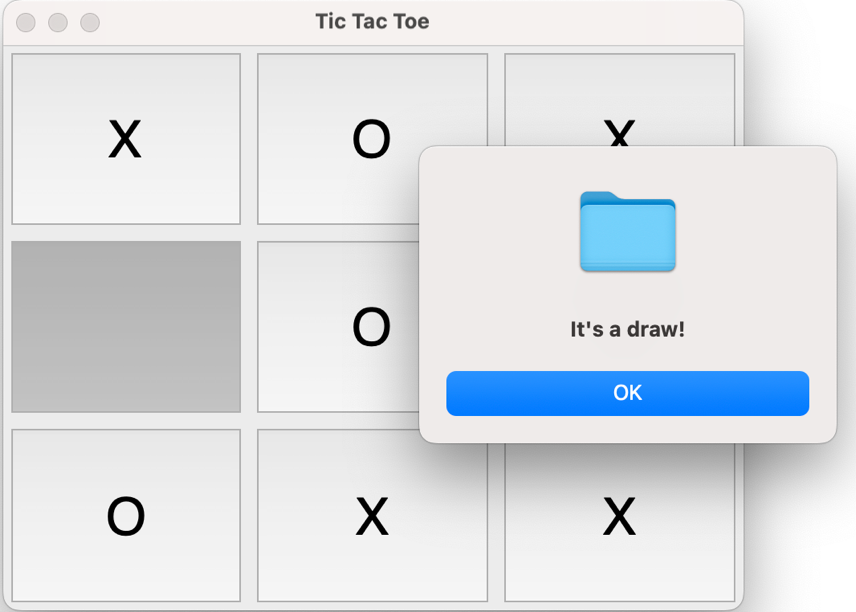
widget.destroy()

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.

1. **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()

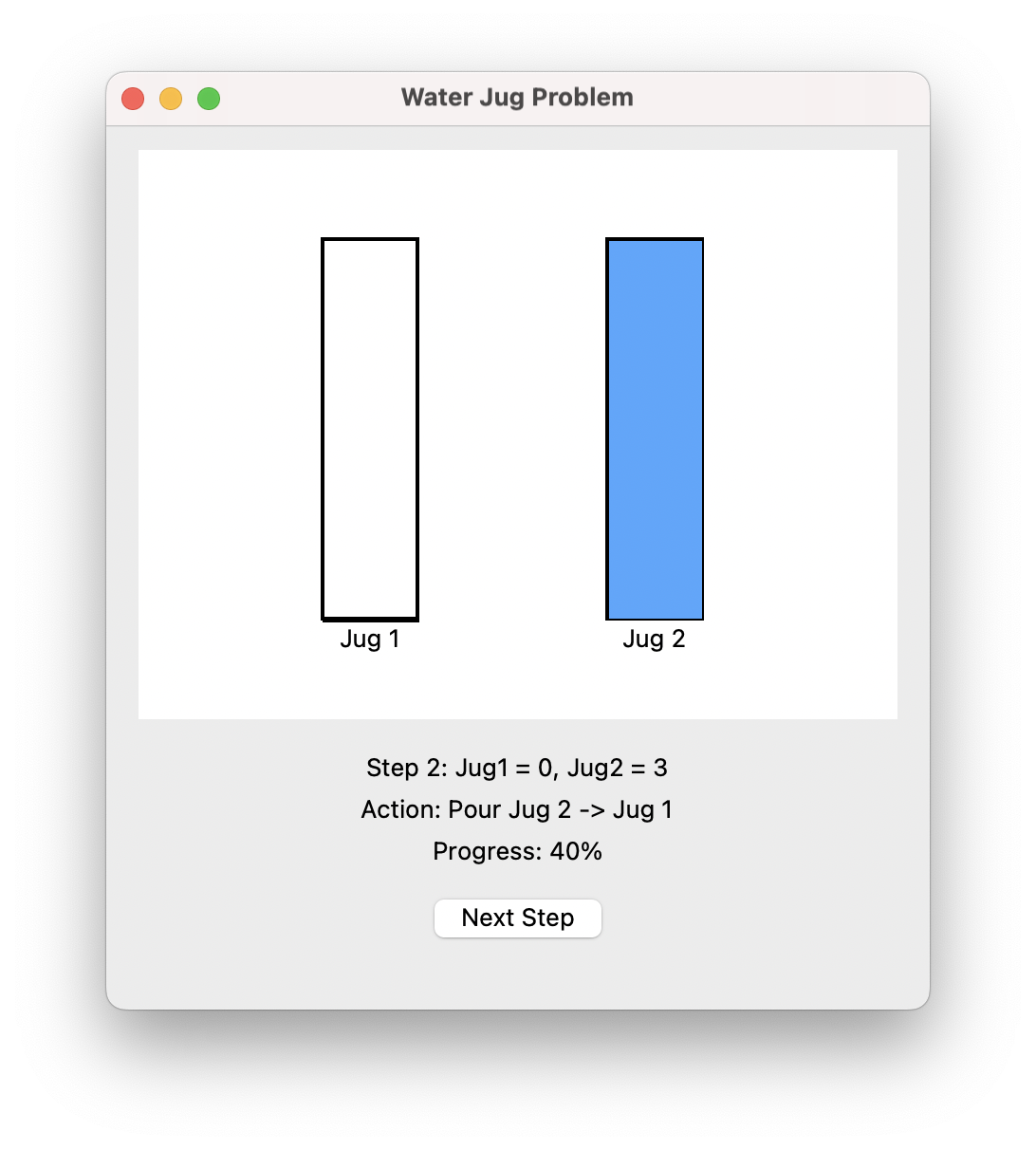
if \_\_name\_\_ == '\_\_main\_\_':

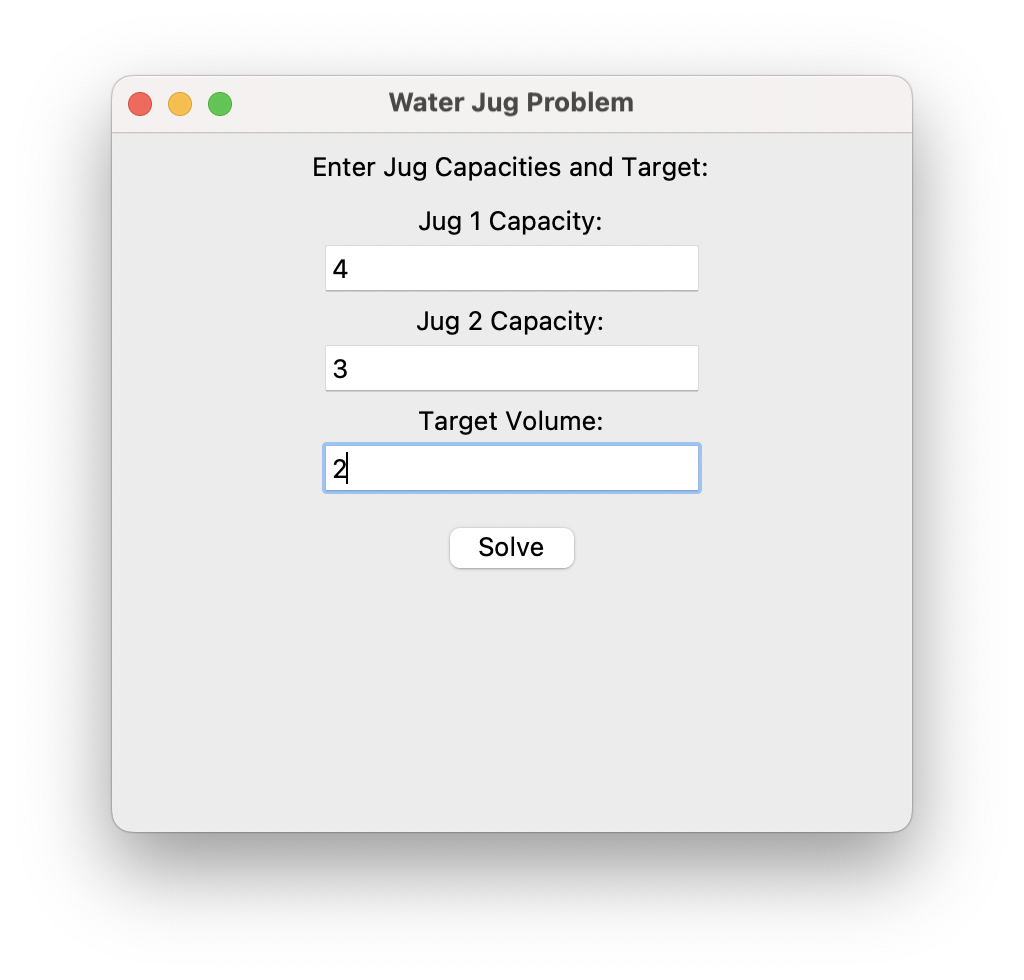
root = tk.Tk()

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)

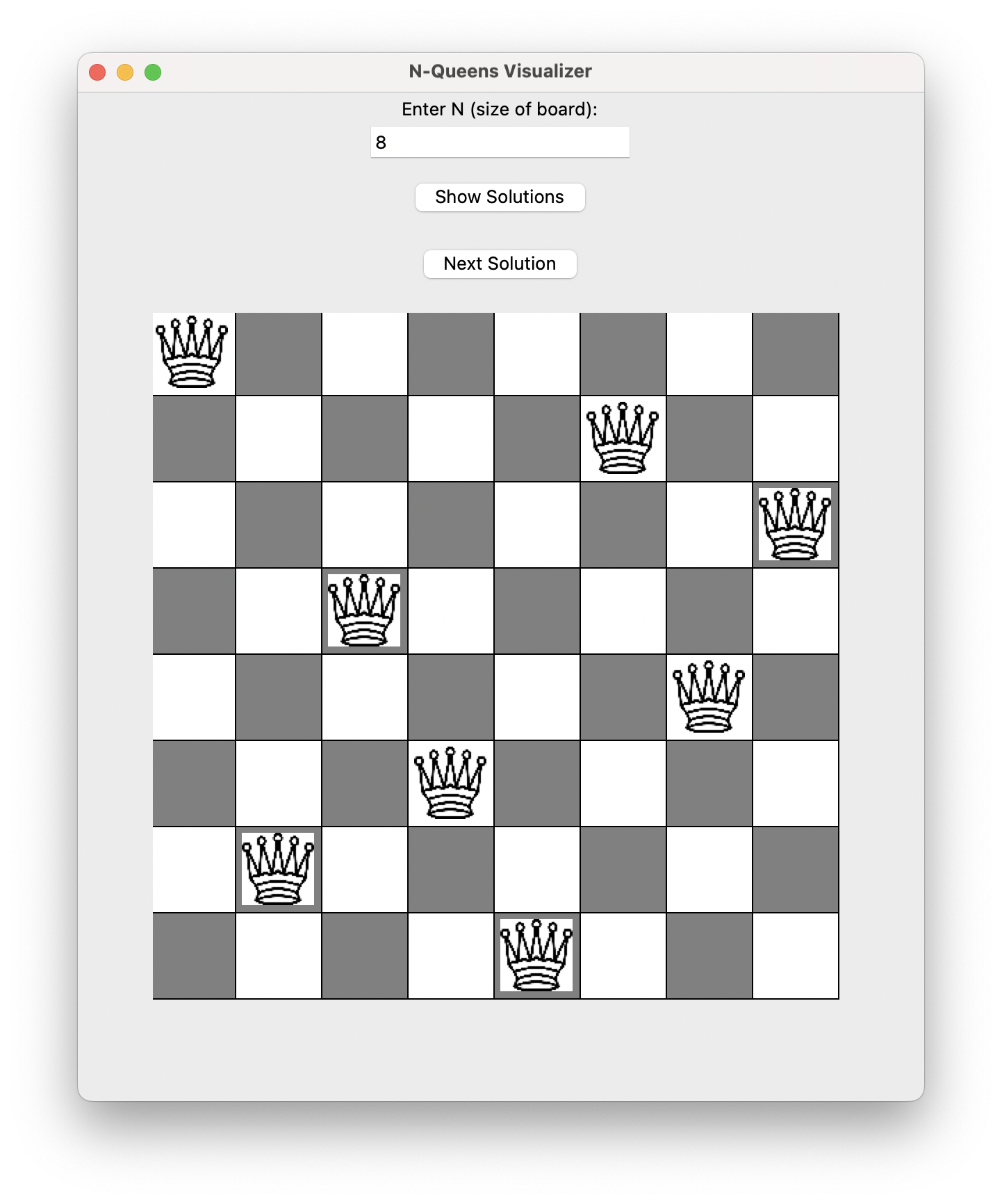
if \_\_name\_\_ == '\_\_main\_\_':

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.