

|  |  |
| --- | --- |
| **Student Name** | **Arun Adhikari** |
| **SRN No** | 202100406 |
| **Roll No** | 31 |
| **Program** | **Department of Computer Engineering** |
| **Year** | 2024-2025 |
| **Division** | G |
| **Subject** | AI LAB |
| **Assignment No** | 1 |

**Experiment No : 1**

# Title /Problem Statement:

Implement Non-AI Techniques for the following problems:

1. Tic Tac Toe
2. N Queens
3. Magic square

# Description:

A non-AI technique employs traditional algorithms or rule-based logic, leveraging predefined conditions and deterministic operations to solve problems systematically, ensuring transparency, predictability, and minimal computational overhead without machine learning.

# Code:

1. Tic Tac Toe:

import tkinter as tk

from tkinter import messagebox

root = tk.Tk()

root.title("Tic Tac Toe")

current\_player = "X"

board = ["" for \_ in range(9)]

def check\_winner():

win\_combinations = [

(0, 1, 2), (3, 4, 5), (6, 7, 8),

(0, 3, 6), (1, 4, 7), (2, 5, 8),

(0, 4, 8), (2, 4, 6)

]

for combo in win\_combinations:

if board[combo[0]] == board[combo[1]] == board[combo[2]] != "":

return board[combo[0]]

if "" not in board:

return "Draw"

return None

def on\_click(index):

global current\_player

if board[index] == "":

board[index] = current\_player

buttons[index]["text"] = current\_player

winner = check\_winner()

if winner:

if winner == "Draw":

messagebox.showinfo("Game Over", "It's a Draw!")

else:

messagebox.showinfo("Game Over", f"Player {winner} wins!")

reset\_game()

else:

current\_player = "O" if current\_player == "X" else "X"

def reset\_game():

global current\_player, board

current\_player = "X"

board = ["" for \_ in range(9)]

for button in buttons:

button["text"] = ""

buttons = []

for i in range(9):

btn = tk.Button(root, text="", font=("Arial", 24), height=2, width=5,

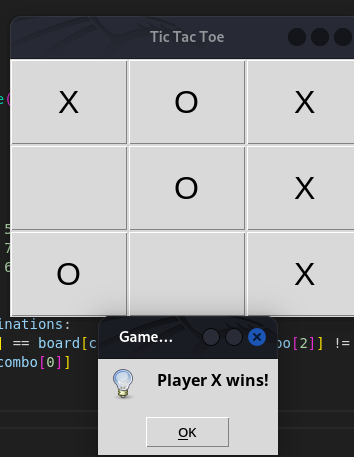
command=lambda idx=i: on\_click(idx))

btn.grid(row=i//3, column=i%3)

buttons.append(btn)

root.mainloop()

**Output :**

****

1. **N Queens**

import tkinter as tk

from tkinter import messagebox

class NQueensGUI:

def \_\_init\_\_(self, n):

self.n = n

self.root = tk.Tk()

self.root.title(f"{n}-Queens Problem")

self.board = [[0] \* n for \_ in range(n)]

self.buttons = [[None for \_ in range(n)] for \_ in range(n)]

self.setup\_board()

def setup\_board(self):

for row in range(self.n):

for col in range(self.n):

btn = tk.Button(self.root, width=4, height=2, bg="white",

command=lambda r=row, c=col: self.toggle\_queen(r, c))

btn.grid(row=row, column=col)

self.buttons[row][col] = btn

def toggle\_queen(self, row, col):

if self.board[row][col] == 1:

self.board[row][col] = 0

self.buttons[row][col].config(text="", bg="white")

else:

if self.is\_safe(row, col):

self.board[row][col] = 1

self.buttons[row][col].config(text="Q", bg="lightblue")

if self.check\_solution():

messagebox.showinfo("Success", "All queens are placed safely!")

else:

messagebox.showwarning("Invalid Move", "This position is not safe!")

def is\_safe(self, row, col):

for i in range(self.n):

if self.board[row][i] == 1 or self.board[i][col] == 1:

return False

for i, j in zip(range(row, -1, -1), range(col, -1, -1)):

if self.board[i][j] == 1:

return False

for i, j in zip(range(row, -1, -1), range(col, self.n)):

if self.board[i][j] == 1:

return False

return True

def check\_solution(self):

queens = sum(row.count(1) for row in self.board)

return queens == self.n

def run(self):

self.root.mainloop()

if \_\_name\_\_ == "\_\_main\_\_":

n = 4

app = NQueensGUI(n)

app.run()

a



1. **Magic square:**

import tkinter as tk

from tkinter import messagebox, simpledialog

class MagicSquareGUI:

def \_\_init\_\_(self, n, magic\_sum):

if n < 3 or n % 2 == 0:

raise ValueError("Magic squares are only defined for odd numbers greater than or equal to 3.")

self.n = n

self.magic\_sum = magic\_sum

self.square = [[0] \* n for \_ in range(n)]

self.root = tk.Tk()

self.root.title(f"{n}x{n} Magic Square with Sum {magic\_sum}")

self.create\_magic\_square()

self.adjust\_to\_magic\_sum()

self.setup\_gui()

def create\_magic\_square(self):

n = self.n

num = 1

i, j = 0, n // 2

while num <= n \*\* 2:

self.square[i][j] = num

num += 1

new\_i = (i - 1) % n

new\_j = (j + 1) % n

if self.square[new\_i][new\_j]:

i += 1

else:

i, j = new\_i, new\_j

def adjust\_to\_magic\_sum(self):

"""Scales the base magic square to match the user-specified magic sum."""

base\_sum = self.n \* (self.n \*\* 2 + 1) // 2 # Standard magic sum formula

factor = self.magic\_sum / base\_sum

for i in range(self.n):

for j in range(self.n):

self.square[i][j] = int(self.square[i][j] \* factor)

def setup\_gui(self):

for i in range(self.n):

for j in range(self.n):

tk.Label(self.root, text=self.square[i][j], font=("Arial", 18),

borderwidth=2, relief="solid", width=4, height=2).grid(row=i, column=j)

def run(self):

self.root.mainloop()

if \_\_name\_\_ == "\_\_main\_\_":

n = 3 # You can change this value for larger magic squares

try:

magic\_sum = simpledialog.askinteger("Input", f"Enter the desired sum for {n}x{n} magic square:")

if magic\_sum is None:

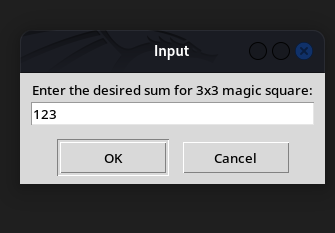
raise ValueError("No sum provided.")

app = MagicSquareGUI(n, magic\_sum)

app.run()

except ValueError as e:

messagebox.showerror("Error", str(e))

****

# 

# Conclusion:

The implementation of non-AI techniques for Tic Tac Toe, N Queens, and Magic Square demonstrates how traditional algorithmic approaches effectively solve well-defined problems.

* 1. Tic Tac Toe: A rule-based approach ensures efficient gameplay, validating moves and determining game outcomes without requiring advanced computational models. This illustrates how deterministic logic can manage simple strategy games.
  2. N Queens: The backtracking algorithm efficiently finds all possible arrangements of queens on the board, ensuring no two queens threaten each other. This highlights the power of recursion and constraint satisfaction for combinatorial problems.
  3. Magic Square: A mathematical formula or iterative construction method generates magic squares systematically, emphasizing the importance of analytical problem-solving in constrained number arrangements.

These solutions underscore the strength of non-AI techniques in providing precise, predictable, and computationally efficient solutions to structured problems without the complexity of machine learning or AI-driven methods.