EXPERIMENT 4

CODE:

```
import heapq

import random

import networkx as nx

import matplotlib.pyplot as plt


def a_star_water_jug(jug1, jug2, target):

    def heuristic(state):

        return abs(state[0] - target) + abs(state[1] - target)


    start = (0, 0)

    queue = [(heuristic(start), 0, start, [])]

    visited = set()

    total_cost = 0


    print("\nRules of the Water Jug Problem:")

    print("1. You can fill a jug to its full capacity.")

    print("2. You can empty a jug completely.")

    print("3. You can transfer water from one jug to another until the receiving jug is full or the pouring jug is empty.")

    print("\nSolution Steps:")


    while queue:

        _, cost, (a, b), path = heapq.heappop(queue)

        if (a, b) in visited:

            continue

        visited.add((a, b))

        path = path + [(a, b)]

        total_cost = cost
```

```
        print(f"Current State: Jug1 = {a}, Jug2 = {b}, Cost = {cost}")

        if a == target or b == target:
            print(f"\nTotal Cost of Operations: {total_cost}")
            return path, total_cost

        possible_moves = [
            (jug1, b),  # Fill jug1
            (a, jug2),  # Fill jug2
            (0, b),     # Empty jug1
            (a, 0),     # Empty jug2
            (a - min(a, jug2 - b), b + min(a, jug2 - b)),  # Pour jug1 to jug2
            (a + min(b, jug1 - a), b - min(b, jug1 - a))   # Pour jug2 to jug1
        ]

        for move in possible_moves:
            if move not in visited:
                heapq.heappush(queue, (cost + heuristic(move), cost + 1, move, path))
    return None, None

def generate_sparse_graph(n=20, density=0.2):
    # Create an empty graph
    G = nx.Graph()
    G.add_nodes_from(range(n))

    # Generate edges until we reach desired density
    target_edges = int(n * (n-1) * density / 2)
    edges_added = 0
```

```
    # Ensure graph is connected first
    nodes = list(range(n))
    random.shuffle(nodes)
    for i in range(len(nodes)-1):
        weight = random.randint(1, 20)
        G.add_edge(nodes[i], nodes[i+1], weight=weight)
        edges_added += 1

    # Add random edges until desired density
    while edges_added < target_edges:
        u, v = random.sample(range(n), 2)
        if not G.has_edge(u, v):
            weight = random.randint(1, 20)
            G.add_edge(u, v, weight=weight)
            edges_added += 1

    # Convert to dictionary format
    graph_dict = {i: {} for i in range(n)}
    for (u, v, w) in G.edges.data('weight'):
        graph_dict[u][v] = w
        graph_dict[v][u] = w

    return graph_dict, G

def a_star_sparse_graph(graph, start, goal):
    def heuristic(node):
        return abs(goal - node)

    queue = [(heuristic(start), 0, start, [])]
```

```
    visited = set()

    while queue:
        _, cost, node, path = heapq.heappop(queue)
        if node in visited:
            continue
        visited.add(node)
        path = path + [node]

        if node == goal:
            return path, cost

        for neighbor, weight in graph[node].items():
            if neighbor not in visited:
                heapq.heappush(queue, (cost + weight + heuristic(neighbor), cost + weight,
neighbor, path))
    return None, None

def display_graph(G, shortest_path, start, goal):
    # Dramatically increased figure size for maximum spacing
    plt.figure(figsize=(20, 16))

    # Increased k parameter significantly for maximum node separation
    pos = nx.spring_layout(G, k=5.0, iterations=200, seed=42)

    # Draw the base graph with increased spacing and sizes
    nx.draw(G, pos, with_labels=True,
            node_color='lightblue',
            edge_color='gray',
            node_size=1200,       # Larger nodes
```

```
    font_size=14,        # Larger node labels
    width=1.5,           # Slightly thicker edges
    alpha=0.9)           # Slightly more opaque

# Draw edge weights with more space and larger font
edge_labels = nx.get_edge_attributes(G, 'weight')
nx.draw_networkx_edge_labels(G, pos,
                   edge_labels=edge_labels,
                   font_size=12,     # Larger weight labels
                   label_pos=0.5,    # Centered on edges
                   bbox=dict(facecolor='white',
                        edgecolor='none',
                        alpha=0.7,  # Background for weight labels
                        pad=2))

# Draw the shortest path if it exists
if shortest_path and len(shortest_path) > 1:
  path_edges = list(zip(shortest_path[:-1], shortest_path[1:]))
  nx.draw_networkx_edges(G, pos,
                 edgelist=path_edges,
                 edge_color='red',
                 width=3.5)

# Highlight start and goal nodes with larger sizes
nx.draw_networkx_nodes(G, pos,
               nodelist=[start],
               node_color='green',
               node_size=1500)
nx.draw_networkx_nodes(G, pos,
```

```
                    nodelist=[goal],
                    node_color='orange',
                    node_size=1500)

    plt.title("Sparse Graph with Shortest Path",
          pad=20,
          size=16,
          fontweight='bold')

    # Added more padding around the graph
    plt.margins(0.2)

    # Increased legend size and moved it outside the graph
    plt.legend(['Nodes', 'Start Node', 'Goal Node', 'Shortest Path'],
          fontsize=12,
          bbox_to_anchor=(1.1, 1.05))

    plt.axis('off')
    # Adjusted layout to account for legend
    plt.tight_layout()
    plt.show()

def main():
    # Water Jug Problem
    print("Water Jug Problem Solver")
    print("-----------------------")
    jug1_capacity = int(input("Enter capacity of first jug: "))
    jug2_capacity = int(input("Enter capacity of second jug: "))
    target_amount = int(input("Enter target amount: "))
```

```
    solution, water_cost = a_star_water_jug(jug1_capacity, jug2_capacity, target_amount)
    if solution:
        print("\nWater Jug Solution Path:", solution)
    else:
        print("\nNo solution exists for the given parameters.")

    # Sparse Graph Problem
    print("\nSparse Graph Problem Solver")
    print("-------------------------")
    graph_dict, G = generate_sparse_graph()
    start_node, goal_node = random.sample(range(20), 2)
    print(f"Randomly Selected Start Node: {start_node}, Goal Node: {goal_node}")

    shortest_path, path_cost = a_star_sparse_graph(graph_dict, start_node, goal_node)
    if shortest_path:
        print(f"\nShortest Path Found: {' -> '.join(map(str, shortest_path))}")
        print(f"Total Path Cost: {path_cost}")
        display_graph(G, shortest_path, start_node, goal_node)
    else:
        print(f"\nNo path exists between nodes {start_node} and {goal_node}")

if __name__ == "__main__":
    main()
```

OUTPUT:

Water Jug Problem Solver

Enter capacity of first jug: 4

Enter capacity of second jug: 3

Enter target amount: 2

Rules of the Water Jug Problem:

1. You can fill a jug to its full capacity.

2. You can empty a jug completely.

3. You can transfer water from one jug to another until the receiving jug is full or the pouring jug is empty.

Solution Steps:

Current State: Jug1 = 0, Jug2 = 0, Cost = 0

Current State: Jug1 = 0, Jug2 = 3, Cost = 1

Current State: Jug1 = 4, Jug2 = 0, Cost = 1

Current State: Jug1 = 1, Jug2 = 3, Cost = 2

Current State: Jug1 = 3, Jug2 = 0, Cost = 2

Current State: Jug1 = 4, Jug2 = 3, Cost = 2

Current State: Jug1 = 3, Jug2 = 3, Cost = 3

Current State: Jug1 = 1, Jug2 = 0, Cost = 3

Current State: Jug1 = 4, Jug2 = 2, Cost =

Total Cost of Operations: 4

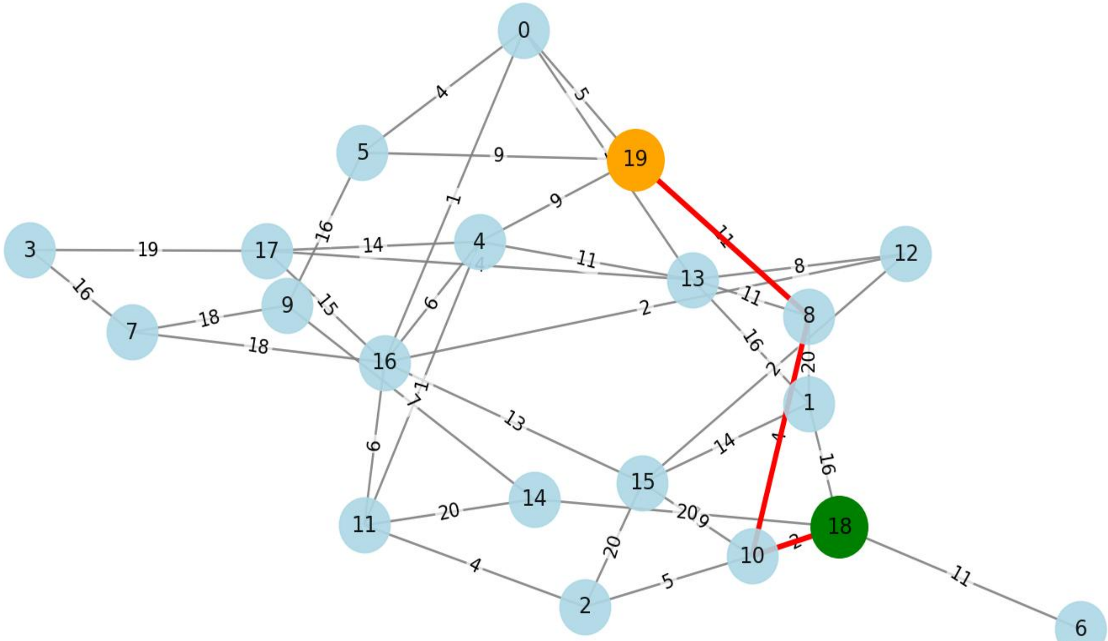
Water Jug Solution Path: [(0, 0), (0, 3), (3, 0), (3, 3), (4, 2)

Sparse Graph Problem Solver

Randomly Selected Start Node: 18, Goal Node: 19

Shortest Path Found: 18 -> 10 -> 8 -> 19

Total Path Cost: 17



0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19

EXPERIMENT 5

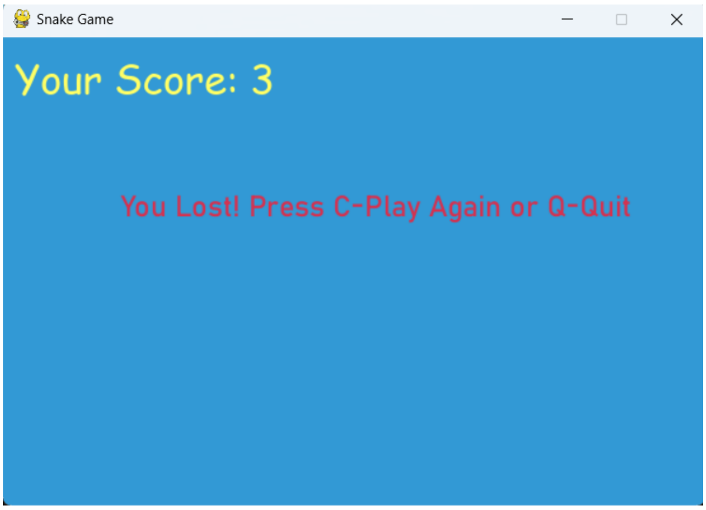
CODE:

```
import pygame

import random

import mat

# Initialize pygame

pygame.init()


# Game Constants

WIDTH, HEIGHT = 600, 400

GRID_SIZE = 20

WHITE, GREEN, RED, BLACK = (255, 255, 255), (34, 139, 34), (255, 0, 0), (0, 0, 0)


# Set up display

screen = pygame.display.set_mode((WIDTH, HEIGHT))

pygame.display.set_caption("Realistic Snake Game")


# Snake properties

snake = [(100, 100), (90, 100), (80, 100)]  # List of segments (x, y)

direction = (GRID_SIZE, 0)  # Moving right initially

food = (random.randrange(0, WIDTH, GRID_SIZE), random.randrange(0, HEIGHT, GRID_SIZE))

clock = pygame.time.Clock()

running = True

score = 0


# Function to draw snake with smooth curves

def draw_snake(snake):

    for i, segment in enumerate(snake):

        size = GRID_SIZE // 2
```

```
        pygame.draw.circle(screen, GREEN, (segment[0] + size, segment[1] + size), size)
        if i == 0:  # Add eyes to the snake head
            eye_offset = 5
            pygame.draw.circle(screen, WHITE, (segment[0] + size - eye_offset, segment[1] + size
- eye_offset), 3)
            pygame.draw.circle(screen, WHITE, (segment[0] + size + eye_offset, segment[1] + size
- eye_offset), 3)

# Function to move snake smoothly
def move_snake():
    global running, food, score
    new_head = (snake[0][0] + direction[0], snake[0][1] + direction[1])

    # Check for collisions (wall or self)
    if (
        new_head in snake or
        new_head[0] < 0 or new_head[0] >= WIDTH or
        new_head[1] < 0 or new_head[1] >= HEIGHT
    ):
        running = False

    snake.insert(0, new_head)  # Move forward

    # Check if food is eaten
    if new_head == food:
        score += 1
        food = (random.randrange(0, WIDTH, GRID_SIZE), random.randrange(0, HEIGHT,
GRID_SIZE))
    else:
        snake.pop()  # Remove tail if no food eaten
```

```
# Main game loop
while running:
   screen.fill(BLACK)

   # Handle events
   for event in pygame.event.get():
      if event.type == pygame.QUIT:
         running = False
      elif event.type == pygame.KEYDOWN:
         if event.key == pygame.K_UP and direction != (0, GRID_SIZE):
            direction = (0, -GRID_SIZE)
         elif event.key == pygame.K_DOWN and direction != (0, -GRID_SIZE):
            direction = (0, GRID_SIZE)
         elif event.key == pygame.K_LEFT and direction != (GRID_SIZE, 0):
            direction = (-GRID_SIZE, 0)
         elif event.key == pygame.K_RIGHT and direction != (-GRID_SIZE, 0):
            direction = (GRID_SIZE, 0)

   move_snake()

   # Draw food
   pygame.draw.circle(screen, RED, (food[0] + GRID_SIZE // 2, food[1] + GRID_SIZE // 2),
GRID_SIZE // 2
   # Draw snake with smoother look
   draw_snake(snake)
   pygame.display.flip()
   clock.tick(10)  # Adjust speed
pygame.quit()
print(f"Game Over! Your Score: {score}")
```

OUTPUT



Game Over! Your Score: 3

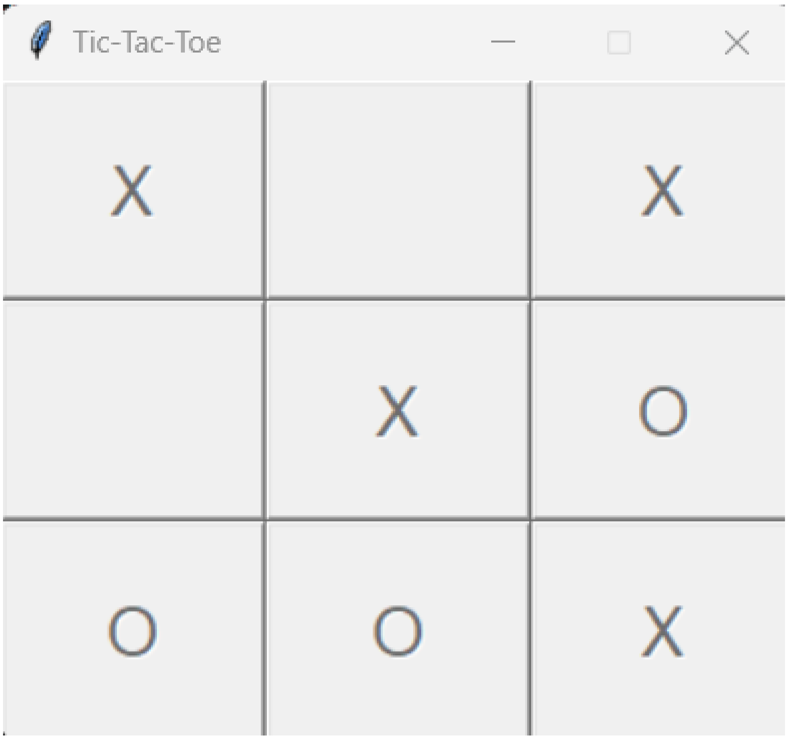
<u>**EXPERIMENT 6**</u>

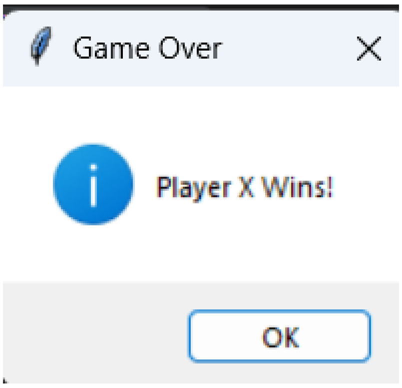
<u>CODE</u>

```
import tkinter as tk

from tkinter import messagebox


# Create the main window

root = tk.Tk()

root.title("Tic-Tac-Toe")

root.resizable(False, False)


# Game variables

current_player = "X"

board = [""] * 9


# Function to check for a winner

def check_winner():

    win_conditions = [(0,1,2), (3,4,5), (6,7,8),  # Rows

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

                      (0,4,8), (2,4,6)]           # Diagonals


    for a, b, c in win_conditions:

        if board[a] == board[b] == board[c] and board[a] != "":

            return board[a]  # Return the winning player (X or O)

    if "" not in board:

        return "Tie"  # If no spaces left, it's a tie

    return None  # No winner yet

# Function to handle button click

def on_click(index):

    global current_player
```

```
    if board[index] == "" and not check_winner():  # Ensure the cell is empty
        board[index] = current_player
        buttons[index].config(text=current_player, state="disabled")
        winner = check_winner()
        if winner:
            if winner == "Tie":
                messagebox.showinfo("Game Over", "It's a Tie!")
            else:
                messagebox.showinfo("Game Over", f"Player {winner} Wins!")
            reset_game()
        else:
            current_player = "O" if current_player == "X" else "X"  # Switch turns
# Function to reset the game
def reset_game():
    global current_player, board
    current_player = "X"
    board = [""] * 9
    for button in buttons:
        button.config(text="", state="normal")
# Create buttons
buttons = []
for i in range(9):
    btn = tk.Button(root, text="", font=("Arial", 20), width=6, height=2,
                    command=lambda i=i: on_click(i))
    btn.grid(row=i//3, column=i%3)
    buttons.append(btn)
# Run the Tkinter event loop
root.mainloop()
```

OUTPUT





EXPERIMENT 7

CODE

```
from itertools import combinations


def knapsack_brute_force(weights, values, capacity):
    n = len(weights)
    max_value = 0
    best_combination = []

    # Generate all subsets of items
    for i in range(1, n + 1):
        for subset in combinations(range(n), i):  # All subsets of size i
            total_weight = sum(weights[j] for j in subset)
            total_value = sum(values[j] for j in subset)
            if total_weight <= capacity and total_value > max_value:
                max_value = total_value
                best_combination = subset
    return max_value, best_combination


# User input
n = int(input("Enter the number of items: "))
weights = []
values = []
print("Enter weight and value for each item:")
for i in range(n):
    w, v = map(int, input(f"Item {i+1} (weight value): ").split())
    weights.append(w)
    values.append(v)
capacity = int(input("Enter knapsack capacity: ")
```

```
# Solve Knapsack
max_value, items = knapsack_brute_force(weights, values, capacity
# Display result
print(f"\nMaximum Value: {max_value}")
print(f"Selected Items: {[i+1 for i in items]} (1-based index)")
```

OUTPUT:

```
Enter the number of items: 5
Enter weight and value for each item:
Item 1 (weight value): 10 8
Item 2 (weight value): 10 5
Item 3 (weight value): 20 6
Item 4 (weight value): 15 7
Item 5 (weight value): 9 6
Enter knapsack capacity: 40

Maximum Value: 21
Selected Items: [1, 4, 5] (1-based index)
```

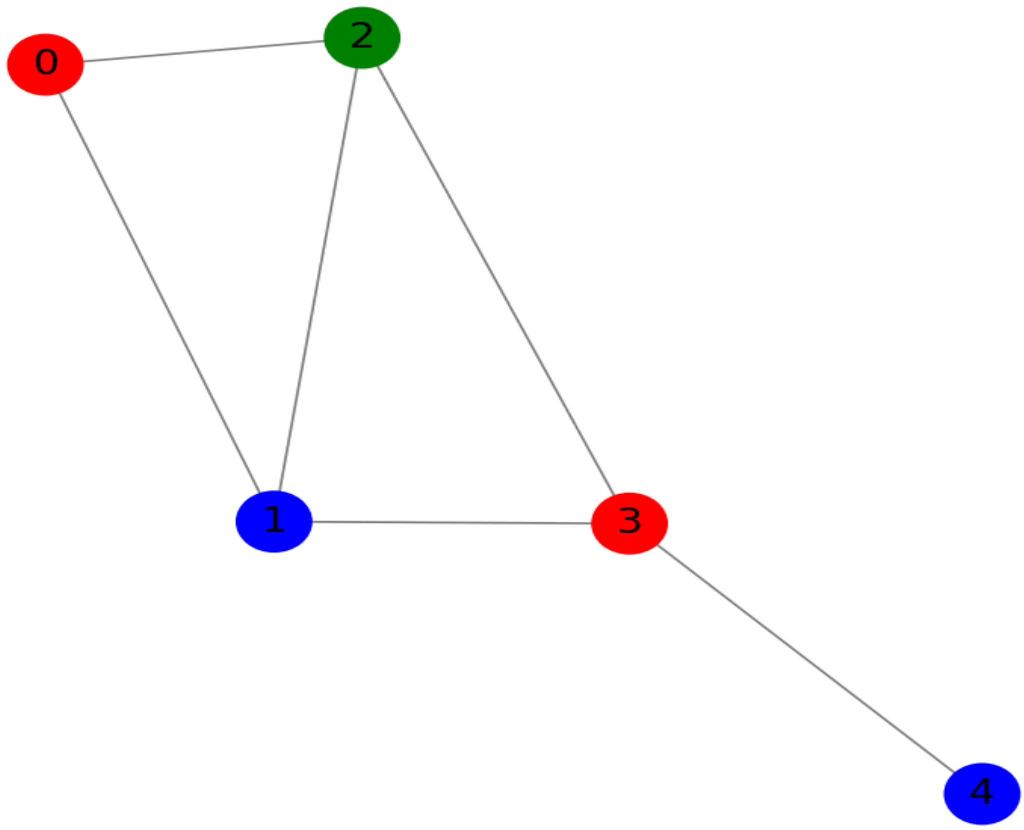
EXPERIMENT 8

CODE

```
import networkx as nx
import matplotlib.pyplot as plt

def is_safe(graph, vertex, color, colors):
    """Check if assigning 'color' to 'vertex' is valid"""
    for neighbor in graph.neighbors(vertex):
        if colors[neighbor] == color:
            return False
    return True

def graph_coloring_util(graph, m, colors, vertex):
    """Recursive backtracking function to assign colors"""
    if vertex == len(graph):  # All vertices are colored
        return True

    for color in range(1, m + 1):  # Try all colors (1 to m)
        if is_safe(graph, vertex, color, colors):
            colors[vertex] = color  # Assign color

            if graph_coloring_util(graph, m, colors, vertex + 1):
                return True

            colors[vertex] = 0  # Backtrack

    return False

def graph_coloring(graph, m):
```

```python
    """Main function to solve graph coloring and visualize it"""
    colors = {node: 0 for node in graph.nodes()}  # Initialize colors

    if not graph_coloring_util(graph, m, colors, 0):
        print("No solution found!")
        return

    # Assign colors
    color_map = {1: "red", 2: "blue", 3: "green", 4: "yellow", 5: "purple"}
    node_colors = [color_map[colors[node]] for node in graph.nodes()]

    # Draw the graph
    plt.figure(figsize=(6, 6))
    pos = nx.spring_layout(graph)
    nx.draw(graph, pos, with_labels=True, node_color=node_colors, node_size=800,
font_size=16, edge_color="gray")
    plt.show()

# Example graph (Adjacency list representation)
G = nx.Graph()
edges = [(0, 1), (0, 2), (1, 2), (1, 3), (2, 3), (3, 4)]
G.add_edges_from(edges)

m = 3  # Number of colors
graph_coloring(G, m)
```

OUTPUT:



EXPERIMENT 9

CODE:

```
from collections import deque


def water_jug_bfs(jug1, jug2, target):
    """Solves the water jug problem using BFS"""
    visited = set()  # To track visited states
    queue = deque([(0, 0)])  # Start state (both jugs empty)

    while queue:
        x, y = queue.popleft()  # Current state of jugs
        # If we reach the target, return the solution
        if x == target or y == target:
            print(f"Solution found: Jug1 = {x}, Jug2 = {y}")
            return

        # If state is already visited, skip it
        if (x, y) in visited:
            continue

        visited.add((x, y))  # Mark state as visited
        # Generate possible states
        next_states = set([
            (jug1, y),     # Fill jug1
            (x, jug2),     # Fill jug2
            (0, y),        # Empty jug1
            (x, 0),        # Empty jug2
            (x - min(x, jug2 - y), y + min(x, jug2 - y)),  # Pour from jug1 to jug2
            (x + min(y, jug1 - x), y - min(y, jug1 - x))   # Pour from jug2 to jug1
```

```
        ])
        # Add new states to the queue
        for state in next_states:
            if state not in visited:
                queue.append(state)
                print(f"New State: Jug1 = {state[0]}, Jug2 = {state[1]}")  # Show steps

    print("No solution found.")

# Input: Jug capacities and target
jug1_capacity = int(input("Enter capacity of Jug 1: "))
jug2_capacity = int(input("Enter capacity of Jug 2: "))
target_amount = int(input("Enter target amount of water: "))
# Solve the problem
water_jug_bfs(jug1_capacity, jug2_capacity, target_amount)
```

OUTPUT:

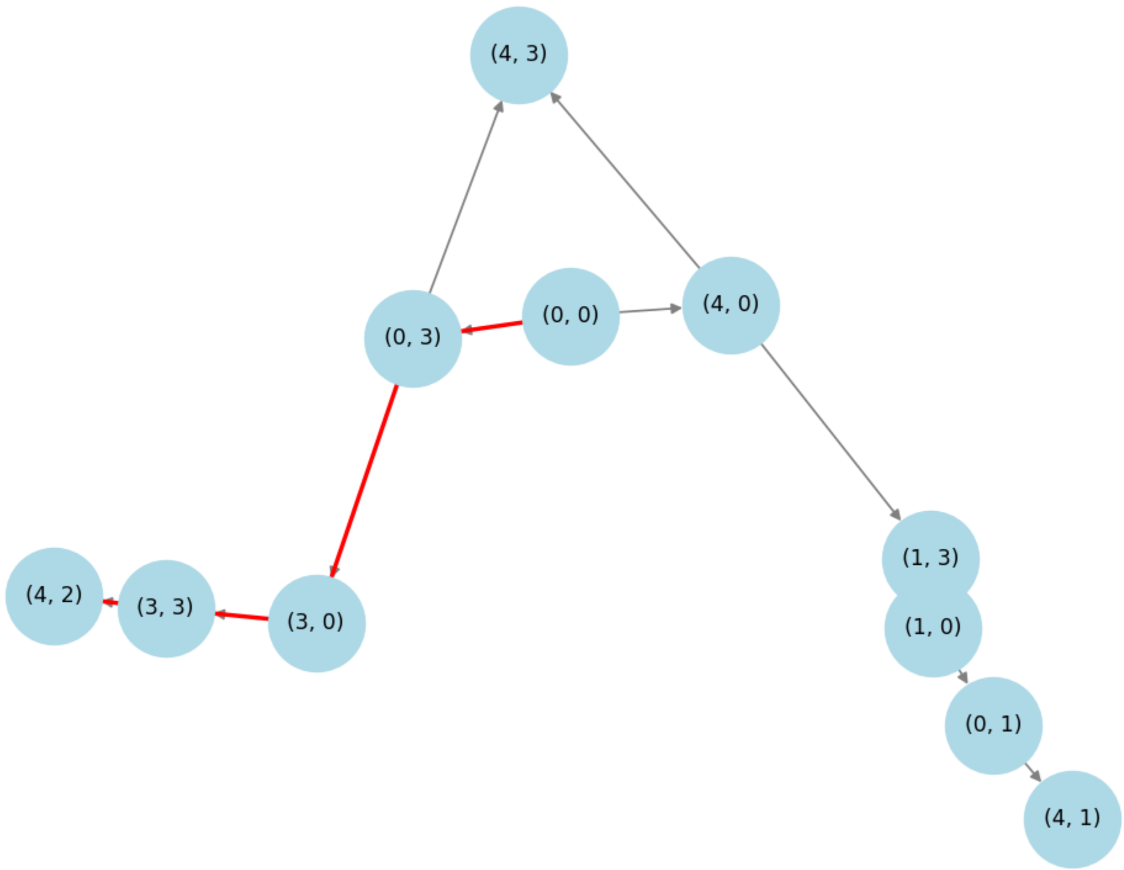
```
Enter capacity of Jug 1: 4
Enter capacity of Jug 2: 3
Enter target amount of water: 2
New State: Jug1 = 4, Jug2 = 0
New State: Jug1 = 0, Jug2 = 3
New State: Jug1 = 1, Jug2 = 3
New State: Jug1 = 4, Jug2 = 3
New State: Jug1 = 3, Jug2 = 0
New State: Jug1 = 4, Jug2 = 3
New State: Jug1 = 4, Jug2 = 3
New State: Jug1 = 1, Jug2 = 0
New State: Jug1 = 3, Jug2 = 3
```

New State: Jug1 = 0, Jug2 = 1

New State: Jug1 = 4, Jug2 = 2

New State: Jug1 = 4, Jug2 = 1

Solution found: Jug1 = 4, Jug2 = 2



EXPERIMENT 10

CODE:

```
import tkinter as tk
from tkinter import messagebox
import random
class UserNavigableMaze:
  def __init__(self, root):
    self.root = root
    self.root.title("Maze Navigation Game”)
    # Maze parameters
    self.rows = 15
    self.cols = 15
    self.cell_size = 30
    # Maze representation
    self.maze = [[0 for _ in range(self.cols)] for _ in range(self.rows)]
    self.start_pos = (0, 0)
    self.end_pos = (self.rows-1, self.cols-1)
    self.player_pos = self.start_pos
    # Colors
    self.colors = {
      0: "white",    # Path
      1: "black",    # Wall
      "start": "green",
      "end": "red",
      "player": "blue",
      "visited": "light grey"
    }
    # Track visited cells
    self.visited = [[False for _ in range(self.cols)] for _ in range(self.rows)]
```

```
        self.visited[self.start_pos[0]][self.start_pos[1]] = True
        # Track moves
        self.moves = 0
        # Create control frame
        self.control_frame = tk.Frame(root)
        self.control_frame.pack(side=tk.TOP, fill=tk.X)
        # Create buttons
        self.generate_btn = tk.Button(self.control_frame, text="Generate New Maze",
command=self.generate_random_maze)
        self.generate_btn.pack(side=tk.LEFT, padx=5, pady=5)
        self.reset_btn = tk.Button(self.control_frame, text="Reset Player",
command=self.reset_player)
        self.reset_btn.pack(side=tk.LEFT, padx=5, pady=5)
        # Create info frame
        self.info_frame = tk.Frame(root)
        self.info_frame.pack(side=tk.TOP, fill=tk.X)
        # Moves counter
        self.moves_label = tk.Label(self.info_frame, text="Moves: 0")
        self.moves_label.pack(side=tk.LEFT, padx=5, pady=5)
        # Create canvas for the maze
        self.canvas_width = self.cols * self.cell_size
        self.canvas_height = self.rows * self.cell_size
        self.canvas = tk.Canvas(root, width=self.canvas_width, height=self.canvas_height,
bg="white")
        self.canvas.pack(padx=10, pady=10)

        # Instructions
        instruction_text = "Use arrow keys to navigate to the red exit. Press 'h' for hint."
        self.instructions = tk.Label(root, text=instruction_text)
        self.instructions.pack(pady=5)
```

```
        # Draw grid
        self.draw_grid()
        # Generate initial maze
        self.generate_random_maze()
        # Bind keyboard events for player movement
        self.root.bind("<KeyPress>", self.on_key_press)
        # Focus on the window to capture key events
        self.root.focus_set()
    def draw_grid(self):
        for row in range(self.rows):
            for col in range(self.cols):
                x1 = col * self.cell_size
                y1 = row * self.cell_size
                x2 = x1 + self.cell_size
                y2 = y1 + self.cell_size
                self.canvas.create_rectangle(x1, y1, x2, y2, fill="white", outline="grey",
tags=f"cell_{row}_{col}")
    def update_maze(self):
        for row in range(self.rows):
            for col in range(self.cols):
                cell = f"cell_{row}_{col}"
                if (row, col) == self.player_pos:
                    color = self.colors["player"]
                elif (row, col) == self.start_pos:
                    color = self.colors["start"]
                elif (row, col) == self.end_pos:
                    color = self.colors["end"]
                elif self.visited[row][col]:
                    color = self.colors["visited"]
```

```
            else:
                color = self.colors[self.maze[row][col]]
            self.canvas.itemconfig(cell, fill=color)


    def reset_player(self):
        self.player_pos = self.start_pos
        self.visited = [[False for _ in range(self.cols)] for _ in range(self.rows)]
        self.visited[self.start_pos[0]][self.start_pos[1]] = True
        self.moves = 0
        self.moves_label.config(text=f"Moves: {self.moves}")
        self.update_maze()
    def generate_random_maze(self):
        # Reset player
        self.reset_player()
        # Generate a maze using a simple algorithm
        # Start with all walls
        self.maze = [[1 for _ in range(self.cols)] for _ in range(self.rows)
        # Use a recursive backtracker algorithm (simplified)
        def carve_passages(row, col, visited):
            visited[row][col] = True
            self.maze[row][col] = 0  # Carve this cell

            # Define directions: right, down, left, up
            directions = [(0, 2), (2, 0), (0, -2), (-2, 0)]
            random.shuffle(directions)

            for dr, dc in directions:
                new_row, new_col = row + dr, col + dc
                if (0 <= new_row < self.rows and 0 <= new_col < self.cols and
```

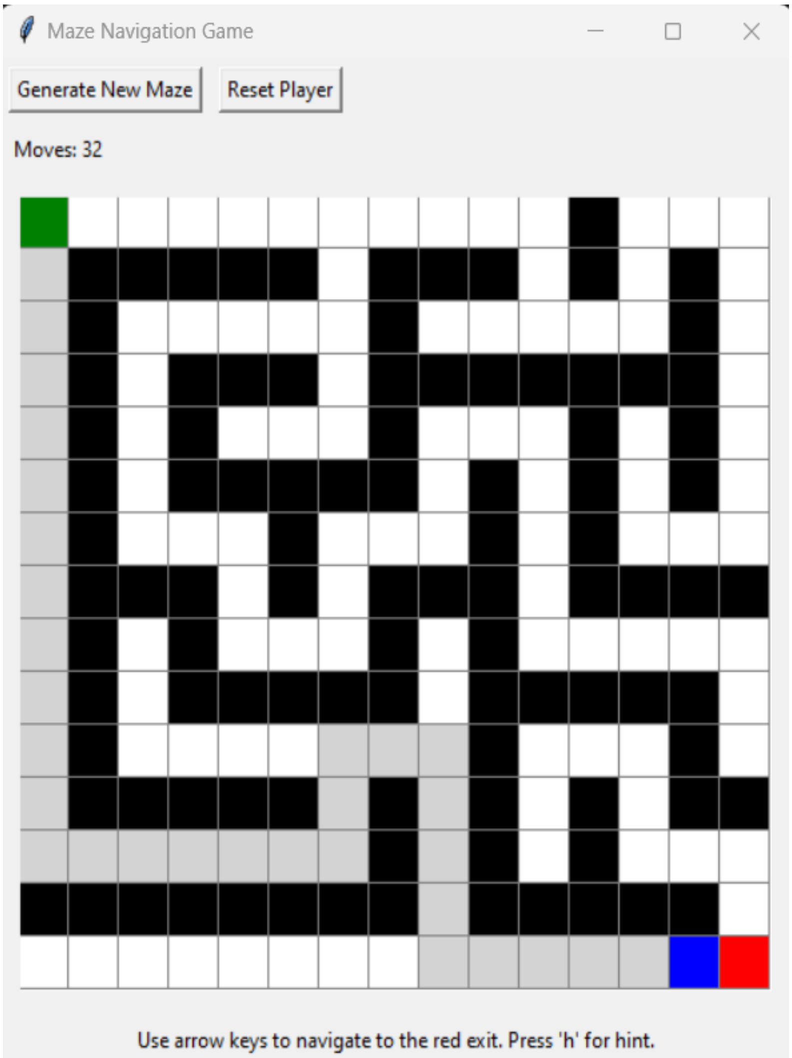
```
            not visited[new_row][new_col]):
            # Carve the wall between
            self.maze[row + dr//2][col + dc//2] = 0
            carve_passages(new_row, new_col, visited)

# Initialize visited array for maze generation
gen_visited = [[False for _ in range(self.cols)] for _ in range(self.rows)]

# Start from a random position
start_row = random.randrange(0, self.rows, 2)
start_col = random.randrange(0, self.cols, 2)
carve_passages(start_row, start_col, gen_visited)

# Ensure start and end positions are paths
self.start_pos = (0, 0)
self.end_pos = (self.rows-1, self.cols-1)
self.maze[self.start_pos[0]][self.start_pos[1]] = 0
self.maze[self.end_pos[0]][self.end_pos[1]] = 0

# Ensure there's a path near the start and end
for dr, dc in [(0, 1), (1, 0)]:
    nr, nc = self.start_pos[0] + dr, self.start_pos[1] + dc
    if 0 <= nr < self.rows and 0 <= nc < self.cols:
        self.maze[nr][nc] = 0

for dr, dc in [(0, -1), (-1, 0)]:
    nr, nc = self.end_pos[0] + dr, self.end_pos[1] + dc
    if 0 <= nr < self.rows and 0 <= nc < self.cols:
        self.maze[nr][nc] = 0
```

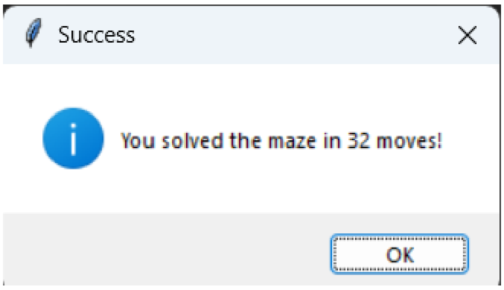
```
        # Reset player position
        self.player_pos = self.start_pos
        self.update_maze()

    def on_key_press(self, event):
        row, col = self.player_pos
        new_row, new_col = row, col

        # Handle movement
        if event.keysym == 'Right' or event.keysym == 'd':
            new_col += 1
        elif event.keysym == 'Left' or event.keysym == 'a':
            new_col -= 1
        elif event.keysym == 'Down' or event.keysym == 's':
            new_row += 1
        elif event.keysym == 'Up' or event.keysym == 'w':
            new_row -= 1
        elif event.keysym == 'h':  # Hint button
            self.show_hint()
            return

        # Check if the new position is valid
        if (0 <= new_row < self.rows and 0 <= new_col < self.cols and
            self.maze[new_row][new_col] == 0):
            self.player_pos = (new_row, new_col)
            self.visited[new_row][new_col] = True
            self.moves += 1
            self.moves_label.config(text=f"Moves: {self.moves}")
```

```
        # Check if reached the end
        if self.player_pos == self.end_pos:
            messagebox.showinfo("Success", f"You solved the maze in {self.moves} moves!")
            self.generate_random_maze()

        self.update_maze()

    def show_hint(self):
        # Use DFS to find a path from current position to end
        visited = [[False for _ in range(self.cols)] for _ in range(self.rows)]
        path = []

        def dfs(row, col, end_row, end_col, visited, path):
            if (row < 0 or row >= self.rows or col < 0 or col >= self.cols or
                self.maze[row][col] == 1 or visited[row][col]):
                return False

            visited[row][col] = True
            path.append((row, col))

            if row == end_row and col == end_col:
                return True

            for dr, dc in [(0, 1), (1, 0), (0, -1), (-1, 0)]:
                if dfs(row + dr, col + dc, end_row, end_col, visited, path):
                    return True

            path.pop()
```

```
            return False

        current_row, current_col = self.player_pos
        end_row, end_col = self.end_pos

        if dfs(current_row, current_col, end_row, end_col, visited, path) and len(path) > 1:
            # Highlight the next step
            next_step = path[1]  # path[0] is the current position
            hint_cell = f"cell_{next_step[0]}_{next_step[1]}"
            original_color = self.canvas.itemcget(hint_cell, "fill")

            # Flash the hint
            for _ in range(3):
                self.canvas.itemconfig(hint_cell, fill="yellow")
                self.root.update()
                self.root.after(200)
                self.canvas.itemconfig(hint_cell, fill=original_color)
                self.root.update()
                self.root.after(200)

            # Update the maze display
            self.update_maze()
        else:
            messagebox.showinfo("Hint", "No path found to the exit!")

if __name__ == "__main__":
    root = tk.Tk()
    app = UserNavigableMaze(root)
    root.mainloop()
```

OUTPUT:


Maze Navigation Game
Generate New Maze
Reset Player
Moves: 32
Use arrow keys to navigate to the red exit. Press 'h' for hint.


Success
You solved the maze in 32 moves!
OK

EXPERIMENT 1

CODE:

```
from collections import deque

from typing import List, Set, Tuple, Optional

import datetime


class State:

    def __init__(self, m_left: int, c_left: int, boat: bool, m_right: int, c_right: int,
parent=None):

        self.m_left = m_left

        self.c_left = c_left

        self.boat = boat

        self.m_right = m_right

        self.c_right = c_right

        self.parent = parent


    def __eq__(self, other):

        if not isinstance(other, State):

            return False

        return (self.m_left == other.m_left and

                self.c_left == other.c_left and

                self.boat == other.boat and

                self.m_right == other.m_right and

                self.c_right == other.c_right)


    def __hash__(self):

        return hash((self.m_left, self.c_left, self.boat, self.m_right, self.c_right))


    def __str__(self):

        left_bank = f"M:{self.m_left} C:{self.c_left}"
```

```python
        right_bank = f"M:{self.m_right} C:{self.c_right}"
        boat = "B" if self.boat else " "
        return f"[{left_bank}] {boat}~~~ [{right_bank}]"

def is_valid_state(state: State, total_missionaries: int, total_cannibals: int) -> bool:
    if (state.m_left < 0 or state.c_left < 0 or
        state.m_right < 0 or state.c_right < 0 or
        state.m_left + state.m_right > total_missionaries or
        state.c_left + state.c_right > total_cannibals):
        return False

    if (state.m_left > 0 and state.m_left < state.c_left) or \
       (state.m_right > 0 and state.m_right < state.c_right):
        return False

    return True

def get_next_states(current: State, total_missionaries: int, total_cannibals: int,
boat_capacity: int) -> List[State]:
    next_states = []
    moves = [(1, 0), (2, 0), (0, 1), (0, 2), (1, 1)]  # Possible moves: (missionaries, cannibals)

    for m, c in moves:
        # Ensure the boat doesn't carry more than its capacity
        if m + c > boat_capacity:
            continue

        new_state = None
        if current.boat:
            new_state = State(
```

```
            current.m_left - m,
            current.c_left - c,
            False,
            current.m_right + m,
            current.c_right + c,
            current
        )
    else:
        new_state = State(
            current.m_left + m,
            current.c_left + c,
            True,
            current.m_right - m,
            current.c_right - c,
            current
        )

    if is_valid_state(new_state, total_missionaries, total_cannibals):
        next_states.append(new_state)

return next_states

def print_and_save_tree(initial_state: State, goal_state: State, file, total_missionaries: int,
total_cannibals: int, boat_capacity: int):
    visited = set()
    queue = deque([initial_state])
    level = 0
    level_nodes = 1
    next_level_nodes = 0
```

```
    header = "\nState Space Search Tree:"
    separator = "=" * 50

    print(header)
    print(separator)
    file.write(header + "\n")
    file.write(separator + "\n")

    while queue:
        current = queue.popleft()
        level_nodes -= 1

        prefix = "  " * level
        state_str = f"{prefix} ├─ {current}"
        print(state_str)
        file.write(state_str + "\n")

        if current == goal_state:
            result = f"\nGoal state reached at level {level}!"
            print(result)
            file.write(result + "\n")
            return

        if current not in visited:
            visited.add(current)
            next_states = get_next_states(current, total_missionaries, total_cannibals, boat_capacity)
            queue.extend(next_states)
            next_level_nodes += len(next_states)
```

```
        if level_nodes == 0:
            level += 1
            level_nodes = next_level_nodes
            next_level_nodes = 0

def get_valid_input(prompt: str, max_value: int) -> int:
    while True:
        try:
            value = int(input(prompt))
            if 0 <= value <= max_value:
                return value
            print(f"Please enter a number between 0 and {max_value}")
        except ValueError:
            print("Please enter a valid number")

def main():
    # Create a unique filename with timestamp
    timestamp = datetime.datetime.now().strftime("%Y%m%d_%H%M%S")
    filename = f"missionary_cannibals_output_{timestamp}.txt"

    print("\nMissionaries and Cannibals Problem - Input Parameters")
    print("=" * 50)

    with open(filename, 'w', encoding='utf-8') as file:
        file.write("Missionaries and Cannibals Problem - User Input\n")
        file.write("=" * 50 + "\n\n")

        # Get total number of missionaries and cannibals
```

```
    total_missionaries = get_valid_input("Enter total number of missionaries: ", 100)
    total_cannibals = get_valid_input("Enter total number of cannibals: ", 100)
    file.write(f"Total missionaries: {total_missionaries}\n")
    file.write(f"Total cannibals: {total_cannibals}\n\n")

    # Get boat capacity input
    boat_capacity = get_valid_input("Enter boat capacity (1 to 2): ", 2)
    file.write(f"Boat capacity: {boat_capacity}\n\n")

    # Get initial state input
    print("\nEnter initial state parameters:")
    file.write("Initial State Parameters:\n")
    m_left = get_valid_input(f"Number of missionaries on left bank (0-
{total_missionaries}): ", total_missionaries)
    c_left = get_valid_input(f"Number of cannibals on left bank (0-{total_cannibals}): ",
total_cannibals)
    boat_side = input("Is boat on left bank? (y/n): ").lower() == 'y'

    file.write(f"  Missionaries on left bank: {m_left}\n")
    file.write(f"  Cannibals on left bank: {c_left}\n")
    file.write(f"  Boat on left bank: {'Yes' if boat_side else 'No'}\n\n")

    # Calculate right bank numbers
    m_right = total_missionaries - m_left
    c_right = total_cannibals - c_left

    # Create initial state
    initial_state = State(m_left, c_left, boat_side, m_right, c_right)

    # Get goal state input
```

```
    print("\nEnter goal state parameters:")
    file.write("Goal State Parameters:\n")
    m_left_goal = get_valid_input(f"Number of missionaries on left bank (0-
{total_missionaries}): ", total_missionaries)
    c_left_goal = get_valid_input(f"Number of cannibals on left bank (0-{total_cannibals}):
", total_cannibals)
    boat_side_goal = input("Is boat on left bank? (y/n): ").lower() == 'y'

    file.write(f"  Missionaries on left bank: {m_left_goal}\n")
    file.write(f"  Cannibals on left bank: {c_left_goal}\n")
    file.write(f"  Boat on left bank: {'Yes' if boat_side_goal else 'No'}\n\n")

    # Calculate right bank numbers for goal
    m_right_goal = total_missionaries - m_left_goal
    c_right_goal = total_cannibals - c_left_goal

    # Create goal state
    goal_state = State(m_left_goal, c_left_goal, boat_side_goal, m_right_goal,
c_right_goal)

    print("\nInitial state:", initial_state)
    print("Goal state:", goal_state)

    file.write("Derived States:\n")
    file.write(f"  Initial state: {initial_state}\n")
    file.write(f"  Goal state: {goal_state}\n\n")

    # Validate initial and goal states
    if not is_valid_state(initial_state, total_missionaries, total_cannibals) or \
      not is_valid_state(goal_state, total_missionaries, total_cannibals):
```

```python
        error_message = "\nError: Invalid initial or goal state configuration!"
        print(error_message)
        file.write(error_message + "\n")
        return

    # Generate and save the state space tree
    print_and_save_tree(initial_state, goal_state, file, total_missionaries, total_cannibals,
boat_capacity)

  print(f"\nOutput has been saved to: {filename}")

if __name__ == "__main__":
  main()
```

Experiment 2

Code:

```
from collections import deque
import copy
import random
import sys
from datetime import datetime

def tee_print(*args, **kwargs):
    """Print to both console and file."""
    print(*args, **kwargs)
    if 'file' in kwargs:
        print(*args)  # Also print to console if writing to file

class StateSpaceTree:
    # [Previous StateSpaceTree class implementation remains the same]
    def __init__(self, initial_state, goal_state=None):
        self.initial_state = initial_state
        self.goal_state = goal_state
        self.visited_states = set()
        self.size = len(initial_state)

    def get_state_id(self, state):
        if isinstance(state, list):
            return tuple(tuple(row) if isinstance(row, list) else row for row in state)
        return tuple(state)

    def format_state(self, state):
        if isinstance(state[0], list):
```

```
        return '\n'.join(' '.join(f"{x:2d}" for x in row) for row in state)
    return ' '.join(map(str, state))

def generate_moves(self, state):
    raise NotImplementedError("Must implement generate_moves for specific problem")

def print_tree(self, max_depth=3, output_file=None):
    queue = deque([(self.initial_state, None, 0, "")])
    self.visited_states = set()
    nodes_at_depth = {i: 0 for i in range(max_depth + 1)}
    total_nodes = 0

    tee_print("\nGenerating State Space Tree...", file=output_file)
    tee_print("=" * 40, file=output_file)

    while queue:
        current_state, parent, depth, prefix = queue.popleft()
        if depth > max_depth:
            continue

        state_id = self.get_state_id(current_state)
        if state_id in self.visited_states:
            continue

        self.visited_states.add(state_id)
        nodes_at_depth[depth] += 1
        total_nodes += 1

        tee_print(file=output_file)
```

```
            if depth == 0:
                tee_print("Root State (Depth 0):", file=output_file)
            else:
                tee_print(f"{prefix}+-- State at depth {depth}", file=output_file)
            tee_print(prefix + "    " + self.format_state(current_state).replace('\n', '\n' + prefix +
"    "), file=output_file)

            children = self.generate_moves(current_state)
            for i, child in enumerate(children):
                child_id = self.get_state_id(child)
                if child_id not in self.visited_states:
                    new_prefix = prefix + ("    " if i == len(children) - 1 else "|   ")
                    queue.append((child, current_state, depth + 1, new_prefix))

        tee_print("\nTree Generation Summary", file=output_file)
        tee_print("=" * 40, file=output_file)
        tee_print(f"Total nodes generated: {total_nodes}", file=output_file)
        for depth, count in nodes_at_depth.items():
            if count > 0:
                tee_print(f"Nodes at depth {depth}: {count}", file=output_file)

class NPuzzle(StateSpaceTree):
    def find_blank(self, state):
        for i, row in enumerate(state):
            for j, val in enumerate(row):
                if val == 0:
                    return i, j
        return None

    def generate_moves(self, state):
```

```
        moves = []
        i, j = self.find_blank(state)
        directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]

        for di, dj in directions:
            new_i, new_j = i + di, j + dj
            if 0 <= new_i < self.size and 0 <= new_j < self.size:
                new_state = [row[:] for row in state]
                new_state[i][j], new_state[new_i][new_j] = new_state[new_i][new_j],
new_state[i][j]
                moves.append(new_state)

        return moves

def is_solvable(state, goal_state, size):
    """
    Check if the puzzle is solvable using the correct logic for both odd and even sized puzzles.
    """
    def get_inversions(state):
        # Convert 2D state to 1D and remove blank (0)
        flat = [num for row in state for num in row if num != 0]
        inversions = 0
        for i in range(len(flat)):
            for j in range(i + 1, len(flat)):
                if flat[i] > flat[j]:
                    inversions += 1
        return inversions

    def get_blank_row(state):
        # Get the row number (0-based) of the blank tile from the top
```

```
        for i in range(size):
            for j in range(size):
                if state[i][j] == 0:
                    return i
        return -1

    # Get inversions count
    inversions = get_inversions(state)

    # Get blank position from top
    blank_row = get_blank_row(state)

    # For odd-sized puzzles (e.g., 3x3)
    if size % 2 == 1:
        return inversions % 2 == 0
    # For even-sized puzzles (e.g., 4x4)
    else:
        blank_row_from_bottom = size - blank_row - 1
        if blank_row_from_bottom % 2 == 0:
            return inversions % 2 == 1
        else:
            return inversions % 2 == 0

def generate_random_state(size):
    """
    Generate a random solvable puzzle state.
    """
    goal_state = [[i + j * size for i in range(1, size + 1)] for j in range(size)]
    goal_state[-1][-1] = 0  # Set last position to blank (0)
```

```
    while True:
        # Create a random state
        numbers = list(range(size * size))
        random.shuffle(numbers)
        state = [numbers[i:i + size] for i in range(0, size * size, size)]

        # Check if it's solvable
        if is_solvable(state, goal_state, size):
            return state

def demo_n_puzzle():
    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
    output_filename = f"puzzle_tree_{timestamp}.txt"

    with open(output_filename, 'w') as output_file:
        tee_print("N-Puzzle State Space Tree Demonstration", file=output_file)
        tee_print("=" * 40, file=output_file)

        while True:
            try:
                size = int(input("\nEnter grid size (2-7 supported): "))
                if size < 2:
                    print("Grid size must be at least 2")
                    continue
                if size > 7:
                    print("Warning: Sizes above 7 may be very slow and consume lots of memory")
                    if input("Do you want to continue anyway? (y/n): ").lower() != 'y':
                        continue
```

```
            break
        except ValueError:
            print("Please enter a valid number")

    # Calculate maximum value for input validation
    max_value = size * size - 1

    initial_state = generate_random_state(size)

    tee_print("\nRandomly generated initial state:", file=output_file)
    tee_print('\n'.join(' '.join(f"{x:{len(str(max_value))}d}" for x in row) for row in
initial_state), file=output_file)

    tee_print(f"\nEnter the goal state row by row (use space-separated numbers, use 0 for
empty tile)", file=output_file)
    tee_print(f"Numbers should be between 0 and {max_value}", file=output_file)

    goal_state = []
    tee_print("\nEnter goal state:", file=output_file)
    used_numbers = set()

    for i in range(size):
        while True:
            try:
                row = list(map(int, input(f"Enter row {i + 1}: ").split()))
                if len(row) != size:
                    print(f"Please enter exactly {size} numbers")
                    continue
                if not all(0 <= x <= max_value for x in row):
                    print(f"Numbers must be between 0 and {max_value}")
```

```
                continue

            # Check for duplicate numbers
            row_set = set(row)
            if len(row_set & used_numbers) > 0:
                print("Duplicate numbers are not allowed")
                continue
            used_numbers.update(row_set)

            goal_state.append(row)
            break
        except ValueError:
            print("Please enter valid numbers")

    # Verify all numbers are used
    if len(used_numbers) != size * size:
        missing = set(range(size * size)) - used_numbers
        print(f"Error: Missing numbers {missing}")
        return

    tee_print("\nEntered goal state:", file=output_file)
    tee_print('\n'.join(' '.join(f"{x:{len(str(max_value))}d}" for x in row) for row in
goal_state), file=output_file)

    if not is_solvable(initial_state, goal_state, size):
        tee_print("\nWarning: The puzzle is not solvable between the initial and goal states.",
file=output_file)
        return

    puzzle = NPuzzle(initial_state, goal_state)
```

```
    while True:
        try:
            max_depth = int(input("\nEnter maximum depth to explore (recommended 1-2 for large puzzles): "))
            if max_depth < 0:
                print("Depth must be non-negative")
                continue
            if max_depth > 3 and size > 4:
                print("Warning: Large depth values with big puzzles may be very slow")
                if input("Do you want to continue anyway? (y/n): ").lower() != 'y':
                    continue
            break
        except ValueError:
            print("Please enter a valid number")

    puzzle.print_tree(max_depth, output_file)
    tee_print(f"\nOutput has been saved to: {output_filename}", file=output_file)


if __name__ == "__main__":
    demo_n_puzzle()
```

Experiment 3

Code

```
class NQueensStateSpaceTree:
    def __init__(self, n):
        self.n = n  # Size of the chessboard
        self.tree = []  # To store the search tree in a structured way
        self.node_count = 0  # Track the total number of nodes
        self.solutions = []  # To store all valid solutions

    def is_safe(self, board, row, col):
        """Check if it's safe to place a queen at board[row][col]."""
        for i in range(row):
            if board[i] == col or \
               board[i] - i == col - row or \
               board[i] + i == col + row:
                return False
        return True

    def generate_tree(self, row=0, board=None, depth=0, parent="Root"):
        """Generate the search tree recursively."""
        if board is None:
            board = [-1] * self.n  # Initialize an empty board

        # Create a copy of the board for storing in the tree
        state = (depth, board[:], parent)
        self.tree.append(state)
        self.node_count += 1

        # If all queens are placed, save the solution
```

```
        if row == self.n:
            self.solutions.append(board[:])
            return

        # Try placing a queen in each column of the current row
        for col in range(self.n):
            if self.is_safe(board, row, col):
                board[row] = col
                self.generate_tree(row + 1, board[:], depth + 1, f"Depth {depth}")
                board[row] = -1  # Backtrack

    def print_tree(self):
        """Print the search tree in the requested format."""
        print("Generating State Space Tree...")
        print("=" * 40)
        print("\nRoot State (Depth 0):")
        self.print_board([-1] * self.n)

        # Iterate through the tree and print each node
        for depth, board, parent in self.tree:
            if depth == 0:
                continue  # Skip the root node here
            indent = "|   " * (depth - 1) + "+-- "
            print(f"{indent}State at depth {depth}")
            self.print_board(board)

        # Summary
        print("\nTree Generation Summary")
        print("=" * 40)
```

```
        print(f"Total nodes generated: {self.node_count}")
        print(f"Nodes at depth 0: 1")
        for d in range(1, self.n + 1):
            count = sum(1 for node in self.tree if node[0] == d)
            print(f"Nodes at depth {d}: {count}")
        # Display solutions
        print("\nSolutions Found")
        print("=" * 40)
        for i, solution in enumerate(self.solutions, 1):
            print(f"Solution {i}:")
            self.print_board(solution)
    def print_board(self, board):
        """Print a board representation based on the current state."""
        for row in range(self.n):
            line = ["Q" if board[row] == col else "." for col in range(self.n)]
            print("    " + " ".join(line))
        print()  # Add spacing between boards
# Input from the user
try:
    n = int(input("Enter the size of the chessboard (N): "))
    if n < 1:
        print("N must be greater than 0.")
    else:
        # Create an instance of the NQueensStateSpaceTree class
        n_queens_tree = NQueensStateSpaceTree(n)
        n_queens_tree.generate_tree()
        n_queens_tree.print_tree()
except ValueError:
    print("Please enter a valid integer.")
```