**Lab 6**

**23/07/2024**

**Brief Summary:**

The script addresses the following tasks and problems:

I. **Puzzle Game:**

 **Initialization**:

* The board is flattened into a tuple for easy manipulation and stored in a dictionary (state\_distances) with its distance from the start state initialized to 0.

 **Breadth-First Search (BFS)**:

* The get\_paths function performs BFS by exploring all possible next states from the current state.
* At each step, the algorithm checks if the current state is the goal state (0, 1, 2, 3, 4, 5, 6, 7, 8).
* If the goal state is reached, the algorithm returns the number of moves taken to reach the goal.

 **State Transitions**:

* The find\_next function generates all possible next states by moving the empty tile (0) to its adjacent positions.
* For each valid move, a new state is created and added to the list of possible next moves.

 **Main Loop**:

* In the get\_paths function, the BFS loop runs until the goal state is found or all possible states are explored.
* At each level of BFS (each distance count), the algorithm prints the possible next moves and updates the state\_distances dictionary.

 **Sleep and Print**:

* The sleep(2) and print(next\_moves) lines are added to visualize the state transitions with a 2-second delay between each step.

 **Execution Flow**

* The solve function initializes the state and calls get\_paths to start the BFS process.
* The get\_paths function repeatedly explores the current level's nodes and generates their next possible states.
* The algorithm prints the next states at each step and checks if any of them match the goal state.
* If the goal state is reached, the function returns the number of moves taken. If no solution is found, it returns -1.

**II. Tic-Tac-Toe Game**

 **Board Creation**:

* The create\_board function initializes a 3x3 board filled with zeros, representing an empty board.

 **Finding Empty Places**:

* The possibilities function identifies and returns the list of empty positions on the board where a move can be made.

 **Making Random Moves**:

* The random\_place function selects a random empty position from the list of possibilities and places the player's mark there.

 **Win Conditions**:

* The row\_win, col\_win, and diag\_win functions check if a player has won by having three of their marks in a row, column, or diagonal, respectively.

 **Game Evaluation**:

* The evaluate function determines the game's outcome. It checks if any player has won or if the board is full, resulting in a tie.

 **Game Loop**:

* The play\_game function manages the game loop. Players take turns making random moves until there is a winner or a tie.
* After each move, the board is printed, and the game pauses for 2 seconds to visualize the moves.

 **Execution Flow**

* The game starts with an empty board, no winner, and a move counter set to 1.
* Players 1 and 2 take turns making random moves.
* After each move, the board is printed, and the state is evaluated to check for a winner or a tie.
* The loop continues until a winner is found or the board is full.
* If a player wins, the game announces the winner.
* If the board is full without a winner, the game ends in a tie.

**Code Puzzle:**

from time import sleep

class Solution:

    def solve(*self*, *board*):

        # Dictionary to store the states and their respective distances from the start state

        state\_distances = {}

        # Flatten the 2D board to a tuple

        flatten = []

        for row in *board*:

            flatten += row

        flatten = tuple(flatten)

        # Initialize the dictionary with the start state

        state\_distances[flatten] = 0

        # Check if the start state is the goal state

        if flatten == (0, 1, 2, 3, 4, 5, 6, 7, 8):

            return 0

        # Perform BFS to find the shortest path to the goal state

        return *self*.get\_paths(state\_distances)

    def get\_paths(*self*, *state\_distances*):

        cnt = 0

        while True:

            # Get all nodes at the current distance

            current\_nodes = [

                node for node in *state\_distances* if *state\_distances*[node] == cnt

            ]

            # If there are no more nodes to explore, return -1 (no solution)

            if not current\_nodes:

                return -1

            for node in current\_nodes:

                # Get all possible next moves from the current node

                next\_moves = *self*.find\_next(node)

                sleep(2)

                print(next\_moves)

                for move in next\_moves:

                    if move not in *state\_distances*:

                        # Assign the distance to the next move

*state\_distances*[move] = cnt + 1

                        # Check if the next move is the goal state

                        if move == (0, 1, 2, 3, 4, 5, 6, 7, 8):

                            return cnt + 1

            cnt += 1

    def find\_next(*self*, *node*):

        # Possible moves for each position on the board

        moves = {

            0: [1, 3],

            1: [0, 2, 4],

            2: [1, 5],

            3: [0, 4, 6],

            4: [1, 3, 5, 7],

            5: [2, 4, 8],

            6: [3, 7],

            7: [4, 6, 8],

            8: [5, 7],

        }

        # Find the position of the empty tile (represented by 0)

        pos\_0 = *node*.index(0)

        results = []

        # Generate new states by swapping the empty tile with its adjacent tiles

        for move in moves[pos\_0]:

            new\_node = list(*node*)

            new\_node[move], new\_node[pos\_0] = new\_node[pos\_0], new\_node[move]

            results.append(tuple(new\_node))

        return results

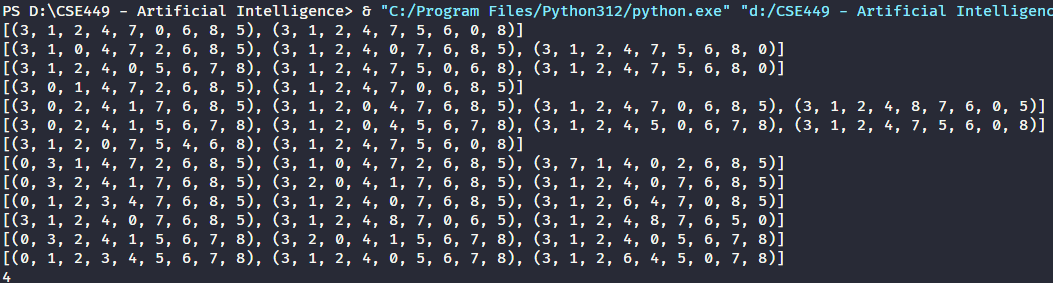
# Initialize the puzzle and solve it

ob = Solution()

matrix = [[3, 1, 2], [4, 7, 5], [6, 8, 0]]

print(ob.solve(matrix))

**Output Puzzle:**



**Code TicTacToe:**

import numpy as np

import random

from time import sleep

# Creates an empty board

def create\_board():

    return np.array([[0, 0, 0], [0, 0, 0], [0, 0, 0]])

# Check for empty places on board

def possibilities(*board*):

    l = []

    for i in range(len(*board*)):

        for j in range(len(*board*)):

            if *board*[i][j] == 0:

                l.append((i, j))

    return l

# Select a random place for the player

def random\_place(*board*, *player*):

    selection = possibilities(*board*)

    current\_loc = random.choice(selection)

*board*[current\_loc] = *player*

    return *board*

# Checks whether the player has three

# of their marks in a horizontal row

def row\_win(*board*, *player*):

    for x in range(len(*board*)):

        win = True

        for y in range(len(*board*)):

            if *board*[x, y] != *player*:

                win = False

                break

        if win:

            return True

    return False

# Checks whether the player has three

# of their marks in a vertical row

def col\_win(*board*, *player*):

    for x in range(len(*board*)):

        win = True

        for y in range(len(*board*)):

            if *board*[y][x] != *player*:

                win = False

                break

        if win:

            return True

    return False

# Checks whether the player has three

# of their marks in a diagonal row

def diag\_win(*board*, *player*):

    win = True

    for x in range(len(*board*)):

        if *board*[x, x] != *player*:

            win = False

            break

    if win:

        return True

    win = True

    for x in range(len(*board*)):

        if *board*[x, len(*board*) - 1 - x] != *player*:

            win = False

            break

    return win

# Evaluates whether there is a winner or a tie

def evaluate(*board*):

    winner = 0

    for player in [1, 2]:

        if row\_win(*board*, player) or col\_win(*board*, player) or diag\_win(*board*, player):

            winner = player

    if np.all(*board* != 0) and winner == 0:

        winner = -1

    return winner

# Main function to start the game

def play\_game():

    board, winner, counter = create\_board(), 0, 1

    print(board)

    sleep(2)

    while winner == 0:

        for player in [1, 2]:

            board = random\_place(board, player)

            print("Board after " + str(counter) + " move")

            print(board)

            sleep(2)

            counter += 1

            winner = evaluate(board)

            if winner != 0:

                break

        if winner != 0:

            break

    return winner

# Driver Code

print("Winner is: " + str(play\_game()))

**Output TicTacToe:**

