**TIC TAC TOE**

import random

print("1BM22CS028 AKASH")

def print\_board(b):

for r in b:

print(" | ".join(r))

print("-"\*10)

def check\_win(b):

for row in b:

if row.count(row[0]) == len(row) and row[0] != ' ':

return True

for col in range(len(b)):

check = []

for row in b:

check.append(row[col])

if check.count(check[0]) == len(check) and check[0] != ' ':

return True

if b[0][0] == b[1][1] == b[2][2] and b[0][0] != ' ':

return True

if b[0][2] == b[1][1] == b[2][0] and b[0][2] != ' ':

return True

return False

def check\_draw(b):

for row in b:

if ' ' in row:

return False

return True

def computer\_move(b):

best\_score = -float('inf')

best\_move = None

for i in range(len(b)):

for j in range(len(b[0])):

if b[i][j] == ' ':

b[i][j] = 'X'

score = minimax(b, 0, False)

b[i][j] = ' '

if score > best\_score:

best\_score = score

best\_move = (i, j)

b[best\_move[0]][best\_move[1]] = 'X'

def minimax(b, depth, is\_maximizing):

if check\_win(b):

if is\_maximizing:

return -1

else:

return 1

elif check\_draw(b):

return 0

if is\_maximizing:

best\_score = -float('inf')

for i in range(len(b)):

for j in range(len(b[0])):

if b[i][j] == ' ':

b[i][j] = 'X'

score = minimax(b, depth + 1, False)

b[i][j] = ' '

best\_score = max(score, best\_score)

return best\_score

else:

best\_score = float('inf')

for i in range(len(b)):

for j in range(len(b[0])):

if b[i][j] == ' ':

b[i][j] = 'O'

score = minimax(b, depth + 1, True)

b[i][j] = ' '

best\_score = min(score, best\_score)

return best\_score

def human\_move(b):

while True:

move = input("Enter your move (row and column number, separated by space): ")

move = move.split()

move = (int(move[0]), int(move[1]))

if b[move[0]][move[1]] != ' ':

print("Invalid move, try again.")

else:

b[move[0]][move[1]] = 'O'

break

def play\_game():

board = [[' ' for \_ in range(3)] for \_ in range(3)]

while True:

print\_board(board)

human\_move(board)

if check\_win(board):

print\_board(board)

print("You win!")

break

elif check\_draw(board):

print\_board(board)

print("It's a draw!")

break

computer\_move(board)

if check\_win(board):

print\_board(board)

print("Computer wins!")

break

elif check\_draw(board):

print\_board(board)

print("It's a draw!")

break

play\_game()

VACCUM

print("Akash ks ")

print("1BM22CS028")

class VacuumCleaner:

def \_\_init\_\_(self, room\_a\_dirt, room\_b\_dirt, starting\_room):

self.current\_state = (room\_a\_dirt, room\_b\_dirt, starting\_room)

def is\_goal\_state(self):

return self.current\_state[0] == 0 and self.current\_state[1] == 0

def clean(self):

if self.current\_state[0] == 1:

self.current\_state = (0, self.current\_state[1], self.current\_state[2])

print("Cleaned room A.")

elif self.current\_state[1] == 1:

self.current\_state = (self.current\_state[0], 0, self.current\_state[2])

print("Cleaned room B.")

def move(self):

if self.current\_state[2] == 'A':

self.current\_state = (self.current\_state[0], self.current\_state[1], 'B')

print("Moved to room B.")

else:

self.current\_state = (self.current\_state[0], self.current\_state[1], 'A')

print("Moved to room A.")

def run(self):

while not self.is\_goal\_state():

print(f"Current state: {self.current\_state}")

self.clean()

if not self.is\_goal\_state():

self.move()

print("Both rooms are clean!")

def get\_initial\_state():

room\_a\_dirt = int(input("Is room A dirty? (1 for yes, 0 for no): "))

room\_b\_dirt = int(input("Is room B dirty? (1 for yes, 0 for no): "))

starting\_room = input("Which room is the vacuum cleaner in? (A or B): ").strip().upper()

if starting\_room not in ['A', 'B'] or room\_a\_dirt not in [0, 1] or room\_b\_dirt not in [0, 1]:

print("Invalid input. Please enter the correct values.")

return get\_initial\_state()

return room\_a\_dirt, room\_b\_dirt, starting\_room

initial\_state = get\_initial\_state()

vacuum = VacuumCleaner(\*initial\_state)

vacuum.run()

**8 puzzle DFS**

import copy

from heapq import heappush, heappop

n = 3

row = [ 1, 0, -1, 0 ]

col = [ 0, -1, 0, 1 ]

class priorityQueue:

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

self.heap = []

def push(self, k):

heappush(self.heap, k)

def pop(self):

return heappop(self.heap)

def empty(self):

if not self.heap:

return True

else:

return False

class node:

def \_\_init\_\_(self, parent, mat, empty\_tile\_pos,

cost, level):

self.parent = parent

self.mat = mat

self.empty\_tile\_pos = empty\_tile\_pos

self.cost = cost

self.level = level

def \_\_lt\_\_(self, nxt):

return self.cost < nxt.cost

def calculateCost(mat, final) -> int:

count = 0

for i in range(n):

for j in range(n):

if ((mat[i][j]) and

(mat[i][j] != final[i][j])):

count += 1

return count

def newNode(mat, empty\_tile\_pos, new\_empty\_tile\_pos,

level, parent, final) -> node:

new\_mat = copy.deepcopy(mat)

x1 = empty\_tile\_pos[0]

y1 = empty\_tile\_pos[1]

x2 = new\_empty\_tile\_pos[0]

y2 = new\_empty\_tile\_pos[1]

new\_mat[x1][y1], new\_mat[x2][y2] = new\_mat[x2][y2], new\_mat[x1][y1]

cost = calculateCost(new\_mat, final)

new\_node = node(parent, new\_mat, new\_empty\_tile\_pos,

cost, level)

return new\_node

def printMatrix(mat):

for i in range(n):

for j in range(n):

print("%d " % (mat[i][j]), end = " ")

print()

def isSafe(x, y):

return x >= 0 and x < n and y >= 0 and y < n

def printPath(root):

if root == None:

return

printPath(root.parent)

printMatrix(root.mat)

print()

def solve(initial, empty\_tile\_pos, final):

pq = priorityQueue()

cost = calculateCost(initial, final)

root = node(None, initial,

empty\_tile\_pos, cost, 0)

pq.push(root)

while not pq.empty():

minimum = pq.pop()

if minimum.cost == 0:

printPath(minimum)

return

for i in range(4):

new\_tile\_pos = [

minimum.empty\_tile\_pos[0] + row[i],

minimum.empty\_tile\_pos[1] + col[i], ]

if isSafe(new\_tile\_pos[0], new\_tile\_pos[1]):

child = newNode(minimum.mat,

minimum.empty\_tile\_pos,

new\_tile\_pos,

minimum.level + 1,

minimum, final,)

pq.push(child)

initial = [ [ 1, 2, 3 ],

[ 5, 6, 0 ],

[ 7, 8, 4 ] ]

final = [ [ 1, 2, 3 ],

[ 5, 8, 6 ],

[ 0, 7, 4 ] ]

empty\_tile\_pos = [ 1, 2 ]

solve(initial, empty\_tile\_pos, final)

**8 Puzzle (IDS)**

class PuzzleIDS:

def \_\_init\_\_(self, start, goal):

self.start = start

self.goal = goal

def find\_zero(self, state):

for i in range(3):

for j in range(3):

if state[i][j] == 0:

return (i, j)

def get\_neighbors(self, state):

x, y = self.find\_zero(state)

neighbors = []

directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]

for dx, dy in directions:

nx, ny = x + dx, y + dy

if 0 <= nx < 3 and 0 <= ny < 3:

new\_state = [row[:] for row in state]

new\_state[x][y], new\_state[nx][ny] = new\_state[nx][ny], new\_state[x][y]

neighbors.append(new\_state)

return neighbors

def dls(self, state, depth, path, visited):

if state == self.goal:

return path + [state]

if depth == 0:

return None

visited.add(tuple(tuple(row) for row in state))

for neighbor in self.get\_neighbors(state):

neighbor\_tuple = tuple(tuple(row) for row in neighbor)

if neighbor\_tuple not in visited:

result = self.dls(neighbor, depth - 1, path + [state], visited)

if result:

return result

visited.remove(tuple(tuple(row) for row in state))

return None

def ids(self, max\_depth=50):

for depth in range(max\_depth):

visited = set()

result = self.dls(self.start, depth, [], visited)

if result:

return result

return None

start\_state = [[1, 2, 3], [4, 0, 5], [7, 8, 6]]

goal\_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]

puzzle = PuzzleIDS(start\_state, goal\_state)

solution = puzzle.ids()

if solution:

for step in solution:

print(step)

else:

print("No solution found.")