**1.BFS**

graph = {

'A': ['B', 'C', 'F'],

'B': ['D'],

'C': [],

'D': [],

'F': []

}

visited = []

queue = []

def bfs(visited, graph, node):

visited.append(node)

queue.append(node)

while queue:

s = queue.pop(0)

print(s, end="")

for neighbour in graph[s]:

if neighbour not in visited:

visited.append(neighbour)

queue.append(neighbour)

bfs(visited, graph, 'A')

**2.DFS**

graph = {

'a': ['b', 'c'],

'b': ['d', 'a'],

'c': ['e'],

'd': [],

'e': [],

'f': []

}

visited = set()

def dfs(node):

if node not in visited:

print(node)

visited.add(node)

for n in graph[node]:

dfs(n)

dfs('a')

**3.TIC-TOC TOE**

board = [" " for x in range(9)]

def print\_board():

row1 = "{} | {} | {}".format(board[0], board[1], board[2])

row2 = "{} | {} | {}".format(board[3], board[4], board[5])

row3 = "{} | {} | {}".format(board[6], board[7], board[8])

print()

print(row1)

print(row2)

print(row3)

print()

def player\_move(icon):

if icon == "x":

number = 1

else:

number = 2

print("Your turn player {}".format(number))

choice = int(input("Enter your move (1-9): ").strip())

if board[choice - 1] == " ":

board[choice - 1] = icon

else:

print()

print("That space is already taken!")

def is\_victory(icon):

return (

(board[0] == icon and board[1] == icon and board[2] == icon) or

(board[3] == icon and board[4] == icon and board[5] == icon) or

(board[6] == icon and board[7] == icon and board[8] == icon) or

(board[0] == icon and board[3] == icon and board[6] == icon) or

(board[1] == icon and board[4] == icon and board[7] == icon) or

(board[2] == icon and board[5] == icon and board[8] == icon) or

(board[0] == icon and board[4] == icon and board[8] == icon) or

(board[2] == icon and board[4] == icon and board[6] == icon)

)

def is\_draw():

return " " not in board

while True:

print\_board()

player\_move("x")

if is\_victory("x"):

print\_board()

print("X wins! Congratulations!")

break

elif is\_draw():

print\_board()

print("It's a draw!")

break

player\_move("o")

if is\_victory("o"):

print\_board()

print("O wins! Congratulations!")

break

elif is\_draw():

print\_board()

print("It's a draw!")

break

**4.WATER JUG -PROBLEM**

from collections import deque

def bfs\_water\_jug(capacity1, capacity2, goal):

initial\_state = (0, 0)

visited = set()

queue = deque([(initial\_state, [])])

def transitions(state):

a, b = state

return [

(capacity1, b),

(a, capacity2),

(0, b),

(a, 0),

(a - min(a, capacity2 - b), b + min(a, capacity2 - b)),

(a + min(b, capacity1 - a), b - min(b, capacity1 - a))

]

while queue:

current\_state, path = queue.popleft()

if current\_state[0] == goal or current\_state[1] == goal:

return path + [current\_state]

for next\_state in transitions(current\_state):

if next\_state not in visited:

visited.add(next\_state)

queue.append((next\_state, path + [current\_state]))

return None

capacity1 = 3

capacity2 = 5

goal = 4

solution = bfs\_water\_jug(capacity1, capacity2, goal)

if solution:

print("Solution path:")

for state in solution:

print(state)

else:

print("No solution found.")

**8-Queen Problem Code:**

def is\_safe(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 solve(board, row):

if row == len(board):

return True

for col in range(len(board)):

if is\_safe(board, row, col):

board[row] = col

if solve(board, row + 1):

return True

board[row] = -1

return False

def print\_board(board):

for row in range(len(board)):

line = ['Q' if board[row] == col else '.' for col in range(len(board))]

print(" ".join(line))

print()

def eight\_queen():

board = [-1] \* 8

if solve(board, 0):

print\_board(board)

else:

print("No Solution found.")

eight\_queen()

**TSP Code**

from itertools import permutations

from sys import maxsize

def travelling\_salesman(graph, start):

nodes = [i for i in range(len(graph)) if i != start]

min\_cost = maxsize

best\_path = []

for perm in permutations(nodes):

cost = sum(graph[k][j] for k, j in zip([start] + list(perm), list(perm) + [start]))

if cost < min\_cost:

min\_cost = cost

best\_path = [start] + list(perm) + [start]

return min\_cost, best\_path

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

graph = [

[0, 12, 18, 24],

[12, 0, 36, 28],

[18, 36, 0, 32],

[24, 28, 32, 0]

]

min\_cost, best\_path = travelling\_salesman(graph, 0)

node\_names = ['A', 'B', 'C', 'D']

print("Optimal path:", " → ".join(node\_names[i] for i in best\_path))

print(f"Minimum Cost: {min\_cost}")

**Tower of Hanoi Code:**

def hanoi(n, source, helper, target):

if n == 1:

print(f"Move disk 1 from {source} to {target}")

return

hanoi(n - 1, source, target, helper)

print(f"Move disk {n} from {source} to {target}")

hanoi(n - 1, helper, source, target)

n = int(input("Enter number of disks: "))

hanoi(n, 'A', 'B', 'C')

OUTPUT:

Enter number of disks: 3

Move disk 1 from A to C

Move disk 2 from A to B

Move disk 1 from C to B

Move disk 3 from A to C

Move disk 1 from B to A

Move disk 2 from B to C

Move disk 1 from A to C

**Alpha-Beta Pruning Code:**

MAX, MIN = 1000, -1000

def minimax(depth, nodeIndex, maximizingPlayer, values, alpha, beta):

if depth == 3:

return values[nodeIndex]

if maximizingPlayer:

best = MIN

for i in range(0, 2):

val = minimax(depth + 1, nodeIndex \* 2 + i, False, values, alpha, beta)

best = max(best, val)

alpha = max(alpha, best)

if beta <= alpha:

break

return best

else:

best = MAX

for i in range(0, 2):

val = minimax(depth + 1, nodeIndex \* 2 + i, True, values, alpha, beta)

best = min(best, val)

beta = min(beta, best)

if beta <= alpha:

break

return best

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

values = [3, 5, 6, 9, 1, 2, 0, -1]

print("The optimal value is:", minimax(0, 0, True, values, MIN, MAX))

**8-Puzzle A\* Algorithm Code**

import heapq

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

moves = [(-1, 0), (1, 0), (0, -1), (0, 1)] # Up, Down, Left, Right

def heuristic(state):

return sum(

abs(r - (v - 1) // 3) + abs(c - (v - 1) % 3)

for r in range(3) for c in range(3)

if (v := state[r][c]) != 0

)

def neighbors(state):

for r in range(3):

for c in range(3):

if state[r][c] == 0:

row, col = r, c

for dr, dc in moves:

nr, nc = row + dr, col + dc

if 0 <= nr < 3 and 0 <= nc < 3:

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

new\_state[row][col], new\_state[nr][nc] = new\_state[nr][nc], new\_state[row][col]

yield new\_state

def serialize(state):

return tuple(tuple(r) for r in state)

def a\_star(start):

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

seen = set()

while heap:

\_, g, state, path = heapq.heappop(heap)

key = serialize(state)

if key in seen:

continue

seen.add(key)

if state == goal:

return path + [state]

for nb in neighbors(state):

heapq.heappush(heap, (g + 1 + heuristic(nb), g + 1, nb, path + [state]))

return None

start = [[1, 2, 3], [4, 1, 6], [7, 5, 8]] # Example input

solution = a\_star(start)

if solution:

for step in solution:

for row in step:

print(row)

print()

else:

print("No solution found.")

OUTPUT

[1, 2, 3]

[4, 1, 6]

[7, 5, 8]

...

[1, 2, 3]

[4, 5, 6]

[7, 8, 0]

class Monkey:

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

self.position = "ground"

self.has\_banana = False

def move\_and\_grab(self):

print("Monkey moves to the ladder.")

self.position = "ladder"

print("Monkey climbs the ladder.")

self.position = "shelf"

print("Monkey grabs the banana!")

self.has\_banana = True

def monkey\_and\_banana():

monkey = Monkey()

monkey.move\_and\_grab()

monkey\_and\_banana()

Monkey moves to the ladder.

Monkey climbs the ladder.

Monkey grabs the banana!

**8-puzzle**

from collections import deque

G = "123456780"

def N(s):

i = s.index("0")

r, x, y = [], i//3, i%3

for dx, dy in [(-1,0),(1,0),(0,-1),(0,1)]:

nx, ny = x+dx, y+dy

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

j = nx\*3+ny; l = list(s)

l[i], l[j] = l[j], l[i]; r.append("".join(l))

return r

def solve(s):

q, v = deque([(s, [])]), set()

while q:

c, p = q.popleft()

if c in v: continue

v.add(c)

if c == G: return p+[c]

for n in N(c): q.append((n, p+[c]))

for step in solve("724506831"): print(step[:3], step[3:6], step[6:], "\n")