1) write a python program to Solve 8- puzzle problem?

Aim:- To implement a python program that solves the 8- puzzle problem using the A* Search algorithm with manhattan distance heuristic.

Code:-

```
import heapq

goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]

Class puzzleState:
    def __init__(self, board, parent=None, g=0):
        self.board = board
        self.parent = parent
        self.g = g
        self.h = self.heuristic()
        self.f = self.g + self.h

def __lt__(self, other):
    h = 0
    for i in range(3):
        for j in range(3):
            value = self.board[i][j]
            if value != 0:
                goal_x, goal_y = divmod
                                 (value - 1, 3)
                h += abs(goal_x - i) + abs(goal_y - j)
```

```
def generate_neighbors(self):
    neighbours = []
    x, y = self.find_zero()
    for dx, dy in moves:
        nx, ny = x+dx, y+dy
    if 0 <= nx < 3 and 0 <= my < 3:
        new_board = [row[:] for row in
                     self.board]
    New_board[x][y], New_board[nx][ny]
    = new_board[nx][ny], new_board
      [x][y].
def is_goal(self):
    return self.board == goal_state
def __hash__(self):
    return hash(str(self.board))
start = [[1, 2, 3],
         [4, 0, 6],
         [7, 5, 8]]
solution = a_star(start)
if solution:
    print("Steps to solve")
    for step in solution:
        for row in step:
            print(row)
        print("...")
```

```
else:
    print("No solution found")
```

Output:-

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

---

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

---

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

Result:- The program successfully finds the optimal sequence of moves to solve the 8-puzzle using A* Algorithm.

2) Write a python program to solve 8-queen problem?

A) Aim:- To place 8 queen on an 8×8 chessboard such that no two queen attack each other.

Code:-

```
N = 8

def print_solution(board):
    for row in board:
        line = ""
        for col in row:
            line += "q" if col else "."
```

```
        print (line)
    print ("\n")

def is_safe (board, row, col):
    for i in range (row):
        if board [i][col]:
            return False

for i, j in zip (range (row-1, -1, -1), range
                 (col-1, -1, -1)):
    if board [i][j]:
        return False

for col in range (N):
    if is_safe (board, row, col):
        board [row][col] = 1
        if solve_n_queen (board, row+1):
            return True
        board [row][col]

board = [[0 for _ in range (N)] for _ in range (N)]
if not solve_n_queen (board, 0):
    print ("No Solution exists")
```

Output :-

```
Q . . . . . .
. . . . Q . . .
. . . . . . . Q
. . . Q . . .
. Q . . . . . .
. . . Q . . . .
```

Result :- The program places 8-queen on the board such that non

3) Write a python program to Solve water jug problem.

A) Aim:- To Solve the classic water jug problem, where you are given two jugs with fixed the capacities and need to measure a specific quantity of water.

Code:-

```
from collection import deque

    def print_steps (path):
        for state in path:
            print (f" Jug A: {state [0]}
                liters, Jug B: {state [1]}
                liters ")

    next_states = [
        (a_capacity, b),
        (a, b_capacity),
        (0, b),
        (a, 0),
        (a - min (a, b_capacity - b), b + min (a, b -
                                    capacity - b)),
        (a + min (b, a_capacity - a), b - min (b, a -
                                    capacity - a))
    ]
    for state in next_states:
        if state not in visited:
            queue. append ((state, path +
                                [state]))

    print ("No solution found")
```

a - Capacity : 4

b - Capacity : 3

goal : 2

water - jug - bfs (a - Capacity, b - Capacity, goal)

Output :-

Jug A : 0 liters, Jug B : 0 liters.

Jug A : 4 liters, Jug B : 0 liters.

Jug A : 1 liters. Jug B : 3 liters.

Jug A : 1 liters, Jug B : 0 liters.

Jug A : 0 liters, Jug B : 1 liters.

4) Write a python program to Solve Cript orithematic problem.

A) Aim:- To Solve a Cryptarithmetic puzzle by assignment digit (0-9) to letters such as the arithmetic equation is satisfied and each letter.

Code:-

```
import itertools

def Solve_Cryptarithmetic():

    letters = "Send money"

    for perm in itertools.permutations(range
    (10), len(letters)):

        mapping = dict(zip(letters, perm)).
```

```
if mapping['S'] == 0 or mapping['M'] == 0:
    Continue

Send = 1000 * mapping['S'] + 100 * mapping['E']
    + 10 * mapping['N'] + mapping['D']

if Send + more == money:
    print(f"Send = {Send}")
    print(f"more = {more}")
    print(f"money = {money}")
    print("Mapping:", mapping)
    return

print("No Solution found")
Solve = cryptarithmetic()
```

Output:-

Send = 9567

more = 1085

money = 10652

Mapping: {'S': 9, 'E': 5, 'N': 6, 'D': 7, 'M': 1, 'O': 0, 'R': 8, 'Y': 2}

Result:- The program finds a valid digit assignment for the given cryptarithmetic equation that satisfies the sum.

5) write the python program for missionaries Cannibal problem.

A) Aim:- To transport all missionaries and the Cannibals across a river using a boat that can carry at most two people.

Code:-

```
Class State:
    def __init__(self, m_left, c_left,
                 boat, path = []):
        Self.m_left = m_left
        Self.c_left = c_left
        Self.boat = boat
        self.path = path + [self]
    if (Self.m_left > 0 and self.m_left
            < Self.c_left):
        return false.
    if (m_right > 0 and m_right < c_right):
        return false.
def Solve():
    Start = State(3,3,1)
    queue = deque([Start])
    visited = set()
    While queue:
        Current = queue.popleft()
        if Current.is_goal():
            print("Solution Steps")
```

```
        for state in current.path:
            print(state)
        return

    for next_state in current.get_next_states():
        if next_state not in visited:
            visited.add(next_state)
            queue.append(next_state)
    print("No solution found").

Solve()
```

Output:-

(M_left = 3, C_left = 3, Boat = left)

(M_left = 3, C_left = 1, Boat = Right)

(M_left = 3, C_left = 2, Boat = left)

(M_left = 3, C_left = 1, Boat = left)

Result:- The program prints a valid sequence of moves that solves the problem.

6) Write the python program for vaccum Cleaner problem.

A) Aim:- To Simulate an intelligent agent (vaccum Cleaner) that can clean two rooms (Room A and Room B) by sensing and acting accordingly to reach a clean environment.

Code:-

```
def vacuum_Cleaner (state):
    location = state [0]
    Status = state [1]
    Stp = []
if location == "A":
    if status ["A"] == 1:
        Stp . append ("suck in Room A")
        Status ["A"] = 0
    Steps . append ("move to Room B")
    if status ["B"] == 1:
elif location == "B":
    if status ["B"] == 1:
        Stps . append ("Suck in Room
                        B")
        Status ["B"] = 0
    Stps . appends ("move to Room A")
    if Status ["A"] == 1:
        Stps . append ("Suck in Room A")
```

return steps, status

```
print("Steps Taken by Vaccum Cleaner")
for step in steps_taken:
    print(step)
print("\n Final Room Status")
print(final_status)
```

Input :-

```
('A', {'A': 1, 'B': 1})
```

Output :-

```
Steps Taken by Vaccum Cleaner:
Suck in Room A
move to Room B
Suck in Room B
Final Room Status:
{"A": 0, "B": 0}
```

Result :- The vaccum cleaner (agent) perform the correct sequence of action.

7) Write the python program to implement BFS.

Aim:- To implement the Breadth - first Search algorithm to traverse or search through a graph using python.

Code:-

```
from Collection import deque
def bfs (graph, start):
    visited = set()
    queue = deque ([start])
    print ("BFS Traversal order")
    while queue:
        vertex = queue.popleft()
        if vertex not in visited:
            print (vertex, end = " ")
            visited.add (vertex)
            queue.extend ([neighbor for
                neighbor in graph [vertex] if
                neighbor not in visited])
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': []
}
bfs (graph, 'A')
```

Output:-

BFS Traversal order:

A B C D E F

Result:- The program Successfully perform Breadth-first search, visiting nodes level by level, starting from the given source node.

8) Write a python program to implement DFS.

Aim:- To implement the Depth-first Search (DFS) algorithm for graph traversal using python.

Code:-

```
def dfs(graph, start, visited = None):
    if visited is None:
        visited = set()
    visited.add(start)
    print(start, end = " ")
    for neighbor in graph[start]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited)

graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': []
```

```
start_node = input("Enter the starting node for
                    DFS").upper()
print("\nDFS Traversal starting from node:
      ", start_node, ":")
dfs(graph, start_node)
```

Input:-

Enter the starting node for DFS: A

Output:-

DFS Traversal starting from node A:

A B D E F C

Result:- The program succesfully implements Depth first search (DFS). It traverses all the reachable nodes from the given starting node in depth-first order using recursion

9) Write a python program to implement travelling Salesman problem.

Aim:- To implement a solution for the Travelling Salesman problem (TSP) using brute-force in python, which finds the shortest possible route that visits each city exactly once and return.

Code:-

```
import itertools

def calculate_total_distance(graph, path)
    distance = 0
for i in range(len(path)-1):
```

```
    distance += graph[path[i]][path[i+1]]
    distance += graph[path[-1]][path[0]]
    return distance

def travelling_Salesman(graph):
    num_cities = len(graph)
    cities = list(range(num_cities))
    min_path = None
    min_distance = float('inf')

    for perm in itertools.permutations(cities[1:]):
        current_path = [cities[0]] + list(perm)
        current_distance = calculate_total_distance
                           (graph, current_path)
        if current_distance < min_distance:
            min_distance = current_distance
            min_path = current_path.
    return min_path, min_distance.

graph = [
    [0, 10, 15, 20],
    [10, 0, 35, 25],
    [15, 35, 0, 30],
    [20, 25, 30, 0],
]

path, distance = travelling_Salesman(graph)
print("Shortest path (by city index):", path)
print("Minimum total distance:", distance)
```

output :-

Shortest path (by city index):

[0, 1, 3, 2]

minum total distance : 80

Result :-

The program succesfully implemented a brute force solution to the Travelling Salesman problem (TSP). It finds the minimum distance and optimal path by checking all permutations.

10) write a python program to implement A* algorithm.

Aim :- To implement the A* Search algorithm in Python to find the shortest path from a start node to a goal node using a heuristic function.

Code :-

```
import heapq

def a-star (graph, start, goal, heuristic)
    queue = [(0 + heuristic[start], start,
                [start], 0)]
    Visited = set()
    while queue:
        est-total-cost, current, path, cost-
            -so-far = heapq.heappop(queue)
    if current == goal.
```

```
if current in visited:
        continue
    visited.add(current)
    for neighbor, weight in graph[current]:
        if neighbor not in visited:
            new_cost = cost_so_far + weight
            est_cost = new_cost + heuristic[neighbor]
            heapq.heappush(queue, (est_cost, neighbor,
                path + [neighbor], new_cost))

start_node = 'A'
goal_node = 'E'

path, cost = a_star(graph, start_node, goal_
    node, heuristic)

print("Shortest path from", start_node, "to",
    goal_node, ":", path)
print("Total cost:", cost)
```

Output:-

Shortest path from A to E : ['A', 'B', 'C', 'D', 'E']

Total cost : 7

Result:-

The program successfully implements the A* algorithm, finding the shortest path from the start node to the goal node using both path.

11) • Write the python program for map Coloring to implement CSP.

Aim:-

To implement python program for the Solving the map Coloring problem using the Constraint Satisfaction problem.

Program:-

```
Colour = ["Red", "Green", "Blue"]
State = ['WA', 'NT', 'SA', 'Q', 'NSW', 'V', 'T']
adj = ["NT", "SA", "NT"].

def valid (s, c):
    return all (assign.get(n) != c for n
        in adj (s]).

def Solve (i=0):
    if i == len (States):
        return true.
    S = States [i]
    for c in Colours:
        if valid (s,c):
            assign [s] = c
    if Solved (i+1):
        return True
    del assign [s]
    return False
```

Solved ()

print (assign)

Output :-

{"WA": 'Red' 'NT': 'Green' 'SA': 'Blue'
'Q': 'Red' 'NSW': 'Green' 'V': 'Red'
'T': 'Red'}

Result :-

The program successfully assigned Color to the each State without Conflit.

12) write the python program Tic Tac Toe game.

Aim :-

To write a python program to the implement the Tic Tac Toe game where two players play alternatly until there is a win or a draw.

program :-

```
def print_board(b):
    for row in b:
        print("|".join(row))
        print("-"*5)
def winner(b, p):
    return any(all(Cell == p for
        Cell in row) for row
        in b)
```

```
board = [[" "]*3 for _ in range(3)]
turn = 'x'
print_board(board)
while True:
    try
        r = int(input(f"player {turn}, row
                (0-2):"))
        if winner(board, turn):
        print(f"{turn} wins!")
        turn = "0" if turn == 'x'
    else "x"
except
    print("Invalid ! Enter 0-2")
    game()
```

Output :-

Result :-

13) Minimax Algorithm for Gaming.

Aim :-

To implement the minimax algorithm for decision making in a two-player game like Tic-Tac-Toe.

Program :-

```
def alphabet(depth, node, index, is ma
    Scores, alpha, beta, h):
    if depth == h:
        return Scores[node index]
    if is Max:
        best = -1000
        for i in range(2):
            val = alphabeta(depth+1, node
                index * 2
                i, False, Sc
                alpha, beta).
        best = 1000
        for i in range(2):
            val = alphabeta(dep
    else:
        best = 1000
        for in range(2):
            best = min(best, val)
            beta = min(beta, best)
            if beta <= alpha
                break
```

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

h = 3

alpha = -1000

beta = 1000

print ("The output value is :", alphabet).

Output:-

The optimal value is : 5

Result:- Therefore, the program was executed successfully using python programing language.