# Experiment 1

**Aim:-** To Solve the 8- puzzle problem using the A* algorithm with Manhattan distance

**Algorithm:-**

* Define goal and start state
* Use A* to explore possible moves.
* calculate cost with Manhattan distance

**Code:-**

```
import heapq
goal = [[1,2,3], [4,5,6], [7,8,0]]
def manhattan(p):
    return sum(abs(val-1)%3 - i) + abs((val-1)//3 - j)
    for i, row in enumerate(p) for j, val in
    enumerate(row) if val)
def neighbors(p):
    r, y = [(i,j) for i in range(3) for j in range(3)
        if p[i][j] == 0][0]
    moves = [(-1,0), (1,0), (0,-1) (0,1)]
    result = []
    for dx, dy in moves:
        if state == goal - state
            return True
        explored.add(state)
        for successor in successors(state):
        if successor in successors(state):
    priority = heuristic(successor) + len(explored)
        frontier.put((priority, successor))
        return False.
```

```
explored.add(state)
for successor in successors(state):
    if successor not in explored:
        priority = heuristic(successor) + len(explored)
        frontier.put((priority, successor))
return False.
def successor(state):
    successor = []
    i = state.index(0)
if i % 3 != 0:
    # slide tile to the left
    new_state = list(state)
    new_state[i], new_state[i-1] = new_state
                [i-1], new_state[i]
    successor.append(tuple(new_state))
if i % 3 != 2:
    # slide tile to the right:
    new_state = list(state)
    new_state[i], new_state[i+1] = new_
    state[i+1], new_state[i]
    successor.append(tuple(new_state))
if i // 3 != 0:
    # slide tile UP.
    new_state = list(state)
    new_state[i], new_state[i-3] = new_
    state[i-3], new_state[i]
    successors.append(tuple(new_state))
```

```
if i!=3 and i!=2:
# state file down
    new_state = state_list(state)
    new_state[i], new_state[i+3] = new_state[i+3], new_state[i]
    successors.append(tuple(new_state))
return successors
initial_state = (2,8,3,6,4,7,0,5)
if solve(initial_state):
    print("The puzzle is solvable!")
else:
    print("The puzzle is unsolvable")
```

output:-

```
1 2 3
5 6 0
7 8 4
```

---

```
1 2 3
5 0 6
7 8 4
```

---

```
1 2 3
5 8 6
7 0 4
```

---

```
1 2 3
5 8 6
0 7 4
```

Result:-

Hence, the simple python program for 8-puzzle is done.

Exp: 9

Aim:-

To write a python program to Solve 8- queen

Algorithm:-

1. Start with an empty board of size 8x8.
2. place the first queen in the first row and the first column.
3. For each subsequent row, check all possible column to place to queen.
4. A valid position is a column no other same row.
5. If no solution is found, return failure.

program:-

```
def Solve_queens(n):
    result = []
    def is_valid(row, col):
        for i in range(row):
            if board[i] == col or
                abs(row-i) == abs(col-board[i]):
                return False
        return True
    def back_track(row):
        if row == n:
            result.append(list(board))
            return
        for col in range(n):
            if is_valid(row, col):
                board[row] = col
                back_track(row+1)
```

```
board [row] = -1
backtrack(0)
return result
Solutions = solveNQueens(8)
for solution in solutions:
    for row in range(8):
        line = ""
        for col in range(8):
            if solution[row] == col:
                line += "Q "
            else:
                line += ". "
        print(line)
    print("\n")
```

output:-

```
[1, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 1, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 1]
[0, 0, 0, 0, 0, 1, 0, 0]
[0, 0, 1, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 1, 0]
[0, 1, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 1, 0, 0, 0, 0]
```

result:- Hence, the simple python program for 8-queen problem is done.

Aim: to write python program for water jug problem

Algorithm:

-) Start with two empty jugs of given capacities.
-) Fill one jug to its maximum capacity.
-) Pour the water from the jug into the other jug if it is empty.
-) if the second jug is full pour out the water to empty it
-) Repeat step 2-4 filling and pouring water between the two jugs.

program:

```
from collections import deque
def water_jug_problem(x,y,z):
 Solve the first search.
x: size of jug1
y: size of jug2
z: large amount of water
queue = deque([(0,0)[i]]) # Start with both empty
visited = set()
while queue:
  a,b, steps = queue.popleft()
  if (a,b) in visited:
    continue
  visited.add(a,b)
  if a == z or b == z
    return steps.
```

```
queue.append((x, b, steps + ['fill jug 1']))
queue.append((a, y, steps + ['fill jug 2']))
queue.append((a, b, steps + ['empty jug 1']))
queue.append((a, 0, steps + ['empty jug 2']))
    amount = min(a, y-b)
queue.append((a-amount, b+amount, steps +
            ['pour jug1 into jug2']))
# pour jug 2 into jug1
amount = min(a, y-b)
queue.append(tb((a-amount, b+amount, steps +
            ['pour jug 1 into jug 2'])))
# pour jug 2 into jug 1
    amount = min(x-a, b)
    queue.append((a+amount, b-amount, step +
            ['pour jug 2 into jug1']))
    return None.

steps = water_jug_problem(4,3,2)
if steps:
    print('\n'.join(steps))
else:
    print('No solution found.')
```

output:

path of states by jugs

0, 0
0, 3
3, 0
0, 3
4, 2
0, 2

Result:-

Hence, The simple python program for water jug problem

Aim:-

To write a python program for cip-1 - Arithmetic problem

Algorithm:-

-) write down the arithmetic problem in full
-) Replace each letter with a unique digit from 0 to 9
-) Generate all possible Combinations
-) for each Combination of digits, evaluate the Constraints from Carry over.
-) for each Combinations of digits.
-) if a Solution is found:

Program:-

```
def Solve_Cryptarithmetic(puzzle):
puzzle = "WORD + WORD = RESULT"
words = puzzle.split()
letters = set(''.join(words))
if len(letters) > 10:
    return None
for perm in permutations(range(10), len(letters)):
    mapping = dict(zip(letters, perm))
if all(mapping[word[0]] != 0
    a = sum(mapping[c] * (10** for i, c in
    b = a[::-1]
    a %= 10
        c = sum(mapping[c] * (10** (len(word[2])-
        i - i)) for i, c in enumerate(word[2][::-1]))
```

```
if a == mapping[words[2][-1]] and b+c==
    sum(mapping[c]*(10**(len(word)-i-
)) for word in words[:-1]):
        return mapping
    return None
# Example usage
puzzle = 'SEND + MORE = MONEY'
mapping = solve_cryptarithmetic(puzzle)
if mapping:
    print(mapping)
    print(puzzle.translate(str.maketrans
        (mapping)))
else:
    print('No solution found.')
```

Output:-

if solution is found (i.e., mapping is not None)

if no solution is found, it prints "No solution found"

Result:-

The output has verified.

Exp: 5

Aim - To write a python program for Missionaries Cannibal problem

Algorithm:-

-) Create a data structure to represent the state
-) Create a data structure to represent the state space of the problem
-) Use a search algorithm, such as depth-first search or breath-first-search.
-) At each step of applying (two missionaries. two cannibals, or one of each)
-) Check if the successor state is valid
-) if there are no more possible successor states to explore.

program:-

```
from typing import List, tuple
from collections import deque
def is_valid_state (tuple [int, int int ])->bool
m1, c1, b, m2, c2, _ = state.
if m1 < 0 or c1 < 0 or m 2<0 or c2 <0:
    return false
if (m1 >0 and m1 < c1) or ( m2 >0 and m2 < c2).
    return False
return True
def get_ successors (state: tuple [int, int,
    int, int, int])
```

```
m1, c1, b, m2, c2, d = state
successors = []
if d == 1:
    for i in range(3):
        for j in range(3):
            if i + j >= 1 and i + j <= 2:
                new_state = (m1 - i, c1 - j, 0, m2 + i, c2 + j, 1)
                if is_valid_state(new_state):
                    successor.append(new_state)
else:
    for i in range(3):
        for j in range(3):
            if i + j >= 1 and i + j <= 2:
                new_state = (m1 + i, c1 + j, 1, m2 - i, c2 - j, 0)
                if is_valid_state(new_state):
                    successor.append(new_state)
    return successors
def breadth_first_search() -> list[Tuple[int, int, int, int, int, int]]:
initial_state = (3, 3, 1, 0, 0, 1)
goal_state = (0, 0, 0, 3, 3, 0))
visited = set()
queue = deque([initial_state, []])
while queue:
    state, path = queue.popleft()
    if state == goal_state:
        return path + [state]
    visited.add(state)
```

```
for successor in get_successors(state):
    if successor not in visited:
        queue.append((successor, path + [state]))
return []
if __name__ == '__main__':
    solution = breadth_first_search()
    print("Solution:")
    for state in solution:
        print(state)
```

output :-

Game start

1. The boat can carry at most two people
2. if cannibals num greater than missionaries then the cannibals would eat it
3. The boat cannot cross the river by people on board

left side → right side river travel

Enter number of missionaries travel =)

Invalid input plese retry!!

Result :-

The program was executed and the output was found to be correct.

Exp:- 6

Aim:-

To write a python program for vaccum cleaner problem.

Algorithm:-

Step1:- Create a data structure to store the problem including the initial state, the goal state, and the possible between states.

-) use the search algorithm, such as depth-first search or depth-first search

-) At each step of the search (move up, down, left, right, or clean)

-) Check if the successor state is valid.

-) if the successor state is valid, add it to the search tree.

-) if the successor is the goal state return the path.

-) back track to the previous state and try another possible action.

program:-

```
from typing import List, Tuple
def get_successor(state: Tuple[Tuple[int,int],
    list[list[int]]]) -> list[Tuple[Tuple[
    int,int], List[List[int]]]:
```

```
position, grid = state
successor = []
for dx, dy in [(0, 1), (0,-1) (1,0), (-1,0)]:
    x,y = position
new_x, new_y = x + dx, y+dy
if 0 <= new_x < len(grid) and 0 <= new_y <
len(grid[0]):
    new_grid [new_x[new_y]] == 1.
    new_grid [new_x)[new_y] = 0
    successor . append ([[new_x, new_y], new_grid])
return successors

def depth_first_search (state: tuple[tuple[int
int], List [List [int ]]], visited : set )->bool.
if all(all(cell == 0 for cell in row) for
    row in state [i]):
    return True
visited. add(state)
for successor in get_ successor s(state).
    if successor in get_ successors (state).
if successor not in visited:
    if depth_ first_ search (successor, visited).
        return True
    return false
if __name__ == '__main__':
        grid = [[0, 1, 1, 1],
                [0, 0, 1,0],
                [1,0,0,1]    [1,1,1,0]]
```

initial_state = ((0, 0), grid)

if depth_first_search(initial_state):

print("The room can be cleaned.")

else:

print("The room cannot be cleaned.")

~~Result~~ output:-

All the room are dirty

[[1, 1, 1, 1], [1, 1, 1, 1], [1, 1, 1, 1], [1, 1, 1, 1]]

Before cleaning the room I detect all of the dirty

[[1, 0, 1, 1], [0, 0, 0, 1], [1, 0, 0, 0], [0, 0, 0, 0]]

vaccum in location now, 0 0

cleaned 0 0

vaccum in location now, 0 2

cleaned 0 2

vaccum in location now, 0 3

cleaned 0 3

vaccum in location now, 1 3

cleaned 1 3

vaccum in location now 2 0

cleaned 2 0

Room is clean now, thanks for using:

A. S AFARAJ CLEANER

[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]

performance = 68.75%.

Result:-

The program has exicuted Succesfully

Exp: 7

Aim:-
To write a python program to implement BFS

Algorithm:-
→ Initialize a queue data structure to set to keep track of visited nodes
→ Add the starting node to the queue to set of visited nodes
→ While the queue is empty, dequeue the next node
→ For each neighbor is the dequeued node.
→ Repeat steps 3 & 4 until the queue is empty.

program:-

```
from collections import deque
def bfs(adj_list, start, end):
    queue = deque([[start, [start]]])
    visited = set()
    while queue:
        node, path = queue.popleft()
        if node == end:
            return path
        if node in visited
            continue
        visited.add(node)
        for neighbor in adj_list[node]:
            if neighbor not is visited:
```

```
        queue.append((neighbor, path + [neighbor]))
    return None
if __name__ == "__main__":
    graph = {0: [1, 2],
             1: [0, 2, 3],
             2: [0, 1, 3],
             3: [1, 2, 4],
             4: [3]}
    start_node = 0
    end_node = 4
    shortest_path = bfs(graph, start_node, end_node)
    if shortest_path is not None:
        print(f"The shortest path between {start_node} and {end_node} is {shortest_path}")
    else:
        print(f"There is no path between {start_node} and {end_node}")
```

output :-

The Breadth first search Traversal for the Graph is as follows: 3 1

Result :-

Thus, the code executed and found the output was found to be Correct.

Exp: 8

Aim:- To write a python program to implement DFS

Algorithm:-

-) Initialize a set to keep track of visited nodes
-) call the DFS- visit function set of visited node
-) DFS- visit function, set of visited nodes
-) For each neighbor of the current node
-) Repeat steps 3-4 untill the nodes have been visited.

program:-

```
# define the graph as an adjancency list
graph = {
'A': ['B', 'C'],
'B': ['D', 'E'],
'C': ['F'],
'D': [],
'E': ['F'],
'F': []
}
def dfs (graph, start, visited= None):
    if visited is None:
        visited = set()
    visited.add(start)
    for neighbor in graph[start]:
        if neighbor not in visited:
```

```
dfs(graph, neighbor, visited)
return visited
print(dfs(graph, 'A'))
```

output

A D H C F J G K B E I

Result :-

The program was executed and output found to be correct.

Aim -

python program to implement travelling salesman problem.

Pseudo code -

```
cities = [A, B, C, D]
start = A
visited = [start]
total = 0
while not all cities visited:
    next = nearest unvisited city
    move to next
    add distance to total
    mark next as visited
return to start
add final distance
```

Program -

```
from itertools import permutations
def tsp(graph):
    n = len(graph)
min_cost = float('inf')
    best_path = []
for perm in permutations(range(1,n)):
    path = (0) + perm + (0)
cost = sum(graph[path[i]][path[i+1]]
           for i in range(n))
```

```
if cost < min_cost:
    min_cost, best_path = cost, path
print('shortest path:', best_path, 'min
                                cost')
graph = [[0,10,15,20], [10,0,35,25],
         [15,35,0,30], [20,25,30,0]]
tsp(graph)
```

output :-

shortest path = (0, 1, 3, 2, 0)

Cost = 80.

Result :-

Thus the program implement successfully.

Exp :- 10

Aim :-

Python program to implement A* Algorithm.

Pseudocode :-

```
open_list = priority queue()
open_list.put((start, [start], 0, [start]))
while open_list is not empty:
    current_city, visited, cost, path = open_list.get()
    if all cities are visited and current_city is start:
        for neighbor in unvisited cities:
            new_cost = cost + distance(current_city, neighbor)
            total = new_cost + heuristic
            open_list.put((neighbor, visited + [neighbor], new_cost, path + [neighbor]))
```

Program :-

```
import heapq
def a_star_tone(graph, start, goal, haristic):
    open_list = []
    heapq.heappush(open_list, (cost + heuristic[start], 0, start, []))
    closed_list = set()
    while open_list:
        if node in closed_list:
            continue
```

```
path = path + [node]
if node == goal:
    print("path:", path, "cost", g)
return
closed_list.add(node)
for neighbor, cost in closed_list:
if neighbor not in closed_list:
heapq.heappush(open_list, (g+cost+
    heuristic(neighbor), g+cost,
    neighbor, path))

graph = {
    'A': {('B',1), ('C',3)}
    'B': {('A',1), ('C',1), ('D',2)}
    'C': {('A',3), ('B',1), ('D',1)}
    'D': {('B',2), ('C',1)}
}
heuristic = {'A': 3, 'B': 2, 'C': 1, 'D': 0}
a-star(graph, 'A', 'D', heuristic)
```

output :-

path :- ['A', 'B', 'C', 'D']

Cost :- 4

Result :-

The program implemented successfully.

Exp:- 11

Aim:- Write the python program for Map coloring to implement CSP.

Pseudocode:-

function backtrack (assignment):
  if all variable assigned
    return assignment
  Var = select una assigned variable
  for value in domain [var]:
    if value is consistent with assignment:
      assign value to var
      result = backtrack
      if result · assignment
        return failure.

program:-

```
colors = ['Red', 'Green', 'Blue']
neighbors = {
    'A': ['B', 'C'],
    'B': ['A', 'C', 'D'],
    'C': ['A', 'B', 'D'],
    'D': ['B', 'C']
}
def is_valid(assignment, var, color):
    for neighbor in neighbors[var]:
```

```
for neighbor in neighbors[var]:
    if neighbor in assignment and assignment
                [neighbor] == color:
        return False
    return ~~False~~ True

def backtrack(assignment):
    if len(assignment) == len(neighbors):
        return assignment
    var = next(v for v in neighbors if v not in
        assignment)
    for color in colors:
        if is_valid(assignment, var, colour):
            assignment[var] = colr
            result = backtrack(assignment)
            if result:
                return result
            del assignment[var]
    return None
: Solution = backtrack(g)
print(Solution)
```

output :-

Solution found : (1, 2, 3, 1)

Result :-

The program implemented successfully.

Experiment - 12

Aim - To write python program for tic-tac-toe game

Program

```
def print_board(board)
for row in board
    print("|", join(row))
    print("-" *a)
def check_winner(board, player)
# check rows, columns and
  diagonals
  for row in board
def check_winner(board, player)
# check rows, columns and diag
  for row in board
    if all is == player for
        in row):
      return true:
      return false:
def is_full(board)
```

```
return all (all (cell != " " for cell
in row ; for row in board
def tic_tac_ tic):
board = ((" for in range (3))
for _in range (3))
players = ("x", "o")
turn = 0.
while True.
print board (board)
player = players (turn %. 2)
try:
if check_ winner. (board player);
~~bor~~.
return t = 1
if_ name = "_ main_"
tic_tac_ toe())
```

output :-

Result :-

Thus the program implemented Successfully

# Experiment - 13

Aim:- To write a python program to implement minimax algorithm for gaming

program:-

```
def minmax (board, depth, is_maximizing):
    winner = check_winner (board)
        if winner = 'x': return 1
        if winner = = 'o': return 1
    if all (cell != " for row in board
        for cell in row):
    return 0
    if is_maximizing:
        best = -float (inf)
        for i in range (3):
        for j in range (3):
    if board (i)(j) = " "
        board (i) (j) = 'x'
    best = max(best, minimax (board
                depth + 1, False))
        board (i) (j) = 'o'
```

~~test~~ = [['x', 'o', 'x'], ['', 'o', '']]

move = best_move(board)

output :-

Best move for x :- (2, 0)

Result :-

thus the program implemented Successfully

# Experiment-14

Aim:- To write a python program to implement alpha & Beta pruning algorithm for gaming

program:-

```
def alpha_beta(board, depth, alpha, beta
               is_maximizing):
winner = check_winner(board)
  if winner == 'x': return 1
  if winner == 'o': return -1
  if all(cell != ' ' for row in board
     for cell in row):
        return 0
  if is_maximizing:
     best = float('-inf')
     for i in range(3):
     for j in range(3):
        if board[i][j] == ' ':
        board[i][j] = 'x';
best = max(best, alpha_beta(board
           depth, alpha, beta) False))
```

```
board (i)(j) = ' '
    alpha = max (alpha, best).
if beta <= alpha : break
return best
else:
 best =    float (-inf)
for i in range (3);
    for j in range (3): ;
  if board (i)(j) = = ' ')
     board (i)(j) = 'O' ;
best = min ( best, alpha_ beta_ (board, depth+1),
          alpha, beta, true)
  board (i) (j) = ( )
beta = min (beta . best)
        if beta = alpha : break;
         result best
def best_ move (board);
     best_val = -float ("-inf")
     move = (-1,-1)
alpha, beta =  - float (inf' ) : -float
                    (' inf')
for i in range (3);
```

```
for j in range(3)
move_val = alpha_beta(board, 0, alpha, beta,
False
board[i][j] = ' '
if move_val > best_val:
    best_val = move_val
    return move
board = [['x', 'd', 'x'], ['', 'o', ''], ['', '', '']]
move = best_move(board)
print("if best move for x (move)")
```

output:-

Best move for x :- (2,1)

Result:-

Thus the program executed successfully

Experiment - 13

Aim:- To write a python program to implement decision Tree

program:-

```
from sklearn.tree import DecisionTree
        classifier
x = [[25, 50000], [40, 60000], [35, 70000],
[50, 8000]]
y = [0, 1, 0, 1]
model = file(x, y)
prediction = model.predict([[30, 55000]])
print("predicted", prediction[7])
```

out put:-

predicted: 0 (or) 1, depending on the earned pattern.

Result

Result:- thus the program was executed Successfully.

# Experiment - 16

Aim:- To write a python program to implement feed forward neural network

program:-

```
import tensorflow as tf
from tensorflow import keras
    import numpy as np
x = np.array([[0,0],[0,1],[1,0],[1,1]])
y = np.array([[0,0],[0],[1]])
model = keras.Sequential([
keras = layers.Sequential(2, activation
    = 'relu')
model.Compile(optimizer='adam',
loss = 'binary-Comparopy', metrics=
    ['accuracy'])
model.fit(x,y, epochs=500, ver
    bose=0)
prediction = model.predict([[1,1]])
print("predicted", prediction
    [a:][0])
```

output :-

predict : 0 : 9 for clock to 1 since [1,1] should output for an AND gate]

Result :-

Thus the program was implemented Successfully