Experiment-1

AIM :- To solve the 8-puzzle problem using AI search algorithm.

Algorithm :-

* Defines the goal state (e.g. 1-8 in order with a 0 the blank)
* use a priority queue to store, stored f(n) = g(n) + r(n)
* start from the initial state, push into the queue
* while the queue is not empty.

code :-

```
import heapq
    class puzzle:
    def _init_(self, board, parent=none, move=111);
    self.board = board
    self.parent = parent
    self.move = move
    self.depth = depth
self.priority = self.depth + self.heuristic()
def __lt__(self, other): return self.priority<other.
                priority
def __eq__(self other): return self.board==other.board
                    ): return
```

```
def heuristic(self, goal):
    # Manhattan Distance
    goal = [1,2,3,4,5,6,7,8,0]
    return sum(abs((val-1)//3 - goal.index(val)//3) + abs((val-1)%3 - goal.index(val)%3)
               for val in range(1, 9) if self.board[val])

neighbors = self.board.index(0)
{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]}

return (self.swap(i,j) for j in neighbors[i])

def swap(self, i, j):  # swap title
    new_board = self.board[:]
    new_board[i], new_board[j] = new_board[j], new_board[i]
    return PuzzleState(new_board, self, f"Move {self.board[j]}")

path.append(state.move); state = state.parent
return path[::-1]

visited.add(tuple(state.board))
for move in state.moves():
    if tuple(move.board) not in visited:
        heapq.heappush(pq, move)
```

Input:-

```
solution = solve_puzzle(start_board)
print("Solution:", solution)

solution: {move: [move 8, move 7, move 4]}
```

Output:-

Thus the program was implemented successfully.

Experiment-2

Aim:- To solve 8-queen problem using python.

Algorithm:-

* Start with an empty board
* Place the first queen in first column.
* For each subsequent column place a queen safe position
* If a safe position is found, place the queen & move next colum.

Code:-

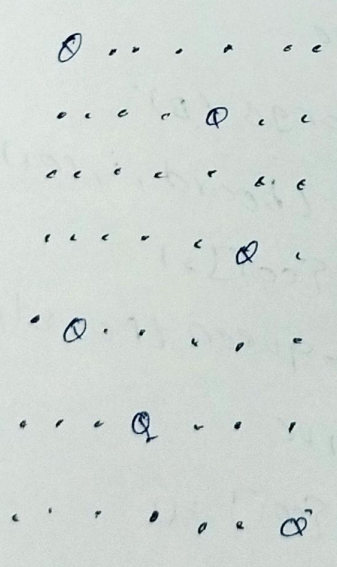
```
def is_safe (board,8);
    if col >=8;
    return True
    for i in range (8):
    if is_safe (board, i, col);
    board [i][col] = 1
    if solve-n-queen (Board, col+1);
    return True
    board [i][col] = 0
    return False
def print = board (board)
print ('' for row in board
    , join ['0' if cell else '.' for
    cell in row]
```

```
if solve_n_queen(board,0):
    print_board(board)
else:
    print("No solution")
if __name__=="__main__":
    main()
```

Input:-

* An 8 x 8 chess board
* 8 queens to be placed

Output:-



Result:- Thus python program 8-queen program successfully executed.

Experiment-3

AIM:- Python program for water jug problem

Algorithm:-

* Full jug B with 5 litres of water
* Pour water jug B into jug A until jug A leaving 2 litres jug B
* empty jug A
* Pour there maining 2 litres jug B jug A.
* fill jug B with 5 litres water gain.

code:-

```
From collections import deque;
    def water_jug (jug1, jug2, target):
        visited = set()
        queue = deque [0,0])
    while queue;
        a, b = queue.popleft()
    if (a,b) in visited
        continue
        print ((a,b))
    visited.add ((a,b))
    if a == target or b == target
        return
queue.extend ([(jug1, b), (a, jug2), (a,b), (ap)
    (a-min (a, jug2-b), 8+min (a, jug2-b)
```

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

3)

Input:-

water-jug (4,3,2)

output:-

(0,0)

(4,0)

(0,3)

(4,3)

(3,1)

(1,3)

(4,2)

Result:-

Thus the program implemented successfully.

AIM:- Python program for tic-tac program game.

program:

```
def print_board (board):
    for row in board:
        print (" ", " ".join(row))
    print (" ".join(a))

def check_winner (board, player):
    for row in row:
        if all (s == player for in row):
            return true
        return false
    def is_full (board):
        return all (all cell != " " for cell in
        a row) for row in board)
    def tac_tac_toe():
        board = ("for i in range (3)] for _ in
            range [3])
        player = ("x", "0")
            turn = 0
        while true
            print board (board)
        player = players [turn %2]
```

```
row, col = map(int, input("play <player> & enter:
row and (0-2) seperate by (Space).split())
board[row][col] = player
if check_winner(board, player):
print_board(winner_board)
print(f"player {player} win")
break
turn += 1
except (value error, index error):
print("invalid input, please enter
the number")
if __name__ == "__main__"
tic_tac
()
```

Out

5) Aim:- To write a python program to implement BFS

Program:-

```
from collection import deqeue
    def bfs (graph, start);
        visited = set()
    queue = deqeue ([start])
    while queue:
        node = queue.people ft)
    if node not is visited;
    print (node, end = " ")
    visited.add node
    visited.add(node)
queue.extend [group r [node]-visited]
    graph = 'A': {'B', 'C'}
            'B': {'A', 'D', 'E'}
            'C': {'A', 'F'}
            'D': {'B'}
            'E': {'B', 'F'}
            'F': {'C', 'E'}
            }
        bfs graph 'A'}
```

Output:- ABCDEF

Result:- Thus the program implemented successfully.

6) Aim:- python program to implement DFS

program:-

```
def dfs (graph, node, visited=set()):
    if node not is visited:
    print (node, End = " ")
    visited.add (node)
    for neighbour in graph [node]:
     dfs [graph, neigbour - visited)
    graph = {
        'A' : {'B', 'C'}
        'B' = {'A', 'D', 'E'}
        'C' = {'A', 'F'}
        'D' = {'B'}
        'E' = {'B', 'F'}
        'F' = {'C', 'E'}
        } dfs [graph, 'A']
```

Output:-

A B D E F C

Result:- Thus the program implemented successfully.

7) Aim :- Python program for vaccum - cleaner program

program:-

```
def vaccum-cleaner(rooms):
for i in range(len(rooms)):
    if rooms[i]==1:
print("f" cleaning rooms{i:y")
    room[i]=0
print("f" Room {i:y is clean")
    room=[1,0,1,1]
    vaccum=clear(rooms)
```

output:-

```
cleaning room 0
Room 0 is clean
room 1 is clean
cleaning room 2
  room 2 is clean.
cleaning room 3
  room 3 is clean.
```

Result:-

Thus the program executed successfully.

8) Aim:- Python program for cript - Arithmetic problem.

program:-

```
from intertools import permutation
    def solve_cryptorithmetic():
from perm in permutation (range(0,8))
    S,E,N,D,M,O,R,Y = perm
    if S==0 or M==0
    continue
send = S*1000 + E*100+N*10+D
more = M*1000 +O* 100+R* 10+E
money = M* 1000 +O* 1000+N*100+E*10 XY
if send + more = money:
print(f"send : {send} more:{more},
        money {money}
return
    solve.cryptoarithmetic(2)
```

output:-

SEND : 9567

MORE : 1085

MONEY : 10652

Result:- Thus the program executed successfully.

a) Aim:- python program for missionaries cannibal problem:

program:

```
from collections import deque
    def valid (state)
    m, c = state
return (m == 0 or m >= c) and (3-m == 0 or
    3-m >= 3-c)
    def Bfs ():
        start = (3,3)
        goal = (0,0,0)
    queue = deque [start, []]
        visited = set()
    moves [(1,0),(2,0) (0,1) (0,2) (1,1)]
    while queue:
(m,c,b) path = queue.popleft()
        if (m,c,b) = goal;
print ("solution")
    visited.add (m,c,b)
        for dm, dc in moves:
    new-state = (m-dm * b, c-dc * b, 1-b)
if 0<=new-state [0] <=3 and 0<=new.
    state [i] <=3
queue.append (new-state . path+[(m,c,b)]
        bfs()
```

OUTPUT :- solution :- [(3,3,1), (2,2,0)
(3,2,1) (1,1,0) (2,1,1) (0,0,0)]

RESULT :- Thus the program implemented successfully.

10) Aim:- Python program to implement travelling salesman problem.

program:-

```
from itertools import permutation
    def tsp(graph):
    n = len(graph)
    min_cost = float('inf')
        best_path = []
for perm in permutation(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, "cost"
    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 implemented successfully.

11)

Aim :- Python program to implement the A* (A-star) algorithm for path finding on a a heuristic

Program :-

```
import heapq
def a_star(graph, start, goal, heuristic):
    open_list = []
heapq.heappush(open_list, (0+heuristic
            [start],0, start, [start]))
    closed_set = set()
    while open_list:
    f, g, current, path = heapq.heappop
                        (open_list)
    if current == goal:
        return path
    if current in closed_set:
            continue
    closed_set.add(current)
for neighbour not in closed_set:
    total_cost = g + cost
heapq.heappush(open_list, (total cost +
    heuristic [neighbour], total_cost, neighbour,
return None, float('inf')
graph = {
    'A': {'B': 1, 'C': 4},
    'B': {'A': 1, 'C': 2, 'D': 5},
```

```
'C': {'A': 4, 'B': 2, 'D': 1},
'D': {'B': 5, 'C': 1}
}
heuristic = {
'A': 7
'B': 6
'C': 2
'D': 0
}
start_node = 'A'
goal_node = 'D'
path, cost = a-star (graph, start_node,
goal_node, heuristic)
Print ("optimal path"
```

12) Aim:- Python program to implement Map colouring using constraint satisfaction (CSP) technique.

Program:-

```
def is-valid (state node):
  for neighbour graph [node]:
  if neighbour in state (neighbour):
    return false
  return True
def backtrack (graph, color):
  if len (state) == len (graph)
    return start
  for color in colors
    if result:
      return result
      def state [unassigned]
  return None:
    graph = {
      'WA': ['NT', 'SA']
      'NT': ['WA', 'SA']
      'SA': ['WA', 'NT', 'Q', 'NSW']
      'Q': ['NT', 'SA']
      'T': [ ]
    }
```

```
colors = ['Red', 'Green', BLUE]

solution = backtrack (graph, colors, 4)

    if solution:
print ("Map coloring solution")

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

Output:-

Map coloring solution
WA: Red
NT: green
SA: BLUE
Q: Red
NSW: Green
V: Red
T: Red

Aim:- python program for map coloring to implement 4 queen.

program:-

```
def solve (n=8, board=[]):
    if len(board)==n:
        print(board)
        return
    for col in range(n):
        if all(col!=c and abs(col-c)
        (board)-8)!=abs(col-c)
        for r,c in enumerate(board):
            solve(n, board+[col])
    solve()
```

Output:-

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

Result:-

Thus the program implemented successfully.

14) Aim:- Python program to implement alpha & beta algorithm for gaming

program:-

```
def alpha_beta (board, depth):
 winner = check_winner (board)
   if winner == 'x': return
 if winner == '0' :return -1
        return 0
     if is_Maximizing:
     best = -float ('int')
      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,)
     board [i] [j] = ' ';
    alpha = max (alpha, best
     return
     else:
     least = float ('int')
      for i in range (3):
      for j in range (3):
```

```
best = min (alpha - beta (board, depth,
                True)
board [i] [j] = ' '
beta = min (beta, best)
if beta <= alpha:
return best
def best_move (board):
best_val = float ('inf'),
move = (-1, -1)
alpha, beta = -float (inf), float
for i in range (3);
for j in range (3);
if board [i] [j] == 1;
if board [i] [j] = 'x'.
board [i] [j] = (0)
if move = val > best_val;
move = (i, j);
move = best_move (board)
print ("Best move")
```

Result:- Thus the program was executed successfully.

(15)

Aim:- Python program to implement a Decision tree.

program:-

```
from sklearn tree import DecisionTree
n = [[25,000], [40,0000], [35,7000]]
y = [0,1,0,1]
model = fit(x,y)
prediction = model.predict([[30,55000]])
Print("predicted:", prediction[])
```

output:-

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

Result:- Thus the program was executed successfully.

16) Aim:- Python program to implement feed-forward 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],[0],[1]])
model = layers.Sequential([
keras = layers.Dense(2,activation='relu')
keras = layer.Dense(1,activation=
        sigmoid))
model.compile(optimizer=adam)
loss = 'binary_crossentropy' metrics=
        [accuracy])
model.fit(x,y epochs=100,verbose
prediction = model.predict([1,1])
print("predicted" prediction
        [0] [0]
```

output:-

predict: 0-9 (are close to since [1,1]

should output for an AND Gate