CSA1783 – Artificial Intelligence with Societal Impact

NAME: P.AKHIL

REG NO: 192124015

1. Write the python program to solve 8-Puzzle problem.

AIM: To write a python program to solve 8-Puzzle problem.

ALGORITHM:

1. **Initialize:** Create a starting node with the initial state of the puzzle. Calculate its heuristic value (h) using a heuristic function (like Manhattan distance) and set its cost from the start (g) to 0. Calculate the initial f-value as f = g + h.
2. **Open Set:** Create an open set (priority queue) and add the starting node to it.
3. **Closed Set:** Create a closed set (a set to store visited states).
4. **While Open Set is not empty:**
    - a. Pop the node with the lowest f-value from the open set.
    - b. If the popped node's state is the goal state, return the solution path (sequence of moves from the start to the goal).
    - c. Otherwise, move the current state to the closed set.
    - d. Generate successors by moving the empty tile in all possible directions (up, down, left, right).
    - e. For each successor state:
        - i. If the successor state is already in the closed set, skip it.
        - ii. Calculate the successor's g-value (cost from start) as the parent's g-value + 1.
        - iii. Calculate the successor's h-value (heuristic value) using a heuristic function.
        - iv. Calculate the successor's f-value as f = g + h.
        - v. If the successor state is not in the open set, add it to the open set.
        - vi. If the successor state is already in the open set but with a higher f-value, skip it.

5. **No Solution:** If the open set becomes empty and the goal state is not reached, return that no solution exists.

PROGRAM:

```
import heapq
goal_state = (1, 2, 3, 4, 5, 6, 7, 8, 0)
moves = [(0, 1), (0, -1), (1, 0), (-1, 0)]
def get_neighbors(state):
    neighbors = []
    empty_index = state.index(0)
    empty_row, empty_col = empty_index // 3, empty_index % 3
    for dr, dc in moves:
        new_row, new_col = empty_row + dr, empty_col + dc
        if 0 <= new_row < 3 and 0 <= new_col < 3:
            neighbor_state = list(state)
            neighbor_index = new_row * 3 + new_col
            neighbor_state[empty_index], neighbor_state[neighbor_index] =
neighbor_state[neighbor_index], neighbor_state[empty_index]
            neighbors.append(tuple(neighbor_state))
    return neighbors
def manhattan_distance(state):
    distance = 0
    for i in range(9):
        if state[i] != 0:
            current_row, current_col = i // 3, i % 3
            target_row, target_col = (state[i] - 1) // 3, (state[i] - 1) % 3
            distance += abs(current_row - target_row) + abs(current_col - target_col)
    return distance
def solve_puzzle(initial_state):
    open_list = [(manhattan_distance(initial_state), initial_state)]
    closed_set = set()
    while open_list:
        current_state = heapq.heappop(open_list)[1]
        if current_state == goal_state:
            return current_state
        closed_set.add(current_state)
        for neighbor in get_neighbors(current_state):
            if neighbor not in closed_set:
                heapq.heappush(open_list, (manhattan_distance(neighbor), neighbor))
    return None
initial_state = (1, 0, 3, 4, 2, 5, 7, 8, 6)
solution = solve_puzzle(initial_state)
if solution:
    print("Solution found:")
    for i in range(0, 9, 3):
        print(solution[i:i+3])
else:
    print("No solution found.")
```

OUTPUT:

```
Solution found:
(1, 2, 3)
(4, 5, 6)
(7, 8, 0)
```

RESULT:

Thus the python program for 8-Puzzle problem is successfully executed.

2. Write the python program to solve 8-Queen problem.

AIM: To write a python program to solve 8-Queen problem.

ALGORITHM:

```
1. Print "Enter the number of queens"
2. Read an integer N from the user
3. Create a 2D list "board" of size N x N, initialized with all zeros
4. Define a function "attack(i, j)" to check if a queen at position (i, j) can be attacked by any other queen:
  4.1. Loop over k from 0 to N-1:
    4.1.1. If board[i][k] is 1 or board[k][j] is 1, return True (there's an attacking queen)
  4.2. Loop over k from 0 to N-1:
    4.2.1. Loop over l from 0 to N-1:
      4.2.1.1. If (k + l == i + j) or (k - l == i - j):
        4.2.1.1.1. If board[k][l] is 1, return True (there's an attacking queen)
  4.3. Return False (no attacking queens found)
5. Define a recursive function "N_queens(n)" to solve the N Queens problem:
  5.1. If n is 0, return True (all queens are placed)
  5.2. Loop over i from 0 to N-1:
    5.2.1. Loop over j from 0 to N-1:
      5.2.1.1. If there is no attack at position (i, j) and board[i][j] is not 1:
        5.2.1.1.1. Place a queen at position (i, j) by setting board[i][j] to 1
        5.2.1.1.2. Recursively call N_queens(n-1)
          5.2.1.1.2.1. If the recursive call returns True, then return True (solution found)
        5.2.1.1.3. If the recursive call returned False, backtrack by setting board[i][j] to 0
  5.3. Return False (no solution found for the current configuration)
6. Call the function N_queens(N) to solve the N Queens problem
7. Loop over each row in the board:
  7.1. Print the row
8. End of the program.
```

PROGRAM:

```
print ("Enter the number of queens:")
N = int(input())
board = [[0]*N for _ in range(N)]

def attack(i, j):
    #checking vertically and horizontally
    for k in range(0,N):
        if board[i][k]==1 or board[k][j]==1:
            return True
    #checking diagonally
    for k in range(0,N):
        for l in range(0,N):
            if (k+l==i+j) or (k-l==i-j):
                if board[k][l]==1:
                    return True
    return False

def N_queens(n):
    if n==0:
        return True
    for i in range(0,N):
        for j in range(0,N):
            if (not(attack(i,j))) and (board[i][j]!=1):
                board[i][j] = 1
                if N_queens(n-1)==True:
                    return True
                board[i][j] = 0
```

```
return False
N_queens(N)
for i in board:
    print (i)
```

OUTPUT:

```
Enter the number of queens:
8
[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:

Thus the python program for 8-Queen problem is successfully executed.

3. Write the python program for Water Jug Problem.

AIM: To write a python program to solve Water Jug problem.

ALGORITHM:

```
1. Read the capacities of Jug A (a) and Jug B (b) from the user
2. Read the initial water amounts in Jug A (ai) and Jug B (bi) from the user
3. Read the final desired water amounts in Jug A (af) and Jug B (bf) from the user
4. Print the list of operations that can be performed on the jugs
5. Initialize a loop while the current state of the jugs (ai, bi) is not equal to the desired final state (af, bf):
    5.1. Read an operation (op) from the user
```

```
5.2. If op is 1:
    Set ai to the capacity of Jug A (a)
    5.3. If op is 2:
    Set bi to the capacity of Jug B (b)
5.4. If op is 3:
    Set ai to 0
5.5. If op is 4:
    Set bi to 0
5.6. If op is 5:
    If the amount that can be poured from Jug B to fill Jug A is greater than the current amount in Jug A:
        Increase the amount in Jug B by the amount needed to fill Jug A
        Set ai to 0
    Else:
        Decrease the amount in Jug A by the amount that can be poured from Jug B
        Set bi to the capacity of Jug B (b)
5.7. If op is 6:
    If the amount that can be poured from Jug A to fill Jug B is greater than the current amount in Jug B:
        Increase the amount in Jug A by the amount needed to fill Jug B
        Set bi to 0
    Else:
        Decrease the amount in Jug B by the amount that can be poured from Jug A
        Set ai to the capacity of Jug A (a)
5.8. If op is 7:
    Increase the amount in Jug A by the amount in Jug B
    Set bi to 0
5.9. If op is 8:
    Increase the amount in Jug B by the amount in Jug A
```

```
        Set ai to 0
    5.10. Print the current amounts in Jug A and Jug B (ai, bi)
6. End of the program.
```

PROGRAM:

```
a = int(input("Enter Jug A Capacity: "))
b = int(input("Enter Jug B Capacity: "))
ai = int(input("Initially Water in Jug A: "))
bi = int(input("Initially Water in Jug B: "))
af = int(input("Final State of Jug A: "))
bf = int(input("Final State of Jug B: "))
#print List Of Operations
print("List Of Operations You can Do:\n")
print("1.Fill Jug A Completely\n")
print("2.Fill Jug B Completely\n")
print("3.Empty Jug A Completely\n")
print("4.Empty Jug B Completely\n")
print("5.Pour From Jug A till Jug B filled Completely or A becomes empty\n")
print("6.Pour From Jug B till Jug A filled Completely or B becomes empty\n")
print("7.Pour all From Jug B to Jug A\n")
print("8.Pour all From Jug A to Jug B\n")
#loop
while ((ai != af or bi != bf)):
    op = int(input("Enter the Operation: "))
    if (op == 1):
        ai = a
    elif (op == 2):
        bi = b
    elif (op == 3):
        ai = 0
    elif (op == 4):
        bi = 0
    elif (op == 5):
        if (b-bi > ai):
            bi = ai+bi
            ai = 0
        else:
            ai = ai-(b-bi)
            bi = b
    elif (op == 6):
            if (a-ai > bi):
                ai = ai+bi
                bi = 0
            else:
                bi = bi-(a-ai)
                ai = a
    elif (op == 7):
        ai = ai+bi
        bi = 0
    elif (op == 8):
        bi = bi+ai
        ai = 0
    print(ai, bi)
```

<u>OUTPUT:</u>

```
Enter Jug A Capacity: 5
Enter Jug B Capacity: 4
Initially Water in Jug A: 2
Initially Water in Jug B: 3
Final State of Jug A: 3
Final State of Jug B: 3
List Of Operations You can Do:
1.Fill Jug A Completely
2.Fill Jug B Completely
3.Empty Jug A Completely
4.Empty Jug B Completely
5.Pour From Jug A till Jug B filled Completely or A becomes empty
6.Pour From Jug B till Jug A filled Completely or B becomes empty
7.Pour all From Jug B to Jug A
8.Pour all From Jug A to Jug B
Enter the Operation: 1
5 3
Enter the Operation: 2
5 4
```

RESULT:

Thus the python program for water jug problem is successfully executed.

4. Write the python program for Cript-Arithmetic problem.

AIM: To write a python program for Cript-Arithmetic problem.

ALGORITHM:

```
1. Define a function "is_valid_assignment(assignment, letters)":
   1.1. Convert the values of the assignment dictionary to a list "values"
   1.2. Return True if the length of "values" is equal to the length of the set of "values" (no
repeated digits)
   1.3. Return False otherwise
```

```
2. Define a function "evaluate_expression(expression, assignment)":
   2.1. Initialize a variable "value" to 0
   2.2. Loop over each character "char" in the expression:
      2.2.1. Update "value" by multiplying it by 10 and adding the value of "char" in the assignment dictionary
   2.3. Return the final "value"
3. Define a function "solve_cryptarithmetic(equation)":
   3.1. Split the equation into words by replacing "+" and "=" with spaces and then splitting
   3.2. Create a set "unique_letters" containing all unique letters from the words
   3.3. Create a set "leading_letters" containing the first letters of each word
   3.4. Create a list "letters" containing all unique letters
   3.5. Import "permutations" from itertools module
   3.6. Loop over all permutations of digits (0-9) with the same length as "letters" using "perm":
      3.6.1. Create an assignment dictionary by zipping "letters" and "perm"
      3.6.2. Check if all letters in "leading_letters" have a non-zero assignment in the dictionary:
         3.6.2.1. Evaluate the left-hand side value by calling "evaluate_expression" on the first word and assignment
         3.6.2.2. Evaluate the right-hand side value by calling "evaluate_expression" on the second word and assignment
         3.6.2.3. Evaluate the result value by calling "evaluate_expression" on the third word and assignment
         3.6.2.4. Check if the sum of left_value and right_value is equal to result_value:
            3.6.2.4.1. If true, return the assignment dictionary
   3.7. Return None if no valid assignment is found
4. Define the main part of the program:
   4.1. Create an example cryptarithmetic problem equation string
   4.2. Call "solve_cryptarithmetic" with the equation string and store the result in "solution"
   4.3. If a solution is found (not None):
      4.3.1. Print "Solution found:"
      4.3.2. Loop over each letter and digit in the solution dictionary:
        4.3.2.1. Print the letter and its corresponding digit
    4.4. Else:
        4.4.1. Print "No solution found."
5. End of the program.
```

PROGRAM:

```
def is_valid_assignment(assignment, letters):
    # Check if the assignment is valid (no repeated digits)
    values = list(assignment.values())
    return len(values) == len(set(values))
def evaluate_expression(expression, assignment):
    # Evaluate the expression using the given assignment
    value = 0
    for char in expression:
        value = value * 10 + assignment[char]
    return value
def solve_cryptarithmetic(equation):
    words = equation.replace("+", " ").replace("=", " ").split()
    unique_letters = set("".join(words))
    leading_letters = set(word[0] for word in words)
    letters = list(unique_letters)
    # Try all possible digit assignments
    from itertools import permutations
    for perm in permutations(range(10), len(letters)):
        assignment = dict(zip(letters, perm))
        if all(assignment[letter] != 0 for letter in leading_letters):
            left_value = evaluate_expression(words[0], assignment)
            right_value = evaluate_expression(words[1], assignment)
            result_value = evaluate_expression(words[2], assignment)
            if left_value + right_value == result_value:
                return assignment
    return None
# Example problem: SEND + MORE = MONEY
equation = "SEND + MORE = MONEY"
solution = solve_cryptarithmetic(equation)
if solution:
    print("Solution found:")
    for letter, digit in solution.items():
        print(f"{letter}: {digit}")
else:
    print("No solution found.")
```

OUTPUT:

```
Solution found:
N: 6
D: 7
E: 5
S: 9
O: 0
R: 8
Y: 2
M: 1
```

RESULT:

Thus the python program for cript-arithmetic problem is successfully executed.

5. Write the python program for Missionaries Cannibal problem.

AIM: To write a python program for Missionaries Cannibal problem.

ALGORITHM:

```
1. Print introductory messages explaining the game rules and goal
2. Initialize the initial states of missionaries and cannibals on the left side (lM = 3, lC = 3)
   and the right side (rM = 0, rC = 0)
3. Initialize variables to track the user's input for traveling missionaries and cannibals (userM, userC)
   and a variable "k" to count the number of attempts
4. Print the initial state of the left and right sides of the river
5. Create an outer loop that continues until the game is won or lost:
    5.1. Create an inner loop to get valid user input for traveling missionaries and cannibals:
        5.1.1. Read the number of missionaries and cannibals the user wants to travel (userM, userC)
        5.1.2. If both inputs are zero, print a message and ask for valid input
        5.1.3. Else if the total number of people traveling is less than or equal to 2,
              and the number of missionaries and cannibals can be subtracted from the left side,
              break the inner loop
6. Update the states of missionaries and cannibals on both sides:
    6.1. Subtract the user inputs (userM and userC) from the right side (rM, rC)
    6.2. Add the user inputs to the left side (lM, lC)
    6.3. Increment "k" by 1
7. Print the current state of the left and right sides of the river
8. Check for game-ending conditions:
    8.1. If any of the following conditions are met, print a losing message and break the outer loop:
        - There are 1 missionary and 3 cannibals on the left side
        - There are 2 missionaries and 3 cannibals on the left side
        - There are 1 missionary and 2 cannibals on the left side
        - There are 1 missionary and 3 cannibals on the right side
        - There are 2 missionaries and 3 cannibals on the right side
        - There are 1 missionary and 2 cannibals on the right side
```

8.2. If the total number of people on the right side is 6, print a winning message and the total attempts, then break the outer loop

9. Create another inner loop to get valid user input for traveling missionaries and cannibals from the right to the left side

10. Update the states of missionaries and cannibals on both sides (similar to step 6)

11. Print the current state of the left and right sides of the river

12. Go back to step 8 and repeat the game-ending condition checks

13. Handle any invalid input exceptions with an error message

14. End of the program

PROGRAM:

```
print("\n")

print("\tGame Start\nNow the task is to move all of them to right side of the river")

print("rules:\n1. The boat can carry at most two people\n2. If cannibals num greater than missionaries then the cannibals would eat the missionaries\n3. The boat cannot cross the river by itself with no people on board")

lM = 3

lC = 3

rM=0

rC=0

userM = 0

userC = 0

k = 0

print("\nM M M C C C |          --- | \n")

try:

        while(True):

                while(True):

                        print("Left side -> right side river travel")

                        uM = int(input("Enter number of Missionaries travel => "))

                        uC = int(input("Enter number of Cannibals travel => "))
                        if((uM==0)and(uC==0)):
                            print("Empty travel not possible")
                            print("Re-enter : ")
                        elif(((uM+uC) <= 2)and((lM-uM)>=0)and((lC-uC)>=0)):
                            break
                        else:
                            print("Wrong input re-enter : ")
                lM = (lM-uM)
                lC = (lC-uC)
                rM += uM
                rC += uC
                print("\n")
                for i in range(0,lM):
                        print("M ",end="")
                for i in range(0,lC):
                        print("C ",end="")
                print("| --> | ",end="")
                for i in range(0,rM):
                        print("M ",end="")
                for i in range(0,rC):
                        print("C ",end="")
                print("\n")
                k +=1
                if(((lC==3)and (lM ==
1))or((lC==3)and(lM==2))or((lC==2)and(lM==1))or((rC==3)and (rM ==
1))or((rC==3)and(rM==2))or((rC==2)and(rM==1))):
                        print("Cannibals eat missionaries:\nYou lost the game")
                        break
```

```
if((rM+rC) == 6):
        print("You won the game : \n\tCongrats")
        print("Total attempt")
        print(k)
        break
while(True):
        print("Right side -> Left side river travel")
        userM = int(input("Enter number of Missionaries travel => "))
        userC = int(input("Enter number of Cannibals travel => "))
        if((userM==0)and(userC==0)):
                        print("Empty travel not possible")
                        print("Re-enter : ")
        elif(((userM+userC) <= 2)and((rM-userM)>=0)and((rC-userC)>=0)):
                break
        else:
                print("Wrong input re-enter : ")
lM += userM
lC += userC
rM -= userM
rC -= userC
k +=1
print("\n")
for i in range(0,lM):
        print("M ",end="")
for i in range(0,lC):
        print("C ",end="")
print("| <-- | ",end="")
for i in range(0,rM):
```

```
                    print("M ",end="")
                for i in range(0,rC):
                    print("C ",end="")
                print("\n")

                if(((lC==3)and (lM ==
1))or((lC==3)and(lM==2))or((lC==2)and(lM==1))or((rC==3)and (rM ==
1))or((rC==3)and(rM==2))or((rC==2)and(rM==1))):
                    print("Cannibals eat missionaries:\nYou lost the game")
                    break
except EOFError as e:
    print("\nInvalid input please retry !!")
```

OUTPUT:

```
Game Start
Now the task is to move all of them to right side of the river
rules:
1. The boat can carry at most two people
2. If cannibals num greater than missionaries then the cannibals would eat the missionaries
3. The boat cannot cross the river by itself with no people on board
M M M C C C |          --- |
Left side -> right side river travel
Enter number of Missionaries travel => 1
Enter number of Cannibals travel => 1
M M C C | --> | M C
Right side -> Left side river travel
```

RESULT: Thus the program for Missionaries Cannibal problem is exexcuted successfully.

6. Write the python program for Vacuum Cleaner problem.

AIM: To write a python program for Vaccum Cleaner problem.

ALGORITHM:

```
1. Define a class "VaccumCleaner":
   1.1. Define a constructor "__init__" that takes an optional parameter "position" with default value "A":
      1.1.1. Initialize the instance variable "position" with the provided or default position
      1.2. Define a method "move_to" that takes a parameter "new_position":
      1.2.1. Update the "position" instance variable with the new position
      1.2.2. Print a message indicating that the vacuum cleaner has moved to the new position
   1.3. Define a method "clean":
      1.3.1. Print a message indicating that the vacuum cleaner is cleaning at the current position
   1.4. Define a method "run" that takes a parameter "actions":
      1.4.1. Loop over each "action" in the "actions" list:
         1.4.1.1. If the action is "MoveA", call the "move_to" method with position "A"
         1.4.1.2. If the action is "MoveB", call the "move_to" method with position "B"
         1.4.1.3. If the action is "Clean", call the "clean" method
         1.4.1.4. If the action is not recognized, print an "Invalid action" message
2. Create a list "actions" containing a sequence of actions to simulate the vacuum cleaner's behavior
3. Create an instance of the "VaccumCleaner" class and assign it to the variable "vaccum"
4. Print a message indicating the start of the vacuum cleaner simulation
5. Call the "run" method on the "vaccum" instance, passing the "actions" list
6. End of the program.
```

PROGRAM:

```
class VaccumCleaner:
   def _init_(self, position="A"):
      self.position = position
      def move_to(self, new_position):
      self.position = new_position
```

```
        print(f"Vaccum cleaner moved to position {self.position}")
        def clean(self):
        print(f"Cleaning at position {self.position}")
        def run(self, actions):
        for action in actions:
            if action == "MoveA":
                self.move_to("A")
            elif action == "MoveB":
                self.move_to("B")
            elif action == "Clean":
                self.clean()
            else:
                print(f"Invalid action: {action}")
actions = ["MoveA", "Clean", "MoveB", "Clean"]
vaccum = VaccumCleaner()
print("Starting vacuum cleaner simulation...")
vaccum.run(actions)
```

OUTPUT:

```
Starting vacuum cleaner simulation...
Vaccum cleaner moved to position A
Cleaning at position A
Vaccum cleaner moved to position B
Cleaning at position B
```

RESULT: Thus the program for Vaccum Cleaner problem is executed successfully.

7. Write the python program to implement BFS.

AIM: To write a python program to implement BFS.

ALGORITHM:

```
1. Define a function "create_graph" to create a graph:
   1.1. Initialize an empty dictionary "graph"
   1.2. Read the number of vertices from the user and store it in "vertices"
   1.3. Loop "vertices" times:
      1.3.1. Read a vertex from the user and store it in "vertex"
      1.3.2. Read a comma-separated list of neighbors from the user, split it, and store it in "neighbors"
      1.3.3. Add an entry to the "graph" dictionary with the "vertex" as the key and "neighbors" as the value
   1.4. Return the "graph"
2. Define a function "bfs(graph, start_node)" to perform Breadth-First Search:
   2.1. Initialize an empty list "visited" to store visited nodes
   2.2. Initialize a queue "queue" with the "start_node" as the only element
   2.3. While "queue" is not empty:
      2.3.1. Pop the first element "node" from "queue"
      2.3.2. If "node" is not in "visited":
         2.3.2.1. Add "node" to "visited"
         2.3.2.2. Extend "queue" with neighbors of "node" that are not in "visited"
   2.4. Return the "visited" list
3. Define the main function "main":
   3.1. Call "create_graph" to create the graph and store it in "graph"
   3.2. Read the starting node from the user and store it in "start_node"
   3.3. Print a message indicating the start of the Breadth-First Search
   3.4. Call "bfs" with the "graph" and "start_node" and store the result in "result"
   3.5. Print the nodes visited in BFS order separated by ' -> '
4. Check if the script is being run as the main module using "__name__":
   4.1. If yes, call the "main" function
```

5. End of the program.

PROGRAM:

```
def create_graph():
    graph = {}
    vertices = int(input("Enter the number of vertices: "))
    for _ in range(vertices):
        vertex = input(f"Enter vertex: ")
        neighbors = input(f"Enter neighbors of {vertex} (comma-separated): ").split(',')
        graph[vertex] = neighbors
    return graph

def bfs(graph, start_node):
    visited = []
    queue = [start_node]

    while queue:
        node = queue.pop(0)
        if node not in visited:
            visited.append(node)
            queue.extend(neighbour for neighbour in graph[node] if neighbour not in visited)
    return visited

def main():
    graph = create_graph()
    start_node = input("Enter the starting node: ")
    print("Following is the Breadth-First Search:")
    result = bfs(graph, start_node)
    print(' -> '.join(result))
if __name__ == "__main__":
    main()
```

OUTPUT:

```
Enter the number of vertices: 4
Enter vertex: A
Enter neighbors of A (comma-separated): B,C
Enter vertex: B
Enter neighbors of B (comma-separated): A,C,D
Enter vertex: C
Enter neighbors of C (comma-separated): A,B
Enter vertex: D
Enter neighbors of D (comma-separated): B
Enter the starting node: A
Following is the Breadth-First Search:
A -> B -> C -> D
```

RESULT: Thus the python program for BFS is executed successfully.

8. Write the python program to implement DFS.

AIM: To write a python program to implement DFS.

ALGORITHM:

```
1. Define a function "dfs(graph, start, visited=None)" to perform Depth-First Search:
    1.1. If "visited" is None, initialize it as an empty set
    1.2. Add "start" to the "visited" set
    1.3. Print "start" node
    1.4. Loop over each "next_node" in the difference between the neighbors of "start" and the "visited" set:
        1.4.1. Recursively call "dfs" with "graph", "next_node", and "visited"
    1.5. Return the updated "visited" set
2. Initialize an empty dictionary "graph" to represent the graph
3. Read the number of edges from the user and store it in "num_edges"
4. Loop "num_edges" times:
    4.1. Read an edge in the format "node1 node2" from the user and split it into "node1" and "node2"
    4.2. If "node1" is not in the "graph", add it as a key with an empty set value
    4.3. If "node2" is not in the "graph", add it as a key with an empty set value
    4.4. Add "node2" to the set of neighbors of "node1" in the "graph"
5. Read the starting node from the user and store it in "start_node"
6. Call the "dfs" function with the "graph" and "start_node" as arguments
7. End of the program.
```

PROGRAM:

```python
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)

    print(start)

    for next_node in graph[start] - visited:
        dfs(graph, next_node, visited)
    return visited

# Take input from the user to build the graph
graph = {}
num_edges = int(input("Enter the number of edges: "))
```

```
for _ in range(num_edges):
    edge = input("Enter an edge in the format 'node1 node2': ").split()
    node1, node2 = edge
    if node1 not in graph:
        graph[node1] = set()
    if node2 not in graph:
        graph[node2] = set()
    graph[node1].add(node2)

start_node = input("Enter the starting node: ")
dfs(graph, start_node)
```

OUTPUT:

```
Enter the number of edges: 4
Enter an edge in the format 'node1 node2': A B
Enter an edge in the format 'node1 node2': B C
Enter an edge in the format 'node1 node2': C A
Enter an edge in the format 'node1 node2': D B
Enter the starting node: B
B
C
A
```

RESULT: Thus the python program for DFS is successfully executed.

9. Write the python to implement Travelling Salesman Problem.

AIM: To write a python program to implement Travelling Salesman Problem.

ALGORITHM:

```
1. Define a function "distance(city1, city2)" to calculate the distance between two cities:
   1.1. Calculate the absolute difference in x-coordinates and y-coordinates between the two cities
   1.2. Return the sum of the absolute differences
2. Define a function "total_distance(path, cities)" to calculate the total distance of a path through cities:
   2.1. Initialize a variable "total" to 0
   2.2. Loop "i" over the range from 0 to the length of "path" - 1:
      2.2.1. Add the distance between city at index "path[i]" and "path[i + 1]" to "total"
   2.3. Add the distance between the last city in "path" and the starting city to "total"
   2.4. Return "total"
3. Define a function "brute_force_tsp(cities)" to solve the Travelling Salesman Problem using brute force:
   3.1. Initialize "num_cities" as the length of "cities"
   3.2. Initialize "min_distance" as positive infinity and "best_path" as an empty list
      3.3. Loop over all permutations of cities using itertools.permutations(range(num_cities)):
      3.3.1. Calculate the total distance of the current permutation using "total_distance"
      3.3.2. If the calculated distance is smaller than "min_distance":
         3.3.2.1. Update "min_distance" with the calculated distance
         3.3.2.2. Update "best_path" with the current permutation
   3.4. Return "best_path" and "min_distance"
4. Read the number of cities from the user and store it in "num_of_cities"
5. Initialize an empty list "cities" to store city coordinates
6. Loop "i" over the range from 0 to "num_of_cities":
   6.1. Read the x and y coordinates of the city from the user
   6.2. Append a tuple of (x, y) coordinates to the "cities" list
7. Call the "brute_force_tsp" function with the "cities" list and store the results in "best_path" and "min_distance"
8. Print the best path and the minimum distance.
```

PROGRAM:

```
import itertools
def distance(city1, city2):
    return abs(city1[0] - city2[0]) + abs(city1[1] - city2[1])


def total_distance(path, cities):
    total = 0
    for i in range(len(path) - 1):
        total += distance(cities[path[i]], cities[path[i + 1]])
    total += distance(cities[path[-1]], cities[path[0]])  # Return to starting city
    return total


def brute_force_tsp(cities):
    num_cities = len(cities)
    min_distance = float('inf')
    best_path = []

    for path in itertools.permutations(range(num_cities)):
        d = total_distance(path, cities)
        if d < min_distance:
            min_distance = d
            best_path = path

    return best_path, min_distance

# Get user input for cities
num_of_cities = int(input("Enter the number of cities: "))
cities = []
```

```
for i in range(num_of_cities):
    x, y = map(int, input(f"Enter coordinates for city {i+1} (x y): ").split())
    cities.append((x, y))

best_path, min_distance = brute_force_tsp(cities)
print("Best Path:", best_path)
print("Min Distance:", min_distance)
```

OUTPUT:

```
Enter the number of cities: 2
Enter coordinates for city 1 (x y): 1 2
Enter coordinates for city 2 (x y): 2 0
Best Path: (0, 1)
Min Distance: 6
```

RESULT: Thus the python program for Travelling Salesman Problem is executed successfully.

10. Write the python program to implement A* algorithm.

AIM: To write a python program to implement A* algorithm.

ALGORITHM:

```
1. Create a base class "State":
   1.1. Initialize instance variables "children", "parent", "value", "dist", "path", "start", and "goal"
   1.2. Define the constructor "__init__":
      1.2.1. Initialize "children" as an empty list
      1.2.2. Initialize "parent" with the provided parent value
      1.2.3. Initialize "value" with the provided value
      1.2.4. Initialize "dist" as 0
      1.2.5. If a parent exists:
          1.2.5.1. Set "start" and "goal" from the parent's "start" and "goal"
          1.2.5.2. Copy "path" from the parent's path and append "value" to it
      1.2.6. If no parent exists:
          1.2.6.1. Initialize "path" as a list with "value"
          1.2.6.2. Initialize "start" and "goal" with the provided start and goal values
2. Create a subclass "State_String" that inherits from "State":
   2.1. Define the constructor "__init__":
      2.1.1. Call the parent class constructor and pass the provided values
      2.1.2. Calculate "dist" using the "GetDistance" method


   2.2. Define the method "GetDistance":
      2.2.1. If the value matches the goal, return 0
      2.2.2. Initialize "dist" as 0
      2.2.3. Loop through the characters in the goal:
          2.2.3.1. Calculate the distance between the character's index in the goal and the index of the same character in the value
          2.2.3.2. Add the calculated distance to "dist"
      2.2.4. Return "dist"
   2.3. Define the method "CreateChildren":
      2.3.1. If children list is empty:
          2.3.1.1. Loop through the range of characters except the last one in the goal:
              2.3.1.1.1. Swap the characters at the current index and the next index in the value
              2.3.1.1.2. Create a new child with the swapped value and the current instance as the parent
              2.3.1.1.3. Append the child to the children list
3. Create a class "A_Star_Solver":
   3.1. Define the constructor "__init__":
      3.1.1. Initialize instance variables "path", "visitedQueue", "priorityQueue", "start", and "goal"
      3.1.2. Create an instance of "State_String" as "startState"
   3.2. Define the method "Solve":
      3.2.1. Initialize "count" as 0
      3.2.2. Add the "startState" to the priority queue with distance 0 and count as the priority
      3.2.3. Loop while "path" is empty and the priority queue is not empty:
          3.2.3.1. Get the closest child from the priority queue
          3.2.3.2. Create children for the closest child
          3.2.3.3. Add the closest child's value to the visited queue
          3.2.3.4. Loop through the children of the closest child:
              3.2.3.4.1. If the child's value is not in the visited queue:
                  3.2.3.4.1.1. Increment "count"
                  3.2.3.4.1.2. If the child's distance is 0, set "path" to the child's path and break loop
                  3.2.3.4.1.3. Put the child in the priority queue with distance and count as the priority
      3.2.4. If "path" is still empty, print "Goal is not possible: " + the goal value
      3.2.5. Return the "path"
4. If the script is being run as the main module using "__name__":
   4.1. Read the starting string from the user and store it in "start1"
   4.2. Read the goal string from the user and store it in "goal1"
   4.3. Create an instance of "A_Star_Solver" with "start1" and "goal1"
   4.4. Call the "Solve" method and store the result in "a.path"
   4.5. Loop through the indexes and paths in "a.path":
      4.5.1. Print the index and the path at that index
5. End of the program
```

PROGRAM:

```
from queue import PriorityQueue
# Creating Base Class
class State(object):
    def __init__(self, value, parent, start=0, goal=0):
        self.children = []
        self.parent = parent
        self.value = value
        self.dist = 0
        if parent:
            self.start = parent.start
            self.goal = parent.goal
            self.path = parent.path[:]
            self.path.append(value)
        else:
            self.path = [value]
            self.start = start
            self.goal = goal

    def GetDistance(self):
        pass

    def CreateChildren(self):
        pass

# Creating subclass
class State_String(State):
    def __init__(self, value, parent, start=0, goal=0):
        super(State_String, self).__init__(value, parent, start, goal)
        self.dist = self.GetDistance()

    def GetDistance(self):
        if self.value == self.goal:
            return 0
        dist = 0
        for i in range(len(self.goal)):
            letter = self.goal[i]
            dist += abs(i - self.value.index(letter))
        return dist

    def CreateChildren(self):
        if not self.children:
            for i in range(len(self.goal) - 1):
                val = self.value
                val = val[:i] + val[i + 1] + val[i] + val[i + 2 :]
                child = State_String(val, self)
                self.children.append(child)

# Creating a class that holds the final magic
class A_Star_Solver:
    def __init__(self, start, goal):
        self.path = []
        self.visitedQueue = []
        self.priorityQueue = PriorityQueue()
        self.start = start
        self.goal = goal

    def Solve(self):
        startState = State_String(self.start, 0, self.start, self.goal)

        count = 0
        self.priorityQueue.put((0, count, startState))
        while not self.path and self.priorityQueue.qsize():
            closestChild = self.priorityQueue.get()[2]
            closestChild.CreateChildren()
            self.visitedQueue.append(closestChild.value)
            for child in closestChild.children:
                if child.value not in self.visitedQueue:
                    count += 1
                    if not child.dist:
                        self.path = child.path
                        break
                    self.priorityQueue.put((child.dist, count, child))
        if not self.path:
            print("Goal is not possible: " + self.goal)
        return self.path


if __name__ == "__main__":
    start1 = input("Enter the starting string: ")
    goal1 = input("Enter the goal string: ")
    print("Starting....")
    a = A_Star_Solver(start1, goal1)
    a.Solve()
    for i in range(len(a.path)):
        print("{0}){1}".format(i, a.path[i]))
```

OUTPUT:

```
Enter the starting string: MURALI
Enter the goal string: RALIMU
Starting....
```

```
0)MURALI
1)MRUALI
2)RMUALI
3)RMAULI
4)RAMULI
5)RAMLUI
6)RALMUI
7)RALMIU
8)RALIMU
```

RESULT: Thus the python program for A* algorithm is executed successfully.

11. Write the python program for Map Coloring to implement CSP.

AIM: To write a python program for Map Coloring to implement CSP.

ALGORITHM:

```
1. Define a function "map_coloring(graph)" to perform map coloring:
  1.1. Create an empty dictionary "colors" to store assigned colors for each region
  1.2. Define a list "available_colors" containing available colors
    1.3. Loop through each "region" in the "graph":
    1.3.1. Create a set "used_colors" containing colors assigned to neighbors of the current "region"
    1.3.2. Loop through each "color" in "available_colors":
      1.3.2.1. If the "color" is not in "used_colors":
        1.3.2.1.1. Assign the "color" to the current "region" in the "colors" dictionary
        1.3.2.1.2. Break the loop
    1.4. Return the "colors" dictionary containing region-color pairs
2. Define the main function "main":
  2.1. Create an empty dictionary "map_graph" to store region-neighbors pairs
  2.2. Read the number of regions from the user and store it in "num_regions"
```

```
    2.3. Loop "num_regions" times:
    2.3.1. Read the name of the "region" from the user and store it in "region_name"
    2.3.2. Read the neighbors of the "region" separated by spaces and split them into a list "neighbors"
    2.3.3. Add an entry to the "map_graph" dictionary with "region_name" as the key and "neighbors" as the value
    2.4. Call the "map_coloring" function with "map_graph" as an argument and store the result in "colored_map"
  2.5. Loop through each "region" and "color" pair in "colored_map":
    2.5.1. Print the message indicating the region and the color assigned to it
3. Check if the script is being run as the main module using "__name__":
  3.1. If yes, call the "main" function
4. End of the program
```

PROGRAM:

```
def map_coloring(graph):
    colors = {}  # Dictionary to store assigned colors for each region

    # List of available colors
    available_colors = ['red', 'green', 'blue', 'yellow', 'purple', 'orange']

    for region in graph:
        used_colors = set(colors.get(neighbour) for neighbour in graph[region] if neighbour in colors)
        for color in available_colors:
            if color not in used_colors:
                colors[region] = color
                break

    return colors
```

```
def main():
    map_graph = {}
    num_regions = int(input("Enter the number of regions: "))

    for i in range(num_regions):
        region_name = input(f"Enter the name of region {i+1}: ")
        neighbors = input(f"Enter the neighbors of {region_name} separated by spaces: ").split()
        map_graph[region_name] = neighbors

    colored_map = map_coloring(map_graph)
    for region, color in colored_map.items():
        print(f"Region {region} is colored {color}")

if __name__ == "__main__":
    main()
```

OUTPUT:

```
Enter the number of regions: 2
Enter the name of region 1: A
Enter the neighbors of A separated by spaces: B C
Enter the name of region 2: B
Enter the neighbors of B separated by spaces: A C D
Region A is colored red
Region B is colored green
```

RESULT: Thus the python program for map coloring is executed successfully.

12. Write the python program for Tic Tac Toe game.

AIM: To write a python program for Tic Tac Toe game.

ALGORITHM:

1. Define a function "sum(a, b, c)" that calculates the sum of three numbers a, b, and c.

2. Define a function "printBoard(xState, zState)" to print the current state of the Tic Tac Toe board:

    2.1. For each position, determine the value to print (X, O, or the position number itself).

    2.2. Print the Tic Tac Toe board with values in their respective positions.

3. Define a function "checkWin(xState, zState)" to check if a player has won the game:

    3.1. Define a list "wins" containing all possible winning combinations of positions.

    3.2. Loop through each combination in "wins":

        3.2.1. If the sum of the positions in "xState" at the current combination is 3, X has won the game.

        3.2.2. If the sum of the positions in "zState" at the current combination is 3, O has won the game.

    3.3. Return 1 if X has won, 0 if O has won, or -1 if no player has won yet.

4. Check if the script is being run as the main module using "__name__":

    4.1. Initialize the total number of turns as 9.

    4.2. Initialize two lists "xState" and "zState" to represent the state of X and O respectively.

    4.3. Initialize "turn" as 1 (1 for X and 0 for O).

    4.4. Print the welcome message for the Tic Tac Toe game.

    4.5. Enter a loop that continues until the game is over or all turns are used up:

        4.5.1. Print the current state of the board using the "printBoard" function.

        4.5.2. If it's X's turn (turn = 1), prompt the user for a value and mark the corresponding position in "xState".

        4.5.3. If it's O's turn (turn = 0), prompt the user for a value and mark the corresponding position in "zState".

        4.5.4. Decrease the total turns count by 1.

        4.5.5. Check if a player has won or if all turns are used up:

            - If a player has won or all turns are used up, print "GAME OVER" and the final state of the board.

- Break the loop.

5. End of the program.

PROGRAM:

```
print("Murali")
def sum(a,b,c):
    return a+b+c
def printBoard(xState , zState):
    zero =  'X' if xState[0] else('@' if zState[0] else 0 )
    one =  'X' if xState[1] else('@' if zState[1] else 1 )
    two =  'X' if xState[2] else('@' if zState[2] else 2 )
    three =  'X' if xState[3] else('@' if zState[3] else 3 )
    four =  'X' if xState[4] else('@' if zState[4] else 4 )
    five =  'X' if xState[5] else('@' if zState[5] else 5 )
    six =  'X' if xState[6] else('@' if zState[6] else 6 )
    seven =  'X' if xState[7] else('@' if zState[7] else 7 )
    eight =  'X' if xState[8] else('@' if zState[8] else 8 )
    print(f" {zero} | {one} | {two} ")
    print(f"---|---|---")
    print(f" {three} | {four} | {five} ")
    print(f"---|---|---")
    print(f" {six} | {seven} | {eight} ")

def checkWin(xState,zState) :
    wins = [[0,1,2], [3,4,5], [6,7,8], [0,3,6], [1,4,7], [2,5,8], [0,4,8], [2,4,6]]
    for win in wins :
        if(sum(xState[win[0]], xState[win[1]], xState[win[2]]) == 3) :
            print("X won the game")
            return 1
        if(sum(zState[win[0]], zState[win[1]], zState[win[2]]) == 3) :
            print("O won the game.")
            return 0
    return -1


if __name__ == "__main__" :
    total_turns = 9
    xState = [0, 0, 0, 0, 0, 0, 0, 0, 0 ]
    zState = [0, 0, 0, 0, 0, 0, 0, 0, 0 ]
    turn = 1 # 1 for X and 0 for O
    print("Welcome to TIC-TAC-TOE")
    while(True) :
        printBoard(xState, zState)
        if(turn == 1):
            print("X's Chance")
            value = int(input("Please enter a value : "))
            xState[value] = 1
        else :
            print("O's Chance")
            value = int(input("Please enter a value : "))
            zState[value] = 1
        total_turns = total_turns - 1
        if(checkWin(xState, zState) != -1 or total_turns == 0 ):
            print("GAME OVER")
            printBoard(xState, zState)
            break
        turn = 1 -  turn
```

OUTPUT:

```
Murali
Welcome to TIC-TAC-TOE
 0 | 1 | 2
---|---|---
 3 | 4 | 5
---|---|---
 6 | 7 | 8
X's Chance

Please enter a value : 1
 0 | X | 2
---|---|---
 3 | 4 | 5
---|---|---
 6 | 7 | 8
O's Chance

Please enter a value : 2
 0 | X | @
---|---|---
 3 | 4 | 5
---|---|---
 6 | 7 | 8
X's Chance

Please enter a value : 3
 0 | X | @
---|---|---
```

```
 X | 4 | 5
---|---|---
 6 | 7 | 8
O's Chance

Please enter a value : 5
 0 | X | @
---|---|---
 X | 4 | @
---|---|---
 6 | 7 | 8
X's Chance

Please enter a value : 8
 0 | X | @
---|---|---
 X | 4 | @
---|---|---
 6 | 7 | X
O's Chance

Please enter a value : 6
 0 | X | @
---|---|---
 X | 4 | @
---|---|---
 @ | 7 | X
X's Chance
```

```
Please enter a value : 4
 0 | X | @
---|---|---
 X | X | @
---|---|---
 @ | 7 | X
O's Chance

Please enter a value : 7
 0 | X | @
---|---|---
 X | X | @
---|---|---
 @ | @ | X
X's Chance

Please enter a value : 0
X won the game
GAME OVER
 X | X | @
---|---|---
 X | X | @
---|---|---
 @ | @ | X
```

RESULT: Thus the python program for Tic Tac Toe game is successfully executed.

13. Write the python program to implement Minimax algorithm for gaming.

AIM: To write a python program to implement Minmax algorithm for gaming.

ALGORITHM:

```
1. Import the "math" module.
2. Define the function "minimax(curDepth, nodeIndex, maxTurn, scores, targetDepth)":
   2.1. Base case: If "curDepth" is equal to "targetDepth", return the score at the "nodeIndex".
      2.2. If "maxTurn" is True (maximizing player's turn):
      2.2.1. Return the maximum of the two recursive calls:
         - Call minimax with incremented "curDepth" and "nodeIndex" * 2, and set "maxTurn" to False.
         - Call minimax with incremented "curDepth" and "nodeIndex" * 2 + 1, and set "maxTurn" to False.
      2.3. If "maxTurn" is False (minimizing player's turn):
      2.3.1. Return the minimum of the two recursive calls:
         - Call minimax with incremented "curDepth" and "nodeIndex" * 2, and set "maxTurn" to True.
         - Call minimax with incremented "curDepth" and "nodeIndex" * 2 + 1, and set "maxTurn" to True.
3. Get input from the user:
   3.1. Create an empty list "scores" to store the input scores.
   3.2. Read the number of scores from the user and store it in "num_scores".
   3.3. Loop "num_scores" times:
      3.3.1. Read a score from the user and append it to the "scores" list.
   3.4. Read the target depth from the user and store it in "targetDepth".
4. Calculate the "treeDepth" by taking the base-2 logarithm of the length of "scores".
5. Print "The optimal value is:" followed by the result of the "minimax" function called with the initial arguments:
   - curDepth = 0
   - nodeIndex = 0
   - maxTurn = True (since it's the maximizing player's turn initially)
   - "scores" list as input
```

    - "targetDepth" as input
6. End of the program.

PROGRAM:

```
import math
def minimax(curDepth, nodeIndex, maxTurn, scores, targetDepth):
    if curDepth == targetDepth:
        return scores[nodeIndex]


    if maxTurn:
        return max(
            minimax(curDepth + 1, nodeIndex * 2, False, scores, targetDepth),
            minimax(curDepth + 1, nodeIndex * 2 + 1, False, scores, targetDepth)
        )
    else:
        return min(
            minimax(curDepth + 1, nodeIndex * 2, True, scores, targetDepth),
            minimax(curDepth + 1, nodeIndex * 2 + 1, True, scores, targetDepth)
        )

# Get input from the user
scores = []
num_scores = int(input("Enter the number of scores: "))
for _ in range(num_scores):
    score = int(input("Enter a score: "))
    scores.append(score)

targetDepth = int(input("Enter the target depth: "))
```

```
treeDepth = math.log(len(scores), 2)
print("The optimal value is:", minimax(0, 0, True, scores, targetDepth))
```

OUTPUT:

```
Enter the number of scores: 3
Enter a score: 5
Enter a score: 8
Enter a score: 1
Enter the target depth: 1
The optimal value is: 8
```

RESULT: Thus the python program for Min max Algorithm is successfully executed.

14. Write the python program to implement Alpha & Beta pruning algorithm for gaming.

AIM: To write a python program to implement Alpha & Beta pruning algorithm for gaming.

ALGORITHM:

```
1. Define constants MAX and MIN with values 1000 and -1000 respectively.
2. Define the function "minimax(depth, nodeIndex, maximizingPlayer, values, alpha, beta)":
  2.1. Base case: If "depth" is equal to 3, return the value at the "nodeIndex".
    2.2. If "maximizingPlayer" is True (maximizing player's turn):
    2.2.1. Initialize "best" with MIN.
    2.2.2. Loop through the two children nodes (i = 0, 1):
      2.2.2.1. Call minimax recursively with incremented "depth" and "nodeIndex * 2 + i",
           "maximizingPlayer" set to False, and pass "alpha" and "beta".
      2.2.2.2. Update "best" with the maximum of "best" and "val".
      2.2.2.3. Update "alpha" with the maximum of "alpha" and "best".
```

```
            2.2.2.4. If "beta" is less than or equal to "alpha", break the loop.
        2.2.3. Return "best".
    2.3. If "maximizingPlayer" is False (minimizing player's turn):
        2.3.1. Initialize "best" with MAX.
        2.3.2. Loop through the two children nodes (i = 0, 1):
            2.3.2.1. Call minimax recursively with incremented "depth" and "nodeIndex * 2 + i",
                "maximizingPlayer" set to True, and pass "alpha" and "beta".
            2.3.2.2. Update "best" with the minimum of "best" and "val".
            2.3.2.3. Update "beta" with the minimum of "beta" and "best".
            2.3.2.4. If "beta" is less than or equal to "alpha", break the loop.
        2.3.3. Return "best".
3. Check if the script is being run as the main module using "__name__":
    3.1. Create an empty list "values" to store input values.
    3.2. Loop 8 times:
        3.2.1. Read a value from the user and append it to the "values" list.
    3.3. Print "Values:" followed by the "values" list.
    3.4. Print "The optimal value is:" followed by the result of the "minimax" function called with
initial arguments:
        - depth = 0
        - nodeIndex = 0
        - maximizingPlayer = True (since it's the maximizing player's turn initially)
        - "values" list as input
        - alpha = MIN
        - beta = MAX
4. End of the program.
```

PROGRAM:

```
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 = []
    for i in range(8):
        value = int(input(f"Enter value {i+1}: "))
        values.append(value)
    print("Values:", values)
    print("The optimal value is:", minimax(0, 0, True, values, MIN, MAX))
```

OUTPUT:

```
Enter value 1: 2
Enter value 2: 5
Enter value 3: 6
Enter value 4: 8
Enter value 5: 7
Enter value 6: 1
Enter value 7: 9
Enter value 8: 4
Values: [2, 5, 6, 8, 7, 1, 9, 4]
The optimal value is: 7
```

RESULT: Thus the program for Alpha & Beta pruning is executed successfully.

15. Write the python program to implement Decision Tree.

AIM: To write a python program to implement Decision Tree.

ALGORITHM:

1. Import necessary modules:
   - Import "load_iris" from "sklearn.datasets"
   - Import "train_test_split" from "sklearn.model_selection"
   - Import "DecisionTreeClassifier" from "sklearn.tree"
   - Import "accuracy_score" from "sklearn.metrics"

2. Load the Iris dataset using "load_iris()" function and store the features in "X" and targets in "y".

3. Take input from the user for:

   - "test_size" (proportion of the dataset to include in the test split)

   - "random_state" (seed used by the random number generator)

4. Split the data into training and testing sets using "train_test_split()" function:

   - Input: "X" (features), "y" (targets), "test_size", and "random_state"

   - Outputs: "X_train" (features for training), "X_test" (features for testing),
     "y_train" (targets for training), "y_test" (targets for testing)

5. Create a Decision Tree classifier using "DecisionTreeClassifier()":

   - Create an instance of the DecisionTreeClassifier class.

6. Train the classifier on the training data using the "fit()" method:

   - Inputs: "X_train" (features for training), "y_train" (targets for training)

7. Make predictions on the test data using the "predict()" method:

   - Input: "X_test" (features for testing)

   - Output: "y_pred" (predicted targets)

8. Calculate the accuracy of the classifier using the "accuracy_score()" function:

   - Inputs: "y_test" (true targets), "y_pred" (predicted targets)

   - Output: "accuracy" (accuracy score)

9. Print the calculated accuracy.

10. End of the program.

PROGRAM:

```
from sklearn.datasets import load_iris

from sklearn.model_selection import train_test_split

from sklearn.tree import DecisionTreeClassifier

from sklearn.metrics import accuracy_score


# Load the Iris dataset

iris = load_iris()
```

```
X = iris.data
y = iris.target
# Take input from the user for test size and random state
test_size = float(input("Enter the test size (between 0 and 1): "))
random_state = int(input("Enter the random state: "))

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=test_size,
random_state=random_state)

# Create a Decision Tree classifier
clf = DecisionTreeClassifier()

# Train the classifier on the training data
clf.fit(X_train, y_train)

# Make predictions on the test data
y_pred = clf.predict(X_test)
# Calculate the accuracy of the classifier
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
```

OUTPUT:

```
Enter the test size (between 0 and 1): 0.5
Enter the random state: 1
Accuracy: 0.9733333333333334
```

RESULT: Thus the python program for Decision Tree is executed successfully.

16. Write the python program to implement Feed forward neural Network.

AIM: To write a python program to implement Feed forward neural Network.

ALGORITHM:

```
1. Import necessary modules:
   - Import "numpy" as "np" for numerical operations
   - Import "expit" from "scipy.special" for the sigmoid activation function
2. Define a class "NeuralNetwork":
   2.1. Define the initializer method "__init__(self, input_size, hidden_size, output_size)":
      2.1.1. Initialize instance variables "input_size", "hidden_size", and "output_size".
      2.1.2. Initialize weights and biases for input-hidden and hidden-output layers using random
values.
   2.2. Define the "sigmoid(self, x)" method:
      2.2.1. Use the "expit" function to compute the sigmoid activation of the input "x".
   2.3. Define the "forward(self, inputs)" method:
      2.3.1. Compute the activations of the hidden layer:
         2.3.1.1. Calculate the dot product of "inputs" and "weights_input_hidden".
         2.3.1.2. Add the bias_hidden.
         2.3.1.3. Apply the sigmoid activation function to the result.
      2.3.2. Compute the final output:
         2.3.2.1. Calculate the dot product of "hidden_output" and "weights_hidden_output".
         2.3.2.2. Add the bias_output.
         2.3.2.3. Apply the sigmoid activation function to the result.
      2.3.3. Return the "predicted_output".
3. Take user input for network parameters:
   - Input: "input_size", "hidden_size", "output_size"
4. Create a neural network instance using the defined network parameters.
5. Take user input for input data:
   - Input: "num_samples" (number of input samples)
   - Loop "num_samples" times:
      5.1. Take input for "input_data" as a list of floats.
```

6. Perform the forward pass for each "input_data" in "inputs":

   6.1. Calculate the "predicted_output" using the "forward" method of the neural network.

   6.2. Print the "input_data" and the corresponding "predicted_output".

7. End of the program.

PROGRAM:

```
import numpy as np
from scipy.special import expit
class NeuralNetwork:
    def __init__(self, input_size, hidden_size, output_size):
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size

        # Initialize weights and biases
        self.weights_input_hidden = np.random.randn(self.input_size, self.hidden_size)
        self.bias_hidden = np.zeros((1, self.hidden_size))

        self.weights_hidden_output = np.random.randn(self.hidden_size, self.output_size)
        self.bias_output = np.zeros((1, self.output_size))

    def sigmoid(self, x):
        return expit(x)

    def forward(self, inputs):
        # Calculate hidden layer activations
        hidden_input = np.dot(inputs, self.weights_input_hidden) + self.bias_hidden
        hidden_output = self.sigmoid(hidden_input)
        # Calculate final output
        output_input = np.dot(hidden_output, self.weights_hidden_output) + self.bias_output
        predicted_output = self.sigmoid(output_input)
        return predicted_output

# Take user input for network parameters
input_size = int(input("Enter input size: "))
hidden_size = int(input("Enter hidden size: "))
output_size = int(input("Enter output size: "))

# Create a neural network instance
nn = NeuralNetwork(input_size, hidden_size, output_size)

# Take user input for input data
num_samples = int(input("Enter the number of input samples: "))
inputs = []
for _ in range(num_samples):
    input_data = [float(x) for x in input("Enter input data (space-separated values): ").split()]
    inputs.append(input_data)

# Perform forward pass
for input_data in inputs:
    predicted_output = nn.forward(input_data)
    print(f"Input: {input_data}, Predicted Output: {predicted_output}")
```

OUTPUT:

```
Enter input size: 2
Enter hidden size: 3
Enter output size: 1
Enter the number of input samples: 2
Enter input data (space-separated values): 0.2 0.5
Enter input data (space-separated values): 0.8 0.3
Input: [0.2, 0.5], Predicted Output: [[0.30658089]]
Input: [0.8, 0.3], Predicted Output: [[0.32649034]]
```

RESULT: Thus the python program for Feed forward neural network is executed successfully.

17. Write a Prolog Program to Sum the Integers from 1 to n.

AIM: To write a Prolog Program to Sum the Integers from 1 to n.

ALGORITHM:

```
1. Define a predicate "sum(N, Result)" that calculates the sum of numbers from 1 to "N":
  1.1. Base Case:
      If "N" is 1:
         - Set "Result" as 1.
  1.2. Recursive Case:
      If "N" is greater than 1:
         - Calculate "N1" as "N - 1".
         - Recursively call "sum(N1, Subsum)" to compute the sum of numbers from 1 to "N1".
         - Calculate "Result" as "N + Subsum".
2. End of the program.
```

PROGRAM:

```
sum(1,1).
sum(N,Result):-
        N>1,
        N1 is N-1,
        sum(N1,Subsum),
        Result is N+Subsum.
```

OUTPUT:

```
% c:/Users/91934/OneDrive/Documents/Prolog/AI/17_Sum.pl compiled 0.00 sec, 3 clauses
```

RESULT: Thus the prolog program for sum of integers from 1 to N.

18. Write a Prolog Program for A DB WITH NAME, DOB.

AIM: To write a Prolog Program for A DB with NAME,DOB.

ALGORITHM:

```
1. Define facts using the "born" predicate to represent birth dates of individuals:
  1.1. Fact: born(murali, 23, 06, 2004)
     - The individual "murali" was born on the 23rd of June, 2004.
  1.2. Fact: born(mahesh, 09, 08, 1975)
     - The individual "mahesh" was born on the 9th of August, 1975.
  1.3. Fact: born(rohit, 30, 04, 1987)
     - The individual "rohit" was born on the 30th of April, 1987.
  1.4. Fact: born(ramu, 1, 9, 1994)
     - The individual "ramu" was born on the 1st of September, 1994.
  1.5. Fact: born(radha, 7, 8, 2003)
     - The individual "radha" was born on the 7th of August, 2003.
2. End of the program.
```

PROGRAM:

```
born(murali,23,06,2004).
born(mahesh,09,08,1975).
born(rohit,30,04,1987).
born(ramu,1,9,1994).
born(radha,7,8,2003).
```

OUTPUT:

```
% c:/Users/91934/OneDrive/Documents/Prolog/AI/18_DOB.pl compiled 0.00 sec, 6 clauses
```

RESULT: Thus the prolog program for NAME, DOB is executed successfully.

19. Write a Prolog Program for STUDENT-TEACHER-SUB-CODE.

AIM: To write a Prolog Program for STUDENT-TEACHER-SUB-CODE.

ALGORITHM:

```
1. Define facts using the "takes" predicate to represent students and their course codes:
  1.1. Fact: takes(murali_krishna, csa123)
     - The student "murali_krishna" takes the course "csa123".
  1.2. Fact: takes(mahesh_babu, uba027)
     - The student "mahesh_babu" takes the course "uba027".
  1.3. Fact: takes(murali_krishna, dsa456)
     - The student "murali_krishna" takes the course "dsa456".
  1.4. Fact: takes(rohit_sharma, his264)
     - The student "rohit_sharma" takes the course "his264".
2. Define a rule using the "classmates" predicate to determine if two students are classmates:
  2.1. Rule: classmates(X, Y) :-
     - For two students "X" and "Y" to be classmates:
     - Check if both students take the same course (Z).
     - The students are classmates if they take the same course.
3. End of the program.
```

PROGRAM:

```
takes(murali_krishna, csa123).
takes(mahesh_babu, uba027).
takes(murali_krishna, dsa456).

takes(rohit_sharma, his264).

classmates(X, Y):- takes(X, Z), takes(Y, Z).
```

OUTPUT:

```
% c:/Users/91934/OneDrive/Documents/Prolog/AI/19_Student_Teacher_Subcode.pl compiled
0.00 sec, 6 clauses
```

RESULT: Thus the prolog program for Student Teacher Subcode is executed successfully.

20. Write a Prolog Program for PLANETS DB.

AIM: To write a Prolog Program for PLANETS DB.

ALGORITHM:

1. Define facts using the "orbit" predicate to represent the orbital relationships between celestial bodies:

   1.1. Fact: orbit(moon, earth)

      - The celestial body "moon" orbits around the celestial body "earth".

   1.2. Fact: orbit(earth, sun)

      - The celestial body "earth" orbits around the celestial body "sun".

   1.3. Fact: orbit(mercury, sun)

      - The celestial body "mercury" orbits around the celestial body "sun".

   1.4. Fact: orbit(venus, sun)

      - The celestial body "venus" orbits around the celestial body "sun".

   1.5. Fact: orbit(mars, sun)

      - The celestial body "mars" orbits around the celestial body "sun".

2. End of the program.

PROGRAM:

```
orbit(moon,earth).

orbit(earth,sun).

orbit(mercury,sun).
orbit(venus,sun).
orbit(mars,sun).
```

OUTPUT:

```
% c:/Users/91934/OneDrive/Documents/Prolog/AI/20_Planets.pl compiled 0.00 sec, 6 clause
```

RESULT: Thus the prolog program for Planets is executed successfully.

21. Write a Prolog Program to implement Towers of Hanoi.

AIM: To write a Prolog Program to implement towers of Hanoi.

ALGORITHM:

```
1. Rule: move(1, X, Y, _):
   - If there is only one disk (1) to move from peg X to peg Y:
   - Print a message indicating the movement of the top disk from peg X to peg Y.
   - End the line.
2. Rule: move(N, X, Y, Z):
   - For moving N disks from peg X to peg Y using peg Z as an auxiliary peg:
   - If N is greater than 1:
      - Calculate M as N - 1 (decrement N by 1).
      - Move M disks from peg X to peg Z using peg Y as an auxiliary peg.
      - Move the top disk (1) from peg X to peg Y using no auxiliary peg.
      - Move M disks from peg Z to peg Y using peg X as an auxiliary peg.
3. End of the program.
```

PROGRAM:

```
move(1,X,Y,_) :-
        write('Move top disk from '),
        write(X),
        write(' to '),
        write(Y),
nl.
move(N,X,Y,Z) :-
        N>1,
        M is N-1,
        move(M,X,Z,Y),
        move(1,X,Y,_),
        move(M,Z,Y,X).
```

OUTPUT:

```
% c:/Users/91934/OneDrive/Documents/Prolog/AI/21_Towers_of_Hanoi.pl compiled 0.00 sec,
3 clauses
```

RESULT: Thus the prolog program for Towers of Hanoi is executed successfully.

22. Write a Prolog Program to print particular bird can fly or not. Incorporate required queries.

AIM: To write a Prolog Program to print particular bird can fly or not.

ALGORITHM:

```
1. Define facts using the "bird" predicate to represent different types of birds:
  1.1. Fact: bird(sparrow)
     - The type "sparrow" is a bird.
  1.2. Fact: bird(crow)
     - The type "crow" is a bird.
  1.3. Fact: bird(pegion)
     - The type "pegion" is a bird.
  1.4. Fact: bird(owl)
     - The type "owl" is a bird.
  1.5. Fact: bird(peacock)
     - The type "peacock" is a bird.
  1.6. Fact: bird(eagle)
```

- The type "eagle" is a bird.

1.7. Fact: bird(kingfisher)

- The type "kingfisher" is a bird.

2. End of the program.

PROGRAM:

```
bird(sparrow).
bird(crow).
bird(pegion).
bird(owl).
bird(peacock).
bird(eagle).
bird(kingfisher).
```

OUTPUT:

```
% c:/Users/91934/OneDrive/Documents/Prolog/AI/22_Bird.pl compiled 0.00 sec, 8 clauses
```

RESULT: Thus the Prolog Program for Bird can fly or not is executed successfully.

23. Write the prolog program to implement family tree.

AIM: To write a Prolog Program for implementing family tree.

ALGORITHM:

1. Define facts using the "female" and "male" predicates to represent individuals' genders:

1.1. Fact: female(sita)

- The individual "sita" is female.

1.2. Fact: female(radha)

- The individual "radha" is female.

1.3. Fact: male(ram)

- The individual "ram" is male.

1.4. Fact: male(krishna)

- The individual "krishna" is male.

2. Define facts using the "parent" predicate to represent parent-child relationships:

   2.1. Fact: parent(sita, ram)

   - The individual "sita" is the parent of "ram".

   2.2. Fact: parent(radha, krishna)

   - The individual "radha" is the parent of "krishna".

3. Define rules to derive familial relationships:

   3.1. Rule: mother(X, Y)

   - If individual X is a parent of individual Y and X is female, then X is the mother of Y.

   3.2. Rule: father(X, Y)

   - If individual X is a parent of individual Y and X is male, then X is the father of Y.

   3.3. Rule: sister(X, Y)

   - If there exists an individual Z such that Z is a parent of both X and Y, and X and Y are female and not the same individual, then X is the sister of Y.

   3.4. Rule: brother(X, Y)

   - If there exists an individual Z such that Z is a parent of both X and Y, and X and Y are male and not the same individual, then X is the brother of Y.

4. End of the program.

PROGRAM:

```
female(sita).

female(radha).

male(ram).

male(krishna).

parent(sita,ram).

parent(radha,krishna).

mother(X,Y):- parent(X,Y), female(X).

father(X,Y):- parent(X,Y), male(Y).

sister(X,Y):- parent(Z,X), parent(Z,Y), female(X),X\==Y.

brother(X,Y):- parent(Z,X), parent(Z,Y), male(X),X\==Y.
```

OUTPUT:

```
% c:/Users/91934/OneDrive/Documents/Prolog/AI/23_Family_Tree.pl compiled 0.00 sec, 11
clauses
```

RESULT: Thus the Prolog Program for Family Tree is executed successfully.

24. Write a Prolog Program to suggest Dieting System based on Disease.

AIM: To write a Prolog Program to suggest dieting system based on Disease.

ALGORITHM:

1. Define facts using the "symptom" predicate to represent symptoms associated with patients:

   1.1. Fact: symptom(murali, fever)

      - The patient "murali" has the symptom "fever".

   1.2. Fact: symptom(murali, cold)

      - The patient "murali" has the symptom "cold".

   1.3. Fact: symptom(murali, cough)

      - The patient "murali" has the symptom "cough".

   1.4. Fact: symptom(murali, difficulty_in_breathing)

      - The patient "murali" has the symptom "difficulty_in_breathing".

2. Define rules to make hypotheses about potential diagnoses:

   2.1. Rule: hypothesis(Patient, covid)

      - If the patient has symptoms of fever, cold, cough, and difficulty in breathing, then they might have COVID-19.

      - Print the message "Follow quarantine for 15 days".

   2.2. Rule: hypothesis(Patient, normal_fever)

      - If the patient has symptoms of fever, sneezing, headache, and body pains, then they might have a normal fever.

      - Print the message "Eat healthy food".

   2.3. Rule: hypothesis(Patient, dengue)

      - If the patient has symptoms of fever, cold, platelets reduction, and throat pain, then they might have dengue.

- Print the messages "Do the blood test immediately" and "Eat fruits and drink fruit juices frequently".

3. End of the program.

PROGRAM:

```
symptom(murali, fever).
symptom(murali, cold).
symptom(murali, cough).
symptom(murali, difficulty_in_breathing).

symptom(krishna, fever).
symptom(krishna, sneezing).
symptom(krishna, headache).
symptom(krishna, bodypains).

symptom(mohan, fever).
symptom(mohan, cold).
symptom(mohan, platelets_reduction).

hypothesis(Patient, covid):-
        symptom(Patient, fever),
        symptom(Patient, cold),
        symptom(Patient, cough),
        symptom(Patient, difficulty_in_breathing),
        write('Follow quarentine for 15 days').

hypothesis(Patient, normal_fever):-
        symptom(Patient, fever),
        symptom(Patient, sneezing),
        symptom(Patient, headache),
        symptom(Patient, bodypains),
        write('Eat healthy food').

hypothesis(Patient, dengue):-
        symptom(Patient, fever),
        symptom(Patient, cold),
        symptom(Patient, platelets_reduction),
        symptom(Patient, throat_pain),
        write('Do the blood test immediately'),
        write('Eat fruits and drink fruit juices frequently').
```

OUTPUT:

```
% c:/Users/91934/OneDrive/Documents/Prolog/AI/24_disease.pl compiled 0.00 sec, 15 clauses
```

RESULT: Thus the Prolog program for Diet based on disease is executed successfully.

25. Write a Prolog program to implement Monkey Banana Problem.

AIM: To write a prolog program to implement Monkey Banana Problem.

ALGORITHM:

```
1. Define facts to represent the initial state of the scenario:
  1.1. Fact: on(floor, monkey)
     - The monkey is initially on the floor.
  1.2. Fact: on(floor, chair)
     - The chair is initially on the floor.
  1.3. Fact: in(room, monkey)
     - The monkey is initially in the room.
  1.4. Fact: in(room, chair)
     - The chair is initially in the room.
  1.5. Fact: in(room, banana)
```

- The banana is initially in the room.

1.6. Fact: at(ceiling, banana)

- The banana is initially at the ceiling.

1.7. Fact: strong(monkey)

- The monkey is initially strong.

1.8. Fact: grasp(monkey)

- The monkey initially has the ability to grasp objects.

2. Define rules for actions and conditions:

2.1. Rule: climb(monkey, chair)

- The monkey can climb the chair.

2.2. Rule: push(monkey, chair)

- The monkey can push the chair if it is strong.

2.3. Rule: under(banana, chair)

- The banana is under the chair if the monkey pushes the chair.

3. Define rules to determine if the monkey can reach and get the banana:

3.1. Rule: canreach(banana, monkey)

- The banana can be reached by the monkey if it is on the floor, in the room, at the ceiling, under the chair (pushed by the monkey), and the monkey climbs the chair.

3.2. Rule: canget(banana, monkey)

- The monkey can get the banana if it can reach the banana and has the ability to grasp objects.

4. End of the program.

PROGRAM:

```
on(floor,monkey).

on(floor,chair).

in(room,monkey).

in(room,chair).

in(room,banana).

at(cieling,banana).
```

```
strong(monkey).

grasp(monkey).

climb(monkey,chair).

push(monkey,chair):- strong(monkey).

under(banana,chair):- push(monkey,chair).


canreach(banana,monkey):-

        on(floor,banana),

        in(room,banana),

        at(cieling,banana),

        under(banana,chair),

        climb(banana,chair).


canget(banana,monkey):-

        canreach(banana,monkey),grasp(monkey).
```

OUTPUT:

```
% c:/Users/91934/OneDrive/Documents/Prolog/AI/25_Monkey_Banana.pl compiled 0.00 sec,
14 clause
```

RESULT: Thus the prolog program for Monkey Banana Problem is executed successfully.

26. Write a Prolog Program for fruit and its color using Back Tracking.

AIM: To write a Prolog Program for fruit and its color using Back Tracking.

ALGORITHM:

```
1. Define facts to represent fruit colors:
  1.1. Fact: fruit(banana, yellow)
     - The fruit "banana" is yellow in color.
  1.2. Fact: fruit(apple, red)
     - The fruit "apple" is red in color.
```

1.3. Fact: fruit(orange, orange)

- The fruit "orange" is orange in color.

1.4. Fact: fruit(cherry, red)

- The fruit "cherry" is red in color.

1.5. Fact: fruit(grape, black)

- The fruit "grape" is black in color.

1.6. Fact: fruit(mango, yellow)

- The fruit "mango" is yellow in color.

2. End of the program.

PROGRAM:

```
fruit(banana,yellow).

fruit(apple,red).

fruit(orange,orange).

fruit(cherry,red).

fruit(grape,black).

fruit(mango,yellow).
```

OUTPUT:

```
% c:/Users/91934/OneDrive/Documents/Prolog/AI/26_Fruit_and_color.pl compiled 0.00 sec, 7
clauses
```

RESULT: Thus the Prolog Program for Fruit and color is executed successfully.

27. Write a Prolog Program to implement Best First Search algorithm.

AIM: To write a prolog program to implement BFS.

ALGORITHM:

1. Define Distance Between Towns:
    - Use the "arc" predicate to define distances between different towns.
2. Heuristic Function for Goal State:
    - Use the "hdist" predicate to define heuristic values for different towns.

- The "h" predicate checks if a heuristic value exists for a town.
- If no heuristic value exists, a default value of 1 is assigned.

3. Best-First Search Algorithm:
    - Define the "best_first" predicate that takes initial state ([[Start]]) and the goal state (Goal) as input.
    - Base case: If the current path has the goal state, return the final path.
    - Recursive case: Extend the current path to new possible paths.
        - Use the "extend" predicate to generate new paths by adding new nodes to the path.
    - Append new paths to the queue and sort the queue based on the heuristic value.
        - Use the "sort_queue1" predicate to sort the queue based on heuristic values.
    - Recurse with the updated queue and goal state to find the optimal path.
    - Calculate the number of explored nodes (N) and return it.
4. Path Extension:
    - Define the "extend" predicate that takes a path and generates new paths by adding new nodes.
    - Find all new nodes connected to the last node in the path using the "arc" predicate.
    - Ensure that the new node is not already in the path to avoid loops.
5. Queue Sorting:
    - Define the "sort_queue1" predicate to sort the queue based on heuristic values.
    - Swap nodes in the queue if the heuristic value of the first node is greater than the second.
6. Heuristic Evaluation:
    - Define the "hh" predicate to check if the heuristic function is correct.
    - Calculate the heuristic value for a state using the "h" predicate.
    - Ensure that the heuristic value is a number, otherwise print an error message and abort.
7. Display Queue Length:
    - Define the "wrq" predicate to display the length of the queue

PROGRAM:

```
arc(arad,zerind,75).
arc(arad,sibiu,140).
arc(arad,timisoara,118).
arc(bucharest,fagaras,211).
arc(bucharest,pitesti,101).
arc(bucharest,giurgiu,90).
arc(bucharest,urziceni,85).
arc(craiova,dobreta,120).
arc(craiova,rimnicu,146).
arc(craiova,pitesti,138).
arc(dobreta,mehadia,75).
arc(dobreta,craiova,120).
arc(eforie,hirsova,86).
arc(fagaras,sibiu,99).
arc(fagaras,bucharest,211).
arc(giurgiu,bucharest,90).
arc(hirsova,urziceni,98).
arc(hirsova,eforie,86).
arc(iasi,neamt,87).
arc(iasi,vaslui,92).
arc(lugoj,timisoara,111).
arc(lugoj,mehadia,70).
arc(mehadia,lugoj,70).
arc(mehadia,dobreta,75).
arc(neamt,iasi,87).
arc(oradea,zerind,71).
arc(oradea,sibiu,151).
arc(pitesti,rimnicu,97).
arc(pitesti,craiova,138).
arc(pitesti,bucharest,101).
arc(rimnicu,sibiu,80).
arc(rimnicu,pitesti,97).
arc(rimnicu,craiova,146).
arc(sibiu,arad,140).
arc(sibiu,oradea,151).
arc(sibiu,fagaras,99).
arc(sibiu,rimnicu,80).
arc(timisoara,arad,118).
arc(timisoara,lugoj,111).
arc(urziceni,bucharest,85).
arc(urziceni,hirsova,98).
arc(urziceni,vaslui,142).
arc(vaslui,iasi,92).
arc(vaslui,urziceni,142).
arc(zerind,arad,75).
arc(zerind,oradea,71).

% heuristic function: stright line distance to bucharest
% (works only if the goal state is bucharest)

h(Node, Value) :-
   hdist(Node, Value).
h(_,1). % A default value
```

```
hdist(arad     ,366).
hdist(bucharest,  0).
hdist(craiova  ,160).
hdist(dobreta  ,242).
hdist(eforie   ,161).
hdist(fagaras  ,178).
hdist(giurgiu  , 77).
hdist(hirsova  ,151).
hdist(iasi     ,266).
hdist(lugoj    ,244).
hdist(mehadia  ,241).
hdist(neamt    ,234).
hdist(oradea   ,380).
hdist(pitesti  , 98).
hdist(rimnicu  ,193).
hdist(sibiu    ,253).
hdist(timisoara,329).
hdist(urziceni , 80).
hdist(vaslui   ,199).
hdist(zerind   ,374).
best_first([[Goal|Path]|_],Goal,[Goal|Path],0).
best_first([Path|Queue],Goal,FinalPath,N) :-
   extend(Path,NewPaths),
   append(Queue,NewPaths,Queue1),
   sort_queue1(Queue1,NewQueue), wrq(NewQueue),
   best_first(NewQueue,Goal,FinalPath,M),
   N is M+1.
```

```
extend([Node|Path],NewPaths) :-
    findall([NewNode,Node|Path],
            (arc(Node,NewNode,_),
            \+ member(NewNode,Path)), % for avoiding loops
            NewPaths).

sort_queue1(L,L2) :-
    swap1(L,L1), !,
    sort_queue1(L1,L2).
sort_queue1(L,L).

swap1([[A1|B1],[A2|B2]|T],[[A2|B2],[A1|B1]|T]) :-
    hh(A1,W1),
    hh(A2,W2),
    W1>W2.
swap1([X|T],[X|V]) :-
    swap1(T,V).
hh(State, Value) :-
    h(State,Value),
    number(Value), !.
hh(State, Value) :-
   write('Incorrect heuristic functionh: '),
   write(h(State, Value)), nl,
   abort.
wrq(Q) :- length(Q,N), writeln(N).
```

OUTPUT:

```
% c:/Users/91934/OneDrive/Documents/Prolog/AI/27_BFS.pl compiled 0.00 sec, 79 clauses
```

RESULT: Thus the Prolog Program for BFS is executed successfully.

28. Write the prolog program for Medical Diagnosis.

AIM: To write a prolog program for Medical Diagnosis.

ALGORITHM:

1. Define symptoms and corresponding treatments for different diseases.

   - Use the "symptom" and "treatment" predicates to store this information.

2. Define the "input" predicate for gathering patient's symptoms and storing their responses.

   - Parameters: X (symptom)

   - Ask the user whether the patient has the symptom X.

   - Read the response and assert the symptom-patient relationship using the "patient" predicate.

   - Repeat this process until "Stomach Pain" symptom is reached.

   - Check for "output" predicate after each response.

3. Define the "disease" predicates to infer diseases based on symptoms.

   - Each disease predicate corresponds to a specific set of symptoms.

   - Use the "not" predicate to ensure that the same symptoms are not part of other diseases.

4. Define the "possible_diseases" predicate to display possible diseases the patient may suffer from.

   - Iterate through disease predicates and print the possible diseases.

5. Define the "advice" predicate to display treatment advice based on patient's symptoms.

   - Iterate through symptoms.

   - If the patient has the symptom, retrieve the corresponding treatment using the "treatment" predicate.

   - Print the treatment advice, unless the symptom is "Stomach Pain".

6. Define the "output" predicate that is used to initiate the display of possible diseases and advice.

   - Call "possible_diseases" and "advice" predicates.

PROGRAM:

```
symptom('Flu').

symptom('Yellowish eyes and skin').
symptom('Dark color urine').

symptom('Pale bowel movement').

symptom('Fatigue').

symptom('Vomitting').

symptom('Fever').

symptom('Pain in joints').

symptom('Weakness').

symptom('Stomach Pain').


treatment('Flu', 'Drink hot water, avoid cold eatables.').

treatment('Yellowish eyes and skin', 'Put eye drops, have healthy sleep, do not strain your eyes.').

treatment('Dark color urine', 'Drink lots of water, juices and eat fruits. Avoid alcohol
consumption.').

treatment('Pale bowel movement', 'Drink lots of water and exercise regularly.').

treatment('Fatigue', 'Drink lots of water, juices and eat fruits.').

treatment('Vomitting', 'Drink salt and water.').

treatment('Fever', 'Put hot water cloth on head and take crocin.').

treatment('Pain in Joints', 'Apply pain killer and take crocin.').

treatment('Weakness', 'Drink salt and water, eat fruits.').

treatment('Stomach Pain', 'Avoid outside food and eat fruits.').


input :- dynamic(patient/2),

   repeat,

   symptom(X),

   write('Does the patient have '),

   write(X),

   write('? '),

   read(Y),

   assert(patient(X,Y)),
   \+ not(X='Stomach Pain'),
   not(output).

disease(hemochromatosis) :-
   patient('Stomach Pain',yes),
   patient('Pain in joints',yes),
   patient('Weakness',yes),
   patient('Dark color urine',yes),
   patient('Yellowish eyes and skin',yes).

disease(hepatitis_c) :-
   not(disease(hemochromatosis)),
   patient('Pain in joints',yes),
   patient('Fever',yes),
   patient('Fatigue',yes),
   patient('Vomitting',yes),
   patient('Pale bowel movement',yes).

disease(hepatitis_b) :-
   not(disease(hemochromatosis)),
   not(disease(hepatitis_c)),
   patient('Pale bowel movement',yes),
   patient('Dark color urine',yes),
   patient('Yellowish eyes and skin',yes).

disease(hepatitis_a) :-
   not(disease(hemochromatosis)),
   not(disease(hepatitis_c)),
   not(disease(hepatitis_b)),
   patient('Flu',yes),
   patient('Yellowish eyes and skin',yes).

disease(jaundice) :-
   not(disease(hemochromatosis)),
   not(disease(hepatitis_c)),
   not(disease(hepatitis_b)),
   not(disease(hepatitis_a)),
   patient('Yellowish eyes and skin',yes).

disease(flu) :-
   not(disease(hemochromatosis)),
   not(disease(hepatitis_c)),
   not(disease(hepatitis_b)),
   not(disease(hepatitis_a)),
   patient('Flu',yes).

output:-
   nl,
   possible_diseases,
   nl,
   advice.

possible_diseases :- disease(X), write('The patient may suffer from '), write(X),nl.
advice :- symptom(X), patient(X,yes), treatment(X,Y), write(Y),nl, \+ not(X='Stomach Pain').
```

OUTPUT:

```
% c:/Users/91934/OneDrive/Documents/Prolog/AI/28_Medical_Diagnosis.pl compiled 0.00 sec,
31 clauses
```

RESULT: Thus the Prolog Program for Medical Diagnosis is executed successfully.

29. Write a Prolog Program for forward Chaining. Incorporate required queries.

AIM: To write a Prolog Program for forward chaining.

ALGORITHM:

1. Define Weather Conditions:
    - Use the predicates "rain," "sunny," and "cold" to define different weather conditions for cities.
2. Snowy Weather Check:
    - Define the "snow" predicate that checks if a city has snowy weather.
    - A city has snowy weather if it satisfies the conditions of raining, being sunny, and being cold.
3. Iterate Through Cities:
    - To determine if a city has snowy weather:
        - Check if the city has rain using the "rain" predicate.
        - Check if the city is sunny using the "sunny" predicate.
        - Check if the city is cold using the "cold" predicate.
4. Display Snowy Cities:
    - If a city satisfies all the conditions for snow, print that the city has snowy weather.

PROGRAM:

```
rain(chennai).
rain(tirupathi).
sunny(vijayawada).
sunny(hyderabad).
cold(ooty).
cold(araku).
snow(X):- rain(X),sunny(X),cold(X).
```

OUTPUT:

```
% c:/Users/91934/OneDrive/Documents/Prolog/AI/29_Forward_Chaining.pl compiled 0.00 sec,
8 clauses
```

RESULT: Thus the prolog program for forward chaining is executed successfully.

30. Write a Prolog Program for backward Chaining. Incorporate required queries.

AIM: To write a prolog program for backward chaining.

ALGORITHM:

1. Define Ancestors:
    - Define the "ancestor" predicate to check if one person is an ancestor of another person.
    - The base case is when the direct parent relationship exists between two individuals. In this case, "parent(X, Y)" implies "ancestor(X, Y)."
2. Recursive Ancestor Search:
    - For a given pair of individuals (X, Y):
        - Check if there is a direct parent-child relationship between X and Y using the "parent(X, Y)" predicate.
        - If there's no direct parent-child relationship, recursively search for an intermediary ancestor Z:
            - Use the "parent(X, Z)" predicate to find a Z such that X is the parent of Z.
            - Check if Z is an ancestor of Y by calling the "ancestor(Z, Y)" predicate recursively.
3. Display Ancestors:
    - Display the ancestors for a given pair (X, Y) by finding all the individuals Z for which "ancestor(X, Z)" holds true.

PROGRAM:

```
ancestor(X,Y):- parent(X,Y).

ancestor(X,Y):- parent(X,Z), ancestor(Z,Y).

parent(sai,ram).
parent(ram,krishna).

parent(ram,devi).
```

OUTPUT:

```
% c:/Users/91934/OneDrive/Documents/Prolog/AI/30_Backward_Chaining.pl compiled 0.00
sec, 4 clauses
```

RESULT: Thus the prolog program for Backward chaining is executed successfully.

31. Create a Web Blog using Word press to demonstrate Anchor Tag, Title Tag, etc.

AIM: To create a Web Blog using Word Press to demonstrate Anchor Tag, Title Tag.

ALGORITHM:

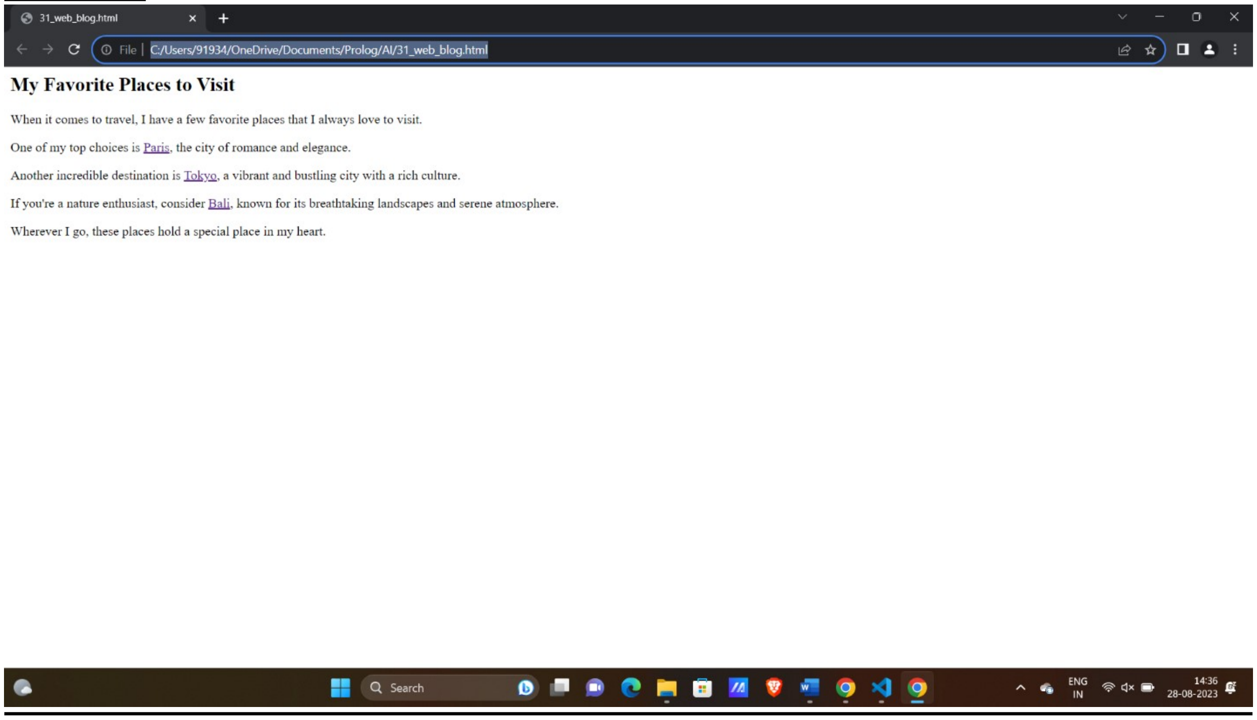
1. Output Headings and Paragraphs:
    - Print the heading <h2>My Favorite Places to Visit</h2>.
    - Print the paragraph <p>When it comes to travel, I have a few favorite places that I always love to visit.</p>.
2. Output Place Descriptions:
    - Print the paragraph <p>One of my top choices is <a href="https://www.visitparisregion.com/en" title="Visit Paris">Paris</a>, the city of romance and elegance.</p>.
    - Print the paragraph <p>Another incredible destination is <a href="https://www.gotokyo.org/en/" title="Explore Tokyo">Tokyo</a>, a vibrant and bustling city with a rich culture.</p>.
    - Print the paragraph <p>If you're a nature enthusiast, consider <a href="https://bali.com/" title="Experience Bali">Bali</a>, known for its breathtaking landscapes and serene atmosphere.</p>.
3. Output Conclusion:
    - Print the paragraph <p>Wherever I go, these places hold a special place in my heart.</p>.

PROGRAM:

```
<h2>My Favorite Places to Visit</h2>
```

```
<p>When it comes to travel, I have a few favorite places that I always love to visit.</p>

<p>One of my top choices is <a href="https://www.visitparisregion.com/en" title="Visit
Paris">Paris</a>, the city of romance and elegance.</p>

<p>Another incredible destination is <a href="https://www.gotokyo.org/en/" title="Explore
Tokyo">Tokyo</a>, a vibrant and bustling city with a rich culture.</p>

<p>If you're a nature enthusiast, consider <a href="https://bali.com/" title="Experience
Bali">Bali</a>, known for its breathtaking landscapes and serene atmosphere.</p>

<p>Wherever I go, these places hold a special place in my heart.</p>
```

OUTPUT:



31_web_blog.html

C:/Users/91934/OneDrive/Documents/Prolog/AI/31_web_blog.html

My Favorite Places to Visit

When it comes to travel, I have a few favorite places that I always love to visit.

One of my top choices is Paris, the city of romance and elegance.

Another incredible destination is Tokyo, a vibrant and bustling city with a rich culture.

If you're a nature enthusiast, consider Bali, known for its breathtaking landscapes and serene atmosphere.

Wherever I go, these places hold a special place in my heart.

RESULT: Thus we have created a blog using html which includes anchor tags and title tags.