SRI KRISHNA COLLEGE OF ENGINEERING AND TECHNOLOGY
COIMBATORE – 641008
(An AUTONOMOUS INSTITUTION,
AFFILIATED TO ANNA UNIVERSITY, CHENNAI)

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

21CS605 - ARTIFICIAL INTELLIGENCE LABORATORY

CONTINUOUS ASSESSMENT RECORD

Submitted by

Name: ……….…………………..…………….

Register No. : ……….…………………..……...

Degree & Branch: B.E. Computer Science and Engineering

Class: III B.E. CSE ‘C’

SRI KRISHNA COLLEGE OF ENGINEERING AND TECHNOLOGY
COIMBATORE – 641008
(An AUTONOMOUS INSTITUTION, AFFILIATED TO ANNA UNIVERSITY, CHENNAI)

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

21CS605 - ARTIFICIAL INTELLIGENCE LABORATORY

Continuous Assessment Record
Submitted by

Name : ……..…….……………………………..………… **Register No. : ....……………………**

Class : III B.E CSE 'C' **Degree & Branch: B.E CSE**

BONAFIDE CERTIFICATE

This is to certify that this record is the bonafide record of work done by Mr./Ms. _________________ during the academic year 2023 – 2024.

Faculty-In Charge **HOD**

Submitted for End Semester practical Examination, held on……………………….

INTERNAL EXAMINER **EXTERNAL EXAMINER**

SRI KRISHNA COLLEGE OF ENGINEERING AND TECHNOLOGY
KUNIAMUTHUR, COIMBATORE - 641 008

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

EVEN SEMESTER 2023-2024

Name of Lab Course	**21CS605 - ARTIFICIAL INTELLIGENCE LABORATORY**
Semester & Year	
Name of the student	
Class	
Name of the Evaluator	
Marks scored out of 100	

RUBRIC ASSESSMENT FOR ARTIFICIAL INTELLIGENCE LABORATORY

RUBRICS - RECORD

Criteria	Range of Marks			
	Excellent	Good	Average	Below Average
Observation (80)	71-80	61-70	41-60	0-40
Record- Timing and Presentation (10)	9-10	7-8	5-6	0-4
Viva (10)	9-10	7-8	5-6	0-4

Components	EXP 1	EXP 2	EXP 3	EXP 4	EXP 5	EXP 6	EXP 7	EXP 8	EXP 9	EXP 10	EXP 11	EXP 12
Observation (80)												
Record-Timing and Presentation (10)												
Viva (10)												
TOTAL												
AVERAGE												

INDEX

EXP :	IMPLEMENTATION OF N-QUEEN ALGORITHM
DATE :	

AIM

To implement the n-queen problem using Python.

ALGORITHM

1. Initialize an empty NxN chessboard.

2. Start placing queens, beginning with the first row (row 0).

3. Call the recursive function `placeQueens(board, 0, N)` to place queens recursively starting from row 0.

4. Inside the `placeQueens` function:

   a. Base case: If `row` is equal to `N`, return true, indicating all queens are successfully placed.

   b. Loop through each column (`col`) in the current row (`row`).

   c. Check if placing a queen at position (`row`, `col`) is safe by calling the `isSafe` function.

   d. If safe, place the queen at (`row`, `col`) and recursively call `placeQueens` for the next row (`row + 1`).

   e. If placing the queen leads to a successful solution (recursive call returns true), return true.

   f. If no successful solution is found in the current configuration, backtrack by removing the queen from (`row`, `col`) and try the next column.

5. If `placeQueens` returns true, indicating a solution is found, return the solved board. Otherwise, return "No solution exists".

6. In the `isSafe` function, check if there is no queen in the same column (`col`).

7. Check the upper-left diagonal to ensure no queen is attacking from that direction.

8. Check the upper-right diagonal to ensure no queen is attacking from that direction.

9.If no conflicts are found, return true, indicating it's safe to place a queen at position (`row`, `col`).

PROGRAM

```
def is_safe(board, row, col, n):
    for i in range(col):
        if board[row][i] == 1:
```

```
            return False

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

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

    return True

def solve_n_queens_util(board, col, n):
    if col >= n:
        return True

    for i in range(n):
        if is_safe(board, i, col, n):
            board[i][col] = 1

            if solve_n_queens_util(board, col + 1, n):
                return True

            board[i][col] = 0

    return False

def solve_n_queens(n):
    board = [[0 for _ in range(n)] for _ in range(n)]

    if not solve_n_queens_util(board, 0, n):
        print("Solution does not exist")
```

```
        return False

    print("Solution for", n, "queens:")
    for i in range(n):
        for j in range(n):
            print(board[i][j], end=" ")
        print()
    return True

if __name__ == "__main__":
    while True:
        try:
            size = int(input("Enter the size of the chessboard (N): "))
            if size <= 0:
                print("Please enter a positive integer greater than 0.")
            else:
                break
        except ValueError:
            print("Invalid input! Please enter an integer.")

    solve_n_queens(size)
```

OUTPUT

```
Enter the size of the chessboard (N): 8
Solution for 8 queens:
1 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 1
0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 0 1 0 0
0 0 1 0 0 0 0 0
>
```

RESULTS

Thus the Python program to implement n-queen problem has been successfully executed.

EXP :	IMPLEMENTATION OF UNINFORMED FORMED SEARCH BREADTH FIRST SEARCH
DATE :	

AIM

To implement breadth first search using Python.

ALGORITHM

1. Create a graph data structure to store the vertices and edges.
2. Prompt the user to input the number of edges (connections between vertices).
3. Iterate through the input number of edges: a. Prompt the user to input an edge as a pair of vertices (u, v). b. Add the edge (u, v) to the graph.
4. Prompt the user to input the starting vertex for BFS traversal.
5. Initialize a queue and a visited array.
6. Enqueue the starting vertex into the queue and mark it as visited.
7. While the queue is not empty: a. Dequeue a vertex from the queue. b. Print the dequeued vertex. c. Iterate through all adjacent vertices of the dequeued vertex: i. If an adjacent vertex is not visited, enqueue it into the queue and mark it as visited.
8. Repeat step 7 until the queue becomes empty.
9. End of Algorithm.

PROGRAM

```
from collections import defaultdict
class Graph:
    def __init__(self):
        self.graph = defaultdict(list)

    def add_edge(self, u, v):
        self.graph[u].append(v)

    def bfs(self, start):
        visited = [False] * len(self.graph)
        queue = []
```

```
        queue.append(start)
        visited[start] = True

        while queue:
            start = queue.pop(0)
            print(start, end=" ")

            for i in self.graph[start]:
                if not visited[i]:
                    queue.append(i)
                    visited[i] = True

if __name__ == "__main__":
    g = Graph()
    while True:
        try:
            n = int(input("Enter the number of edges: "))
            break
        except ValueError:
            print("Invalid input! Please enter an integer.")

    for _ in range(n):
        while True:
            try:
                u, v = map(int, input("Enter an edge (u v): ").split())
                break
            except ValueError:
                print("Invalid input! Please enter two integers separated by space.")
        g.add_edge(u, v)

    start_vertex = int(input("Enter the start vertex: "))
    print("Breadth First Traversal starting from vertex", start_vertex)
```

```
g.bfs(start_vertex)
```

OUTPUT

```
Enter the number of edges: 10
Enter an edge (node1 node2): A B
Enter an edge (node1 node2): A S
Enter an edge (node1 node2): S G
Enter an edge (node1 node2): S C
Enter an edge (node1 node2): C F
Enter an edge (node1 node2): G F
Enter an edge (node1 node2): C D
Enter an edge (node1 node2): C E
Enter an edge (node1 node2): E H
Enter an edge (node1 node2): G H
Enter the start node: A
Enter the goal node: H
Expand Node | Fringe
--------------------
-           | A
A           | B, S
S           | B, G, C
C           | B, G, D, F, E
E           | B, G, D, F, H
H           |
Path: A => S => C => E => H
>
```

RESULTS

Thus the Python program to implement breath-first search has been successfully executed.

EXP :	IMPLEMENTATION OF UNINFORMED FORMED SEARCH - DEPTH FIRST SEARCH
DATE :	

AIM

To implement depth first search using Python.

ALGORITHM

1. Create a graph data structure to store the vertices and edges.
2. Prompt the user to input the number of edges (connections between vertices).
3. Iterate through the input number of edges: a. Prompt the user to input an edge as a pair of vertices (u, v). b. Add the edge (u, v) to the graph.
4. Prompt the user to input the starting vertex for DFS traversal.
5. Initialize a stack and a visited array.
6. Push the starting vertex onto the stack and mark it as visited.
7. While the stack is not empty:

   a. Pop a vertex from the stack.

   b. Print the popped vertex.

   c. Iterate through all adjacent vertices of the popped vertex: i. If an adjacent vertex is not visited, push it onto the stack and mark it as visited.
8. Repeat step 7 until the stack becomes empty.
9. End of Algorithm.

PROGRAM

```
from collections import deque

class Graph:

    def __init__(self, directed=True):

        self.edges = {}

        self.directed = directed


    def add_edge(self, node1, node2, __reversed=False):

        try:

            neighbors = self.edges[node1]

        except KeyError:
```

```
        neighbors = set()
    neighbors.add(node2)
    self.edges[node1] = neighbors
    if not self.directed and not __reversed:
        self.add_edge(node2, node1, True)

def neighbors(self, node):
    try:
        return self.edges[node]
    except KeyError:
        return []

def breadth_first_search(self, start, goal):
    found, fringe, visited, came_from = False, deque([start]), set([start]), {start: None}
    print('{:11s} | {}'.format('Expand Node', 'Fringe'))
    print('--------------------')
    print('{:11s} | {}'.format('-', start))
    while not found and len(fringe):
        current = fringe.pop()
        print('{:11s}'.format(current), end=' | ')
        if current == goal:
            found = True
            break
        for node in self.neighbors(current):
            if node not in visited:
                visited.add(node)
                fringe.append(node)
                came_from[node] = current
        print(', '.join(fringe))
    if found:
        print()
        return came_from
```

```
        else:
            print('No path from {} to {}'.format(start, goal))

    @staticmethod
    def print_path(came_from, goal):
        parent = came_from[goal]
        if parent:
            Graph.print_path(came_from, parent)
        else:
            print(goal, end='')
            return
        print(' =>', goal, end='')

    def __str__(self):
        return str(self.edges)
graph = Graph(directed=False)
while True:
    try:
        num_edges = int(input("Enter the number of edges: "))
        break
    except ValueError:
        print("Invalid input! Please enter an integer.")

for _ in range(num_edges):
    while True:
        try:
            node1, node2 = input("Enter an edge (node1 node2): ").split()
            break
        except ValueError:
            print("Invalid input! Please enter two nodes separated by space.")
    graph.add_edge(node1, node2)
start = input("Enter the start node: ")
```

```
goal = input("Enter the goal node: ")
traced_path = graph.breadth_first_search(start, goal)
if traced_path:
    print('Path:', end=' ')
    Graph.print_path(traced_path, goal)
    print()
```

OUTPUT

```
Enter the number of edges: 4
Enter an edge (node1 node2): A B
Enter an edge (node1 node2): A C
Enter an edge (node1 node2): B D
Enter an edge (node1 node2): C D
Enter the start node: A
Enter the goal node: D
Expand Node | Fringe
--------------------
-           | A
A           | B, C
C           | B, D
D           |
Path: A => C => D
>
```

RESULTS

Thus the Python program to implement depth-first search has been successfully executed.

EXP :	IMPLEMENTATION OF UNINFORMED FORMED SEARCH - UNIFORM COST SEARCH
DATE :	

AIM

To implement uniform cost search using Python.

ALGORITHM

1. Initialize: Create a graph data structure to store the vertices and edges.
2. Input Edges and Costs: Prompt the user to input the number of edges. Then, for each edge, prompt the user to input the source node, destination node, and the cost of the edge.
3. Input Start and Goal Nodes: Prompt the user to input the start node and the goal node for the search.
4. Uniform Cost Search: Perform Uniform Cost Search algorithm to find the shortest path from the start node to the goal node.
5. Print Expansion Table: Print the expansion table showing the expanded nodes and the fringe at each step.
6. Print Path and Cost: If a path is found, print the path from the start node to the goal node along with the total cost. If no path is found, print that there is no path.
7. End.

PROGRAM

```
from queue import heappop, heappush

from math import inf


class Graph:

    def __init__(self, directed=True):

        self.edges = {}

        self.directed = directed


    def add_edge(self, node1, node2, cost = 1, __reversed=False):

        try: neighbors = self.edges[node1]

        except KeyError: neighbors = {}

        neighbors[node2] = cost

        self.edges[node1] = neighbors
```

```
        if not self.directed and not __reversed: self.add_edge(node2, node1, cost, True)

    def neighbors(self, node):
        try: return self.edges[node]
        except KeyError: return []

    def cost(self, node1, node2):
        try: return self.edges[node1][node2]
        except: return inf

    def uniform_cost_search(self, start, goal):
        found, fringe, visited, came_from, cost_so_far = False, [(0, start)], set([start]), {start: None},
{start: 0}
        print('{:11s} | {}'.format('Expand Node', 'Fringe'))
        print('--------------------')
        print('{:11s} | {}'.format('-', str((0, start))))
        while not found and len(fringe):
            _, current = heappop(fringe)
            print('{:11s}'.format(current), end=' | ')
            if current == goal: found = True; break
            for node in self.neighbors(current):
                new_cost = cost_so_far[current] + self.cost(current, node)
                if node not in visited or cost_so_far[node] > new_cost:
                    visited.add(node); came_from[node] = current; cost_so_far[node] = new_cost
                    heappush(fringe, (new_cost, node))
            print(', '.join([str(n) for n in fringe]))
        if found: print(); return came_from, cost_so_far[goal]
        else: print('No path from {} to {}'.format(start, goal)); return None, inf

    @staticmethod
    def print_path(came_from, goal):
        parent = came_from[goal]
        if parent:
```

```
            Graph.print_path(came_from, parent)
        else: print(goal, end='');return
        print(' =>', goal, end='')

    def __str__(self):
        return str(self.edges)



graph = Graph(directed=True)


num_edges = int(input("Enter the number of edges: "))
for _ in range(num_edges):
    while True:
        try:
            node1, node2, cost = input("Enter an edge and its cost (node1 node2 cost): ").split()
            cost = int(cost)
            break
        except ValueError:
            print("Invalid input! Please enter three values separated by space (node1 node2 cost).")
    graph.add_edge(node1, node2, cost)

start = input("Enter the start node: ")
goal = input("Enter the goal node: ")

traced_path, cost = graph.uniform_cost_search(start, goal)
if traced_path:
    print('Path:', end=' ')
    Graph.print_path(traced_path, goal)
    print('\nCost:', cost)
```

OUTPUT

```
Enter the number of edges: 6
Enter an edge and its cost (node1 node2 cost): A B 4
Enter an edge and its cost (node1 node2 cost): A C 1
Enter an edge and its cost (node1 node2 cost): B D 1
Enter an edge and its cost (node1 node2 cost): B E 8
Enter an edge and its cost (node1 node2 cost): C C 0
Enter an edge and its cost (node1 node2 cost): C D 7
Enter the start node: A
Enter the goal node: E
Expand Node | Fringe
--------------------
-           | (0, 'A')
A           | (1, 'C'), (4, 'B')
C           | (4, 'B'), (8, 'D')
B           | (5, 'D'), (8, 'D'), (12, 'E')
D           | (8, 'D'), (12, 'E')
D           | (12, 'E')
E           |
Path: A => B => E
Cost: 12
>
```

RESULTS

Thus the Python program to implement uniform cost search has been successfully executed.

EXP :	IMPLEMENTATION OF UNIFORMED SEARCH - DEPTH LIMITED SEARCH
DATE :	

AIM

To implement depth limited search using Python.

ALGORITHM

1. Initialize: Create a graph data structure to store the vertices and edges.
2. Input Edges: Prompt the user to input the number of edges. Then, for each edge, prompt the user to input the source node and the destination node.
3. Input Start and Goal Nodes: Prompt the user to input the start node and the goal node for the search.
4. Input Depth Limit: Prompt the user to input the depth limit for the search. If -1 is entered, it signifies an unlimited depth.
5. Depth-Limited Search: Perform Depth-Limited Search algorithm to find a path from the start node to the goal node within the specified depth limit.
6. Print Expansion Table: Print the expansion table showing the expanded nodes and the fringe at each step.
7. Print Path: If a path is found, print the path from the start node to the goal node. If no path is found, print that there is no path.
8. End.

PROGRAM

```
from collections import deque


class Graph:
    def __init__(self, directed=True):
        self.edges = {}
        self.directed = directed


    def add_edge(self, node1, node2, __reversed=False):
        try:
            neighbors = self.edges[node1]
        except KeyError:
```

```
        neighbors = set()
    neighbors.add(node2)
    self.edges[node1] = neighbors
    if not self.directed and not __reversed:
        self.add_edge(node2, node1, True)

def neighbors(self, node):
    try:
        return self.edges[node]
    except KeyError:
        return []

def depth_limited_search(self, start, goal, limit=-1):
    print('Depth limit =', limit)
    found, fringe, visited, came_from = False, deque([(0, start)]), set([start]), {start: None}
    print('{:11s} | {}'.format('Expand Node', 'Fringe'))
    print('--------------------')
    print('{:11s} | {}'.format('-', start))
    while not found and len(fringe):
        depth, current = fringe.pop()
        print('{:11s}'.format(current), end=' | ')
        if current == goal:
            found = True
            break
        if limit == -1 or depth < limit:
            for node in self.neighbors(current):
                if node not in visited:
                    visited.add(node)
                    fringe.append((depth + 1, node))
                    came_from[node] = current
        print(', '.join([n for _, n in fringe]))
    if found:
```

```
            print()
            return came_from
        else:
            print('No path from {} to {}'.format(start, goal))

    @staticmethod
    def print_path(came_from, goal):
        parent = came_from[goal]
        if parent:
            Graph.print_path(came_from, parent)
        else:
            print(goal, end='')
            return
        print(' =>', goal, end='')

    def __str__(self):
        return str(self.edges)



graph = Graph(directed=False)



while True:
    try:
        num_edges = int(input("Enter the number of edges: "))
        break
    except ValueError:
        print("Invalid input! Please enter an integer.")

for _ in range(num_edges):
    while True:
        try:
```

```
            node1, node2 = input("Enter an edge (node1 node2): ").split()
            break
        except ValueError:
            print("Invalid input! Please enter two nodes separated by space.")
    graph.add_edge(node1, node2)

start = input("Enter the start node: ")
goal = input("Enter the goal node: ")
depth_limit = int(input("Enter the depth limit (or -1 for unlimited): "))

traced_path = graph.depth_limited_search(start, goal, depth_limit)
if traced_path:
    print('Path:', end=' ')
    Graph.print_path(traced_path, goal)
    print()
```

OUTPUT

```
Enter the number of edges: 5
Enter an edge (node1 node2): A B
Enter an edge (node1 node2): A S
Enter an edge (node1 node2): S G
Enter an edge (node1 node2): S C
Enter an edge (node1 node2): C F
Enter the start node: A
Enter the goal node: F
Enter the depth limit (or -1 for unlimited): 3
Depth limit = 3
Expand Node | Fringe
--------------------
-           | A
A           |
No path from A  to F
```

RESULTS

Thus the Python program to implement depth- limited search has been successfully executed.

EXP :	IMPLEMENTATION OF UNINFORMED FORMED SEARCH - ITERATIVE DEEPENING DEPTH FIRST SEARCH
DATE :	

AIM

To implement iterative deepening depth first search using Python.

ALGORITHM

1. Initialize: Create a graph data structure to store the vertices and edges.
2. Input Edges: Prompt the user to input the number of edges. Then, for each edge, prompt the user to input the source node and the destination node.
3. Input Start and Goal Nodes: Prompt the user to input the start node and the goal node for the search.
4. Iterative Deepening Depth-First Search: Perform iterative deepening depth-first search to find a path from the start node to the goal node.
5. Depth-Limited Search: Perform depth-limited search with increasing depth limits until the goal node is found or until the depth limit exceeds the maximum depth of the graph.
6. Print Expansion Table: Print the expansion table showing the expanded nodes and the fringe at each step.
7. Print Path: If a path is found, print the path from the start node to the goal node. If no path is found, print that there is no path.
8. End.

PROGRAM

```
from collections import deque

class Graph:

    def __init__(self, directed=True):

        self.edges = {}

        self.directed = directed


    def add_edge(self, node1, node2, __reversed=False):

        try:

            neighbors = self.edges[node1]

        except KeyError:

            neighbors = set()
```

```
        neighbors.add(node2)
        self.edges[node1] = neighbors
        if not self.directed and not __reversed:
            self.add_edge(node2, node1, True)

    def neighbors(self, node):
        try:
            return self.edges[node]
        except KeyError:
            return []

    def iterative_deepening_dfs(self, start, goal):
        prev_iter_visited, depth = [], 0
        while True:
            traced_path, visited = self.depth_limited_search(start, goal, depth)
            if traced_path or len(visited) == len(prev_iter_visited):
                return traced_path
            else:
                prev_iter_visited = visited
                depth += 1

    def depth_limited_search(self, start, goal, limit=-1):
        print('Depth limit =', limit)
        found, fringe, visited, came_from = False, deque([(0, start)]), set([start]), {start: None}
        print('{:11s} | {}'.format('Expand Node', 'Fringe'))
        print('--------------------')
        print('{:11s} | {}'.format('-', start))
        while not found and len(fringe):
            depth, current = fringe.pop()
            print('{:11s}'.format(current), end=' | ')
            if current == goal:
                found = True
```

```
                break
            if limit == -1 or depth < limit:
                for node in self.neighbors(current):
                    if node not in visited:
                        visited.add(node)
                        fringe.append((depth + 1, node))
                        came_from[node] = current
            print(', '.join([n for _, n in fringe]))
        if found:
            print()
            return came_from, visited
        else:
            print('No path from {} to {}'.format(start, goal))
            return None, visited

    @staticmethod
    def print_path(came_from, goal):
        parent = came_from[goal]
        if parent:
            Graph.print_path(came_from, parent)
        else:
            print(goal, end='')
            return
        print(' =>', goal, end='')

    def __str__(self):
        return str(self.edges)
graph = Graph(directed=False)
while True:
    try:
        num_edges = int(input("Enter the number of edges: "))
        break
```

```
        except ValueError:
            print("Invalid input! Please enter an integer.")

for _ in range(num_edges):
    while True:
        try:
            node1, node2 = input("Enter an edge (node1 node2): ").split()
            break
        except ValueError:
            print("Invalid input! Please enter two nodes separated by space.")
    graph.add_edge(node1, node2)

start = input("Enter the start node: ")
goal = input("Enter the goal node: ")

traced_path = graph.iterative_deepening_dfs(start, goal)
if traced_path:
    print('Path:', end=' ')
    Graph.print_path(traced_path, goal)
    print()
```

OUTPUT

```
Enter the number of edges: 5
Enter an edge (node1 node2): A B
Enter an edge (node1 node2): A S
Enter an edge (node1 node2): S G
Enter an edge (node1 node2): S C
Enter an edge (node1 node2): C F
Enter the start node: A
Enter the goal node: F
Depth limit = 0
Expand Node | Fringe
--------------------
-           | A
A           |
No path from A  to F
Depth limit = 1
Expand Node | Fringe
--------------------
-           | A
A           |
No path from A  to F
```

RESULT

Thus the Python program to implement iterative deepening depth first search has been successfully executed.

EXP :	IMPLEMENTATION OF INFORMED FORMED SEARCH - BEST FIRST SEARCH
DATE :	

AIM

To implement best first search using Python.

ALGORITHM

1. Initialize an empty priority queue Q.
2. Add the initial state of the problem to Q.
3. While Q is not empty:

   3.1 Remove the node with the lowest heuristic value from Q.
   3.2 If the node is a goal state, return the solution.
   3.3 Expand the node and add its children to Q.

PROGRAM

```
from collections import deque
class Graph:
    def __init__(self, directed=True):
        self.edges = {}
        self.directed = directed

    def add_edge(self, node1, node2, __reversed=False):
        try: neighbors = self.edges[node1]
        except KeyError: neighbors = set()
        neighbors.add(node2)
        self.edges[node1] = neighbors
        if not self.directed and not __reversed: self.add_edge(node2, node1, True)

    def neighbors(self, node):
        try: return self.edges[node]
        except KeyError: return []

    def breadth_first_search(self, start, goal):
```

```
        found, fringe, visited, came_from = False, deque([start]), set([start]), {start: None}
        print('{:11s} | {}'.format('Expand Node', 'Fringe'))
        print('--------------------')
        print('{:11s} | {}'.format('-', start))
        while not found and len(fringe):
            current = fringe.pop()
            print('{:11s}'.format(current), end=' | ')
            if current == goal: found = True; break
            for node in self.neighbors(current):
                if node not in visited: visited.add(node); fringe.appendleft(node); came_from[node] = current
            print(', '.join(fringe))
        if found: print(); return came_from
        else: print('No path from {} to {}'.format(start, goal))

    @staticmethod
    def print_path(came_from, goal):
        parent = came_from[goal]
        if parent:
            Graph.print_path(came_from, parent)
        else: print(goal, end='');return
        print(' =>', goal, end='')

    def __str__(self):
        return str(self.edges)
graph = Graph(directed=False)
graph.add_edge('A', 'B')
graph.add_edge('A', 'S')
graph.add_edge('S', 'G')
graph.add_edge('S', 'C')
graph.add_edge('C', 'F')
graph.add_edge('G', 'F')
graph.add_edge('C', 'D')
```

```
graph.add_edge('C', 'E')

graph.add_edge('E', 'H')

graph.add_edge('G', 'H')

start, goal = 'A', 'H'

traced_path = graph.breadth_first_search(start, goal)

if (traced_path): print('Path:', end=' '); Graph.print_path(traced_path, goal);print()
```

OUTPUT

```
Expand Node | Fringe
--------------------
-           | A
A           | S, B
B           | S
S           | G, C
C           | F, E, D, G
G           | H, F, E, D
D           | H, F, E
E           | H, F
F           | H
H           |
Path: A => S => G => H
```

RESULT

Thus the Python program to implement best first search has been successfully executed.

EXP :	IMPLEMENTATION OF INFORMED FORMED SEARCH - ITERATIVE DEEPENING A* SEARCH
DATE :	

AIM

To implement iterative deepening a* search using Python.

ALGORITHM

1. Initialize a threshold to the heuristic value of the initial state.
2. Repeat until a solution is found:
    1. Perform a depth-first search with depth-bound limited by the threshold.
    2. If a solution is found, return it.
    3. Update the threshold to the minimum f value exceeding the current threshold.
3. If no solution is found, return failure.

PROGRAM

```
from queue import heappop, heappush

from math import inf


class Graph:

    def __init__(self, directed=True):

        self.edges = {}

        self.huristics = {}

        self.directed = directed


    def add_edge(self, node1, node2, cost = 1, __reversed=False):

        try: neighbors = self.edges[node1]

        except KeyError: neighbors = {}

        neighbors[node2] = cost

        self.edges[node1] = neighbors

        if not self.directed and not __reversed: self.add_edge(node2, node1, cost, True)

    def set_huristics(self, huristics={}):

        self.huristics = huristics
```

```
def neighbors(self, node):
    try: return self.edges[node]
    except KeyError: return []

def cost(self, node1, node2):
    try: return self.edges[node1][node2]
    except: return inf

def iterative_deepening_astar_search(self, start, goal):
    prev_visited, depth = 0, 0
    while True:
        trace, cost, visited = self.dept_limited_astar_search(start, goal, depth)
        if trace or visited == prev_visited: return trace, cost
        prev_visited = visited
        depth += 1

def dept_limited_astar_search(self, start, goal, limit=-1):
    print('Depth Limit =', limit)
    found, fringe, visited = False, [(self.huristics[start], start, 0)], set([start])
    came_from, cost_so_far = {start: None}, {start: 0}
    print('{:11s} | {}'.format('Expand Node', 'Fringe'))
    print('--------------------')
    print('{:11s} | {}'.format('-', str(fringe[0][:-1])))
    while not found and len(fringe):
        _, current, depth = heappop(fringe)
        print('{:11s}'.format(current), end=' | ')
        if current == goal: found = True; break
        if limit == -1 or depth < limit:
            for node in self.neighbors(current):
                new_cost = cost_so_far[current] + self.cost(current, node)
                if node not in visited or cost_so_far[node] > new_cost:
```

```
                visited.add(node); came_from[node] = current; cost_so_far[node] = new_cost
                heappush(fringe, (new_cost + self.huristics[node], node, depth + 1))
            print(', '.join([str(n[:-1]) for n in fringe]))
        if found: print(); return came_from, cost_so_far[goal], len(visited)
        else: print('No path from {} to {}'.format(start, goal)); return None, inf, len(visited)

    @staticmethod
    def print_path(came_from, goal):
        parent = came_from[goal]
        if parent:
            Graph.print_path(came_from, parent)
        else: print(goal, end='');return
        print(' =>', goal, end='')

    def __str__(self):
        return str(self.edges)

graph = Graph(directed=True)
graph.add_edge('A', 'B', 4)
graph.add_edge('A', 'C', 1)
graph.add_edge('B', 'D', 3)
graph.add_edge('B', 'E', 8)
graph.add_edge('C', 'C', 0)
graph.add_edge('C', 'D', 7)
graph.add_edge('C', 'F', 6)
graph.add_edge('D', 'C', 2)
graph.add_edge('D', 'E', 4)
graph.add_edge('E', 'G', 2)
graph.add_edge('F', 'G', 8)
graph.set_huristics({'A': 8, 'B': 8, 'C': 6, 'D': 5, 'E': 1, 'F': 4, 'G': 0})
start, goal, limit = 'A', 'G', 3
traced_path, cost = graph.iterative_deepening_astar_search(start, goal)
```

if (traced_path): print('Path:', end=' '); Graph.print_path(traced_path, goal); print('\nCost:', cost)

OUTPUT

```
Depth Limit = 0
Expand Node | Fringe
--------------------
-           | (8, 'A')
A           |
No path from A to G
Depth Limit = 1
Expand Node | Fringe
--------------------
-           | (8, 'A')
A           | (7, 'C'), (12, 'B')
C           | (12, 'B')
B           |
No path from A to G
```

RESULT

Thus the Python program to implement iterative deepening a* search has been successfully executed.

EXP :	IMPLEMENTATION OF INFORMED FORMED SEARCH - BIDIRECTIONAL SEARCH
DATE :	

AIM

To implement bidirectional search using Python.

ALGORITHM

1. Initialize two empty queues: Q_start and Q_goal.
2. Add the initial state to Q_start and the goal state to Q_goal.
3. While both Q_start and Q_goal are not empty:
    1. Expand one node from Q_start and one node from Q_goal.
    2. If a node in Q_start is also in Q_goal, return the solution.
    3. If a node in Q_start can reach a node in Q_goal or vice versa, return the solution.
    4. Add the children of expanded nodes to their respective queues.
4. If no solution is found, return failure.

PROGRAM

```
from collections import deque
class Graph:
    def __init__(self, directed=True):
        self.edges = {}
        self.directed = directed

    def add_edge(self, node1, node2, __reversed=False):
        try: neighbors = self.edges[node1]
        except KeyError: neighbors = set()
        neighbors.add(node2)
        self.edges[node1] = neighbors
        if not self.directed and not __reversed: self.add_edge(node2, node1, True)

    def neighbors(self, node):
        try: return self.edges[node]
```

```
        except KeyError: return []


    def bi_directional_search(self, start, goal):
        found, fringe1, visited1, came_from1 = False, deque([start]), set([start]), {start: None}
        meet, fringe2, visited2, came_from2 = None, deque([goal]), set([goal]), {goal: None}
        while not found and (len(fringe1) or len(fringe2)):
            print('FringeStart: {:30s} | FringeGoal: {}'.format(str(fringe1), str(fringe2)))
            if len(fringe1):
                current1 = fringe1.pop()
                if current1 in visited2: meet = current1; found = True; break
                for node in self.neighbors(current1):
                    if node not in visited1: visited1.add(node); fringe1.appendleft(node); came_from1[node]
= current1
            if len(fringe2):
                current2 = fringe2.pop()
                if current2 in visited1: meet = current2; found = True; break
                for node in self.neighbors(current2):
                    if node not in visited2: visited2.add(node); fringe2.appendleft(node); came_from2[node]
= current2
        if found: print(); return came_from1, came_from2, meet
        else: print('No path between {} and {}'.format(start, goal)); return None, None, None


    @staticmethod
    def print_path(came_from, goal):
        parent = came_from[goal]
        if parent:
            Graph.print_path(came_from, parent)
        else: print(goal, end='');return
        print(' =>', goal, end='')
    def __str__(self):
        return str(self.edges)


graph = Graph(directed=False)
```

```
graph.add_edge('A', 'B'); graph.add_edge('A', 'S'); graph.add_edge('S', 'G')
graph.add_edge('S', 'C'); graph.add_edge('C', 'F'); graph.add_edge('G', 'F')
graph.add_edge('C', 'D'); graph.add_edge('C', 'E'); graph.add_edge('E', 'H')
graph.add_edge('G', 'H')
start, goal = 'A', 'H'
traced_path1, traced_path2, meet = graph.bi_directional_search(start, goal)
if meet:
    print('Meeting Node:', meet)
    print('Path From Start:', end=' '); Graph.print_path(traced_path1, meet); print()
    print('Path From Goal:', end=' '); Graph.print_path(traced_path2, meet); print()
```

OUTPUT

```
FringeStart: deque(['A'])              | FringeGoal: deque(['H'])
FringeStart: deque(['B', 'S'])         | FringeGoal: deque(['G', 'E'])
FringeStart: deque(['G', 'C', 'B'])    | FringeGoal: deque(['C', 'G'])

Meeting Node: G
Path From Start: A => S => G
Path From Goal: H => G
```

RESULT

Thus the Python program to implement bidirectional search has been successfully executed.

EXP :	8-PUZZLE PROBLEM USING BEST FIRST SEARCH
DATE :	

AIM

To implement 8-puzzle problem using best first search using Python.

ALGORITHM

1. Initialize an empty priority queue Q.
2. Add the initial state of the problem to Q.
3. While Q is not empty:

   3.1 Remove the node with the lowest heuristic value from Q.
   3.2 If the node is a goal state, return the solution.
   3.3 Expand the node and add its children to Q.

PROGRAM

```
import heapq

class PuzzleState:
    def __init__(self, puzzle, parent=None):
        self.puzzle = puzzle
        self.parent = parent
        self.cost = 0
        self.heuristic = 0
        self.depth = 0


    def __lt__(self, other):
        return (self.cost + self.heuristic) < (other.cost + other.heuristic)


    def __eq__(self, other):
        return self.puzzle == other.puzzle


    def __hash__(self):
        return hash(str(self.puzzle))
    def goal_test(self):
```

```
        return self.puzzle == [0, 1, 2, 3, 4, 5, 6, 7, 8]

    def generate_children(self):
        children = []
        zero_index = self.puzzle.index(0)
        moves = [(1, 0), (-1, 0), (0, 1), (0, -1)]  # Right, Left, Down, Up

        for dx, dy in moves:
            new_x, new_y = zero_index // 3 + dx, zero_index % 3 + dy
            if 0 <= new_x < 3 and 0 <= new_y < 3:
                new_puzzle = self.puzzle[:]
                new_zero_index = new_x * 3 + new_y
                new_puzzle[zero_index], new_puzzle[new_zero_index] = new_puzzle[new_zero_index],
new_puzzle[zero_index]
                new_state = PuzzleState(new_puzzle, self)
                children.append(new_state)

        return children

def manhattan_distance(state):
    distance = 0
    for i in range(1, 9):
        current_x, current_y = state.puzzle.index(i) // 3, state.puzzle.index(i) % 3
        goal_x, goal_y = (i - 1) // 3, (i - 1) % 3
        distance += abs(current_x - goal_x) + abs(current_y - goal_y)
    return distance

def best_first_search(initial_state):
    frontier = []
    heapq.heappush(frontier, initial_state)

    explored = set()
    while frontier:
```

```python
        current_state = heapq.heappop(frontier)

        if current_state.goal_test():
            return current_state

        explored.add(current_state)

        for child in current_state.generate_children():
            if child not in explored:
                child.cost = current_state.cost + 1
                child.heuristic = manhattan_distance(child)
                heapq.heappush(frontier, child)

    return None

def print_solution(solution):
    path = []
    current_state = solution
    while current_state:
        path.append(current_state.puzzle)
        current_state = current_state.parent
    path.reverse()
    for i, state in enumerate(path):
        print(f"Move {i}:")
        print_puzzle(state)
        print()

def print_puzzle(puzzle):
    for i in range(0, 9, 3):
        print(puzzle[i:i+3])

if __name__ == "__main__":
```

```
initial_puzzle = [1, 2, 3, 0, 4, 5, 6, 7, 8]  # Initial state of the puzzle
initial_state = PuzzleState(initial_puzzle)


solution = best_first_search(initial_state)


if solution:
    print("Solution found:")
    print_solution(solution)
else:
    print("No solution found.")
```

OUTPUT

```
Solution found:
```

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

RESULT

Thus the Python program to implement 8-puzzle problem using best first search has been successfully executed.

EXP :	IMPLEMENTATION OF INFORMED FORMED SEARCH - A* SEARCH
DATE :	

AIM

To implement A* search using Python.

ALGORITHM

1. Initialize an empty priority queue Q.
2. Add the initial state of the problem to Q with priority f(initial_state) = g(initial_state) + h(initial_state).
3. While Q is not empty:
   a. Remove the node with the lowest f value from Q.
   b. If the node is a goal state, return the solution.
   c. Expand the node and compute the f values for its children.
   d. Add the children to Q with their respective f values.
4. If no solution is found, return failure.

PROGRAM

```
from queue import heappop, heappush

from math import inf


class Graph:

    def __init__(self, directed=True):

        self.edges = {}

        self.huristics = {}

        self.directed = directed


    def add_edge(self, node1, node2, cost = 1, __reversed=False):

        try: neighbors = self.edges[node1]

        except KeyError: neighbors = {}

        neighbors[node2] = cost

        self.edges[node1] = neighbors

        if not self.directed and not __reversed: self.add_edge(node2, node1, cost, True)
```

```
    def set_huristics(self, huristics={}):
        self.huristics = huristics

    def neighbors(self, node):
        try: return self.edges[node]
        except KeyError: return []

    def cost(self, node1, node2):
        try: return self.edges[node1][node2]
        except: return inf
    def a_star_search(self, start, goal):
        found, fringe, visited, came_from, cost_so_far = False, [(self.huristics[start], start)], set([start]),
{start: None}, {start: 0}
        print('{:11s} | {}'.format('Expand Node', 'Fringe'))
        print('--------------------')
        print('{:11s} | {}'.format('-', str(fringe[0])))
        while not found and len(fringe):
            _, current = heappop(fringe)
            print('{:11s}'.format(current), end=' | ')
            if current == goal: found = True; break
            for node in self.neighbors(current):
                new_cost = cost_so_far[current] + self.cost(current, node)
                if node not in visited or cost_so_far[node] > new_cost:
                    visited.add(node); came_from[node] = current; cost_so_far[node] = new_cost
                    heappush(fringe, (new_cost + self.huristics[node], node))
            print(', '.join([str(n) for n in fringe]))
        if found: print(); return came_from, cost_so_far[goal]
        else: print('No path from {} to {}'.format(start, goal)); return None, inf

    @staticmethod
    def print_path(came_from, goal):
        parent = came_from[goal]
```

```
        if parent:
            Graph.print_path(came_from, parent)
        else: print(goal, end='');return
        print(' =>', goal, end='')
    def __str__(self):
        return str(self.edges)

graph = Graph(directed=True)
graph.add_edge('A', 'B', 4)
graph.add_edge('A', 'C', 1)
graph.add_edge('B', 'D', 3)
graph.add_edge('B', 'E', 8)
graph.add_edge('C', 'C', 0)
graph.add_edge('C', 'D', 7)
graph.add_edge('C', 'F', 6)
graph.add_edge('D', 'C', 2)
graph.add_edge('D', 'E', 4)
graph.add_edge('E', 'G', 2)
graph.add_edge('F', 'G', 8)
graph.set_huristics({'A': 8, 'B': 8, 'C': 6, 'D': 5, 'E': 1, 'F': 4, 'G': 0})
start, goal = 'A', 'G'
traced_path, cost = graph.a_star_search(start, goal)
if (traced_path): print('Path:', end=' '); Graph.print_path(traced_path, goal); print('\nCost:', cost)
```

OUTPUT

```
Expand Node | Fringe
--------------------
-           | (8, 'A')
A           | (7, 'C'), (12, 'B')
C           | (11, 'F'), (13, 'D'), (12, 'B')
F           | (12, 'B'), (13, 'D'), (15, 'G')
B           | (12, 'D'), (13, 'E'), (13, 'D'), (15, 'G')
D           | (12, 'E'), (13, 'D'), (15, 'G'), (13, 'E')
E           | (13, 'D'), (13, 'E'), (15, 'G'), (13, 'G')
D           | (13, 'E'), (13, 'G'), (15, 'G')
E           | (13, 'G'), (15, 'G')
G           |
Path: A => B => D => E => G
Cost: 13
```

RESULT

Thus the Python program to implement A* search has been successfully executed.

EXP :	TRAVELLING SALESMAN PROBLEM
DATE :	

AIM

To implement travelling salesman problem using Python.

ALGORITHM

1. Generate all possible permutations of cities.
2. For each permutation, calculate the total distance of the route by summing up the distances between consecutive cities.
3. Identify the permutation with the shortest total distance.
4. Output the shortest route found along with its total distance.
5. Implement optimization techniques to improve the efficiency of the algorithm heuristic methods for larger instances of the problem

PROGRAM

```
import itertools
# Function to calculate the distance between two cities (Euclidean distance)
def distance(city1, city2):
    return ((city1[0] - city2[0]) ** 2 + (city1[1] - city2[1]) ** 2) ** 0.5
def total_distance(route, cities):
    total = 0
    for i in range(len(route) - 1):
        total += distance(cities[route[i]], cities[route[i+1]])
    # Add distance from last city back to the starting city
    total += distance(cities[route[-1]], cities[route[0]])
    return total
def tsp_brute_force(cities):
    shortest_distance = float('inf')
    shortest_route = None

    # Generate all possible permutations of cities
    for route in itertools.permutations(range(len(cities))):
```

```
        route_distance = total_distance(route, cities)
        if route_distance < shortest_distance:
            shortest_distance = route_distance
            shortest_route = route

    return shortest_route, shortest_distance
# Example usage
if __name__ == "__main__":
    # Example cities (format: [x, y])
    cities = [(0, 0), (1, 2), (3, 1), (5, 3)]

    # Find the shortest route and its distance
    shortest_route, shortest_distance = tsp_brute_force(cities)

    print("Shortest route:", shortest_route)
    print("Shortest distance:", shortest_distance)
```

OUTPUT

```
Shortest route: (2, 0, 1, 3)
Shortest distance: 12.349878388032O2
```

RESULT

Thus the Python program to implement travelling salesman problem has been successfully executed.

EXP :	GENETIC ALGORITHM OF TRAVELLING SALESMAN PROBLEM
DATE :	

AIM

To implement the travelling salesman problem using genetic algorithm using Python.

ALGORITHM

1. Generate an initial population of routes, each consisting of a random permutation of cities.
2. Calculate the fitness of each route in the population based on the total distance traveled. Higher fitness corresponds to shorter distances.
3. Evolve Population:
    - Select individuals from the population to serve as parents for the next generation.
    - Generate offspring through crossover and mutation operations.
    - Replace the current population with the new generation of individuals.
4. Repeat the evolution process for a specified number of generations.
5. After the specified number of generations, select the route with the highest fitness as the best route found.

PROGRAM

```
import random
import numpy as np
class GeneticAlgorithmTSP:
    def __init__(self, cities, population_size=50, mutation_rate=0.01, generations=100):
        self.cities = cities
        self.population_size = population_size
        self.mutation_rate = mutation_rate
        self.generations = generations

    def initial_population(self):
        return [random.sample(self.cities, len(self.cities)) for _ in range(self.population_size)]
    def fitness(self, route):
        total_distance = 0
        for i in range(len(route)):
```

```
        total_distance += np.linalg.norm(np.array(route[i-1]) - np.array(route[i]))
    return 1 / total_distance

def mutate(self, route):
    for swap in range(len(route)):
        if random.random() < self.mutation_rate:
            swap_with = int(random.random() * len(route))
            route[swap], route[swap_with] = route[swap_with], route[swap]
    return route

def crossover(self, parent1, parent2):
    start = int(random.random() * len(parent1))
    end = int(random.random() * len(parent1))
    if start > end:
        start, end = end, start

    child = [None] * len(parent1)
    for i in range(start, end):
        child[i] = parent1[i]

    for i in range(len(parent2)):
        if parent2[i] not in child:
            for j in range(len(child)):
                if child[j] is None:
                    child[j] = parent2[i]
                    break

    return child

def evolve(self, population):
    graded = [(self.fitness(route), route) for route in population]
    graded = [route[1] for route in sorted(graded, key=lambda x: x[0], reverse=True)]
```

```
        retain_length = int(len(graded) * 0.2)
        parents = graded[:retain_length]

        for individual in graded[retain_length:]:
            if random.random() < 0.5:
                parents.append(individual)

        parents_length = len(parents)
        desired_length = len(population) - parents_length
        children = []

        while len(children) < desired_length:
            parent1 = random.choice(parents)
            parent2 = random.choice(parents)
            child = self.crossover(parent1, parent2)
            child = self.mutate(child)
            children.append(child)

        parents.extend(children)
        return parents

    def optimize(self):
        population = self.initial_population()
        for i in range(self.generations):
            population = self.evolve(population)
        best_route = max([(self.fitness(route), route) for route in population], key=lambda x: x[0])[1]
        return best_route

# Example usage:
if __name__ == "__main__":
    # Example cities (format: [x, y])
    cities = [(0, 0), (1, 2), (3, 1), (5, 3)]
```

```
# Initialize and run Genetic Algorithm
ga_tsp = GeneticAlgorithmTSP(cities)
best_route = ga_tsp.optimize()

print("Best Route:", best_route)
```

OUTPUT

```
Best Route: [(3, 1), (5, 3), (1, 2), (0, 0)]
```

RESULT

Thus the Python program to implement travelling salesman problem using genetic algorithm has been successfully executed.

EXP :	SIMULATED ANNEALING ALGORITHM OF TRAVELLING SALESMAN PROBLEM
DATE :	

AIM

To implement simulated annealing algorithm of travelling salesman problem using Python.

ALGORITHM

1. Generate an initial solution (route) randomly. This could be a random permutation of the cities.
2. Define a fitness function to evaluate the quality of a solution (route). In TSP, this could be the inverse of the total distance travelled.
3. Repeat for a specified number of iterations:
    a. Perturb the current solution to obtain a neighbouring solution. This could involve swapping two cities in the route.
    b. Evaluate the fitness of the new solution.
    c. If the new solution is better (has higher fitness), accept it as the current solution.
    d. If the new solution is worse, accept it probabilistically based on the change in fitness and the current temperature according to the acceptance probability formula.
    e. Update the temperature according to the cooling schedule.
4. Repeat the annealing process for a predetermined number of iterations or until a stopping criterion is met.
5. Output the best solution found during the annealing process, i.e., the solution with the highest fitness.

PROGRAM

```
import random

import math

import numpy as np

class SimulatedAnnealingTSP:

    def __init__(self, cities, temperature=10000, cooling_rate=0.003, num_iterations=1000):

        self.cities = cities

        self.temperature = temperature

        self.cooling_rate = cooling_rate
```

```
        self.num_iterations = num_iterations

    def initial_solution(self):
        return random.sample(self.cities, len(self.cities))

    def fitness(self, route):
        total_distance = 0
        for i in range(len(route)):
            total_distance += np.linalg.norm(np.array(route[i-1]) - np.array(route[i]))
        return 1 / total_distance

    def acceptance_probability(self, cost_diff, temperature):
        if cost_diff < 0:
            return 1.0
        return math.exp(-cost_diff / temperature)

    def anneal(self):
        current_solution = self.initial_solution()
        current_cost = self.fitness(current_solution)
        best_solution = current_solution
        best_cost = current_cost

        for i in range(self.num_iterations):
            new_solution = current_solution[:]
            city1, city2 = random.sample(range(len(new_solution)), 2)
            new_solution[city1], new_solution[city2] = new_solution[city2], new_solution[city1]

            new_cost = self.fitness(new_solution)
            cost_diff = new_cost - current_cost

            if self.acceptance_probability(cost_diff, self.temperature) > random.random():
                current_solution = new_solution
```

```
                current_cost = new_cost

            if current_cost > best_cost:
                best_solution = current_solution
                best_cost = current_cost

            self.temperature *= 1 - self.cooling_rate
        return best_solution

if __name__ == "__main__":
    # Example cities (format: [x, y])
    cities = [(0, 0), (2, 3), (3, 8), (3, 9)]
    # Initialize and run Simulated Annealing Algorithm
    sa_tsp = SimulatedAnnealingTSP(cities)
    best_route = sa_tsp.anneal()
    print("Best Route:", best_route)
```

OUTPUT

```
Best Route: [(2, 3), (3, 8), (3, 9), (0, 0)]
```

RESULT

Thus the Python program to implement annealing algorithm of travelling salesman problem has been successfully executed.

EXP :	MAP COLOURING USING CONSTRAINT SATISFACTION PROBLEM (CSP)
DATE :	

AIM

To implement map colouring using constraint satisfaction problem (CSP) using Python.

ALGORITHM

1. Define a set of colours that can be used to colour the map.
2. Define the states (regions) of the map and their neighbouring states.
3. For each state in the map: Determine the available colours that haven't been used by neighbouring states.
4. Assign a colour to the current state from the set of available colours.
5. Repeat this process for each state, ensuring that neighbouring states have different colours.
6. If a valid colouring is found (i.e., all states are assigned colours without violating the constraint of neighbouring states having the same colour), output the colouring.
7. If no valid colouring is possible (i.e., conflicting constraints arise), report that no solution exists.

PROGRAM

```
def map_coloring(states, neighbors):
    colors = {}
    for state in states:
        available_colors = set(["red", "green", "blue"])
        for neighbor in neighbors[state]:
            if neighbor in colors:
                available_colors.discard(colors[neighbor])
        if available_colors:
            colors[state] = available_colors.pop()
        else:
            return None
    return colors
# Define states and their neighbors
states = ["WA", "NT", "SA", "Q", "NSW", "V", "T"]
```

```
neighbors = {
    "WA": ["NT", "SA"],
    "NT": ["WA", "SA", "Q"],
    "SA": ["WA", "NT", "Q", "NSW", "V"],
    "Q": ["NT", "SA", "NSW", "V"],
    "NSW": ["SA", "Q", "V"],
    "V": ["SA", "Q", "NSW", "T"],
    "T": ["V"]}
# Solve the problem
solution = map_coloring(states, neighbors)
# Print the solution
if solution:
    print("Solution found:")
    for state, color in solution.items():
        print(f"{state}: {color}")
else:
    print("No solution found.")
```

OUTPUT

```
Solution found:
WA: red
NT: green
SA: blue
Q: red
NSW: green
V: red
T: blue
```

RESULT

Thus the Python program to implement map colouring using constraint satisfaction problem has been successfully executed.

EXP :	ADVERSARIAL SEARCH FOR TIC-TAC-TOE PROBLEM
DATE :	

AIM

To implement adversarial search for tic-tac-toe using Python.

ALGORITHM

1. Set up the Tic-tac-toe board and print it.
2. Ask the player to input their move (row and column).
3. Validate the move and update the board.
4. Check if the player wins or if it's a tie. If so, end the game.
5. Implement the minimax algorithm to find the best move for the AI.
6. Update the board with the AI's move.
7. Check if the AI wins or if it's a tie. If so, end the game.
8. Alternate between steps 2 and 3 until the game ends.
9. Print the final state of the board.
10. Print the result (whether the player wins, the AI wins, or it's a tie).
11. End the game.

PROGRAM

```
# Constants for players
PLAYER_X = 'X'
PLAYER_O = 'O'
EMPTY = ' '

def print_board(board):
    for row in board:
        print("|".join(row))
        print("-----")
    print()

def evaluate(board):
```

```
    # Check rows, columns, and diagonals for a win
    for row in board:
        if row.count(PLAYER_X) == 3:
            return 10
        elif row.count(PLAYER_O) == 3:
            return -10

    for col in range(3):
        if board[0][col] == board[1][col] == board[2][col] and board[0][col] != EMPTY:
            if board[0][col] == PLAYER_X:
                return 10
            else:
                return -10

    if board[0][0] == board[1][1] == board[2][2] and board[0][0] != EMPTY:
        if board[0][0] == PLAYER_X:
            return 10
        else:
            return -10

    if board[0][2] == board[1][1] == board[2][0] and board[0][2] != EMPTY:
        if board[0][2] == PLAYER_X:
            return 10
        else:
            return -10

    # If no winner, it's a tie
    return 0

def is_moves_left(board):
    for row in board:
        if EMPTY in row:
```

```
            return True
    return False

def minimax(board, depth, is_maximizing):
    score = evaluate(board)

    if score == 10:
        return score - depth
    if score == -10:
        return score + depth
    if not is_moves_left(board):
        return 0

    if is_maximizing:
        best = -float('inf')
        for i in range(3):
            for j in range(3):
                if board[i][j] == EMPTY:
                    board[i][j] = PLAYER_X
                    best = max(best, minimax(board, depth+1, not is_maximizing))
                    board[i][j] = EMPTY
        return best
    else:
        best = float('inf')
        for i in range(3):
            for j in range(3):
                if board[i][j] == EMPTY:
                    board[i][j] = PLAYER_O
                    best = min(best, minimax(board, depth+1, not is_maximizing))
                    board[i][j] = EMPTY
        return best
```

```
def find_best_move(board):
    best_val = -float('inf')
    best_move = (-1, -1)

    for i in range(3):
        for j in range(3):
            if board[i][j] == EMPTY:
                board[i][j] = PLAYER_X
                move_val = minimax(board, 0, False)
                board[i][j] = EMPTY
                if move_val > best_val:
                    best_move = (i, j)
                    best_val = move_val

    return best_move

def play_game():
    board = [[EMPTY]*3 for _ in range(3)]
    print("Welcome to Tic Tac Toe!")
    print_board(board)

    while True:
        # Player's move
        while True:
            row = int(input("Enter row (0, 1, or 2): "))
            col = int(input("Enter column (0, 1, or 2): "))
            if board[row][col] == EMPTY:
                board[row][col] = PLAYER_O
                break
            else:
                print("That spot is already taken. Try again.")
```

```
        print_board(board)

        # Check if player wins
        if evaluate(board) == -10:
            print("You win!")
            break

        # Check for tie
        if not is_moves_left(board):
            print("It's a tie!")
            break

        # AI's move
        print("AI's move:")
        ai_move = find_best_move(board)
        board[ai_move[0]][ai_move[1]] = PLAYER_X
        print_board(board)

        # Check if AI wins
        if evaluate(board) == 10:
            print("AI wins!")
            break

        # Check for tie
        if not is_moves_left(board):
            print("It's a tie!")
            break

play_game()
```

OUTPUT

```
Welcome to Tic Tac Toe!
 | |
-----
 | |
-----
 | |
-----

Enter row (0, 1, or 2): 0
Enter column (0, 1, or 2): 1
 |O|
-----
 | |
-----
 | |
-----

AI's move:
X|O|
-----
 | |
-----
 | |
-----
```

RESULT

Thus the Python program to implement adversarial search for tic-tac-toe problem has been successfully executed.

EXP :	KINSHIP DOMAIN USING PROLOG
DATE :	

AIM

To establish the various relationships of Kinship Domain using SWISH Prolog and generate queries for such relationships respectively.

ALGORITHM

1. Create a New Program in SWISH Prolog.
2. Define the various Facts/Premises of the related Problem.
3. Define the Rules of the problem.
4. Execute the Query
5. Note down the respective output for the Query.

PROGRAM

FACTS:

```
female(tamil).
female(sita).
female(anu).
female(pavi).
female(agalya).
female(preeti).
male(ram).
male(damodar).
male(ramesh).
male(ravi).
male(udhay).
male(kishore).
father(ram,udhay).
father(ram,preeti).
father(damodar,ram).
father(damodar,ravi).
father(ramesh,tamil).
father(ramesh,anu).
mom(tamil,preeti).
mom(tamil,udhay).
mom(agalya,ram).
mom(agalya,ravi).
mom(sita,tamil).
mom(sita,anu).
spouse(ram,tamil).
spouse(tamil,ram).
spouse(damodar,agalya).
spouse(agalya,damodar).
spouse(ramesh,sita).
spouse(sita,ramesh).
spouse(ravi,pavi).
spouse(pavi,ravi).
spouse(kishore,anu).
spouse(anu,kishore).
```

RULES:

child(X,Y):-(male(X);female(X)),(father(Y,X);mom(Y,X)).

daughter(X,Y):-female(X),(father(Y,X);mom(Y,X)).

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

sibling(X,Y):-mom(A,X),mom(A,Y).

brother(X,Y):-male(X),father(A,X),father(A,Y).

sister(X,Y):-female(X),mom(A,X),mom(A,Y).

grandparent(X,Y):-child(A,X),child(Y,A).

grandmother(X,Y):-female(X),child(A,X),child(Y,A).

uncle(X,Y):- male(X),child(X,Z),child(A,Z),child(Y,A).

sister_in_law(X,Y):-

female(X),child(Y,A),child(Z,A),spouse(X,Z);female(X),child(X,A),child(Z,A),spouse(Y,Z).

mother_in_law(X,Y):-female(X),spouse(Y,Z),child(Z,X).

wife(X,Y):-female(X),spouse(X,Y).

ancestor(X,Y):-child(Y,X);child(A,X),child(Y,A).

descendant(X,Y):-child(X,Y);child(X,A),child(A,Y).

relative_by_blood(X,Y):-
child(Y,X);child(X,Y);child(A,X),child(Y,A);child(A,Y),child(X,A);child(X,A),child(Y,A).

OUTPUT

QUERIES:

1. child(X,Y) :- % true if X is a child of Y

0. daughter(X,Y) :- % true if X is a daughter of Y

```
daughter(tamil,sita).
true
?- daughter(tamil,sita).
```

```
daughter(tamil,agalya).
false
?- daughter(tamil,agalya).
```

0. mother(X,Y) :- % true if X is the mother of Y.

```
mother(agalya,ram).
true
Next 10 100 1,000 Stop
?- mother(agalya,ram).
```

```
mother(preeti,agalya).
false
?- mother(preeti,agalya).
```

0. sibling(X,Y) :- % true if X and Y are siblings (i.e. have the same biological % parents). Be sure your definition does not lead to one being ones sibling

```
sibling(tamil,anu).
true
?- sibling(tamil,anu).
```

```
sibling(anu,ravi).
false
?- sibling(anu,ravi).
```

0. brother(X,Y) :- % true if X is a brother of Y, e.g., X is a male sibling of y

```
brother(ravi,ram).
true

?- brother(ravi,ram).
```

```
brother(ravi,anu).
false

?- brother(ravi,anu).
```

0. sister(X,Y) :- % true if X is a sister of Y, e.g., X is a female sibling of y

```
sister(tamil,anu).
true

?- sister(tamil,anu).
```

```
sister(preeti,ravi).
false

?- sister(preeti,ravi).
```

0. grandparent(X,Y) :- % true of X is a grandparent of Y, e.g., X is a parent of some Z who % is also a parent of Y

```
grandparent(sita,udhay).
true
Next | 10 | 100 | 1,000 | Stop

?- grandparent(sita,udhay).
```

```
grandparent(ravi,udhay).
false

?- grandparent(ravi,udhay).
```

0. grandmother(X,Y) :- % true of X is a grandmother of Y, e.g., X is a female grandparent of Y

grandmother(sita,udhay).

true

Next 10 100 1,000 Stop

```
?- grandmother(sita,udhay).
```

grandmother(anu,udhay).

false

```
?- grandmother(anu,udhay).
```

0. uncle(X,Y) :- % true if X is an uncle of Y, i.e. the brother of a parent.

uncle(ravi,udhay).

true

Next 10 100 1,000 Stop

```
?- uncle(ravi,udhay).
```

uncle(ramesh,udhay).

false

```
?- uncle(ramesh,udhay).
```

0. uncle(X,Y) :- % true if X is an uncle of Y, i.e. the male spouse of a sibling of a % parent

uncle(ravi,udhay).

true

Next 10 100 1,000 Stop

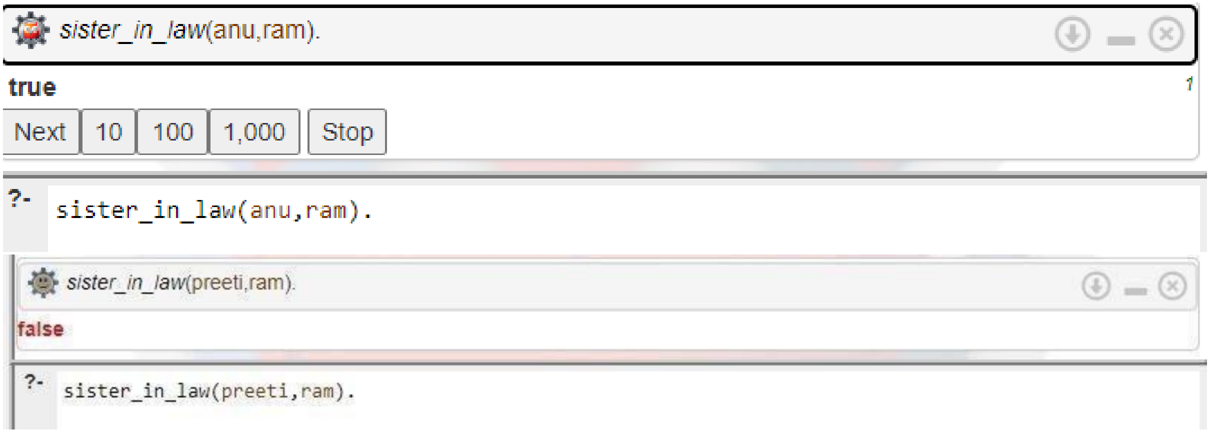
```
?- uncle(ravi,udhay).
```

uncle(ramesh,udhay).

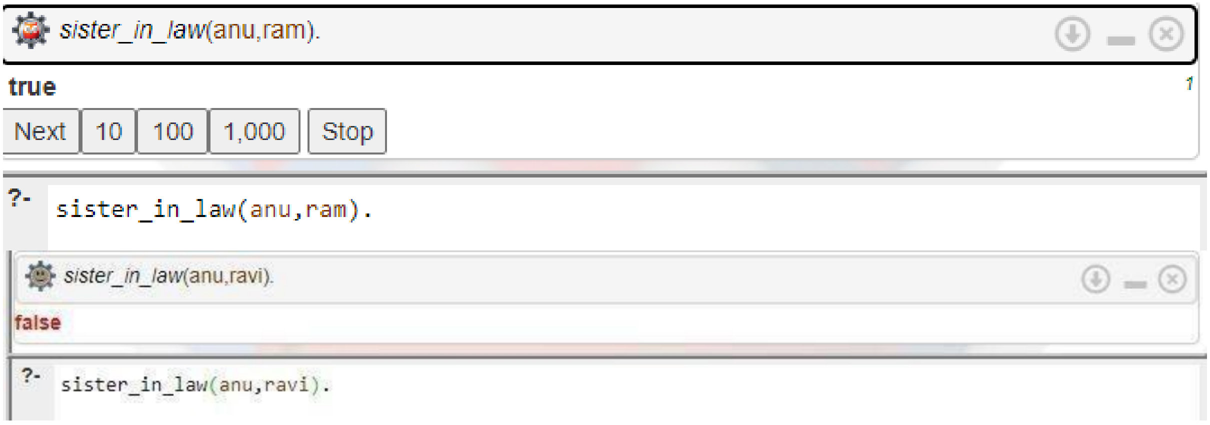
false

```
?- uncle(ramesh,udhay).
```

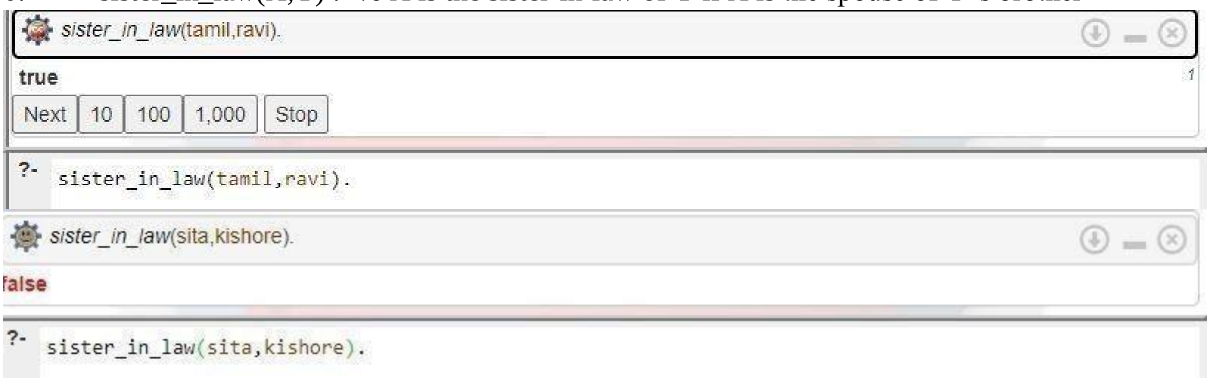
0. sister_in_law(X,Y) :-% X is the sister-in-law of Y if X is the female spouse of Y‟s sibling



0. sister_in_law(X,Y) :- % X is the sister-in-law of Y if X is the sister of Y‟;s spouse



0. sister_in_law(X,Y) :- % X is the sister-in-law of Y if X is the spouse of Y‟s brother



0. mother_in_law(X,Y) :- % X is the mother_in_law of Y if if X is the mother of Y's spouse

mother_in_law(sita,ram).

true

?- mother_in_law(sita,ram).

mother_in_law(anu,ram).

false

?- mother_in_law(anu,ram).

0. wife(X,Y) :- % X is the wife of Y if X is Y's female spouse

wife(tamil,ram).

true

?- wife(tamil,ram).

wife(anu,ravi).

false

?- wife(anu,ravi).

0. ancestor(X,Y) :- % X is an ancestor of Y if X is a parent of Y

ancestor(damodar,ram).

true

Next | 10 | 100 | 1,000 | Stop

?- ancestor(damodar,ram).

ancestor(ravi,ram).

false

?- ancestor(ravi,ram).

0. ancestor(X,Y) :- % X is an ancestor of Y if P is a parent of Y and X is an ancestor of P

ancestor(ramesh,preeti).

true

Next | 10 | 100 | 1,000 | Stop

?- ancestor(ramesh,preeti).

ancestor(agalya,kishore).

false

?- ancestor(agalya,kishore).

0. descendant(X,Y) :-% X is a descendant of Y if Y is an ancestor of X

0. relative_by_blood(X,Y) :- % X is an relative_by_blood of Y if X and Y share an ancestor

RESULT

Thus, the required queries for Kinship Domain is executed and the outputs are verified and documented accordingly.

EXP :	K-MEANS CLUSTERING ALGORITHM
DATE :	

AIM

To implement K-means clustering algorithm using Python.

ALGORITHM

1. Initialize Centroids:
    - Choose the number of clusters *k*.
    - Initialize *k* centroids randomly from the dataset.
2. Assign Data Points to Nearest Centroid:
    - Compute the distance between each data point and each centroid.
    - Assign each data point to the nearest centroid based on the distance.
3. Update Centroids:
    - Calculate the mean of all data points assigned to each centroid.
    - Update the centroid coordinates to the mean values.
4. Repeat Steps 2 and 3:
    - Repeat the assignment of data points to centroids and centroid updates until convergence or until a maximum number of iterations is reached.
5. Convergence Check:
    - Check for convergence by comparing the updated centroids with the previous centroids.
    - If the centroids do not change significantly (below a predefined threshold), stop the algorithm. Otherwise, repeat Steps 2 and 3.

PROGRAM

```
import numpy as np
class KMeans:
    def __init__(self, n_clusters, max_iterations=100):
        self.n_clusters = n_clusters
        self.max_iterations = max_iterations


    def fit(self, X):
        # Initialize centroids randomly
```

```
        self.centroids = X[np.random.choice(X.shape[0], self.n_clusters, replace=False)]

        for _ in range(self.max_iterations):
            # Assign each data point to the nearest centroid
            labels = self._assign_clusters(X)

            # Update centroids based on the mean of the points assigned to each cluster
            new_centroids = self._update_centroids(X, labels)

            # Check for convergence
            if np.allclose(self.centroids, new_centroids):
                break

            self.centroids = new_centroids

    def _assign_clusters(self, X):
        distances = np.sqrt(((X - self.centroids[:, np.newaxis])**2).sum(axis=2))
        return np.argmin(distances, axis=0)

    def _update_centroids(self, X, labels):
        new_centroids = np.zeros_like(self.centroids)
        for i in range(self.n_clusters):
            points = X[labels == i]
            if len(points) > 0:
                new_centroids[i] = np.mean(points, axis=0)
            else:
                new_centroids[i] = self.centroids[i]
        return new_centroids

# Example usage:
if __name__ == "__main__":
    # Generate random data points
```

```
np.random.seed(0)
X = np.random.rand(100, 2)

# Specify number of clusters
kmeans = KMeans(n_clusters=3)

# Fit the KMeans model to the data
kmeans.fit(X)

# Print the centroids
print("Final centroids:\n", kmeans.centroids)
```

OUTPUT

```
Final centroids:
 [[0.32472932 0.33625072]
 [0.69556481 0.28243836]
 [0.65142505 0.7296778 ]]
```

RESULT

Thus the Python program to implement K-means clustering algorithm has been successfully executed.

EXP :	DECISION MAKING TREE USING MACHINE LEARNING
DATE :	

AIM

To implement a simple decision-making tree using machine learning using Prolog.

THEORY

Basic Primitives:

- name/1 - defines the name for an individual
- names/1- lists all the individuals
- attribute/1 - defines the name for an attribute
- attributes/1- lists all the attributes
- class/1 defines the classes. classes/1 defines the list of all the classes.

To each constant atr that satisfies attribute(atr), a predicate atr/1 exists that defines which individuals have said attribute.

The examples are defined by the predicate class o/2.

The predicate examples(?Ns) makes Ns the list of all the names N that have a class C that verifies classof (N,C).

And so we have the following predicates in Prolog.

1. sameclass(+Ns, ?C): all the names in N are of class C.
2. partition(+Ns, +A, ?NT, ?NF): partitions the list Ns into two lists: NT and NF. NT has the names that have attribute A, and NF contains the names that don't.
3. proportion(+Ns, +C, +A, ?P): P is the proportion of elements of class C in the Ns list, for which an element A is true.
4. entropy(+Ns, +As, +A, ?E): E is the entropy of list Ns for an attribute A if it belongs to the list of attributes As, or 1.0 if it doesn't. The entropy for an attribute is calculated as: $(-1) * \Sigma c \in clases\ p(c) * \log p(c)$. Where p(c) is the proportion mentioned previously. If p(c) is 0, then p(c) * log p(c) is replaced by 0.
5. minatr(+Ns, +As, ?M): M is the attribute of the list As which has minimum entropy in Ns. If there are many possible values it returns the first in the list.
6. maxcla(+Ns, ?C): C is the most representative class of Ns.
7. id3(+Ns, +As, +C, ?T), T is the decision tree for a list of names Ns, a list of attributes As and class C. Each leaf of T is a leaf-node, leaf(C), where C is a class. Each non terminal node for an attribute A represents the class label of the final subset of this branch. It has the shape node(A, T1, T2) where A is an attribute and T1 and T2 are trees.
8. tree(?T): T is the tree we get from using id3 for the set of examples. The initial attribute list is the list of all the attributes. The initial class is the most representative of the training set examples.
9. classify(+T, +N, ?C): C is the class of the name according to a classifing tree T.

10.clasification(Ns, ?Xs): Xs is the list of clasifications obtained for the list Ns. Each element of Xs is a tuple (N,C) so that N is an element of Ns and C is a class.

PROGRAM

```
male(kasthuriraja).
male(dhanush).
male(selva).
male(yatra).
male(linga).
female(vijaya).
female(aishwarya).
female(geethanjali).
female(anjali).
parent(kasthuriraja,dhanush).
parent(vijaya,dhanush).
parent(kasthuriraja,selva).
parent(vijaya,selva).

parent(dhanush,linga).
parent(aishwarya,linga).
parent(dhanush,yatra).
parent(aishwarya,yatra).
parent(selva,anjali).
parent(geethanjali,anjali).

father(F,X):-male(F),parent(F,X).
mother(M,X):-female(M),parent(M,X).
sibling(X,Y):-father(Z,X),father(Z,Y),X\=Y.
child(C,P):-parent(P,C).
brother(B,X):-male(B),sibling(B,X).
sister(S,X):-female(S),sibling(S,X).
daughter(D,X):-female(D),parent(X,D).
son(S,X):-male(S),parent(X,S).
spouse(X,Y):-child(Z,X),child(Z,Y).
wife(W,X):-female(W),male(X),spouse(W,X).
husband(H,X):-
male(H),female(X),spouse(H,X).
grandfather(GP,GC):-
male(GP),parent(GP,X),parent(X,GC).
grandmother(GP,GC):-
female(GP),parent(GP,X),parent(X,GC).
grandchild(GC,GP):-grandmother(GP,GC).
grandchild(GC,GP):-grandfather(GP,GC).
aunt(A,X):-
female(A),father(Z,X),brother(Z,A).
aunt(A,X):-
female(A),mother(Z,X),sister(Z,A).
uncle(U,X):-
male(U),father(Z,X),brother(Z,U).
uncle(U,X):-male(U),mother(Z,X),sister(Z,U).
uncle(U,X):-
male(U),father(Z,X),sister(S,Z),husband(U,S).

cousin(X,Y):-
parent(Z,X),parent(P,Y),sibling(Z,P).
```

OUTPUT

1 ?- male(Y).

Y = kasthuriraja ;

Y = dhanush ;

Y = selva ;

Y = yatra ;

Y = linga.

2 ?- female(Y).

Y = vijaya ;

Y = aishwarya ;

Y = geethanjali ;

Y = anjali.

3 ?- parent(dhanush,X).

X = linga ;

X = yatra.

4 ?- parent(selva,anjali).

true.

5 ?- father(selva,anjali).

true.

6 ?- father(kasthuriraja,X).

X = dhanush ;

X = selva.

7 ?- mother(vijaya,X).

X = dhanush ;

X = selva.

8 ?- mother(vijaya,yatra).

false.

9 ?- sibling(yatra,linga).

true .

10 ?- sibiling(dhanush,X).

Correct to: "sibling(dhanush,X)"?

Please answer 'y' or 'n'? yes

X = dhanush .

11 ?- child(selva,kasthuriraja).

true.

12 ?- daughter(geethanjali,X).
false.

RESULT

Thus, the program for Decision – Making tree using Machine Learning was implemented using Prolog.