Vel Tech
Rangarajan Dr. Sagunthala
R&D Institute of Science and Technology
(Deemed to be University Estd. u/s 3 of UGC Act, 1956)

Department of Artificial Intelligence &Machine learning

School of Computing

ACADEMIC YEAR 2024 – 2025

Summer Semester

10211AM211: Artificial Intelligence Techniques

LAB MANUAL

LIST OF EXPERIMENTS

Task No.	Task Name
1	Implementation of Graph search algorithms (**Breadth first search and Depth First Search**) using following constraints.
2	Implementation of **Hill climbing algorithm for Heuristic search** approach using following constraints in python.
3	Implementation of **A * Algorithm** to find the optimal path using Python by following constraints.
4	Implementation of **Mini-Max algorithm** using recursion to search through the Game - tree using python by applying following constraints.
5	Implementation of **Ant Colony Optimization** to Optimize Ride-Sharing Trip Duration using Python by following constraints.
6	Solve a **Map Coloring problem** using constraint satisfaction approach by applying following constraints
7	Implementation of **Monkey Banana Problem** in Goal Stack planning using python by applying following constraints.
8	Implementation of **N-queen problem using backtracking algorithm** using prolog In the 4 Queens problem the object is to place 4 queens on a chessboard in such a way that no queens can capture a piece.
9	To Build an Intelligent **Chatbot system** with Python and Dialog-flow using Interactive Text Mining Framework for Exploration of Semantic Flows in Large Corpus of Text.
10	Implement simple fact using python

TASK:1

Implementation of Graph search algorithms (**Breadth first search and Depth First Search**) using following constraints.

Aim: To Implement of Graph search algorithms (Breadth first search and Depth First Search) using Python.

Task 1A

Algorithm:

BFS

Step 1: Start by putting any one of the graph's vertices at the back of the queue.

Step 2: Now take the front item of the queue and add it to the visited list.

Step 3: Create a list of that vertex's adjacent nodes. Add those which are not within the visited list to the rear of the queue.

Step 4: Keep continuing steps two and three till the queue is empty.

Program

```
graph = {
  '5' : ['3','7'],
  '3' : ['2', '4'],
  '7' : ['8'],
  '2' : [],
  '4' : ['8'],
  '8' : []
}

visited = [] # List for visited nodes.
queue = []     #Initialize a queue

def bfs(visited, graph, node): #function for BFS
  visited.append(node)
  queue.append(node)

  while queue:          # Creating loop to visit each node
    m = queue.pop(0)
```

```
    print (m, end = " ")

    for neighbour in graph[m]:
      if neighbour not in visited:
        visited.append(neighbour)
        queue.append(neighbour)

# Driver Code
print("Following is the Breadth-First Search")
bfs(visited, graph, '5')    # function calling
```

Output:
Following is the Breadth-First Search
5 3 7 2 4 8

Task1 b

Algorithm

DFS –

Step 1: Declare a queue and insert the starting Vertex.

Step 2: Initialize a visited array and mark the starting Vertex as visited.

Step3: Remove the First vertex of queue.

Step 4: Mark that vertex as visited

Step 5: Insert all the unvisited neighbors of the vertex into queue.

Step 6: stop.

Program

```
graph = {
  '5' : ['3','7'],
  '3' : ['2', '4'],
  '7' : ['8'],
  '2' : [],
  '4' : ['8'],
  '8' : []
}
```

```
def dfs(graph, start):
    visited = set()
    stack = [start]

    while stack:
        node = stack.pop()

        if node not in visited:
            print(node)
            visited.add(node)

             for neighbor in graph[node]:
                 if neighbor not in visited:
                     stack.append(neighbor)

print("Following is the Depth-First Search")
dfs(graph, '5')
```

Output:

```
Following is the Depth-First Search
5
3
2
4
8
7
```

Result:

Thus the Implementation of Graph search algorithms (Breadth first search and Depth First Search) using Python was successfully executed and output was verified.

Task :2

Implementation of **Hill climbing algorithm for Heuristic search** approach using following constraints in python.

Aim: To Implement Hill climbing algorithm for Heuristic search approach for travelling salesman problem using python

Algorithm:

Step 1: start

Step 2: define TSP with (graph, s) and assign value for vertex.

Step 3: store all vertex apart from source vertex.

Step 4: store minimum weight hamiltonian cycle and assign permutation (vertex).

Step 5: store current path weight (cost) and compute current path weight.

Step 6: Update minimum and matrix representation of the graph values and print it.

Step 7: stop

Program:

```
from sys import maxsize
from itertools import permutations
V = 4
def travellingSalesmanProblem(graph, s):
    vertex = []  # Changed variable name to lowercase 'vertex'
    for i in range(V):  # Fixed capitalization of 'for'
        if i != s:  # Changed capitalization of 'if'
            vertex.append(i)


    min_path = maxsize  # Changed variable name to lowercase 'min_path'
    next_permutation = permutations(vertex)  # Changed variable name to lowercase 'next_permutation'
    for i in next_permutation:  # Fixed capitalization of 'for'
        current_pathweight = 0  # Changed variable name to lowercase 'current_pathweight'
        k = s  # Changed variable name to lowercase 'k'
        for j in i:  # Fixed capitalization of 'for'
            current_pathweight += graph[k][j]
            k = j
```

```
        current_pathweight += graph[k][s]
        min_path = min(min_path, current_pathweight)

    return min_path  # Changed capitalization of 'return'
if __name___== "__main__":
    graph = [[0, 10, 15, 20], [10, 0, 35, 25],
             [15, 35, 0, 30], [20, 25, 30, 0]]
    s = 0
    print(travellingSalesmanProblem(graph, s))  # Changed capitalization of 'print'
```

Output:

80

Result:

Thus the Implementation of Hill climbing algorithm for Heuristic search approach for travelling salesman problem using python was successfully executed and output was verified.

TASK:3

Implementation of **A * Algorithm** to find the optimal path using Python by following constraints.

3(A) A* Algorithm

Aim : To implement of A * Algorithm to find the optimal path using Jupiter notebook.

Algorithm:

Step 1: start

Step 2: Place the starting node into open and find its f(n) [start node] value.

Step 3: Remove the node from OPEN, having the smallest f(n) value, if it is x goal node, then stop and return to success.

Step 4: Else remove the node from OPEN, and find all its successors.

Step 5: Find the f(n) value of all the successors, Place them into OPEN and place the removed node into close

Step 6: Go to step 2.

Step 7: Exit.

Program :

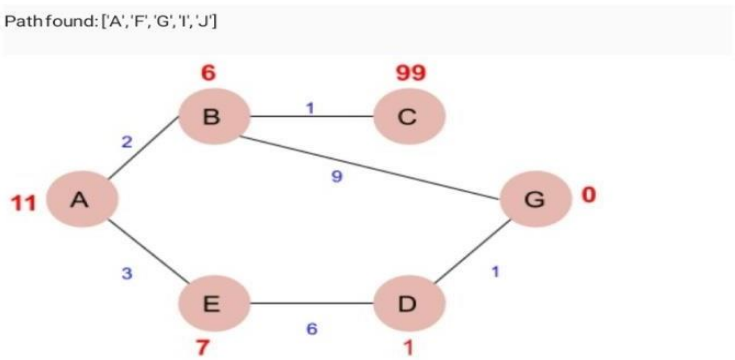
```
def aStarAlgo(start_node, stop_node):
    open_set = set([start_node])
    closed_set = set()
    g = {}  # store distance from starting node
    parents = {}  # parents contain an adjacency map of all nodes

    # distance of starting node from itself is zero
    g[start_node] = 0
    # start_node is the root node, so it has no parent nodes
    # so start_node is set to its own parent node
    parents[start_node] = start_node

    while len(open_set) > 0:
        n = None
        # node with the lowest f() is found
        for v in open_set:
```

```
            if n is None or g[v] + heuristic(v) < g[n] + heuristic(n):
                n = v

        if n == stop_node or n is None or Graph_nodes[n] is None:
            break
        else:
            for m, weight in get_neighbors(n):
                # nodes 'm' not in open_set and closed_set are added to open_set
                # n is set as its parent
                if m not in open_set and m not in closed_set:
                    open_set.add(m)
                    parents[m] = n
                    g[m] = g[n] + weight
                # for each node m, compare its distance from start i.e g(m)
                # to the from start through n node
                else:
                    if g[m] > g[n] + weight:
                        # update g(m)
                        g[m] = g[n] + weight
                        # change parent of m to n
                        parents[m] = n
                        # if m is in closed_set, remove and add to open_set
                        if m in closed_set:
                            closed_set.remove(m)
                            open_set.add(m)

        # remove n from the open_set and add it to closed_set
        # because all of its neighbors were inspected
        open_set.remove(n)
        closed_set.add(n)

    if n is None:
        print('Path does not exist!')
        return None
```

```
    # if the current node is the stop_node,
    # then we begin reconstructing the path from it to the start_node
    if n == stop_node:
        path = []
        while parents[n] != n:
            path.append(n)
            n = parents[n]
        path.append(start_node)
        path.reverse()
        print('Path found:', path)
        return path

    print('Path does not exist!')
    return None


# define function to return neighbors and their distances from the passed node
def get_neighbors(v):
    if v in Graph_nodes:
        return Graph_nodes[v]
    else:
        return None


# for simplicity, we'll consider heuristic distances given
# and this function returns heuristic distance for all nodes
def heuristic(n):
    h_dist = {
        'A': 11,
        'B': 6,
        'C': 5,
        'D': 7,
        'E': 3,
        'F': 6,
        'G': 5,
```

```
        'H': 3,
        'I': 1,
        'J': 0
    }
    return h_dist[n]


# Describe your graph here
Graph_nodes = {
    'A': [('B', 6), ('F', 3)],
    'B': [('A', 6), ('C', 3), ('D', 2)],
    'C': [('B', 3), ('D', 1), ('E', 5)],
    'D': [('B', 2), ('C', 1), ('E', 8)],
    'E': [('C', 5), ('D', 8), ('I', 5), ('J', 5)],
    'F': [('A', 3), ('G', 1), ('H', 7)],
    'G': [('F', 1), ('I', 3)],
    'H': [('F', 7), ('I', 2)],
    'I': [('E', 5), ('G', 3), ('H', 2), ('J', 3)],
}

print("Following is the A* Algorithm:")
aStarAlgo('A', 'J')
```

Output:



Result:

Thus the Implementation of A * Algorithm to find the optimal path using Python Was successfully executed and output was verified.

3(B) – Simplified A* Algorithm.

Aim: To implement the simplified A*Algorithm using Jupiter notebook.

Algorithm:

Step 1 : start.

Step 2: place the starting node into open and find its f(n) value

Step 3: Remove the node from OPEN , having the smallest f(n) value, if it is a goal node , then stop and return to success.

Step 4: else remove the node from OPEN, and find all its successors

Step 5: Find the f(n) value of all the successors, Place them into OPEN and place the removed node into close

Step 6: Go to step 2.

Step 7: Exit.

Program:

```
def aStarAlgo(start_node, stop_node):
    open_set = set([start_node])
    closed_set = set()
    g = {}  # store distance from starting node
    parents = {}  # parents contain an adjacency map of all nodes

    # distance of starting node from itself is zero
    g[start_node] = 0
    # start_node is the root node, so it has no parent nodes
    # so start_node is set to its own parent node
    parents[start_node] = start_node

    while len(open_set) > 0:
        n = None
        # node with the lowest f() is found
        for v in open_set:
            if n is None or g[v] + heuristic(v) < g[n] + heuristic(n):
                n = v

        if n == stop_node or n is None or n not in Graph_nodes:
```

```
            break
        else:
            for m, weight in get_neighbors(n):
                # nodes 'm' not in open_set and closed_set are added to open_set
                # n is set as its parent
                if m not in open_set and m not in closed_set:
                    open_set.add(m)
                    parents[m] = n
                    g[m] = g[n] + weight
                # for each node m, compare its distance from start i.e g(m)
                # to the from start through n node
                else:
                    if g[m] > g[n] + weight:
                        # update g(m)
                        g[m] = g[n] + weight
                        # change parent of m to n
                        parents[m] = n
                        # if m is in closed_set, remove and add to open_set
                        if m in closed_set:
                            closed_set.remove(m)
                            open_set.add(m)

        # remove n from the open_set and add it to closed_set
        # because all of its neighbors were inspected
        open_set.remove(n)
        closed_set.add(n)

    if n is None:
        print('Path does not exist!')
        return None

    # if the current node is the stop_node,
    # then we begin reconstructing the path from it to the start_node
    if n == stop_node:
        path = []
```

```
        while parents[n] != n:
            path.append(n)
            n = parents[n]
        path.append(start_node)
        path.reverse()
        print('Path found:', path)
        return path

    print('Path does not exist!')
    return None

# define function to return neighbors and their distances from the passed node
def get_neighbors(v):
    if v in Graph_nodes:
        return Graph_nodes[v]
    else:
        return None

# for simplicity, we'll consider heuristic distances given
# and this function returns heuristic distance for all nodes
def heuristic(n):
    h_dist = {
        'A': 11,
        'B': 6,
        'C': 99,
        'D': 1,
        'E': 7,
        'G': 0
    }
    return h_dist[n]

# Describe your graph here
Graph_nodes = {
    'A': [('B', 2), ('E', 3)],
    'B': [('A', 2), ('C', 1), ('G', 9)],
```

```
    'C': [('B', 1)],
    'D': [('E', 6), ('G', 1)],
    'E': [('A', 3), ('D', 6)],
    'G': [('B', 9), ('D', 1)]
}

print("Following is the A* Algorithm:")
aStarAlgo('A', 'G')
```

Output:

Path found: [‘A’, ‘E’, ‘D’, ‘G’]

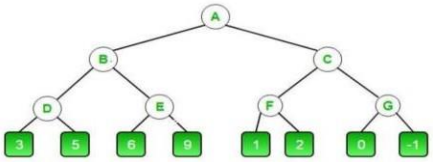
Result:

Thus the implementation of the simplified A*Algorithm using Jupiter notebook was successfully executed and output was verified.

TASK:4

Implementation of **Mini-Max algorithm** using recursion to search through the Game - tree using python by applying following constraints.

Aim: To create a program for searching problem using Mini-max algorithm with Alpha-Beta pruning approach.



Algorithm:

Step 1: At the first stop the, Max player will start first move from node A where α=-infinity and beta = +infinity, these values of alpha and beta passed down to node B. Node B transmitting the identical value to its off spring D.

Step2: As Max's turn at Node D approaches, the value of α will be decided. when the value of α is compared to 3 then 5 the value at node D is max (3,5) = 5. Hence the node value is also 5

Step 3: The algorithm returns to node B, where the value of beta will change since this a turn of min

Step 4: Max will take over at node E and change alpha's value.

Step 5: We know traverse the tree backward, from node B to node A

Step 6: As a result, in this case, the ideal value for the maximizer is 5.

Program:

```
# Initial values of Alpha and Beta
MAX, MIN = 1000, -1000
# Returns optimal value for current player
# (Initially called for root and maximizer)

def minimax(depth, nodeIndex, maximizingPlayer, values, alpha, beta):
    # Terminating condition. i.e. leaf node is reached
    if depth == 3:
        return values[nodeIndex]

    if maximizingPlayer:
        best = MIN
        # Recur for left and right children
```

```
        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)

            # Alpha Beta Pruning
            if beta <= alpha:
                break

        return best
    else:
        best = MAX
        # Recur for left and right children
        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)

            # Alpha Beta Pruning
            if beta <= alpha:
                break

        return best

# Driver Code
if __name__ == "__main__":
    values = [3, 5, 6, 9, 1, 2, 0, -1]
    print("The optimal value is:", minimax(0, 0, True, values, MIN, MAX))
```

Sample output :

The optimal value is :5

Result:

Thus creating a program for searching problem using Mini-max algorithm with Alpha-Beta pruning approach was successfully executed and output was verified.

TASK:5

Implementation of **Ant Colony Optimization** to Optimize Ride-Sharing Trip Duration using Python by following constraints.

Aim: To Implement Ant Colony Optimization to Optimize Ride-Sharing Trip Duration using Python.

Algorithm:

Step 1:[Initialization]

t=0;NC=0;for each edge (I,j),initialize trail intensity.

Step 2:[starting node]

For each ant k:place ant k on a randomly chosen city and store this information in tablet.

Step 3: Build a tour for each ant.

Step 4:global update of trail.

Step 5: termination conditions,memorize the shortest tour found to this point.

Program:

```
import numpy as np
from numpy import inf

#given values for the problems

d = np.array([[0,10,12,11,14]
        ,[10,0,13,15,8]
        ,[12,13,0,9,14]
        ,[11,15,9,0,16]
        ,[14,8,14,16,0]])

iteration = 100
n_ants = 5
n_citys = 5

# intialization part

m = n_ants
```

```
n = n_citys
e = .5          #evaporation rate
alpha = 1       #pheromone factor
beta = 2        #visibility factor

#calculating the visibility of the next city visibility(i,j)=1/d(i,j)

visibility = 1/d
visibility[visibility == inf ] = 0

#intializing pheromne present at the paths to the cities

pheromne = .1*np.ones((m,n))

#intializing the rute of the ants with size rute(n_ants,n_citys+1)
#note adding 1 because we want to come back to the source city

rute = np.ones((m,n+1))

for ite in range(iteration):

    rute[:,0] = 1          #initial starting and ending positon of every ants '1' i.e city '1'

    for i in range(m):

        temp_visibility = np.array(visibility)          #creating a copy of visibility

        for j in range(n-1):
            #print(rute)

            combine_feature = np.zeros(5)     #intializing combine_feature array to zero
            cum_prob = np.zeros(5)            #intializing cummulative probability array to zeros

            cur_loc = int(rute[i,j]-1)        #current city of the ant
```

```
            temp_visibility[:,cur_loc] = 0     #making visibility of the current city as zero

            p_feature = np.power(pheromne[cur_loc,:],beta)         #calculating pheromne feature
            v_feature = np.power(temp_visibility[cur_loc,:],alpha)  #calculating visibility feature

            p_feature = p_feature[:,np.newaxis]                  #adding axis to make a size[5,1]
            v_feature = v_feature[:,np.newaxis]                  #adding axis to make a size[5,1]

            combine_feature = np.multiply(p_feature,v_feature)   #calculating the combine feature

            total = np.sum(combine_feature)                      #sum of all the feature

            probs = combine_feature/total          #finding probability of element probs(i) = comine_feature(i)/total

            cum_prob = np.cumsum(probs)     #calculating cummulative sum
            #print(cum_prob)
            r = np.random.random_sample()  #randon no in [0,1)
            #print(r)
            city = np.nonzero(cum_prob>r)[0][0]+1        #finding the next city having probability higher then random(r)
            #print(city)

            rute[i,j+1] = city             #adding city to route

        left = list(set([i for i in range(1,n+1)])-set(rute[i,:-2]))[0]      #finding the last untraversed city to route

        rute[i,-2] = left                  #adding untraversed city to route

    rute_opt = np.array(rute)              #intializing optimal route

    dist_cost = np.zeros((m,1))            #intializing total_distance_of_tour with zero

    for i in range(m):
```

```
    s = 0
    for j in range(n-1):

        s = s + d[int(rute_opt[i,j])-1,int(rute_opt[i,j+1])-1]  #calcualting total tour distance

    dist_cost[i]=s                  #storing distance of tour for 'i'th ant at location 'i'

  dist_min_loc = np.argmin(dist_cost)           #finding location of minimum of dist_cost
  dist_min_cost = dist_cost[dist_min_loc]        #finging min of dist_cost

  best_route = rute[dist_min_loc,:]             #intializing current traversed as best route
  pheromne = (1-e)*pheromne                      #evaporation of pheromne with (1-e)

  for i in range(m):
    for j in range(n-1):
      dt = 1/dist_cost[i]
      pheromne[int(rute_opt[i,j])-1,int(rute_opt[i,j+1])-1]     =     pheromne[int(rute_opt[i,j])-
1,int(rute_opt[i,j+1])-1] + dt
      #updating the pheromne with delta_distance
      #delta_distance will be more with min_dist i.e adding more weight to that route
peromne


print('route of all the ants at the end :')
print(rute_opt)
print()
print('best path :',best_route)
print('cost of the best path',int(dist_min_cost[0]) + d[int(best_route[-2])-1,0])
```

Output:

```
Route of all ants at the end:
[[1.4.3.5.2.1]
 [1.4.3.5.2.1]
 [1.4.3.5.2.1]
 [1.4.3.5.2.1]
 [1.4.3.5.2.1]]
```

Best path: [1.4.3.5.2.1]

Cost of the best path=52.

Result:

Thus the Implementation of Ant Colony Optimization to Optimize Ride-Sharing Trip Duration using Python was successfully executed and output was verified.

TASK:6

Solve a **Map Coloring problem** using constraint satisfaction approach by applying following constraints

Aim: To Solve a Map Coloring problem using constraint satisfaction approach using Graphonline and visualago online simulator

Algorithm:

Step 1: Confirm whether it is valid to color the current vertex worth the current color (by checking whether any of its adjacent vertices are colored with the same color)

Step 2: If yes then color it and otherwise try a different color

Step 3: check if all vertices are colored or not

Step 4: If not then move to the next adjacent uncolored vertex

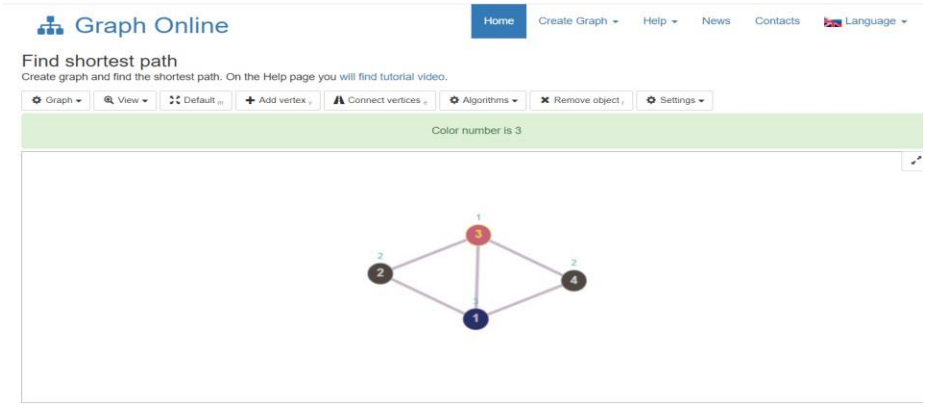
Step 5: Here backtracking means to stop further recursive calls on adjacent vertices.

Program:

```
class Graph:
    def __init__(self, vertices):
        self.v = vertices
        self.graph = [[0 for column in range(vertices)] for row in range(vertices)]

    # A utility function to check if the current color assignment is safe for vertex v
    def is_safe(self, v, color, c):
        for i in range(self.v):
            if self.graph[v][i] == 1 and color[i] == c:
                return False
        return True

    # A recursive utility function to solve m-coloring problem
    def graph_color_util(self, m, color, v):
        if v == self.v:
            return True

        for c in range(1, m+1):
            if self.is_safe(v, color, c):
                color[v] = c
```

```
            if self.graph_color_util(m, color, v+1):
                return True
            color[v] = 0

    def graph_coloring(self, m):
        color = [0] * self.v
        if not self.graph_color_util(m, color, 0):
            return False

        # Print the solution
        print("Solution exists and following are the assigned colors:")
        for c in color:
            print(c, end=" ")

# Driver Code
if __name___== '__main__':
    g = Graph(4)
    g.graph = [[0, 1, 1, 1], [1, 0, 1, 0], [1, 1, 0, 1], [1, 0, 1, 0]]
    m = 3
    # Function call
    g.graph_coloring(m)
```

Output:



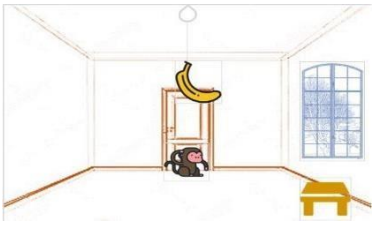
Result:

Thus Solving a Map Coloring problem using constraint satisfaction approach using Graphonline and visulago online simulator was successfully executed and output was verified.

TASK:7

Implementation of **Monkey Banana Problem** in Goal Stack planning using python by applying following constraints.

Aim: To Implement the Monkey Banana Problem in Goal Stack planning using python



Algorithm:

Step 1: when the block is at the middle, and monkey is on top of the block and monkey does not have the banana (i.e. has not state), then using the grasp action,it will change from has not state to have state.

Step 2: from the floor, it can move to the top of the block Lie on top state & by performing the action climb.

Step3: The push or drag operation moves the block from one place to another.

Step 4: Monkey can move from one place to another using walk or move clauses.

Step 5: Another predicate will be canget ().

Program:

```
# Operators
def move(subject, x1, x2):
    return f"Move {subject} from {x1} to {x2}"

def push_box(x1, x2):
    return f"Push box from {x1} to {x2}"

def climb_box(x, direction):
    return f"Climb box at {x} {direction}"

def have_banana(x):
    return f"Have banana at {x}"

# Initial State
```

```
initial_state = {
    'monkeyAt0': True,
    'monkeyLevel': 'Down',
    'bananaAt1': True,
    'boxAt2': True
}

# Goal State
goal_state = {
    'GetBanana': True,
    'at': 1
}

# Planning Algorithm
def plan_actions(initial_state, goal_state):
    actions = []

    # Example planning algorithm to achieve the goal state
    if initial_state['monkeyAt0'] and initial_state['bananaAt1']:
        actions.append(move('Monkey', 0, 1))
        actions.append(climb_box(1, 'Up'))
        actions.append(have_banana(1))

    return actions

# Execute the planning algorithm
actions = plan_actions(initial_state, goal_state)

# Print the actions in the plan
print("Plan:")
for action in actions:
    print(action)
```

Output:

Plan:

Move Monkey from 0 to 1

Climb box at 1 Up

Have banana at 1

Result:

Thus the Implementation the Monkey Banana Problem in Goal Stack planning using python was successfully executed and output was verified.

TASK:8

Implementation of **N-queen problem using backtracking algorithm** using prolog In the 4 Queens problem the object is to place 4 queens on a chessboard in such a way that no queens can capture a piece.

Aim: To Implement N-Queen's problem by using backtracking algorithm using python

Algorithm:

Step 1: k=queen and I is column number in which queen k is placed

Step 2: where x[] is a global array whose first k-1 values have been set

Step 3: Queen-place (k, i) returns true if a queen can be placed in the kth row and ith column otherwise return false

Step 4:ABS (r) returns the absolute value of r.

Step 5: for j<-1 to k-1 do if x[j]=1 or ABS(x[j]-1)= ABS (j-k) then return false

Step 6:for i<-1 to n do if Queen-place (k,i) then x[k] <- i if k=n

then write (x[i---n]) else N-Queen (k+1,n).

Program:

```
# Python program to solve N Queen
# Problem using backtracking

global N
N = 4

def printSolution(board):
    for i in range(N):
        for j in range(N):
            print (board[i][j],end=' ')
        print()

def isSafe(board, row, col):

    # Check this row on left side
    for i in range(col):
        if board[row][i] == 1:
            return False
```

```
    # Check upper diagonal on left side
    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False

    # Check lower diagonal on left side
    for i, j in zip(range(row, N, 1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False

    return True

def solveNQUtil(board, col):

    if col >= N:
        return True

       for i in range(N):

        if isSafe(board, i, col):

            board[i][col] = 1

                if solveNQUtil(board, col + 1) == True:
                return True

                    board[i][col] = 0

        return False

def solveNQ():
    board = [ [0, 0, 0, 0],
              [0, 0, 0, 0],
              [0, 0, 0, 0],
              [0, 0, 0, 0]
```

```
        ]

    if solveNQUtil(board, 0) == False:
        print ("Solution does not exist")
        return False

    printSolution(board)
    return True

solveNQ()
```

Output:

0 0 1 0

1 0 0 0

0 0 0 1

0 1 0 0

Result:

Thus the Implementation of N-queen problem using backtracking algorithm using Python was successfully executed and output was verified.

TASK:9

To Build an Intelligent **Chatbot system** with Python and Dialog-flow using Interactive Text Mining Framework for Exploration of Semantic Flows in Large Corpus of Text.

Aim: To build an intelligent chatbox system with Python and dialog-flow using interactive text mining framework for exploration of semantic flow in large corpus of Text

Algorithm:

Steps to create an intelligent chatbot using OpenAI APIs:

1. Sign up for OpenAI API access at https://beta.openai.com/signup/. Once you sign up, you will receive your API key.

2. Choose the type of chatbot you want to create. For example, you can create an FAQ chatbot, a customer support chatbot, or a conversational chatbot.

3. Use OpenAI's GPT-3 language model to generate responses to user input. You can use the API to train the language model on your chatbot's intended use case/s.

4. Use Natural Language Processing (NLP) techniques to understand user input and provide relevant responses. You can use OpenAI's API to extract entities (such as dates and names) from user input.

5. Use Machine Learning to continually improve the chatbot's ability to understand and respond to user input.

6. Integrate the chatbot with your preferred messaging platform or channel (e.g., web chat, social media, etc.) using API connectors.

7. Test your chatbot frequently, and use user feedback to improve its performance and provide the best possible experience for your users.

a. Simple ChatGPT using openai

Code:

```
Pip install openai

import openai


openai.api_key = "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"


completion = openai.ChatCompletion.create(model="gpt-3.5-turbo", messages=[{"role": "user", "content": "Give me 3 ideas that i could build using openai apis"}])
```

```
print(completion.choices[0].message.content)
```

Output:

1. Personalized Content Recommendation System: Develop an AI-powered content recommendation system that suggests personalized content to users based on their interests and search history. Use OpenAI's language generation APIs to generate relevant content descriptions and summaries, and employ their natural language processing (NLP) APIs to understand user preferences and interests.

2. Intelligent Chatbot: Build a conversational AI-enabled chatbot that can answer customer queries, provide helpful recommendations, and complete transactions seamlessly. Use OpenAI's language processing APIs to train the chatbot to understand user inputs and respond in natural language. Integration with other APIs such as payment gateways and customer databases can make the chatbot efficient and effective.

3. Fraud Detection System: Develop a machine learning model that can identify and prevent fraudulent activities using OpenAI's anomaly detection and classification APIs. Train the model using historical data of fraudulent transactions, and use the APIs to continuously scan for and identify suspicious activities. Such a system can be deployed in a range of applications such as finance or e-commerce platforms.

b. ChatGPT Assistant using openai

Code:

```
import openai


openai.api_key = "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"


messages = []
system_msg = input("What type of chatbot would you like to create?\n")
messages.append({"role": "system", "content": system_msg})


print("Your new assistant is ready! Type your query")
while input != "quit()":
    message = input()
```

```
    messages.append({"role": "user", "content": message})
    response = openai.ChatCompletion.create(model="gpt-3.5-turbo", messages=messages)
    reply = response["choices"][0]["message"]["content"]
    messages.append({"role": "assistant", "content": reply})
    print("\n" + reply + "\n")
```

Output:

What type of chatbot would you like to create?

Nila’s personal chatbot

(ctrl enter)

Your new assistant is ready!

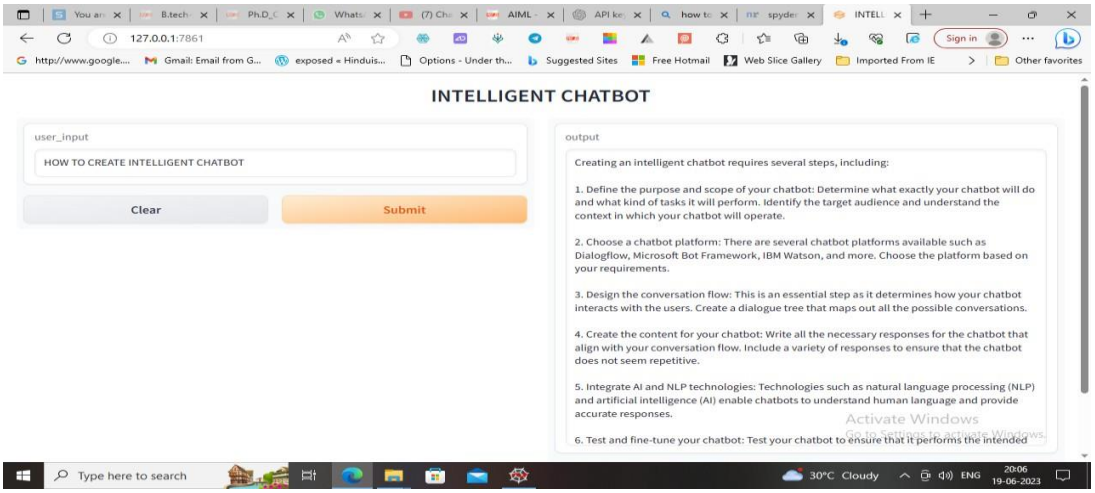
c. CHATBOT CHAT ASSISTANT WEBSITE

Code:

```
import openai

import gradio

openai.api_key = "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"

messages = [{"role": "system", "content": "You are a financial experts that specializes in real estate investment and negotiation"}]

def CustomChatGPT(user_input):
    messages.append({"role": "user", "content": user_input})
    response = openai.ChatCompletion.create(
        model = "gpt-3.5-turbo",
        messages = messages
    )
    ChatGPT_reply = response["choices"][0]["message"]["content"]
    messages.append({"role": "assistant", "content": ChatGPT_reply})
```

```
    return ChatGPT_reply

demo = gradio.Interface(fn=CustomChatGPT, inputs = "text", outputs = "text", title = "INTELLIGENT CHATBOT")

demo.launch(share=True)
```

OUTPUT:



Result:

Thus to build an intelligent chatbox system with Python and dialogue flow was successfully completed and output was verified.

Task 10

Implement simple facts using python

Aim: To implement simple facts and verify using python

Algorithm:

Step:1Define a list of facts containing the statements to be verified.

Step:2 Create a function named verify_fact that takes a fact as input and returns a boolean value indicating whether the fact is true or false.

Step:3 In the verify_fact function:

a. Remove the trailing period from the fact using the rstrip function.

b. Check the fact against the known conditions to determine its truth value. You can use conditional statements (if, elif, else) for this.

- If the fact matches a known condition, return True to indicate that the fact is true.
- If the fact does not match any known condition, return False to indicate that the fact is false.

Step:4 Iterate over each fact in the list of facts:

a. Call the verify_fact function for each fact.

b. Print the fact and the corresponding "Yes" or "No" based on its truth value.

Program:

```
# Define a list of facts
facts = [
    "john_is_cold.",              # john is cold
    "raining.",                   # it is raining
    "john_Forgot_His_Raincoat.",  # john forgot his raincoat
    "fred_lost_his_car_keys.",    # fred lost his car keys
    "peter_footballer."           # peter plays football
]


# Function to check if a fact is true
def verify_fact(fact):
    # Remove the trailing period
    fact = fact.rstrip(".")

    # Perform some logic to verify the fact
```

```
    if fact == "john_Forgot_His_Raincoat":
        return True
    elif fact == "raining":
        return True
    elif fact == "foggy":
        return True
    elif fact == "Cloudy":
        return False  # Assume it's not cloudy
    else:
        return False

# Verify each fact
for fact in facts:
    if verify_fact(fact):
        print(f"{fact} - Yes")
    else:
        print(f"{fact} - No")
```

Output:

john_is_cold. - No

raining. - Yes

john_Forgot_His_Raincoat. - Yes

fred_lost_his_car_keys. - No

peter_footballer. – No

Result:

Thus the implementation of simple facts using python was successfully executed and output was verified.

USE CASE

Intelligent chatbox using Dialog Flow

Aim:- To build an Intelligent chatbox system with python and dialogue flow using Interactive text mining framework for exploration of semantic flour in large corpus of Text.

Procedure:

Step I: Create an Agent.

An Agent is an intelligent program inside. The chatbox, it's that program that interacts with the clients or users. To create an Agent, go to the left section of your screen and click on first button below the Dialog flow logo and go down to the create new agent button, After that the new screen will be loaded, and you will be ask to specify the name of Agent, the language that it should be speak and time zone. For me I type FOOD -bot. for the name and rest, I leave the default values.After that, you must click on CREATE button and DialogFlow will create an agent for your chatbox.

Step 2: Create Intents

Intents is use by the Chabot to understand what the Clients or users want. It's inside the Intents that we should provide to the chatbot the examples of phrases that the clients may ask and some responses that chatbot should use to answer to the clients. Let's chow how we can do it.

NOTE:

When we create a new agent, it comes with two defaults intents named Default fallback Intent and Default welcome Intent.

For create a new Intent, click on the Create Intent button.

After that, you must give the name of your intent. Then go to Training phrases section and click on add Training phrases. This section concerns the way where you should give the example of phrase which represents the different questions that clients may ask to chatbot. We recommend giving many examples to make your chatbot very powerful.

We have added some phrases that clients may ask to our chatbox, for your own chatbox, feel free to add another phrase to improve the power of your chatbot. Dialog Flow recognizes three types of entities such as systems, entities, developer entities, and session entities. This night and today are recognized as systems entities, it refers to date or period of time. Go down to the Response section and click on Add response button, add some responses statements. Start with $ symbol these expressions are considered as variables like $time-Period, $date –time.

Step 3: **Creation of entities**

In reality, entities are Keywords that help Agent to recognize what client wants. To create it. Click on Entities button. After specify the name of entity – your entity, because you have to use it as variable. Make sure to case Define synonyms before, then click on save button.

Step 4: Add our entities inside training phrases expressions.

Back to the Food Intent Interface and go to the training phrases section. When you are there, select an expression, and Inside this expression select word

For the words booking, Food and reserve table.

Step 5: Definition of parameters and actions.

It’s not required, but in some cases, it will be very Important to obligate the user to give to chatbot, some inform Go down to the actions and parameters section, always Inside The foodbot Interface.

For our chatbot, we want that clients provide the reserve table and date of reservation. Make sure to check it. After that, we should specify the prompt text that Agent should display to client when they haven’t specified the required parameters. You need to click on Define prompts. Space on right place of this section, after defining prompt text, close the box dialog.

For the date-time parameter. After this, save the Intent. Now you can test your chatbot.