Ex1: Date: 11.01.2022

Toy Problem Using AI Vacuum Cleaner

AIM: Given 2 rooms in which either one or both is dirty. The vacuum cleaner is in one room. The Vacuum Cleaner can only move right and left. The past cost for each movement is 1. The task is to clean both rooms and achieve the Goal State and print the final cost.

TOOL: GDB Compiler, AWS

ALGORITHM:

1. Initialize cost to 0.
2. Take the input for vacuum cleaner location and state of the room.
3. Check if the vacuum location is dirty or not. If dirty then clean and move to the next room. Else directly move to next room.
4. Check if the next room is dirty. If dirty then clean it. Else update performance cost and print.
5. Repeat steps 3-4 for different initial states.

CODE:

```
def vacuum_world():
      # initializing goal_state
      # 0 indicates Clean and 1 indicates Dirty
   goal_state = {'A': '0', 'B': '0'}
   cost = 0

   location_input = input("Enter Location of Vacuum") #user_input of location vacuum is
placed
   status_input = input("Enter status of " + location_input) #user_input if location is dirty
or clean
   status_input_complement = input("Enter status of other room")
   print("Initial Location Condition" + "{ A:" + status_input + "," + "B:" +
status_input_complement + " }")

   if location_input == 'A':
      # Location A is Dirty.
      print("Vacuum is placed in Location A")
      if status_input == '1':
         print("Location A is Dirty.")
```

```
        # suck the dirt  and mark it as clean
        goal_state['A'] = '0'
        cost += 1                    #cost for suck
        print("Cost for CLEANING A " + str(cost))
        print("Location A has been Cleaned.")

        if status_input_complement == '1':
            # if B is Dirty
            print("Location B is Dirty.")
            print("Moving right to the Location B. ")
            cost += 1                    #cost for moving right
            print("COST for moving RIGHT" + str(cost))
            # suck the dirt and mark it as clean
            goal_state['B'] = '0'
            cost += 1                    #cost for suck
            print("COST for SUCK " + str(cost))
            print("Location B has been Cleaned. ")
        else:
            print("No action" + str(cost))
            # suck and mark clean
            print("Location B is already clean.")

    if status_input == '0':
        print("Location A is already clean ")
        if status_input_complement == '1':# if B is Dirty
            print("Location B is Dirty.")
            print("Moving RIGHT to the Location B. ")
            cost += 1                    #cost for moving right
            print("COST for moving RIGHT " + str(cost))
            # suck the dirt and mark it as clean
            goal_state['B'] = '0'
            cost += 1                    #cost for suck
            print("Cost for SUCK" + str(cost))
            print("Location B has been Cleaned. ")
        else:
            print("No action " + str(cost))
            print(cost)
            # suck and mark clean
            print("Location B is already clean.")
```

```
else:
    print("Vacuum is placed in location B")
    # Location B is Dirty.
    if status_input == '1':
        print("Location B is Dirty.")
        # suck the dirt  and mark it as clean
        goal_state['B'] = '0'
        cost += 1  # cost for suck
        print("COST for CLEANING " + str(cost))
        print("Location B has been Cleaned.")

        if status_input_complement == '1':
            # if A is Dirty
            print("Location A is Dirty.")
            print("Moving LEFT to the Location A. ")
            cost += 1  # cost for moving right
            print("COST for moving LEFT" + str(cost))
            # suck the dirt and mark it as clean
            goal_state['A'] = '0'
            cost += 1  # cost for suck
            print("COST for SUCK " + str(cost))
            print("Location A has been Cleaned.")

    else:
        print(cost)
        # suck and mark clean
        print("Location B is already clean.")

        if status_input_complement == '1':  # if A is Dirty
            print("Location A is Dirty.")
            print("Moving LEFT to the Location A. ")
            cost += 1  # cost for moving right
            print("COST for moving LEFT " + str(cost))
            # suck the dirt and mark it as clean
            goal_state['A'] = '0'
            cost += 1  # cost for suck
            print("Cost for SUCK " + str(cost))
            print("Location A has been Cleaned. ")
        else:
            print("No action " + str(cost))
```

```
        # suck and mark clean
        print("Location A is already clean.")

    # done cleaning
    print("GOAL STATE: ")
    print(goal_state)
    print("Performance Measurement: " + str(cost))

vacuum_world()
```

TIME COMPLEXITY: Constant Runtime

SPACE COMPLEXITY: Constant space

OUTPUT:
Enter location of Vacuum A
Enter status of A 1
Enter status of other room 1
Initial location condition (A:1, B:1)
Vacuum is placed in location B
Location B is dirty
Cost for cleaning 1
Location B has been cleaned
Location A is dirty
Cost for SUCK 3
Location A has been cleaned
GOAL STATE:
{‘A’ : ‘0’ , ‘B’: ‘0’}
Performance Measurement : 3

RESULT: Successfully found out the performance cost of the vacuum problem using AI & achieved the goal state.

Ex2: Date: 28.01.2022

GRAPH COLORING USING PYTHON

AIM: When Given a graph, implement a program to colour it such that no two adjacent vertices have the same colour.

TOOL: AWS , Python3

ALGORITHM:

1. Arrange the vertices of the graph in some random order.
2. Choose the first vertex and colour it with the first colour.
3. Choose the next vertex and colour it with the lowest numbered colour that has not been coloured on the vertices adjacent to it, if all the adjacent vertices are coloured with this colour, assign a new colour to it. Repeat this step until all the vertices are coloured.

CODE:

```
class Graph:

    # Constructor
    def __init__(self, edges, N):

        self.adj = [[] for _ in range(N)]

        # add edges to the undirected graph
        for (src, dest) in edges:
            self.adj[src].append(dest)
            self.adj[dest].append(src)



def colorGraph(graph):

    # stores color assigned to each vertex
    result = {}
    print("The vertices adjacent to each other are as follows")
    print([(0, 1), (0, 4), (0, 5), (4, 5), (1, 4), (1, 3), (2, 3), (2, 4)])
    # assign color to vertex one by one
```

```
    for u in range(N):

        # set to store color of adjacent vertices of u
        # check colors of adjacent vertices of u and store in set
        assigned = set([result.get(i) for i in graph.adj[u] if i in result])

        # check for first free color
        color = 1
        for c in assigned:
            if color != c:
                break
            color = color + 1

        # assigns vertex u the first available color
        result[u] = color

    for v in range(N):
        print("Color assigned to vertex", v, "is", colors[result[v]])
    print("Since no adjacent vertex has same colour, graph colouring is not violated")

# Greedy coloring of graph
if __name__ == '__main__':

    # Add more colors for graphs with many more vertices
    colors = ["", "BLUE", "GREEN", "RED", "YELLOW", "ORANGE", "PINK",
              "BLACK", "BROWN", "WHITE", "PURPLE", "VIOLET"]

    #  of graph edges as per above diagram
    edges = [(0, 1), (0, 4), (0, 5), (4, 5), (1, 4), (1, 3), (2, 3), (2, 4)]

    # Set number of vertices in the graph
    N = 6

    # create a graph from edges
    graph = Graph(edges, N)

    # colour graph using greedy algorithm
    colourGraph(graph)
```

TIME COMPLEXITY: O(m^V)
There is a total O(m^V) combination of colours. So the time complexity is O(m^V)

SPACE COMPLEXITY: O(V)
graphColoring() function will require O(V) space.

OUTPUT:
The vertices adjacent to each other are as follows:
[(0, 1), (0, 4), (0, 5), (4, 5), (1, 4), (1, 3), (2, 3), (2, 4)]

Color assigned to vertex 1 is RED
Color assigned to vertex 2 is GREEN
Color assigned to vertex 3 is YELLOW
Color assigned to vertex 4 is YELLOW
Color assigned to vertex 5 is RED
Rules are not Violated

RESULT: Graph colouring problem was solved & visualized in an optimized way using greedy approach in python.

Ex3: Date:04.02.2022

CONSTRAINT SATISFACTION PROBLEM

AIM: To solve and execute the Cryptarithmetic Problem SEND + MORE = MONEY using an efficient algorithm in AI

TOOL: AWS, Python3

ALGORITHM:

1. M=1, Since it is only carry over possible from the sum of 2 single-digit numbers in 4.
2. To produce a carry from 4 to 5 ‘S+M’ is at least 9 so ‘S=8or9’ so ‘S+M=9or10’ so ‘O=0or1’.But ‘M=1’,so’0’=’0’.
3. If there is carry from 3 to 4 then ‘E=9’ and so ‘N=0’ but ‘0’=’0’ so there is no carry and ‘S=9’ and ‘C3=0’.
4. If there is no carry from columns 2 to 3 then E=N is impossible, therefore there is carry and N=E+1 and C2=1.
5. If there is carry from 1 to 2 then N+R=E mod 10 and N=E+1 so E+1+R=E mod 10, so R=9 but S=9, so there must be carry from column 1 to 2. Therefore C1=1 and R=8.
6. To produce carry C1=1 from 1 to 2, we must have D+E=10+y as y cannot be 0/1 so D+E is at least 12. As D is at most 7 and E is at least 5. N is almost 7 and N=E+1 so E=5 or 6.

CODE:

```
import itertools
def get_value(word, substitution):
    s = 0
    factor = 1
    for letter in reversed(word):
        s += factor * substitution[letter]
        factor *= 10
    return s
def solve2(equation):
```

```
    # split equation in left and right
    left, right = equation.lower().replace(' ', '').split('=')
    # split words in left part
    left = left.split('+')
    # create list of used letters
    letters = set(right)
    for word in left:
        for letter in word:
            letters.add(letter)
    letters = list(letters)

    digits = range(10)
    for perm in itertools.permutations(digits, len(letters)):
        sol = dict(zip(letters, perm))

        if sum(get_value(word, sol) for word in left) == get_value(right, sol):
            print(' + '.join(str(get_value(word, sol)) for word in left) + " = {} (mapping:
{})".format(get_value(right, sol), sol))

if __name__ == '__main__':
    solve2('SEND + MORE = MONEY')
```

TIME COMPLEXITY: O(N^2)

SPACE COMPLEXITY: O(N)

OUTPUT:

Enter the expression: SEND + MORE = MONEY
9567 + 1085 = 10652

RESULT:

We have successfully solved and executed cryptarithmetic problems using AI techniques with optimizations.

Ex4: Date: 11.02.2022

BREADTH FIRST SEARCH & DEPTH FIRST SEARCH

AIM:

To implement Breadth-First Search and Depth First Search in Python3 using the AWS platform.

TOOL: AWS, Python3

ALGORITHM:

BFS

1. Initialize a queue for BFS.
2. Enqueue the starting node A and change out status.
3. Repeat step 4 and 5 until the queue becomes empty.
4. Dequeue a Node N. Process it and set its status.
5. Enqueue all the neighbors of N that are in ready state and set their status.
6. END loop and Exit.

DFS

1. **I**nitialize a stack for DFS.
2. Push the starting node A on the stack and change its status.
   Repeat step 4 and 5 until the stack is empty.
3. Pop the top node N and process it then change its status.
4. Push on stack all neighbors of N that are in ready state.
5. END the loop and Exit.

CODE:

```
def dfs(query_node, parents):
    result = {}
    stack = []
    stack.append( (query_node, 0) )
    while len(stack) > 0:
        print("stack=", stack)
        node, dist = stack.pop()
```

```
        result[node] = dist
        if node in parents:
            for parent in parents[node]:
                stack_members = [x[0] for x in stack]
                if parent not in stack_members:
                    stack.append( (parent, dist+1) )
    return result

def bfs(query_node, parents):
    result = {}
    queue = []
    queue.append( (query_node, 0) )
    while queue:
        print("queue=", queue)
        node, dist = queue.pop(0)
        result[node] = dist
        if node in parents:
            for parent in parents[node]:
                queue_members = [x[0] for x in queue]
                if parent not in result and parent not in queue_members:
                    queue.append( (parent, dist+1) )
    return result
if __name__ == "__main__":
    parents = dict()
    parents = {'N1': ['N2', 'N3', 'N4'], 'N3': ['N6', 'N7'], 'N4': ['N3'], 'N5': ['N4', 'N8'], 'N6':
['N13'],
               'N8': ['N9'], 'N9': ['N11'], 'N10': ['N7', 'N9'], 'N11': ['N14'], 'N12': ['N5']}
    print("Depth-first search:")
    dist = dfs('N1', parents)
    print(dist)
    print("Breadth-first search:")
    dist =bfs('N1', parents)
    print(dist)
```

TIME COMPLEXITY:

BFS and DFS:

Time complexity is O(|V|), where |V| is the number of nodes

SPACE COMPLEXITY:

BFS and DFS:

O(|V|) since at worst case you need to hold all vertices in the queue

OUTPUT:

```
Depth-first search:
stack= [('N1', 0)]
stack= [('N2', 1), ('N3', 1), ('N4', 1)]
stack= [('N2', 1), ('N3', 1)]
stack= [('N2', 1), ('N6', 2), ('N7', 2)]
stack= [('N2', 1), ('N6', 2)]
stack= [('N2', 1), ('N13', 3)]
stack= [('N2', 1)]
{'N1': 0, 'N4': 1, 'N3': 1, 'N7': 2, 'N6': 2, 'N13': 3, 'N2': 1}

Breadth-first search:
queue= [('N1', 0)]
queue= [('N2', 1), ('N3', 1), ('N4', 1)]
queue= [('N3', 1), ('N4', 1)]
queue= [('N4', 1), ('N6', 2), ('N7', 2)]
queue= [('N6', 2), ('N7', 2)]
queue= [('N7', 2), ('N13', 3)]
queue= [('N13', 3)]
{'N1': 0, 'N2': 1, 'N3': 1, 'N4': 1, 'N6': 2, 'N7': 2, 'N13': 3}
```

RESULT: We have successfully executed Breadth-First Search and Depth First Search in the AWS platform using Python3.

Ex 5: Date :18.02.2022

BEST FIRST SEARCH & A* SEARCH

AIM: To implement Best First Search and A* Search in Python3 using AWS platform.

TOOL: AWS , Python3

ALGORITHM:

Best First Search

1. Create an Empty Priority Queue pq.
2. Insert start in pq, pq.insert(start).
3. Until Priority Queue is empty
   u=priority queue. Delete M in
   If u is the goal
   exit

A* Algorithm

1. Initialize the open list.
2. Initialize the cloud list and put the starting node on the open list.
3. While the open list is not empty.

CODE:

(BEST FIRST SEARCH)

```
from queue import PriorityQueue
v = 14
graph = [[] for i in range(v)]

def best_first_search(source, target, n):
    visited = [0] * n
    visited[0] = True
    pq = PriorityQueue()
    pq.put((0, source))
    while pq.empty() == False:
        u = pq.get()[1]
```

```
        print(u, end=" ")
        if u == target:
            break

        for v, c in graph[u]:
            if visited[v] == False:
                visited[v] = True
                pq.put((c, v))
    print()

def addedge(x, y, cost):
    graph[x].append((y, cost))
    graph[y].append((x, cost))

addedge(0, 1, 3)
addedge(0, 2, 6)
addedge(0, 3, 5)
addedge(1, 4, 9)
addedge(1, 5, 8)
addedge(2, 6, 12)
addedge(2, 7, 14)
addedge(3, 8, 7)
addedge(8, 9, 5)
addedge(8, 10, 6)
addedge(9, 11, 1)
addedge(9, 12, 10)
addedge(9, 13, 2)

source = 0
target = 9
best_first_search(source, target, v)
```

(A*)

```
import queue as Q
g3 = {'a': [('b', 2), ('c', 2)],
        'b': [('a', 2), ('d', 1)],
        'c': [('a', 2), ('d', 8), ('f', 3)],
        'd': [('b', 1), ('c', 8), ('e', 2), ('S', 3)],
        'e': [('d', 2,), ('h', 8), ('r', 2), ('S', 9)],
        'f': [('c', 3), ('G', 2), ('r', 2)],
        'G': [('f', 2)],
        'h': [('e', 8), ('p', 4), ('q', 4)],
        'p': [('h', 4), ('q', 15), ('S', 1)],
        'q': [('h', 4), ('p', 15)],
        'r': [('e', 2), ('f', 2)],
        'S': [('d', 3), ('e', 9), ('p', 1)]}

heuristic = {'S': 0, 'a': 5, 'b': 7, 'c': 4, 'd': 7, 'e': 5, 'f': 2, 'G': 0, 'h':11, 'p': 14, 'q': 12, 'r': 3}

def astar(graph, start, goal):
    visited = []
    path = []
    prev = {}
    queue = Q.PriorityQueue()
    queue.put((0, start, None))
    h2= 0

    while queue:
        cost, node, prev_n = queue.get()
        if node not in visited:
            visited.append(node)
            prev[node] = prev_n

            if node == goal:
                while prev[node] != None:
                    path += [node]
                    node = prev[node]
```

```
            path += [start]
            return visited, prev, path[::-1]
        for i, c in graph[node]:
            if i not in visited:
                total_cost = cost + c
                h1 = heuristic[i]
                total = total_cost + h1 - heuristic[node]
                queue.put((total, i, node))


visited, prev, path = (astar(g3, 'S', 'G'))
print("The visited nodes are:")
print(visited)

print("\n The path followed is:")
print(path)

print("\n The List of previous nodes are:")
print(prev)
```

TIME COMPLEXITY:

Best First Search

The worst-case time complexity of Greedy best-first search is $O(b^m)$.

A* Search

The time complexity of the A* search algorithm depends on the heuristic function, and the number of nodes expanded is exponential to the depth of solution d. So the time complexity is O(b^d), where b is the branching factor.

SPACE COMPLEXITY:

Best First Search

The worst-case space complexity of Greedy best-first search is $O(b^m)$. Where m is the maximum depth of the search space

A* Search

The space complexity of A* search algorithm is **O(b^d)**

OUTPUT:

Best First Search - 0 1 3 2 8 9
A* Search -
The visited nodes are:
['S', 'd', 'e', 'r', 'b', 'a', 'f', 'G']

The path followed is:
['S', 'd', 'e', 'r', 'f', 'G']

The List of previous nodes are:
{'S': None, 'd': 'S', 'e': 'd', 'r': 'e', 'b': 'd', 'a': 'b', 'f': 'r', 'G': 'f'}

RESULT: We have successfully executed Best First Search and A* in the AWS platform using Python3.

Ex 6: Date: 25.02.2022

AO* SEARCH

AIM: To implement AO* Search in Python3 using the AWS platform.

TOOL: AWS, Python3

ALGORITHM:

1. Initialize the graph to start Node.
2. Traverse the graph following the current path accumulate nodes that have not yet been expanded or solved.
3. Pick any of these nodes and expand it and if it has no successor call this value FUTILITY otherwise calculate only for each successor.
4. If f is 0 then mark the node as solved.
5. Change the values of f for the newly created node to its successors by backpropagation.
6. Whenever possible use the most promising routes and if the node is marked as solved then mark the present node as solved.
7. If starting node is solved or the value is greater than futility stop, else repeat from step 2.

CODE:

```
class Graph:
    def __init__(self, graph, heuristicNodeList, startNode):
        self.graph = graph
        self.H=heuristicNodeList
        self.start=startNode
        self.parent={}
        self.status={}
        self.solutionGraph={}

    def applyAOStar(self):
        self.aoStar(self.start, False)
```

```
    def getNeighbors(self, v):
        return self.graph.get(v,'')

    def getStatus(self,v):
        return self.status.get(v,0)

    def setStatus(self,v, val):
        self.status[v]=val

    def getHeuristicNodeValue(self, n):
        return self.H.get(n,0)

    def setHeuristicNodeValue(self, n, value):
        self.H[n]=value



    def printSolution(self):
        print("FOR GRAPH SOLUTION, TRAVERSE THE GRAPH FROM THE START
NODE:",self.start)
        print(self.solutionGraph)

    def computeMinimumCostChildNodes(self, v):  # Computes the Minimum Cost of
child nodes of a given node v
        minimumCost=0
        costToChildNodeListDict={}
        costToChildNodeListDict[minimumCost]=[]
        flag=True
        for nodeInfoTupleList in self.getNeighbors(v):  # iterate over all the set of child
node/s
            cost=0
            nodeList=[]
            for c, weight in nodeInfoTupleList:
                cost=cost+self.getHeuristicNodeValue(c)+weight
```

```
            nodeList.append(c)

        if flag==True:                     # initialize Minimum Cost with the cost of first set of
child node/s
            minimumCost=cost
            costToChildNodeListDict[minimumCost]=nodeList      # set the Minimum Cost
child node/s
            flag=False
        else:                              # checking the Minimum Cost nodes with the current
Minimum Cost
            if minimumCost>cost:
                minimumCost=cost
                costToChildNodeListDict[minimumCost]=nodeList  # set the Minimum Cost
child node/s


    return minimumCost, costToChildNodeListDict[minimumCost]   # return Minimum
Cost and Minimum Cost child node/s



  def aoStar(self, v, backTracking):     # AO* algorithm for a start node and
backTracking status flag

    # print("HEURISTIC VALUES  :", self.H)
    # print("SOLUTION GRAPH    :", self.solutionGraph)
    # print("PROCESSING NODE   :", v)
    # print("-----------------------------------------------------------------------------------------")

    if self.getStatus(v) >= 0:        # if status node v >= 0, compute Minimum Cost nodes
of v
        minimumCost, childNodeList = self.computeMinimumCostChildNodes(v)
        self.setHeuristicNodeValue(v, minimumCost)
        self.setStatus(v,len(childNodeList))
```

```
            solved=True                   # check the Minimum Cost nodes of v are solved
            for childNode in childNodeList:
                self.parent[childNode]=v
                if self.getStatus(childNode)!=-1:
                    solved=solved & False

            if solved==True:              # if the Minimum Cost nodes of v are solved, set the current node status as solved(-1)
                self.setStatus(v,-1)
                self.solutionGraph[v]=childNodeList


            if v!=self.start:
                self.aoStar(self.parent[v], True)   # backtracking the current node value with backtracking status set to true

            if backTracking==False:
                for childNode in childNodeList:   # for each Minimum Cost child node
                    self.setStatus(childNode,0)   # set the status of child node to 0(needs exploration)
                    self.aoStar(childNode, False) # Minimum Cost child node is further explored with backtracking status as false



h1 = {'A': 1, 'B': 6, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 5, 'H': 7, 'I': 7, 'J': 1, 'T': 3}
graph1 = {
    'A': [[('B', 1), ('C', 1)], [('D', 1)]],
    'B': [[('G', 1)], [('H', 1)]],
    'C': [[('J', 1)]],
    'D': [[('E', 1), ('F', 1)]],
    'G': [[('I', 1)]]
}
G1= Graph(graph1, h1, 'A')
```

```
G1.applyAOStar()
G1.printSolution()
```

TIME COMPLEXITY:

SPACE COMPLEXITY:

OUTPUT:

FOR GRAPH SOLUTION, TRAVERSE THE GRAPH FROM THE START NODE: A
{'I': [], 'G': ['I'], 'B': ['G'], 'J': [], 'C': ['J'], 'A': ['B', 'C']}

RESULT : We have successfully executed AO* in the AWS platform using Python3.

Ex 7: Date: 11.03.2022

MIN MAX ALGORITHM AND ALPHA BETA PRUNING

AIM: To implement Min-max and Alpha Beta Pruning in AI using Python 3 in AWS

TOOL: AWS, Python3

ALGORITHM:

1. Start traversing the given tree in a top to bottom manner.
2. If the node is a leaf node then return the values of the node.
3. It is maximizing player exists then bestval=-INFINITY.
4. For each child node,value =minmax(node,depth1,false,alpha,beta)
5. bestval=max(bestval,value) and alpha=max(alpha,bestval)
6. If beta<=alpha then stop traversing and return.
7. Else bestval = +INFINITY.
8. For each child node,value=minmax(node,depth+1,tree,alpha,beta)
9. bestval=min(bestval,value) and beta=min(beta,bestval)
10. Is beta <= alpha then stop traversing and return bestval.

CODE:

```
MAX, MIN = 1000, -1000
def minimax(depth, nodeIndex, maximizingPlayer,
                        values, alpha, beta):
    if depth == 3:
        return values[nodeIndex]
    if maximizingPlayer:
        best = MIN
        for i in range(0, 2):
            best = max(best, val)
            alpha = max(alpha, best)
            if beta <= alpha:
                break
        return best
    else:
```

```
            best = MAX
            for i in range(0, 2):
                val = minimax(depth + 1, nodeIndex * 2 + i,
                                            True, values, alpha, beta)
                best = min(best, val)
                beta = min(beta, best)
                if beta <= alpha:
                    break
            return best
if __name__ == "__main__":
   values = []
   for i in range(0, 8):
      x = int(input(f"Enter Value {i}  : "))
      values.append(x)
   print ("The optimal value is :", minimax(0, 0, True, values, MIN, MAX))
```

TIME COMPLEXITY:

Minimax

The time complexity of minimax is O(b^m)

Alpha-Beta Pruning

In the worst case, the node would examine b^2 grandchildren which is O(b^(d/2))

SPACE COMPLEXITY:

MiniMax

Space complexity is O(bm)

Alpha-Beta Pruning

Space complexity is O(b * d)

OUTPUT:

The optimal value is: 4

RESULT: We have successfully executed Min-max and learnt the need for Alpha Beta Pruning in the AWS platform using Python3.

Ex 8: Date: 04.04.2022

IMPLEMENTATION OF UNIFICATION AND RESOLUTION FOR REAL-WORLD PROBLEMS

AIM: To implement UNIFICATION and RESOLUTION for real-world problems Using Python3 in AWS.

TOOL: AWS, Python3

ALGORITHM:

1. Initialize the substitution set to be empty.
2. Recursively unify atomic sentences:

- Check for identical expressions.
- If one expression is a variable and the other is a term that does not contain the variable vi then substitute ti/vi.
- Add ti/vi to the substitution setlist.

CODE:

(UNIFICATION)

```
def get_index_comma(string):
    index_list = list()
    par_count = 0
    for i in range(len(string)):
        if string[i] == ',' and par_count == 0:
            index_list.append(i)
        elif string[i] == '(':
            par_count += 1
        elif string[i] == ')':
            par_count -= 1
    return index_list
def is_variable(expr):
    for i in expr:
        if i == '(' or i == ')':
            return False
```

```
    return True
def process_expression(expr):
    expr = expr.replace(' ', '')
    index = None
    for i in range(len(expr)):
        if expr[i] == '(':
            index = i
            break
    predicate_symbol = expr[:index]
    expr = expr.replace(predicate_symbol, '')
    expr = expr[1:len(expr) - 1]
    arg_list = list()
    indices = get_index_comma(expr)
    if len(indices) == 0:
        arg_list.append(expr)
    else:
        arg_list.append(expr[:indices[0]])
        for i, j in zip(indices, indices[1:]):
            arg_list.append(expr[i + 1:j])
        arg_list.append(expr[indices[len(indices) - 1] + 1:])
    return predicate_symbol, arg_list
def get_arg_list(expr):
    _, arg_list = process_expression(expr)
    flag = True
    while flag:
        flag = False
        for i in arg_list:
            if not is_variable(i):
                flag = True
                _, tmp = process_expression(i)
                for j in tmp:
                    if j not in arg_list:
                        arg_list.append(j)
                arg_list.remove(i)
```

```
    return arg_list
def check_occurs(var, expr):
    arg_list = get_arg_list(expr)
    if var in arg_list:
        return True
    return False
def unify(expr1, expr2):

    if is_variable(expr1) and is_variable(expr2):
        if expr1 == expr2:
            return 'Null'
        else:
            return False
    elif is_variable(expr1) and not is_variable(expr2):
        if check_occurs(expr1, expr2):
            return False
        else:
            tmp = str(expr2) + '/' + str(expr1)
            return tmp
    elif not is_variable(expr1) and is_variable(expr2):
        if check_occurs(expr2, expr1):
            return False
        else:
            tmp = str(expr1) + '/' + str(expr2)
            return tmp
    else:
        predicate_symbol_1, arg_list_1 = process_expression(expr1)
        predicate_symbol_2, arg_list_2 = process_expression(expr2)
        # Step 2
        if predicate_symbol_1 != predicate_symbol_2:
            return False
        # Step 3
        elif len(arg_list_1) != len(arg_list_2):
            return False
```

```
        else:
            # Step 4: Create substitution list
            sub_list = list()
            # Step 5:
            for i in range(len(arg_list_1)):
                tmp = unify(arg_list_1[i], arg_list_2[i])
                if not tmp:
                    return False
                elif tmp == 'Null':
                    pass
                else:
                    if type(tmp) == list:
                        for j in tmp:
                            sub_list.append(j)
                    else:
                        sub_list.append(tmp)
            # Step 6
            return sub_list
if __name__ == '__main__':
    f1 = 'Q(a, g(x, a), f(y))'
    f2 = 'Q(a, g(f(b), a), x)'
    # f1 = input('f1 : ')
    # f2 = input('f2 : ')
    result = unify(f1, f2)
    if not result:
        print('The process of Unification failed!')
    else:
        print('The process of Unification successful!')
        print(result)
```

(RESOLUTION)

```
import copy
import time
class Parameter:
```

```
    variable_count = 1
    def __init__(self, name=None):
        if name:
            self.type = "Constant"
            self.name = name
        else:
            self.type = "Variable"
            self.name = "v" + str(Parameter.variable_count)
            Parameter.variable_count += 1
    def isConstant(self):
        return self.type == "Constant"
    def unify(self, type_, name):
        self.type = type_
        self.name = name
    def __eq__(self, other):
        return self.name == other.name
    def __str__(self):
        return self.name
class Predicate:
    def __init__(self, name, params):
        self.name = name
        self.params = params
    def __eq__(self, other):
        return self.name == other.name and all(a == b for a, b in zip(self.params,
other.params))
    def __str__(self):
        return self.name + "(" + ",".join(str(x) for x in self.params) + ")"
    def getNegatedPredicate(self):
        return Predicate(negatePredicate(self.name), self.params)
class Sentence:
    sentence_count = 0
    def __init__(self, string):
        self.sentence_index = Sentence.sentence_count
        Sentence.sentence_count += 1
```

```
        self.predicates = []
        self.variable_map = {}
        local = {}
        for predicate in string.split("|"):
            name = predicate[:predicate.find("(")]
            params = []
            for param in predicate[predicate.find("(") + 1: predicate.find(")")].split(","):
                if param[0].islower():
                    if param not in local:  # Variable
                        local[param] = Parameter()
                        self.variable_map[local[param].name] = local[param]
                    new_param = local[param]
                else:
                    new_param = Parameter(param)
                    self.variable_map[param] = new_param
                params.append(new_param)
            self.predicates.append(Predicate(name, params))
    def getPredicates(self):
        return [predicate.name for predicate in self.predicates]
    def findPredicates(self, name):
        return [predicate for predicate in self.predicates if predicate.name == name]
    def removePredicate(self, predicate):
        self.predicates.remove(predicate)
        for key, val in self.variable_map.items():
            if not val:
                self.variable_map.pop(key)
    def containsVariable(self):
        return any(not param.isConstant() for param in self.variable_map.values())
    def __eq__(self, other):
        if len(self.predicates) == 1 and self.predicates[0] == other:
            return True
        return False

    def __str__(self):
```

```
        return "".join([str(predicate) for predicate in self.predicates])
class KB:
    def __init__(self, inputSentences):
        self.inputSentences = [x.replace(" ", "") for x in inputSentences]
        self.sentences = []
        self.sentence_map = {}

    def prepareKB(self):
        self.convertSentencesToCNF()
        for sentence_string in self.inputSentences:
            sentence = Sentence(sentence_string)
            for predicate in sentence.getPredicates():
                self.sentence_map[predicate] = self.sentence_map.get(
                    predicate, []) + [sentence]

    def convertSentencesToCNF(self):
        for sentenceIdx in range(len(self.inputSentences)):
            # Do negation of the Premise and add them as literal
            if "=>" in self.inputSentences[sentenceIdx]:
                self.inputSentences[sentenceIdx] = negateAntecedent(
                    self.inputSentences[sentenceIdx])

    def askQueries(self, queryList):
        results = []

        for query in queryList:
            negatedQuery = Sentence(negatePredicate(query.replace(" ", "")))
            negatedPredicate = negatedQuery.predicates[0]
            prev_sentence_map = copy.deepcopy(self.sentence_map)
            self.sentence_map[negatedPredicate.name] = self.sentence_map.get(
                negatedPredicate.name, []) + [negatedQuery]
            self.timeLimit = time.time() + 40

            try:
```

```
            result = self.resolve([negatedPredicate], [
                         False]*(len(self.inputSentences) + 1))
        except:
            result = False
        self.sentence_map = prev_sentence_map
        if result:
            results.append("TRUE")
        else:
            results.append("FALSE")
    return results
def resolve(self, queryStack, visited, depth=0):
    if time.time() > self.timeLimit:
        raise Exception
    if queryStack:
        query = queryStack.pop(-1)
        negatedQuery = query.getNegatedPredicate()
        queryPredicateName = negatedQuery.name
        if queryPredicateName not in self.sentence_map:
            return False
        else:
            queryPredicate = negatedQuery
            for kb_sentence in self.sentence_map[queryPredicateName]:
                if not visited[kb_sentence.sentence_index]:
                    for kbPredicate in kb_sentence.findPredicates(queryPredicateName):
                        canUnify, substitution = performUnification(
                            copy.deepcopy(queryPredicate), copy.deepcopy(kbPredicate))
                        if canUnify:
                            newSentence = copy.deepcopy(kb_sentence)
                           newSentence.removePredicate(kbPredicate)
                            newQueryStack = copy.deepcopy(queryStack)
                            if substitution:
                                for old, new in substitution.items():
                                    if old in newSentence.variable_map:
                                        parameter = newSentence.variable_map[old]
```

```
                        newSentence.variable_map.pop(old)
                        parameter.unify(
                            "Variable" if new[0].islower() else "Constant", new)
                        newSentence.variable_map[new] = parameter
                    for predicate in newQueryStack:
                        for index, param in enumerate(predicate.params):
                            if param.name in substitution:
                                new = substitution[param.name]
                                predicate.params[index].unify(
                                    "Variable" if new[0].islower() else "Constant", new)
                for predicate in newSentence.predicates:
                    newQueryStack.append(predicate)
                new_visited = copy.deepcopy(visited)
                if kb_sentence.containsVariable() and len(kb_sentence.predicates)
> 1:
                    new_visited[kb_sentence.sentence_index] = True
                if self.resolve(newQueryStack, new_visited, depth + 1):
                    return True
        return False
    return True

def performUnification(queryPredicate, kbPredicate):
    substitution = {}
    if queryPredicate == kbPredicate:
        return True, {}
    else:
        for query, kb in zip(queryPredicate.params, kbPredicate.params):
            if query == kb:
                continue
            if kb.isConstant():
                if not query.isConstant():
                    if query.name not in substitution:
                        substitution[query.name] = kb.name
                    elif substitution[query.name] != kb.name:
```

```
                return False, {}
            query.unify("Constant", kb.name)
        else:
            return False, {}
    else:
        if not query.isConstant():
            if kb.name not in substitution:
                substitution[kb.name] = query.name
            elif substitution[kb.name] != query.name:
                return False, {}
            kb.unify("Variable", query.name)
        else:
            if kb.name not in substitution:
                substitution[kb.name] = query.name
            elif substitution[kb.name] != query.name:
                return False, {}
  return True, substitution
def negatePredicate(predicate):
  return predicate[1:] if predicate[0] == "~" else "~" + predicate
def negateAntecedent(sentence):
  antecedent = sentence[:sentence.find("=>")]
  premise = []

  for predicate in antecedent.split("&"):
    premise.append(negatePredicate(predicate))

  premise.append(sentence[sentence.find("=>") + 2:])
  return "|".join(premise)
def getInput(filename):
  with open(filename, "r") as file:
    noOfQueries = int(file.readline().strip())
    inputQueries = [file.readline().strip() for _ in range(noOfQueries)]
    noOfSentences = int(file.readline().strip())
    inputSentences = [file.readline().strip()
```

```
                    for _ in range(noOfSentences)]
        return inputQueries, inputSentences
def printOutput(filename, results):
    print(results)
if __name__ == '__main__':
    inputQueries_, inputSentences_ =
getInput('/home/ubuntu/environment/RA1911029010066/Exp8_4apr/input.txt')
    knowledgeBase = KB(inputSentences_)
    knowledgeBase.prepareKB()
    results_ = knowledgeBase.askQueries(inputQueries_)
    printOutput("output.txt", results_)
```

TIME COMPLEXITY:

SPACE COMPLEXITY:

OUTPUT:

The Process of unification is successful

[‘f(b)/x’ , ‘f(y)/x’]

Resolution - [‘True’ , ‘True’]

RESULT: we have successfully implemented UNIFICATION and RESOLUTION for real world problems Using Python3 in AWS.

Ex9: Date: 11.01.2022

IMPLEMENTATION OF UNCERTAIN METHODS FOR AN APPLICATION

AIM: To implement Uncertain methods for an application in AI using Python3 in AWS.

TOOL: AWS, Python3

ALGORITHM:

1. Locate the input, output, and state variables of the plane under consideration.
2. Split the complete universe of discourse spanned by each variable into a number of fuzzy subsets, assigning each with a linguistic label. The subsets include all the elements in the universe.
3. Obtain the membership function for each fuzzy subset.
4. Assign the fuzzy relationships between the inputs or states of fuzzy subsets on one side and the output of fuzzy subsets on the other side, thereby forming the rule base.
5. Choose appropriate scaling factors for the input and output variables for normalizing the variables between [0, 1] and [-1, I] interval.
6. Carry out the fuzzification process.
7. Identify the output contributed from each rule using fuzzy approximate reasoning.
8. Combine the fuzzy outputs obtained from each rule.
9. Finally, apply defuzzification to form a crisp output.

CODE:

```
import matplotlib.pyplot as plt
import seaborn; seaborn.set_style('whitegrid')
import numpy

from pomegranate import *

numpy.random.seed(0)
numpy.set_printoptions(suppress=True)

# The guests initial door selection is completely random
guest = DiscreteDistribution({'A': 1./3, 'B': 1./3, 'C': 1./3})
```

```
# The door the prize is behind is also completely random
prize = DiscreteDistribution({'A': 1./3, 'B': 1./3, 'C': 1./3})

   # Monty is dependent on both the guest and the prize.
monty = ConditionalProbabilityTable(
      [[ 'A', 'A', 'A', 0.0 ],
       [ 'A', 'A', 'B', 0.5 ],
       [ 'A', 'A', 'C', 0.5 ],
       [ 'A', 'B', 'A', 0.0 ],
       [ 'A', 'B', 'B', 0.0 ],
       [ 'A', 'B', 'C', 1.0 ],
       [ 'A', 'C', 'A', 0.0 ],
       [ 'A', 'C', 'B', 1.0 ],
       [ 'A', 'C', 'C', 0.0 ],
       [ 'B', 'A', 'A', 0.0 ],
       [ 'B', 'A', 'B', 0.0 ],
       [ 'B', 'A', 'C', 1.0 ],
       [ 'B', 'B', 'A', 0.5 ],
       [ 'B', 'B', 'B', 0.0 ],
       [ 'B', 'B', 'C', 0.5 ],
       [ 'B', 'C', 'A', 1.0 ],
       [ 'B', 'C', 'B', 0.0 ],
       [ 'B', 'C', 'C', 0.0 ],
       [ 'C', 'A', 'A', 0.0 ],
       [ 'C', 'A', 'B', 1.0 ],
       [ 'C', 'A', 'C', 0.0 ],
       [ 'C', 'B', 'A', 1.0 ],
       [ 'C', 'B', 'B', 0.0 ],
       [ 'C', 'B', 'C', 0.0 ],
       [ 'C', 'C', 'A', 0.5 ],
       [ 'C', 'C', 'B', 0.5 ],
       [ 'C', 'C', 'C', 0.0 ]], [guest, prize])

       # State objects hold both the distribution, and a high level name.
s1 = State(guest, name="guest")
s2 = State(prize, name="prize")
s3 = State(monty, name="monty")
# Create the Bayesian network object with a useful name
model = BayesianNetwork("Monty Hall Problem")

# Add the three states to the network
model.add_states(s1, s2, s3)
# Add edges which represent conditional dependencies, where the second node is
# conditionally dependent on the first node (Monty is dependent on both guest and
prize)
model.add_edge(s1, s3)
```

```
model.add_edge(s2, s3)
model.bake()
model.probability([['A', 'B', 'C']])
model.probability([['A', 'B', 'C']])
print(model.predict_proba({}))
print(model.predict_proba([[None, None, None]]))
print(model.predict_proba([['A', None, None]]))
print(model.predict_proba([{'guest': 'A', 'monty': 'B'}]))
```

TIME COMPLEXITY:

SPACE COMPLEXITY:

OUTPUT:

RESULT: We have successfully implemented Uncertain methods for an application in AI using Python3 in AWS.

Ex 10: Date: 11.01.2022

IMPLEMENTATION OF BLOCKS WORLD PROBLEM

AIM: To Implement Blocks World problem for an application in AI using Python3 in AWS.

TOOL: AWS , Python3

ALGORITHM:

1. Initialise a stack to store the blocks.
2. Make sure the stack is empty when HEAD NODE.NEXT = NULL
3. Read the pattern of blocks given label it START STATE
4. Compare the given pattern to the given final pattern label it GOAL STATE
5. Now start the movement of the blocks one by on either one on top or to the floor according to the need.
6. Keep recording these movements in the empty stack created by STACK.PUSH and STACK.POP methods.
7. Stop the block manipulation when goal state is reached.

OPTIMIZATION TECHNIQUE:

Here keeping track of movement of the block is the main problem, if we keep traversing the floor again and again after each move, our time complexity will be O(n^2) which is exponentially higher than what is needed and should be avoided. To solve this problem STACK data structure can be used, so whenever a movement is made the movement can be conveniently stored in the stack which will be initialized as empty which HEAD.NEXT = NULL. When the block is to be added to the sequence of blocks simply use STACK.PUSH() to make the movement. And when a block is supposed to be removed from the pattern of blocks STACK.POP() can be used to make that movement. Implementing this will bring down the time complexity from O(n) and worst case of O(n^2) to O(1) that is unit time which is a major optimization from exponential time.

CODE:

```
class PREDICATE:
  def __str__(self):
    pass
```

```
 def __repr__(self):
  pass
 def __eq__(self, other) :
  pass
 def __hash__(self):
  pass
 def get_action(self, world_state):
  pass

#OPERATIONS - Stack, Unstack, Pickup, Putdown
class Operation:
 def __str__(self):
  pass
 def __repr__(self):
  pass
 def __eq__(self, other) :
  pass
 def precondition(self):
  pass
 def delete(self):
  pass
 def add(self):
  pass

class ON(PREDICATE):

 def __init__(self, X, Y):
  self.X = X
  self.Y = Y

 def __str__(self):
  return "ON({X},{Y})".format(X=self.X,Y=self.Y)

 def __repr__(self):
  return self.__str__()

 def __eq__(self, other) :
  return self.__dict__ == other.__dict__ and self.__class__ == other.__class__

 def __hash__(self):
   return hash(str(self))

 def get_action(self, world_state):
  return StackOp(self.X,self.Y)
```

```
class ONTABLE(PREDICATE):

  def __init__(self, X):
    self.X = X

  def __str__(self):
    return "ONTABLE({X})".format(X=self.X)

  def __repr__(self):
    return self.__str__()

  def __eq__(self, other) :
    return self.__dict__ == other.__dict__ and self.__class__ == other.__class__

  def __hash__(self):
      return hash(str(self))

  def get_action(self, world_state):
    return PutdownOp(self.X)

class CLEAR(PREDICATE):

  def __init__(self, X):
    self.X = X

  def __str__(self):
    return "CLEAR({X})".format(X=self.X)
    self.X = X

  def __repr__(self):
    return self.__str__()

  def __eq__(self, other) :
    return self.__dict__ == other.__dict__ and self.__class__ == other.__class__

  def __hash__(self):
    return hash(str(self))

  def get_action(self, world_state):
    for predicate in world_state:
      #If Block is on another block, unstack
      if isinstance(predicate,ON) and predicate.Y==self.X:
        return UnstackOp(predicate.X, predicate.Y)
```

```
    return None

class HOLDING(PREDICATE):

  def __init__(self, X):
    self.X = X

  def __str__(self):
    return "HOLDING({X})".format(X=self.X)

  def __repr__(self):
    return self.__str__()

  def __eq__(self, other) :
    return self.__dict__ == other.__dict__ and self.__class__ == other.__class__

  def __hash__(self):
    return hash(str(self))

  def get_action(self, world_state):
    X = self.X
    #If block is on table, pick up
    if ONTABLE(X) in world_state:
      return PickupOp(X)
    #If block is on another block, unstack
    else:
      for predicate in world_state:
        if isinstance(predicate,ON) and predicate.X==X:
          return UnstackOp(X,predicate.Y)

class ARMEMPTY(PREDICATE):

  def __init__(self):
    pass

  def __str__(self):
    return "ARMEMPTY"

  def __repr__(self):
    return self.__str__()

  def __eq__(self, other) :
    return self.__dict__ == other.__dict__ and self.__class__ == other.__class__
```

```
  def __hash__(self):
    return hash(str(self))

  def get_action(self, world_state=[]):
    for predicate in world_state:
      if isinstance(predicate,HOLDING):
        return PutdownOp(predicate.X)
    return None

class StackOp(Operation):

  def __init__(self, X, Y):
    self.X = X
    self.Y = Y

  def __str__(self):
    return "STACK({X},{Y})".format(X=self.X,Y=self.Y)

  def __repr__(self):
    return self.__str__()

  def __eq__(self, other) :
    return self.__dict__ == other.__dict__ and self.__class__ == other.__class__

  def precondition(self):
    return [ CLEAR(self.Y) , HOLDING(self.X) ]

  def delete(self):
    return [ CLEAR(self.Y) , HOLDING(self.X) ]

  def add(self):
    return [ ARMEMPTY() , ON(self.X,self.Y) ]

class UnstackOp(Operation):

  def __init__(self, X, Y):
    self.X = X
    self.Y = Y

  def __str__(self):
    return "UNSTACK({X},{Y})".format(X=self.X,Y=self.Y)

  def __repr__(self):
    return self.__str__()
```

```
  def __eq__(self, other) :
    return self.__dict__ == other.__dict__ and self.__class__ == other.__class__

  def precondition(self):
    return [ ARMEMPTY() , ON(self.X,self.Y) , CLEAR(self.X) ]

  def delete(self):
    return [ ARMEMPTY() , ON(self.X,self.Y) ]

  def add(self):
    return [ CLEAR(self.Y) , HOLDING(self.X) ]

class PickupOp(Operation):

  def __init__(self, X):
    self.X = X

  def __str__(self):
    return "PICKUP({X})".format(X=self.X)

  def __repr__(self):
    return self.__str__()

  def __eq__(self, other) :
    return self.__dict__ == other.__dict__ and self.__class__ == other.__class__

  def precondition(self):
    return [ CLEAR(self.X) , ONTABLE(self.X) , ARMEMPTY() ]

  def delete(self):
    return [ ARMEMPTY() , ONTABLE(self.X) ]

  def add(self):
    return [ HOLDING(self.X) ]

class PutdownOp(Operation):

  def __init__(self, X):
    self.X = X

  def __str__(self):
    return "PUTDOWN({X})".format(X=self.X)
```

```
  def __repr__(self):
    return self.__str__()

  def __eq__(self, other) :
    return self.__dict__ == other.__dict__ and self.__class__ == other.__class__

  def precondition(self):
    return [ HOLDING(self.X) ]

  def delete(self):
    return [ HOLDING(self.X) ]

  def add(self):
    return [ ARMEMPTY() , ONTABLE(self.X) ]

def isPredicate(obj):
  predicates = [ON, ONTABLE, CLEAR, HOLDING, ARMEMPTY]
  for predicate in predicates:
    if isinstance(obj,predicate):
      return True
  return False

def isOperation(obj):
  operations = [StackOp, UnstackOp, PickupOp, PutdownOp]
  for operation in operations:
    if isinstance(obj,operation):
      return True
  return False

def arm_status(world_state):
  for predicate in world_state:
    if isinstance(predicate, HOLDING):
      return predicate
  return ARMEMPTY()

class GoalStackPlanner:

  def __init__(self, initial_state, goal_state):
    self.initial_state = initial_state
    self.goal_state = goal_state

  def get_steps(self):

    #Store Steps
```

```
steps = []
#Program Stack
stack = []
#World State/Knowledge Base
world_state = self.initial_state.copy()
#Initially push the goal_state as compound goal onto the stack
stack.append(self.goal_state.copy())
#Repeat until the stack is empty
while len(stack)!=0:
  #Get the top of the stack
  stack_top = stack[-1]
  #If Stack Top is Compound Goal, push its unsatisfied goals onto stack
  if type(stack_top) is list:
    compound_goal = stack.pop()
    for goal in compound_goal:
      if goal not in world_state:
        stack.append(goal)
  #If Stack Top is an action
  elif isOperation(stack_top):
    #Peek the operation
    operation = stack[-1]
    all_preconditions_satisfied = True
    #Check if any precondition is unsatisfied and push it onto program stack
    for predicate in operation.delete():
      if predicate not in world_state:
        all_preconditions_satisfied = False
        stack.append(predicate)
    #If all preconditions are satisfied, pop operation from stack and execute it
    if all_preconditions_satisfied:
      stack.pop()
      steps.append(operation)
      for predicate in operation.delete():
        world_state.remove(predicate)
      for predicate in operation.add():
        world_state.append(predicate)
  #If Stack Top is a single satisfied goal
  elif stack_top in world_state:
    stack.pop()
  #If Stack Top is a single unsatisfied goal
  else:
    unsatisfied_goal = stack.pop()

    #Replace Unsatisfied Goal with an action that can complete it
    action = unsatisfied_goal.get_action(world_state)
```

```
        stack.append(action)
        #Push Precondition on the stack
        for predicate in action.precondition():
          if predicate not in world_state:
            stack.append(predicate)
    return steps
if __name__ == '__main__':
  initial_state = [
    ON('B','A'),
    ON('C','B'),
    ONTABLE('A'),ONTABLE('D'),
    CLEAR('C'),CLEAR('D'),
    ARMEMPTY()
  ]
  goal_state = [
    ON('B','D'),ON('C','A'),
    ONTABLE('D'),ONTABLE('A'),
    CLEAR('B'),CLEAR('C'),
    ARMEMPTY()
  ]

  goal_stack = GoalStackPlanner(initial_state=initial_state, goal_state=goal_state)
  steps = goal_stack.get_steps()
  print(steps)
```

TIME COMPLEXITY:

SPACE COMPLEXITY:

OUTPUT:

[UNSTACK(C,B), PUTDOWN(C), UNSTACK(B,A), PUTDOWN(B), PICKUP(C), STACK(C,A), PICKUP(B), STACK(B,D)]

RESULT: We have successfully implemented the Blocks World problem for an application in AI using Python3 in AWS.

Ex 11: Date: 11.01.2022

IMPLEMENTATION OF LEARNING ALGORITHMS FOR AN APPLICATION

AIM: To implement a Machine Learning Algorithm for an application in AI using Python3 in AWS.

TOOL: AWS , Python3

ALGORITHM:

1. Import the required libraries.
2. Import the data file in the program.
3. Clean the data.
4. Find the required features on which the model predicts.For eg. in our program we use -Rooms, Bathroom, Land Size, Latitude, Longitude from our data.
5. We also have to find which field we have to predict. In our model we are predicting the price of the House.
6. After that we import the libraries of the algorithm .Here we have imported Decision tree regression.
7. We train our model with features and data then the predicted value is given by the model.

CODE:

```
import pandas as pd
from sklearn.tree import DecisionTreeRegressor
melbourne_file_path = 'melb_data.csv'
melbourne_data = pd.read_csv(melbourne_file_path) melbourne_data.columns
melbourne_data = melbourne_data.dropna(axis=0)
y = melbourne_data.Price
melbourne_features = ['Rooms', 'Bathroom', 'Landsize', 'Lattitude', 'Longtitude']
X = melbourne_data[melbourne_features] X.describe()
X.head()
# Define model. Specify a number for random_state to ensure same results each run
melbourne_model = DecisionTreeRegressor(random_state=1) # Fit model
```

```
melbourne_model.fit(X, y)
print("Making predictions for the following 5 houses:") print(X.head())
print("The predictions are") print(melbourne_model.predict(X.head()))
```

TIME COMPLEXITY:

SPACE COMPLEXITY:

OUTPUT:

RESULT: We have successfully implemented a Machine Learning Algorithm for an application in AI using Python3 in AWS.

Ex 12: Date: 11.01.2022

TO IMPLEMENT NLP PROGRAMS

AIM: To implement a NLP program for an application in AI using Python3 in AWS.

TOOL: AWS , Python3

ALGORITHM:

CODE:

```
!pip install -q wordcloud
import wordcloud

import nltk
nltk.download('stopwords')
nltk.download('wordnet')
nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')

import pandas as pd
import matplotlib.pyplot as plt
import io
import unicodedata
import numpy as np
import re
import string
# Constants
# POS (Parts Of Speech) for: nouns, adjectives, verbs and adverbs
DI_POS_TYPES = {'NN':'n', 'JJ':'a', 'VB':'v', 'RB':'r'}
POS_TYPES = list(DI_POS_TYPES.keys())

# Constraints on tokens
MIN_STR_LEN = 3
```

```
RE_VALID = '[a-zA-Z]'
# Upload from google drive
from google.colab import files
uploaded = files.upload()
print("len(uploaded.keys():", len(uploaded.keys()))

for fn in uploaded.keys():
  print('User uploaded file "{name}" with length {length} bytes'.format(name=fn,
length=len(uploaded[fn])))

# Get list of quotes
df_quotes = pd.read_csv(io.StringIO(uploaded['quotes.txt'].decode('utf-8')), sep='\t')

# Display
print("df_quotes:")
print(df_quotes.head().to_string())
print(df_quotes.describe())

# Convert quotes to list
li_quotes = df_quotes['Quote'].tolist()
print()
print("len(li_quotes):", len(li_quotes))
# Get stopwords, stemmer and lemmatizer
stopwords = nltk.corpus.stopwords.words('english')
stemmer = nltk.stem.PorterStemmer()
lemmatizer = nltk.stem.WordNetLemmatizer()

# Remove accents function
def remove_accents(data):
    return ''.join(x for x in unicodedata.normalize('NFKD', data) if x in string.ascii_letters
or x == " ")

# Process all quotes
li_tokens = []
```

```
li_token_lists = []
li_lem_strings = []

for i,text in enumerate(li_quotes):
    # Tokenize by sentence, then by lowercase word
    tokens = [word.lower() for sent in nltk.sent_tokenize(text) for word in
nltk.word_tokenize(sent)]

    # Process all tokens per quote
    li_tokens_quote = []
    li_tokens_quote_lem = []
    for token in tokens:
        # Remove accents
        t = remove_accents(token)

        # Remove punctuation
        t = str(t).translate(string.punctuation)
        li_tokens_quote.append(t)

        # Add token that represents "no lemmatization match"
        li_tokens_quote_lem.append("-") # this token will be removed if a lemmatization
match is found below

        # Process each token
        if t not in stopwords:
            if re.search(RE_VALID, t):
                if len(t) >= MIN_STR_LEN:
                    # Note that the POS (Part Of Speech) is necessary as input to the
lemmatizer
                    # (otherwise it assumes the word is a noun)
                    pos = nltk.pos_tag([t])[0][1][:2]
                    pos2 = 'n'  # set default to noun
                    if pos in DI_POS_TYPES:
                      pos2 = DI_POS_TYPES[pos]
```

```
            stem = stemmer.stem(t)
            lem = lemmatizer.lemmatize(t, pos=pos2)  # lemmatize with the correct
POS

            if pos in POS_TYPES:
                li_tokens.append((t, stem, lem, pos))
                # Remove the "-" token and append the lemmatization match
                li_tokens_quote_lem = li_tokens_quote_lem[:-1]
                li_tokens_quote_lem.append(lem)

    # Build list of token lists from lemmatized tokens
    li_token_lists.append(li_tokens_quote)

    # Build list of strings from lemmatized tokens
    str_li_tokens_quote_lem = ' '.join(li_tokens_quote_lem)
    li_lem_strings.append(str_li_tokens_quote_lem)

# Build resulting dataframes from lists
df_token_lists = pd.DataFrame(li_token_lists)

print("df_token_lists.head(5):")
print(df_token_lists.head(5).to_string())

# Replace None with empty string
for c in df_token_lists:
    if str(df_token_lists[c].dtype) in ('object', 'string_', 'unicode_'):
        df_token_lists[c].fillna(value='', inplace=True)

df_lem_strings = pd.DataFrame(li_lem_strings, columns=['lem quote'])

print()
print("")
print("df_lem_strings.head():")
```

```
print(df_lem_strings.head().to_string())
# Add counts
print("Group by lemmatized words, add count and sort:")
df_all_words = pd.DataFrame(li_tokens, columns=['token', 'stem', 'lem', 'pos'])
df_all_words['counts'] = df_all_words.groupby(['lem'])['lem'].transform('count')
df_all_words = df_all_words.sort_values(by=['counts', 'lem'], ascending=[False,
True]).reset_index()
print("Get just the first row in each lemmatized group")
df_words = df_all_words.groupby('lem').first().sort_values(by='counts',
ascending=False).reset_index()
print("df_words.head(10):")
print(df_words.head(10))
df_words = df_words[['lem', 'pos', 'counts']].head(200)
for v in POS_TYPES:
    df_pos = df_words[df_words['pos'] == v]
    print()
    print("POS_TYPE:", v)
    print(df_pos.head(10).to_string())
li_token_lists_flat = [y for x in li_token_lists for y in x]  # flatten the list of token lists to a
single list
print("li_token_lists_flat[:10]:", li_token_lists_flat[:10])
di_freq = nltk.FreqDist(li_token_lists_flat)
del di_freq['']
li_freq_sorted = sorted(di_freq.items(), key=lambda x: x[1], reverse=True)  # sorted list
print(li_freq_sorted)
di_freq.plot(30, cumulative=False)
li_lem_words = df_all_words['lem'].tolist()
di_freq2 = nltk.FreqDist(li_lem_words)
li_freq_sorted2 = sorted(di_freq2.items(), key=lambda x: x[1], reverse=True)  # sorted
list
print(li_freq_sorted2)
di_freq2.plot(30, cumulative=False)
```

TIME COMPLEXITY:
O(N^2) for the above program

SPACE COMPLEXITY:
O(N) for the above program

OUTPUT:

RESULT: We have successfully implemented a NLP program for an application in AI using Python3 in AWS.