Tower of Hanoi

def TowerOfHanoi(n , source, destination, auxiliary):

if n==1:

print ("Move disk 1 from source",source,"to destination",destination)

return

TowerOfHanoi(n-1, source, auxiliary, destination)

print ("Move disk",n,"from source",source,"to destination",destination)

TowerOfHanoi(n-1, auxiliary, destination, source)

n=int(input("Enter the number of disks: "))

TowerOfHanoi(n,'A','B','C')

N queens problem

print ("Enter the number of queens")

N = int(input())

# here we create a chessboard

# NxN matrix with all elements set to 0

board = [[0]\*N for \_ in range(N)]

def attack(i, j):

#checking vertically and horizontally

for k in range(0,N):

if board[i][k]==1 or board[k][j]==1:

return True

#checking diagonally

for k in range(0,N):

for l in range(0,N):

if (k+l==i+j) or (k-l==i-j):

if board[k][l]==1:

return True

return False

def N\_queens(n):

if n==0:

return True

for i in range(0,N):

for j in range(0,N):

if (not(attack(i,j))) and (board[i][j]!=1):

board[i][j] = 1

if N\_queens(n-1)==True:

return True

board[i][j] = 0

return False

N\_queens(N)

for i in board:

print (i)

**Vacuum Cleaner**

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()

**WATER JUG PROBLEM:**

from collections import defaultdict

# jug1 and jug2 contain the value

# for max capacity in respective jugs

# and target is the amount of water to be measured.

jug1=int(input(“Enter the capacity of jug1: “)

jug2=int(input(“Enter the capacity of jug2: “)

target=int(input(“Enter the target capacity: ”)

# Initialize dictionary with

# default value as false.

visited = defaultdict(lambda: False)

# Recursive function which prints the

# intermediate steps to reach the final

# solution and return boolean value

# (True if solution is possible, otherwise False).

# amt1 and amt2 are the amount of water present

# in both jugs at a certain point of time.

def waterJugSolver(x, y):

    # Checks for our goal and

    # returns true if achieved.

    if (x == target and y == 0) or (y== target and x== 0):

        print(x, y)

        return True

    # Checks if we have already visited the

    # combination or not. If not, then it proceeds further.

    if visited[(x,y)] == False:

        print(x,y)

        # Changes the boolean value of

        # the combination as it is visited.

        visited[(x,y)] = True

        # Check for all the 6 possibilities and

        # see if a solution is found in any one of them.

        return (waterJugSolver(0,y) or

                waterJugSolver(x, 0) or

                waterJugSolver(jug1,y) or

                waterJugSolver(x, jug2) or

                waterJugSolver(x+ min(y, (jug1-x)),

                y- min(y, (jug1-x))) or

                waterJugSolver(x - min(x, (jug2-y)),

                y + min(x, (jug2-y))))

    # Return False if the combination is

    # already visited to avoid repetition otherwise

    # recursion will enter an infinite loop.

    else:

        return False

print(&quot;Complete Path is: &quot;)

# Call the function and pass the

# initial amount of water present in both jugs.

waterJugSolver(0, 0)

**Camel and Banana Problem**

total=int(input('Enter total no. of bananas: '))

d=int(input('Enter total distance:'))

max\_load=int(input('Enter max load capacity of the camel: '))

lost=0

s=total

for i in range(d):

while s>0:

s=s-max\_load

if s==1:

lost=lost-1

lost=lost+2

lost=lost-1

s=total-lost

if s==0:

break

print(s)

**Missionaries and Cannibals problem**

import math

class State():

def \_\_init\_\_(self, cannibalLeft, missionaryLeft, boat, cannibalRight, missionaryRight):

self.cannibalLeft = cannibalLeft

self.missionaryLeft = missionaryLeft

self.boat = boat

self.cannibalRight = cannibalRight

self.missionaryRight = missionaryRight

self.parent = None

def is\_goal(self):

if self.cannibalLeft == 0 and self.missionaryLeft == 0:

return True

else:

return False

def is\_valid(self):

if self.missionaryLeft >= 0 and self.missionaryRight >= 0 \

and self.cannibalLeft >= 0 and self.cannibalRight >= 0 \

and (self.missionaryLeft == 0 or self.missionaryLeft >= self.cannibalLeft) \

and (self.missionaryRight == 0 or self.missionaryRight >= self.cannibalRight):

return True

else:

return False

def \_\_eq\_\_(self, other):

return self.cannibalLeft == other.cannibalLeft and self.missionaryLeft == other.missionaryLeft \

and self.boat == other.boat and self.cannibalRight == other.cannibalRight \

and self.missionaryRight == other.missionaryRight

def \_\_hash\_\_(self):

return hash((self.cannibalLeft, self.missionaryLeft, self.boat, self.cannibalRight, self.missionaryRight))

def successors(cur\_state):

children = [];

if cur\_state.boat == 'left':

new\_state = State(cur\_state.cannibalLeft, cur\_state.missionaryLeft - 2, 'right',

cur\_state.cannibalRight, cur\_state.missionaryRight + 2)

## Two missionaries cross left to right.

if new\_state.is\_valid():

new\_state.parent = cur\_state

children.append(new\_state)

new\_state = State(cur\_state.cannibalLeft - 2, cur\_state.missionaryLeft, 'right',

cur\_state.cannibalRight + 2, cur\_state.missionaryRight)

## Two cannibals cross left to right.

if new\_state.is\_valid():

new\_state.parent = cur\_state

children.append(new\_state)

new\_state = State(cur\_state.cannibalLeft - 1, cur\_state.missionaryLeft - 1, 'right',

cur\_state.cannibalRight + 1, cur\_state.missionaryRight + 1)

## One missionary and one cannibal cross left to right.

if new\_state.is\_valid():

new\_state.parent = cur\_state

children.append(new\_state)

new\_state = State(cur\_state.cannibalLeft, cur\_state.missionaryLeft - 1, 'right',

cur\_state.cannibalRight, cur\_state.missionaryRight + 1)

## One missionary crosses left to right.

if new\_state.is\_valid():

new\_state.parent = cur\_state

children.append(new\_state)

new\_state = State(cur\_state.cannibalLeft - 1, cur\_state.missionaryLeft, 'right',

cur\_state.cannibalRight + 1, cur\_state.missionaryRight)

## One cannibal crosses left to right.

if new\_state.is\_valid():

new\_state.parent = cur\_state

children.append(new\_state)

else:

new\_state = State(cur\_state.cannibalLeft, cur\_state.missionaryLeft + 2, 'left',

cur\_state.cannibalRight, cur\_state.missionaryRight - 2)

## Two missionaries cross right to left.

if new\_state.is\_valid():

new\_state.parent = cur\_state

children.append(new\_state)

new\_state = State(cur\_state.cannibalLeft + 2, cur\_state.missionaryLeft, 'left',

cur\_state.cannibalRight - 2, cur\_state.missionaryRight)

## Two cannibals cross right to left.

if new\_state.is\_valid():

new\_state.parent = cur\_state

children.append(new\_state)

new\_state = State(cur\_state.cannibalLeft + 1, cur\_state.missionaryLeft + 1, 'left',

cur\_state.cannibalRight - 1, cur\_state.missionaryRight - 1)

## One missionary and one cannibal cross right to left.

if new\_state.is\_valid():

new\_state.parent = cur\_state

children.append(new\_state)

new\_state = State(cur\_state.cannibalLeft, cur\_state.missionaryLeft + 1, 'left',

cur\_state.cannibalRight, cur\_state.missionaryRight - 1)

## One missionary crosses right to left.

if new\_state.is\_valid():

new\_state.parent = cur\_state

children.append(new\_state)

new\_state = State(cur\_state.cannibalLeft + 1, cur\_state.missionaryLeft, 'left',

cur\_state.cannibalRight - 1, cur\_state.missionaryRight)

## One cannibal crosses right to left.

if new\_state.is\_valid():

new\_state.parent = cur\_state

children.append(new\_state)

return children

def breadth\_first\_search():

initial\_state = State(3,3,'left',0,0)

if initial\_state.is\_goal():

return initial\_state

frontier = list()

explored = set()

frontier.append(initial\_state)

while frontier:

state = frontier.pop(0)

if state.is\_goal():

return state

explored.add(state)

children = successors(state)

for child in children:

if (child not in explored) or (child not in frontier):

frontier.append(child)

return None

def print\_solution(solution):

path = []

path.append(solution)

parent = solution.parent

while parent:

path.append(parent)

parent = parent.parent

for t in range(len(path)):

state = path[len(path) - t - 1]

print ("(" + str(state.cannibalLeft) + "," + str(state.missionaryLeft) \

+ "," + state.boat + "," + str(state.cannibalRight) + "," + \

str(state.missionaryRight) + ")")

def main():

solution = breadth\_first\_search()

print("Missionaries and Cannibals solution:")

print("(cannibalLeft,missionaryLeft,boat,cannibalRight,missionaryRight)")

print\_solution(solution)

# if called from the command line, call main()

if \_\_name\_\_ == "\_\_main\_\_":

main()

**GRAPH COLORING**

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)

**CONSTRAINT SATISFACTION PROBLEM**

**SENDMORY : 95671082**

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')

**BREADTH FIRST SEARCH & DEPTH FIRST SEARCH**

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)

**BEST FIRST SEARCH & A\* SEARCH**

.**(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)

**AO\* SEARCH**

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()

**implementation of knowledge representation schemes**

size = 9

#empty cells have value zero

matrix = [

[5,3,0,0,7,0,0,0,0],

[6,0,0,1,9,5,0,0,0],

[0,9,8,0,0,0,0,6,0],

[8,0,0,0,6,0,0,0,3],

[4,0,0,8,0,3,0,0,1],

[7,0,0,0,2,0,0,0,6],

[0,6,0,0,0,0,2,8,0],

[0,0,0,4,1,9,0,0,5],

[0,0,0,0,8,0,0,7,9]]

#print sudoku

def print\_sudoku():

for i in matrix:

print (i)

#assign cells and check

def number\_unassigned(row, col):

num\_unassign = 0

for i in range(0,size):

for j in range (0,size):

#cell is unassigned

if matrix[i][j] == 0:

row = i

col = j

num\_unassign = 1

a = [row, col, num\_unassign]

return a

a = [-1, -1, num\_unassign]

return a

#check validity of number

def is\_safe(n, r, c):

#checking in row

for i in range(0,size):

#there is a cell with same value

if matrix[r][i] == n:

return False

#checking in column

for i in range(0,size):

#there is a cell with same value

if matrix[i][c] == n:

return False

row\_start = (r//3)\*3

col\_start = (c//3)\*3;

#checking submatrix

for i in range(row\_start,row\_start+3):

for j in range(col\_start,col\_start+3):

if matrix[i][j]==n:

return False

return True

#check validity of number

def solve\_sudoku():

row = 0

col = 0

#if all cells are assigned then the sudoku is already solved

#pass by reference because number\_unassigned will change the values of row and col

a = number\_unassigned(row, col)

if a[2] == 0:

return True

row = a[0]

col = a[1]

#number between 1 to 9

for i in range(1,10):

#if we can assign i to the cell or not

#the cell is matrix[row][col]

if is\_safe(i, row, col):

matrix[row][col] = i

#backtracking

if solve\_sudoku():

return True

#f we can't proceed with this solution

#reassign the cell

matrix[row][col]=0

return False

if solve\_sudoku():

print\_sudoku()

else:

print("No solution")

**MIN MAX ALGORITHM AND ALPHA BETA PRUNING**

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))

**UNIFICATION AND RESOLUTION**

**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\_)

**UNCERTAIN METHODS**

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'}]))

**BLOCKS WORLD**

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)

**LEARNING ALGORITHMS FOR AN APPLICATION**

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()))

**TO IMPLEMENT NLP PROGRAMS**

!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)