# PRACTICAL NO.-8

1. **Write a program to implement alpha beta search.**

**AIM:-**

Write a program to implement alpha beta search.

# PYTHON CODE

tree = [[[5, 1, 2], [8, -8, -9]], [[9, 4, 5], [-3, 4, 3]]]

root = 0

pruned = 0

def children(branch, depth, alpha, beta):

global tree

global root

global pruned

i = 0

for child in branch:

if type(child) is list:

(nalpha, nbeta) = children(child, depth + 1, alpha, beta)

if depth % 2 == 1:

beta = min(beta, nalpha)

else:

alpha = max(alpha, nbeta)

branch[i] = nalpha if depth % 2 == 0 else nbeta

i += 1

else:

if depth % 2 == 0 and alpha < child:

alpha = child

if depth % 2 == 1 and beta > child:

beta = child

if alpha >= beta:

pruned += 1

break

if depth == root:

tree = branch

return (alpha, beta)

def alphabeta(in\_tree=tree, start=root, upper=-15, lower=15):

global tree

global pruned

global root

(alpha, beta) = children(tree, start, upper, lower)

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

print("(alpha, beta): ", alpha, beta)

print("Result: ", tree)

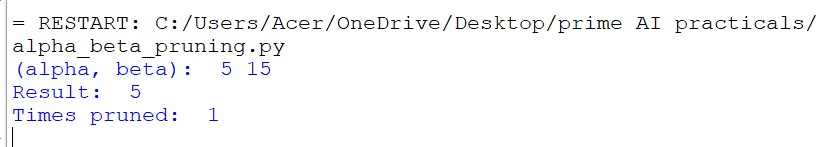
print("Times pruned: ", pruned)

return (alpha, beta, tree, pruned)

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

alphabeta()

**Output:**



# PRACTICAL NO.-9

1. **Write a program for Hill climbing problem.**

**Hill climbing algorithm**

Hill climbing algorithm is a local search algorithm which continuously moves in the direction of increasing elevation/value to find the peak of the mountain or best solution to the problem. It terminates when it reaches a peak value where no neighbor has a higher value. Hill climbing algorithm is a technique which is used for optimizing the mathematical problems. One of the widely discussed examples of Hill climbing algorithm is Traveling-salesman Problem in which we need to minimize the distance traveled by the salesman. It is also called greedy local search as it only looks to its good immediate neighbor state and not beyond that. A node of hill climbing algorithm has two components which are state and value. Hill Climbing is mostly used when a good heuristic is available. In this algorithm, we don't need to maintain and handle the search tree or graph as it only keeps a single current state.

**Algorithm with an example trace**

1. **Algorithm**

Step 1: Evaluate the initial state, if it is goal state then return success and Stop.

Step 2: Loop Until a solution is found or there is no new operator left to apply.

Step 3: Select and apply an operator to the current state.

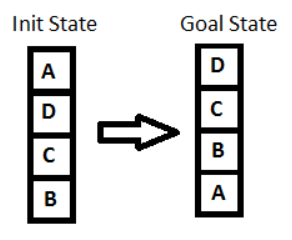
Step 4: Check new state:

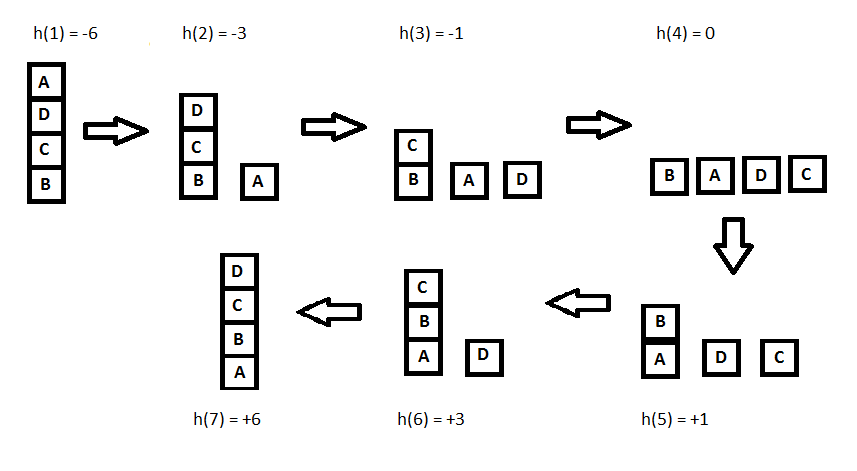
* 1. If it is goal state, then return success and quit.
  2. Else if it is better than the current state then assign new state as a current state.
  3. Else if not better than the current state, then return to step2.

Step 5: Exit.

1. **example trace**

Use Hill Climbing Search to place the initial sate given below to final state





**Source Code**

sussList = {} # This dictionary holds all the nodes with their successors and their corresponding heuristic value

temp\_key\_list = [] # Holds key node whose successor is to be inputted

initial\_node = str(input("Input initial node: ")) # root node

initial\_value = eval(input(f"Input {initial\_node}'s heuristic value: ")) # holds heuristic value of root node

numberNode = eval(input(f"How many successor nodes in node '{initial\_node}': ")) # number of successor of root node

temp\_key\_list.append(initial\_node)

def nodeInput(numberNode): # Function used to input all nodes with their successor and corresponding heuristic value

new\_node = temp\_key\_list[0]

temp\_key\_list.pop(0)

new\_list = []

for i in range(numberNode):

key\_name = str(input(f"Enter {i+1}'th successor of {new\_node}: "))

key\_value = eval(input(f"Enter {key\_name}'s heuristic value: "))

temp = [key\_name,key\_value]

new\_list.append(temp)

sussList[new\_node] = new\_list

temp\_key\_list.append(key\_name)

if len(temp\_key\_list) != 0:

new\_node = temp\_key\_list[0]

new\_numberNode = eval(input(f"How many successor nodes in node {new\_node}?: "))

nodeInput(new\_numberNode)

else:

pass

def sortList(new\_list): #Function to sort the selected list in ascending order

new\_list.sort(key = lambda x: x[1])

return new\_list

def hillClimbing\_search(node,value): #Function to find shortest path using heuristic value

new\_list = list()

if node in sussList.keys():

new\_list = sussList[node]

new\_list = sortList(new\_list)

if (value > new\_list[0][1]):

value = new\_list[0][1]

node = new\_list[0][0]

hillClimbing\_search(node, value)

if (value < new\_list[0][1]):

print(f"ANSWER:\nFor given Data, the local maxima is at node '{node}' with heuristic value {value}")

else:

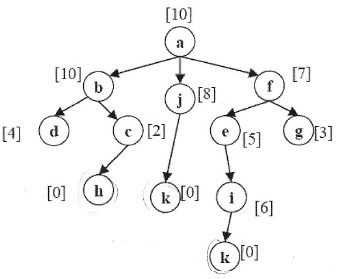
print(f"ANSWER:\nFor given Data, the local maxima is at node '{node}' with heuristic value {value}")

nodeInput(numberNode)

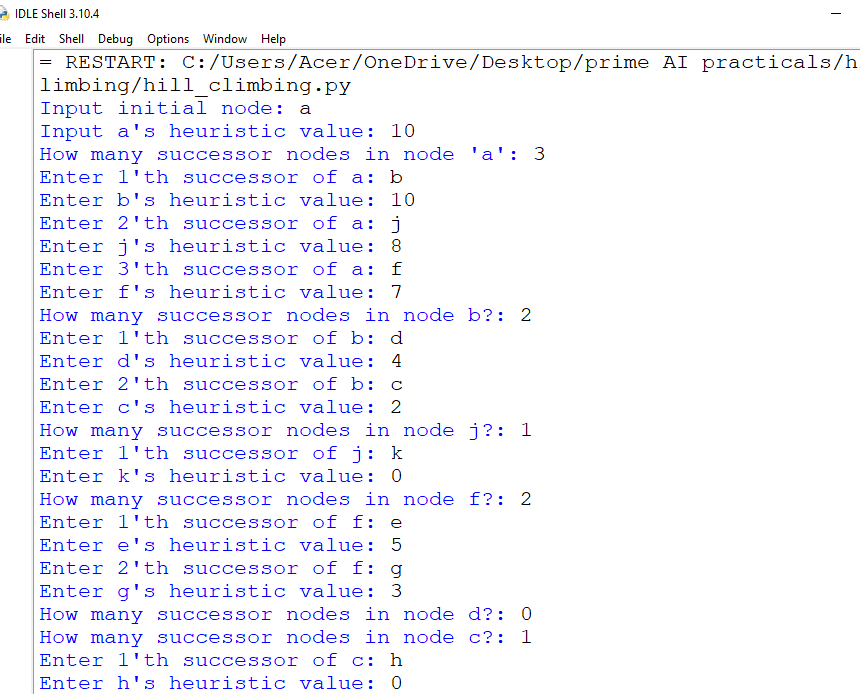
print("The user input is as follows: \n", sussList)

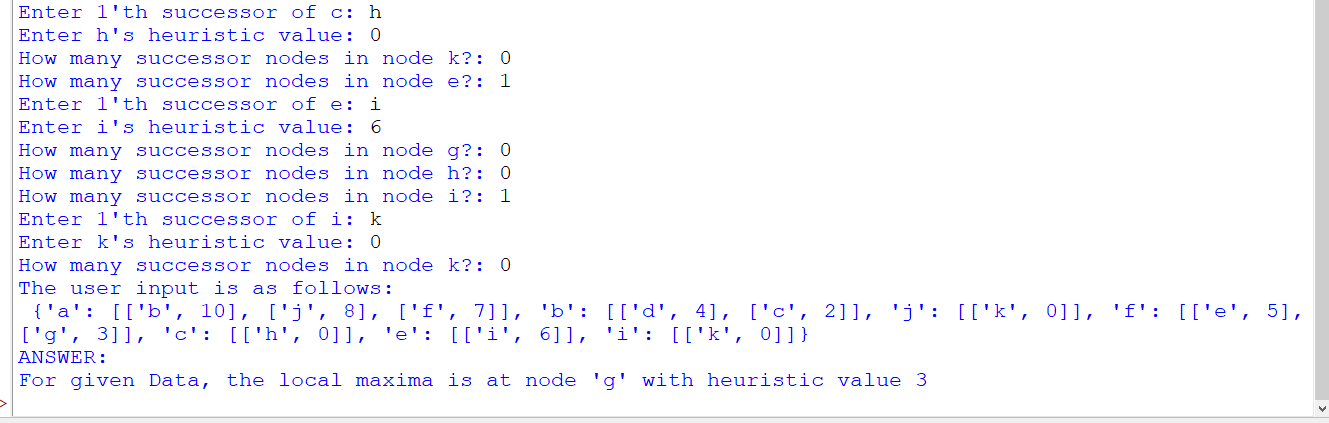
hillClimbing\_search(initial\_node, initial\_value)

**Tree:**



**Output**

****



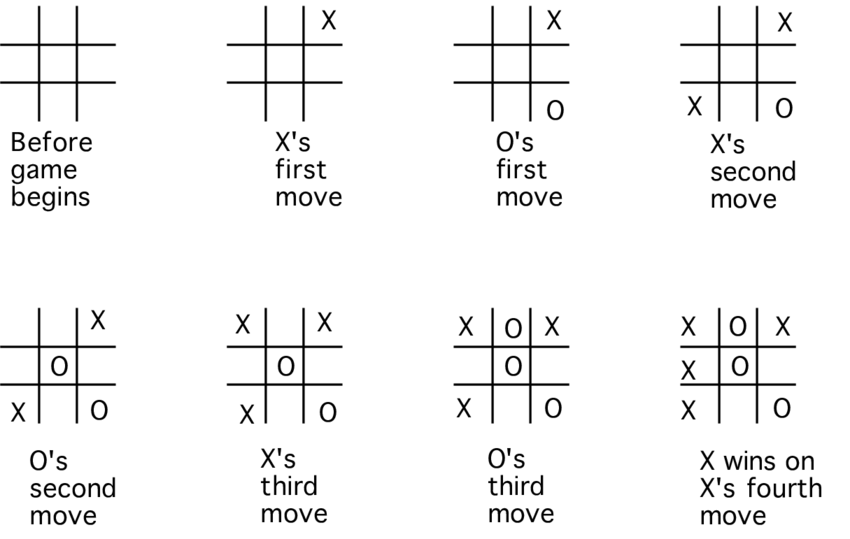
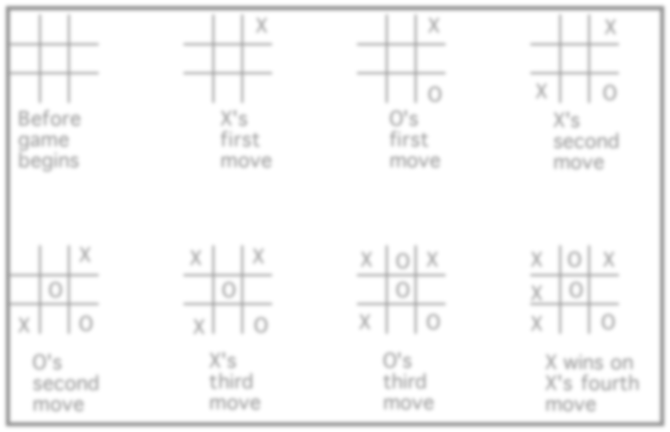
# Practical no-11

1. **Design the simulation of tic – tac – toe game using min-max algorithm.**

**Aim:-**

Design the simulation of TIC – TAC –TOE game using min-max algorithm

# Diagram:-



**Python code:**

import time

import os

board = [' ',' ',' ',' ',' ',' ',' ',' ',' ',' ']

player = 1

########win Flags##########

Win = 1

Draw = -1

Running = 0

Stop = 1

###########################

Game = Running

Mark = 'X'

**#This Function Draws Game Board**

def DrawBoard():

print(" %c | %c | %c " % (board[1],board[2],board[3]))

print("\_\_\_|\_\_\_|\_\_\_")

print(" %c | %c | %c " % (board[4],board[5],board[6]))

print("\_\_\_|\_\_\_|\_\_\_")

print(" %c | %c | %c " % (board[7],board[8],board[9]))

print(" | | ")

**#This Function Checks position is empty or not**

def CheckPosition(x):

if(board[x] == ' '):

return True

else:

return False

**#This Function Checks player has won or not**

def CheckWin():

global Game

#Horizontal winning condition

if(board[1] == board[2] and board[2] == board[3] and board[1] != ' '):

Game = Win

elif(board[4] == board[5] and board[5] == board[6] and board[4] != ' '):

Game = Win

elif(board[7] == board[8] and board[8] == board[9] and board[7] != ' '):

Game = Win

**#Vertical Winning Condition**

elif(board[1] == board[4] and board[4] == board[7] and board[1] != ' '):

Game = Win

elif(board[2] == board[5] and board[5] == board[8] and board[2] != ' '):

Game = Win

elif(board[3] == board[6] and board[6] == board[9] and board[3] != ' '):

Game=Win

**#Diagonal Winning Condition**

elif(board[1] == board[5] and board[5] == board[9] and board[5] != ' '):

Game = Win

elif(board[3] == board[5] and board[5] == board[7] and board[5] != ' '):

Game=Win

#Match Tie or Draw Condition

elif(board[1]!=' ' and board[2]!=' ' and board[3]!=' ' and board[4]!=' ' and board[5]!=' ' and board[6]!=' ' and board[7]!=' ' and board[8]!=' ' and board[9]!=' '):

Game=Draw

else:

Game=Running

print("Tic-Tac-Toe Game")

print("Player 1 [X] --- Player 2 [O]\n")

print()

print()

print("Please Wait...")

time.sleep(1)

while(Game == Running):

os.system('cls')

DrawBoard()

if(player % 2 != 0):

print("Player 1's chance")

Mark = 'X'

else:

print("Player 2's chance")

Mark = 'O'

choice = int(input("Enter the position between [1-9] where you want to mark : "))

if(CheckPosition(choice)):

board[choice] = Mark

player+=1

CheckWin()

os.system('cls')

DrawBoard()

if(Game==Draw):

print("Game Draw")

elif(Game==Win):

player-=1

if(player%2!=0):

print("Player 1 Won")

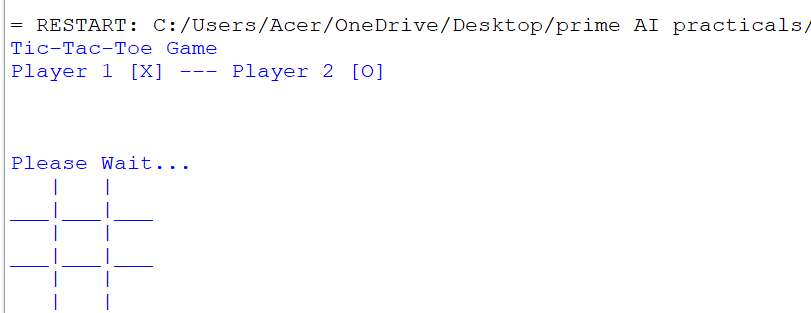
else:

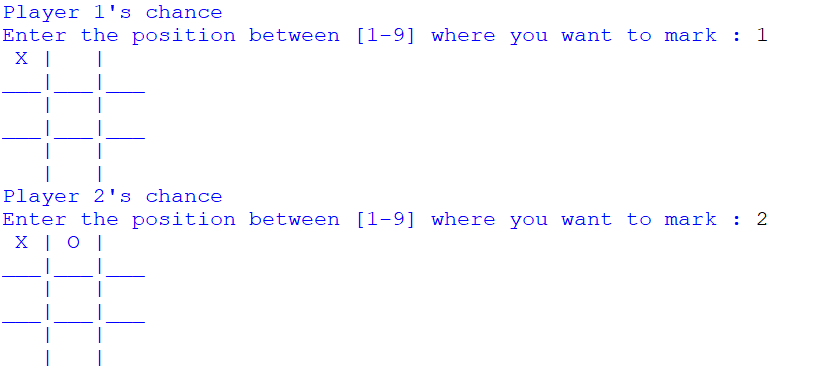
print("Player 2 Won")

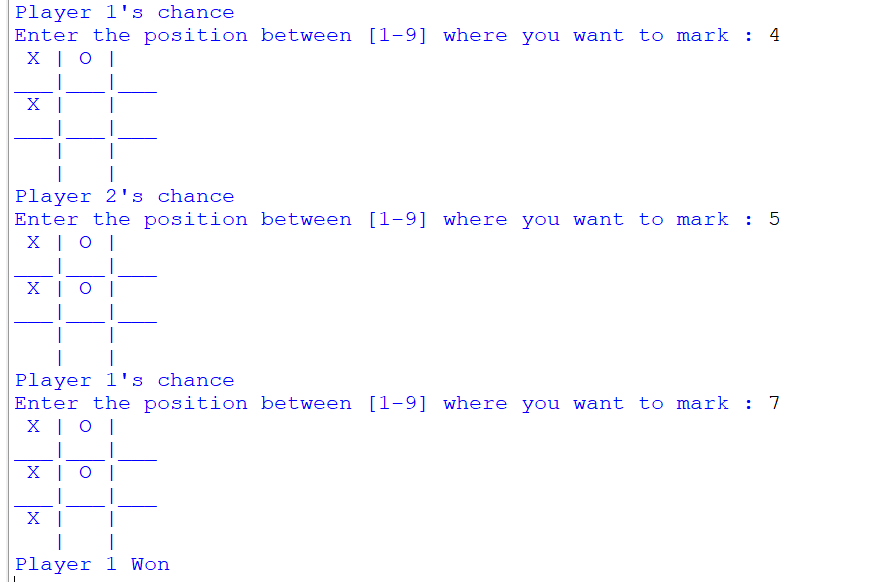
# NOTE:- Game Rules

1. Traditionally the first player plays with "X". So you can decide who wants to go with "X" and who wants go with "O".
2. Only one player can play at a time.
3. If any of the players have filled a square then the other player and the same player cannot override that square.
4. There are only two conditions that may match will be draw or may win.
5. The player that succeeds in placing three respective marks (X or O) in a horizontal, vertical or diagonal row wins the game.

Output:







# PRACTICAL No.-12

1. **Solve constraint satisfaction problem**

# Aim:-

Implementation Of Constraints Satisfactions Problem

**PYTHON CODE:**

from \_\_future\_\_

import print\_function

from simpleai.search

import CspProblem, backtrack, min\_conflicts,

MOST\_CONSTRAINED\_VARIABLE, HIGHEST\_DEGREE\_VARIABLE,

LEAST\_CONSTRAINING\_VALUE

variables = ('WA', 'NT', 'SA', 'Q', 'NSW', 'V', 'T')

domains = dict((v, ['red', 'green', 'blue']) for v in variables)

def const\_different(variables, values):

return values[0] != values[1] **# expect the value of the neighbors to be different**

constraints = [

(('WA', 'NT'), const\_different),

(('WA', 'SA'), const\_different),

(('SA', 'NT'), const\_different),

(('SA', 'Q'), const\_different),

(('NT', 'Q'), const\_different),

(('SA', 'NSW'), const\_different),

(('Q', 'NSW'), const\_different),

(('SA', 'V'), const\_different),

(('NSW', 'V'), const\_different),

]

my\_problem = CspProblem(variables, domains, constraints)

print(backtrack(my\_problem))

print(backtrack(my\_problem,

variable\_heuristic=MOST\_CONSTRAINED\_VARIABLE)) print(backtrack(my\_problem,

variable\_heuristic=HIGHEST\_DEGREE\_VARIABLE)) print(backtrack(my\_problem,

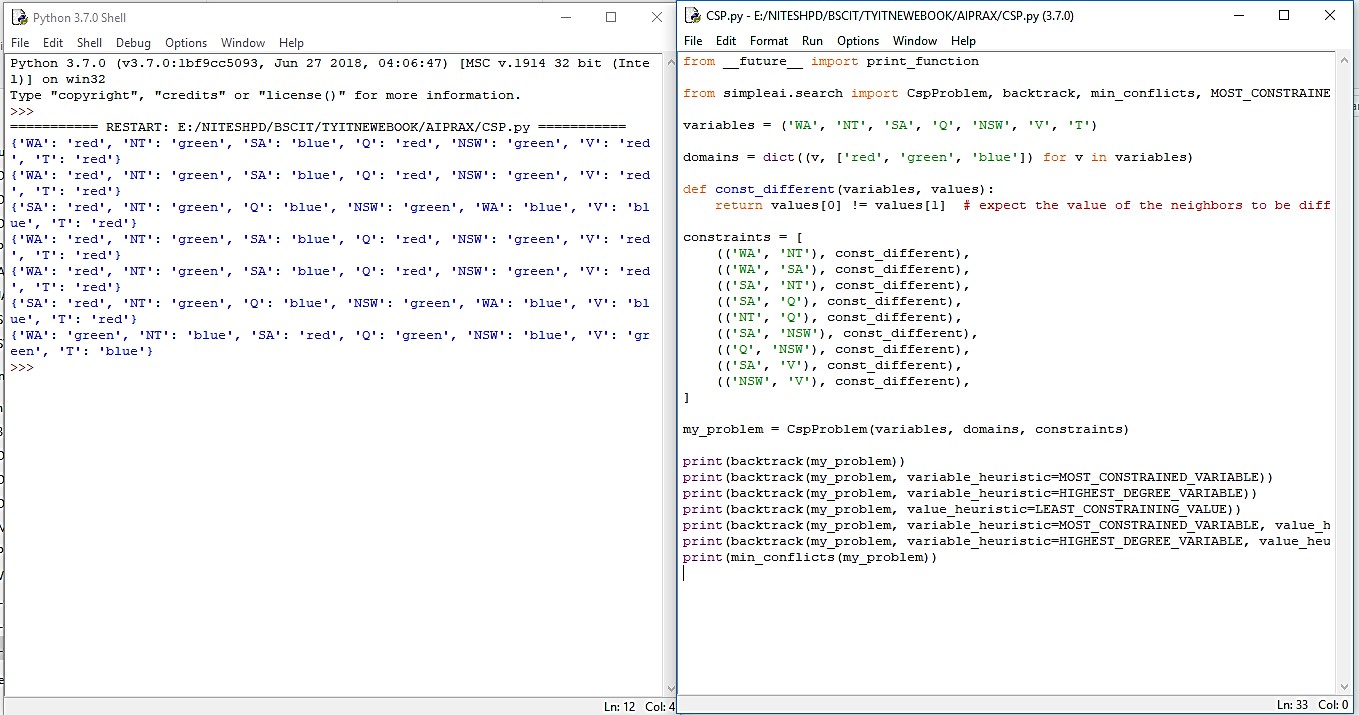
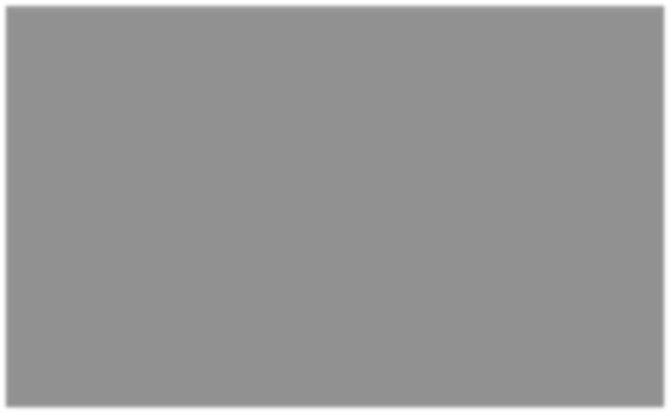
value\_heuristic=LEAST\_CONSTRAINING\_VALUE)) print(backtrack(my\_problem,

variable\_heuristic=MOST\_CONSTRAINED\_VARIABLE,

value\_heuristic=LEAST\_CONSTRAINING\_VALUE)) print(backtrack(my\_problem,

variable\_heuristic=HIGHEST\_DEGREE\_VARIABLE, value\_heuristic=LEAST\_CONSTRAINING\_VALUE)) print(min\_conflicts(my\_problem))

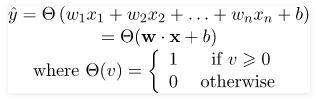
**OUTPUT:-**



# Practical no-13

**Perceptron algorithm for NAND logic gate**

In the field of Machine Learning, the Perceptron is a Supervised Learning Algorithm for binary classifiers. The Perceptron Model implements the following function:



For a particular choice of the weight vector x and bias parameter b, the model predicts output y(cap) for the corresponding input vector x.

**NAND** logical function truth table for ***2-bit binary variables***, i.e, the input vector x: (x1,x2) and the corresponding output y –

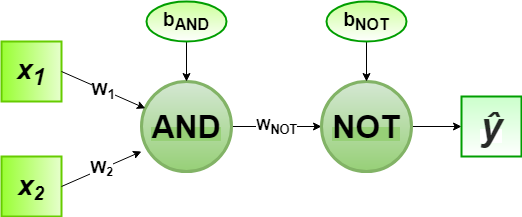
| X1 | X2 | y |
| --- | --- | --- |
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

We can observe that, NAND(x1,x2) = NOT (AND (x1,x2))  
Now for the corresponding weight vector w: (w1,w2) of the input vector x: (x1,x2)  to the AND node, the associated Perceptron Function can be defined as:

Y(cap)I = Θ(w1x1+w2x2+ band)

Later on, the output of AND node Y(cap)I is the input to the NOT node with weight wnot. Then the corresponding output Y(cap) is the final output of the NAND logic function and the associated Perceptron Function can be defined as:

Y(cap) = Θ(wnotY(cap)I + bnot)



For the implementation, considered weight parameters are w1= 1, w2= 1, wnot =-1 and the bias parameters are band=-1.5, bnot = 0.5.

**Source Code:**

# importing Python library

import numpy as np

# define Unit Step Function

def unitStep(v):

    if v >= 0:

        return 1

    else:

        return 0

# design Perceptron Model

def perceptronModel(x, w, b):

    v = np.dot(w, x) + b

    y = unitStep(v)

    return y

# NOT Logic Function

# wNOT = -1, bNOT = 0.5

def NOT\_logicFunction(x):

    wNOT = -1

    bNOT = 0.5

    return perceptronModel(x, wNOT, bNOT)

# AND Logic Function

# w1 = 1, w2 = 1, bAND = -1.5

def AND\_logicFunction(x):

    w = np.array([1, 1])

    bAND = -1.5

    return perceptronModel(x, w, bAND)

# NAND Logic Function

# with AND and NOT

# function calls in sequence

def NAND\_logicFunction(x):

    output\_AND = AND\_logicFunction(x)

    output\_NOT = NOT\_logicFunction(output\_AND)

    return output\_NOT

# testing the Perceptron Model

test1 = np.array([0, 1])

test2 = np.array([1, 1])

test3 = np.array([0, 0])

test4 = np.array([1, 0])

print("NAND({}, {}) = {}".format(0, 1, NAND\_logicFunction(test1)))

print("NAND({}, {}) = {}".format(1, 1, NAND\_logicFunction(test2)))

print("NAND({}, {}) = {}".format(0, 0, NAND\_logicFunction(test3)))

print("NAND({}, {}) = {}".format(1, 0, NAND\_logicFunction(test4)))

**Output:**

NAND(0, 1) = 1

NAND(1, 1) = 0

NAND(0, 0) = 1

NAND(1, 0) = 1

Conclusion:

Here, the model predicted output (ycap) for each of the test inputs are exactly matched with the NAND logic gate conventional output (y) according to the truth table for 2-bit binary input.  
Hence, it is verified that the perceptron algorithm for NAND logic gate is correctly implemented.