|  |  |  |  |
| --- | --- | --- | --- |
| **S. N** | **TITLE** | **DATE** | **SIGNATURE** |
| 1 | WAP to implement BFS. |  |  |
| 2 | WAP to implement DFS. |  |  |
| 3 | WAP to implement IDDFS. |  |  |
| 4 | WAP to implement GBFS. |  |  |
| 5 | WAP to implement A\* Search. |  |  |
| 6 | WAP for Water Jug Problem. |  |  |
| 7 | WAP to implement nqueen problem. |  |  |
| 8 | WAP to implement alpha beta search. |  |  |
| 9 | WAP for Hill Climbing Problem. |  |  |
| 10 | Design the simulation of tic – tac – toe game using min-max algorithm. |  |  |
| 11 | WAP to solve constraint satisfaction problem. |  |  |
| 12 | Implementation of NAND Logic Gate. |  |  |

**INDEX**

**Q. WAP to implement A\* Search.**

# a\* search

adjacency\_list={

's':[('a',1),('g',10)],

'a': [('b', 2), ('c', 1)],

'b': [('d', 5)],

'c': [('d', 3),('g',4)],

'd': [('g',2)],

'g': []

}

heuristic = {

's': 5,

'a': 3,

'b': 4,

'c': 2,

'd': 6,

'g': 0

}

def a\_star\_algorithm(adjacency\_list, heuristic, start\_node,goal\_node):

open\_list = set([start\_node])

closed\_list = set([])

g = {}

g[start\_node] = 0

parents = {}

parents[start\_node] = start\_node

def get\_neighbors(node):

return adjacency\_list[node]

def h(node):

return heuristic[node]

while len(open\_list) > 0:

n = None

for v in open\_list:

if n == None or g[v] + h(v) < g[n] + h(n):

n = v;

if n == None:

print('Path does not exist!')

return None

if n == goal\_node:

reconst\_path = []

while parents[n] != n:

reconst\_path.append(n)

n = parents[n]

reconst\_path.append(start\_node)

reconst\_path.reverse()

print('Path found: {}'.format(reconst\_path))

return reconst\_path

for (m, weight) in get\_neighbors(n):

if m not in open\_list and m not in closed\_list:

open\_list.add(m)

parents[m] = n

g[m] = g[n] + weight

else:

if g[m] > g[n] + weight:

g[m] = g[n] + weight

parents[m] = n

if m in closed\_list:

closed\_list.remove(m)

open\_list.add(m)

open\_list.remove(n)

closed\_list.add(n)

print('Path does not exist!')

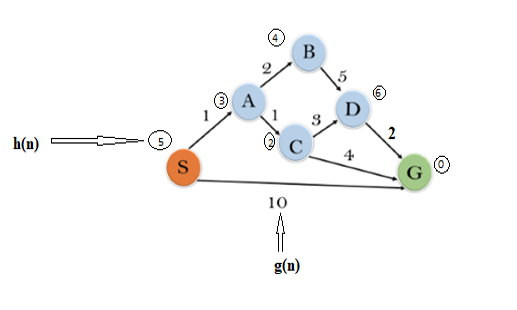
return None

start\_node = input("Enter the start node: ")

goal\_node = input("Enter the stop node: ")

a\_star\_algorithm(adjacency\_list, heuristic, start\_node,goal\_node)

**Graph**

****

**Output**

**Q. WAP for Water Jug Problem.**

1. j1 and j2 are two jugs,
2. (x, y): order pair,
3. x: maximum water storage capacity of jug1 is 4 gallons i.e. x=4,
4. y: maximum water storage capacity of jug2 is 3 gallons i.e. y=3,
5. No mark on jug,
6. Pump to fill the water into the jug,
7. How can you get exactly 2 gallons of water into the 4 gallons jug?

**Solution**

Solution 1

|  |  |  |  |
| --- | --- | --- | --- |
| **States** | **Jug1 (4 Gallons)** | **Jug2 (3 Gallons)** | **Rules** |
| **Initial State** | 0 | 0 | - |
|  | 0 | 3 | 2 |
|  | 3 | 0 | 8 |
|  | 3 | 3 | 2 |
|  | 4 | 2 | 6 |
|  | 0 | 2 | 3 |
| **Final State** | **2** | 0 | 8 |

Solution 2

|  |  |  |  |
| --- | --- | --- | --- |
| **States** | **Jug1 (4 Gallons)** | **Jug2 (3 Gallons)** | **Rules** |
| **Initial State** | 0 | 0 | - |
|  | 4 | 0 | 1 |
|  | 1 | 3 | 5 |
|  | 1 | 0 | 4 |
|  | 0 | 1 | 7 |
|  | 4 | 1 | 1 |
| **Final State** | **2** | 3 | 5 |

|  |  |  |  |
| --- | --- | --- | --- |
| **Rules** | **Jugs** | **Tasks** | **State/logic** |
| 1 | (j1, j2) | Fill the 4-gallon jug | if(r==1):  j1=x; |
| 2 | (j1, j2) | Fill the 3-gallon jug | elif(r==2):  j2=y |
| 3 | (j1, j2) | Empty the 4-gallon jug on the ground | elif(r==3):  j1=0; |
| 4 | (j1, j2) | Empty the 3-gallon jug on the ground | elif(r==4):  j2=0; |
| 5 | (j1, j2) | Pour water from the 4-gallon jug into the 3-gallon jug **until the 3-gallon jug is full** | elif(r==5):  t=y-j2;  **j2=y;**  j1-=t  if j1<0  j1=0 |
| 6 | (j1, j2) | Pour water from the 3-gallon jug into the 4-gallon jug **until the 4-gallon jug is full** | elif(r==6):  t=x-j1;  **j1=x**  j2-=t;  if j2<0:  j2=0; |
| 7 | (j1, j2) | Pour all water from the 4-gallon jug into the 3-gallon jug **until the 4-gallon jug becomes empty** | elif(r==7):  j2+=j1  **j1=0;**  if j2>y  j2=y |
| 8 | (j1, j2) | Pour water from the 3-gallon jug  into the 4-gallon jug **until the 3-gallon jug becomes empty** | elif(r==8):  j1+=j2  **j2=0;**  if j1>x  j1=x |

**Q. WAP to implement alpha beta search.**

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 = nalpha if nalpha < beta else beta

else:

alpha = nbeta if nbeta > alpha else alpha

branch[i] = alpha if depth % 2 == 0 else beta

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 = alpha if root == 0 else beta

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(None)

**Output**

**Q. WAP 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**

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.

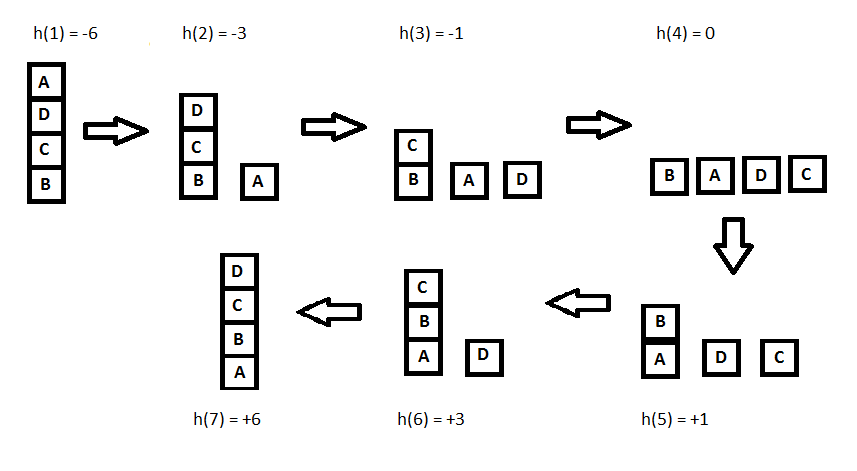
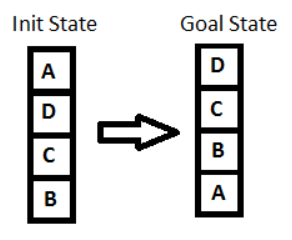
Step 3: Select and apply an operator to the current state.

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 assigns new state as a current state.
3. Else if not better than the current state, then return to step2.

Step 5: Exit.

**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:

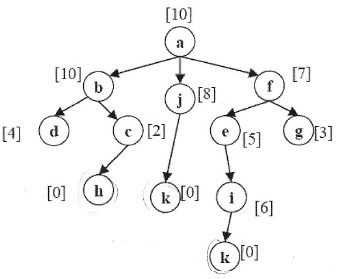
print(f"ANSWER:\nFor given Data, the local maxima is at node '{node}' with heuristic value {value}")

nodeInput(numberNode)

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

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

**Tree**



**Output**

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

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 if the position is empty or not*

def CheckPosition(x):

if board[x] == ' ':

return True

else:

return False

*# This Function Checks if the 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

while Game == Running:

os.system('cls')

print("21-00031-5")

print("Tic-Tac-Toe")

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

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 1 <= choice <= 9 and CheckPosition(choice):

board[choice] = Mark

player += 1

CheckWin()

os.system('cls')

print("21-00031-5")

print("Tic-Tac-Toe")

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

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")

**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**

**Q. WAP to solve constraint satisfaction problem.**

def backtracking\_search(variables, domains, constraints):

return backtrack({}, variables, domains, constraints)

def backtrack(assignment, variables, domains, constraints):

if len(assignment) == len(variables):

return assignment

var = select\_unassigned\_variable(assignment, variables)

for value in order\_domain\_values(var, assignment, domains):

assignment[var] = value

if is\_consistent(var, value, assignment, constraints):

result = backtrack(assignment, variables, domains, constraints)

if result is not None:

return result

del assignment[var] # Backtrack

return None

def select\_unassigned\_variable(assignment, variables):

for var in variables:

if var not in assignment:

return var

return None # All variables are assigned

def order\_domain\_values(var, assignment, domains):

return domains[var] # Return the domain as it is for simplicity

def is\_consistent(var, value, assignment, constraints):

for constraint in constraints:

if not constraint(var, value, assignment):

return False

return True

*# Example constraint function: Different values for A and B*

def different\_values\_constraint(A, B, assignment):

return assignment.get(A) != assignment.get(B)

*# Example usage*:

variables = ['A', 'B', 'C']

domains = {

'A': [1, 2, 3],

'B': [1, 2],

'C': [3, 4]

}

constraints = [different\_values\_constraint]

solution = backtracking\_search(variables, domains, constraints)

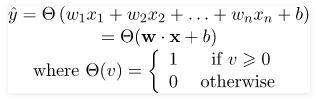
print(solution)

**Output**

**Q. Implementation of NAND Logic Gate.**

**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.

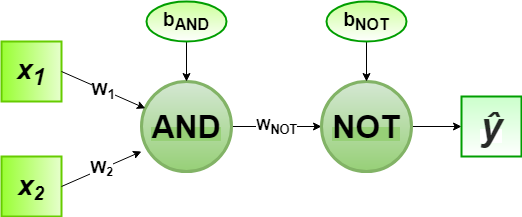
NAND logical function truth table for 2-bit binary variables, i.e, the input vector x: (x1,x2) and the corresponding output y:

| 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**

*# define Unit Step Function*

def unitStep(v):

if v >= 0:

return 1

else:

return 0

*# design Perceptron Model*

def perceptronModel(x, w, b):

v = sum([w[i] \* x[i] for i in range(len(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 = [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 = [0, 1]

test2 = [1, 1]

test3 = [0, 0]

test4 = [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**