01. Breadth-First Search

Breadth-first search (BFS) is an algorithm for traversing or searching for tree or graph data structures. It starts at the tree root (or some arbitrary node of a graph, sometimes referred to as a 'search key'[1]) and explores all of the neighbour nodes at the present depth before moving on to the nodes at the next depth level.

****Algorithm: ****

1. Create a queue and enqueue the source into it. Mark the source as visited.
2. While the queue is not empty, do the following
   1. Dequeue a vertex from queue. Let this be u.
   2. If u is the destination, then return true.
   3. Else, continue following steps
   4. Enqueue all adjacent vertices of u that are not yet visited.
   5. Mark them visited and continue from step 2.

****Implementation****

```
'''
Program to implement Breadth First Search
'''
graph = {
  'A': ['B', 'C'],
  'B': ['D', 'E'],
  'C': ['F'],
  'D': ['G', 'H'],
  'E': ['I'],
  'F': [],
  'G': [],
  'H': [],
  'I': []
}

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


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

  while queue:        # Creating loop to visit each node
    m = queue.pop(0)
    # print '->' after each node except the last one
    print(m, end='->' if m != 'I' else '')

    for neighbour in graph[m]:
```

```
        if neighbour not in visited:
            visited.append(neighbour)
            queue.append(neighbour)


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

****Output****

02. Depth First Search

Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking.

****Algorithm: ****

1. Create a stack and push the source into it—Mark the source as visited.
2. While the stack is not empty, do the following
   1. Pop a vertex from the stack. Let this be u.
   2. If U is the destination, then return true.
   3. Else, continue following steps
   4. Push all adjacent vertices of u which are not yet visited.
   5. Mark them visited and continue from step 2.

****Implementation****

```
'''
Program to implement Depth First Search
'''

# Using a Python dictionary to act as an adjacency list
graph = {
   'A': ['B', 'C'],
   'B': ['D', 'E'],
   'C': ['F'],
   'D': ['G', 'H'],
   'E': ['I'],
   'F': [],
   'G': [],
   'H': [],
   'I': []
}

visited = set()  # Set to keep track of visited nodes of graph.


def dfs(visited, graph, node):  # function for dfs
   if node not in visited:
      # print '->' after each node except the last one
      print(node, end='->' if node != 'F' else '')
      visited.add(node)
      for neighbour in graph[node]:
         dfs(visited, graph, neighbour)
```

```
# Driver Code
print("Following is the Depth-First Search")
dfs(visited, graph, 'A')
```

Output

03. Iterative Deepening Depth First Search

Iterative deepening depth-first search (IDDFS) is an extension to the 'vanilla' depth-first search algorithm, with an added constraint on the total depth explored per iteration. IDDFS is optimal like breadth-first search, but uses much less memory; at each iteration, it visits the nodes in the search tree in the same order as depth-first search, but the cumulative order in which nodes are first visited is effectively breadth-first.

****Algorithm: ****

1. Perform a depth-first search to depth 1, starting from the root.
2. Provided it didn't find the goal, perform a depth-first search to depth 2, starting from the root.
3. Provided it didn't find the goal, perform a depth-first search to depth 3, etc., etc.

****Implementation****

```
'''
Program to implement iterative deepening depth-first search
'''

graph = {
   'A': ['B', 'C'],
   'B': ['D', 'E'],
   'C': ['F', 'G'],
   'D': [],
   'E': [],
   'F': [],
   'G': []
}

nodes_visited_level = {}
current_level = -1


def dfs(node, goal, d_limit, start, visited, path):
   global current_level

   if start == d_limit:
      current_level += 1

      nodes_visited_level[current_level] = []
   nodes_visited_level[current_level].append(node)

   if node == goal:
      return "FOUND", path + [node]
   elif d_limit == 0:
```

```
        return "NOT_FOUND", None
    else:
        visited.add(node)
        for child in graph[node]:
            if child not in visited:
                result, tv_path = dfs(
                    child, goal, d_limit - 1, start, visited, path + [node])
                if result == 'FOUND':
                    return "FOUND", tv_path
        return "NOT_FOUND", None


def iddfs(root, goal):
    d_limit = 0
    while True:
        visited = set()
        start = d_limit
        result, tv_path = dfs(root, goal, d_limit, start, visited, [])

        if result == "FOUND":
            return "Goal node found! Traversed path:" + '->'.join(tv_path)
        elif result == "NOT_FOUND":
            d_limit += 1


root = input("Enter the start node: ")
goal = input("Enter the goal node: ")
result = iddfs(root, goal)

for level, nodes in nodes_visited_level.items():
    print("Depth limit: "+str(level) + " Traversed path: ", end="")
    for node in nodes:
        print(node, end="")
        if node != nodes[-1]:
            print("->", end="")
    print()

print(result)
```

****Output****

04. Greedy Best First Search

Greedy best-first search is a best-first search that uses heuristics to improve speed. It expands the most promising node chosen according to the heuristic function. Greedy best-first search algorithm is a search algorithm that is used to solve many problems among them the most common one is the 8-puzzle. It is an informed search algorithm as it uses information about path cost and also uses heuristics to find the solution. It is a best-first search algorithm in which the cost associated with a node is f(n) = h(n). It expands the node that is estimated to be closest to the goal. It is not optimal as it does not guarantee the shortest path. It is complete as it will always find a solution if one exists. It is faster than the breadth-first search but slower than the A* search.

****Algorithm:****

1. Create a priority queue and enqueue source into it. Mark source as visited.
2. While queue is not empty, do following
   1. Dequeue a vertex from queue. Let this be u.
   2. If u is the destination, then return true.
   3. Else, continue following steps
   4. Enqueue all adjacent vertices of u which are not yet visited.
   5. Mark them visited and continue from step 2.

****Implementation****

```
'''
Program: Greedy Best First Search
'''

graph = {
   'S': {'A': 3, 'B': 2},
   'A': {'C': 4, 'D': 1},
   'B': {'E': 3, 'F': 1},
   'C': {},
   'D': {},
   'E': {'H': 5},
   'F': {'I': 2, 'G': 3},
   'G': {},
   'H': {},
   'I': {},
}

heuristic = {
   'S': 13,
   'A': 12,
   'B': 4,
   'C': 7,
```

```
    'D': 3,
    'E': 8,
    'F': 2,
    'G': 0,
    'H': 4,
    'I': 9,

}


def gbfs(graph, heuristic, start, goal):
    visited = set()
    queue = [(heuristic[start], [start])]
    while queue:
        (h, path) = queue.pop(0)
        current_node = path[-1]
        if current_node == goal:
            return path
        visited.add(current_node)
        for neighbor, distance in graph[current_node].items():
            if neighbor not in visited:
                new_path = path + [neighbor]
                queue.append((heuristic[neighbor], new_path))
        queue.sort()
    return None


print("Greedy best first search")
start = input("Enter the start node: ")
goal = input("Enter the goal node: ")

traversed_path = gbfs(graph, heuristic, start, goal)

if traversed_path:
    print(f"Goal node found: \n{traversed_path}\n")

else:
    print("Goal node not found\n")
```

****Output****

05. A* Search

A* is the most popular choice for pathfinding because it's flexible and can be used in a wide range of contexts. A* is like Dijkstra's Algorithm in that it can be used to find the shortest path. A* is like Greedy Best-First Search in that it can use a heuristic to guide itself.

****Algorithm: ****

1. Create a priority queue and enqueue the source into it. Mark the source as visited.
2. While the queue is not empty, do the following
   1. Dequeue a vertex from queue. Let this be u.
   2. If u is the destination, then return true.
   3. Else, continue following steps
   4. Enqueue all adjacent vertices of u that are not yet visited.
   5. Mark them visited and continue from step 2.

****Implementation****

```
'''
Program to implement A Star Search Algorithm
'''

# Defining the graph nodes in dict with given costs to traverse
adj_list = {
   's': [('a', 1), ('g', 10)],
   'a': [('b', 2), ('c', 1)],
   'b': [('d', 5)],
   'c': [('d', 3), ('g', 4)],
   'd': [('g', 2)],
   'g': []
}

# Defining heuristic values for each nodes
heuristic = {
   's': 5,
   'a': 3,
   'b': 4,
   'c': 2,
   'd': 6,
   'g': 0
}

# A Star Search Algorithm


def astar_search(adj_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 adj_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


print("----- A star search -----")
start_node = input("Enter the start node: ")
goal_node = input("Enter the goal node: ")

astar_search(adj_list, heuristic, start_node, goal_node)
```

****Output****

06. n Queens Problem

The n-queens puzzle is the problem of placing n queens on an n×n chessboard such that no two queens attack each other. Given an integer n, print all distinct solutions to the n-queens puzzle. Each solution contains distinct board configurations of the n-queens' placement, where the solutions are a permutation of [1,2,3..n] in increasing order, here the number in the ith place denotes that the ith-column queen is placed in the row with that number. For eg below figure represents a chessboard [3 1 4 2].

****Algorithm:****

1. Start in the leftmost column
2. If all queens are placed
   1. return true
3. Try all rows in the current column. Do following for every tried row.
   1. If the queen can be placed safely in this row then mark this [row, column] as part of the solution and recursively check if placing queen here leads to a solution.
   2. If placing the queen in [row, column] leads to a solution then return true.
   3. If placing queen doesn't lead to a solution then unmark this [row, column] (Backtrack) and go to step (a) to try other rows.
4. If all rows have been tried and nothing worked, return false to trigger backtracking.

****Implementation****

```
'''
Program to implement for N-Queen problem
'''


'''
logic where the queen must not be placed:
1) row, col+-
2) row--, col--
3) row++, col--
'''



def is_safe(board, row, col, n):
    for c in range(col, -1, -1):  # check for the same row in left side of the board
        if board[row][c] == 'Q':
            return False

    i = row
    j = col

    while i >= 0 and j >= 0:  # check for the left diagonal in the upper side of the board
        if board[i][j] == 'Q':
            return False
```

```
        i -= 1
        j -= 1

    i = row
    j = col

    while i < n and j >= 0:  # check for the left diagonal in the bottom side of the board
        if board[i][j] == 'Q':
            return False

        i += 1
        j -= 1

    return True


def nqueen(board, col, n):
    if col >= n:
        return True
    for i in range(n):
        if is_safe(board, i, col, n):
            board[i][col] = 'Q'
            if nqueen(board, col+1, n):
                return True
            board[i][col] = 0
    return False


n = int(input("Enter the number of queens: "))
board = [[0 for j in range(n)] for i in range(n)]

if nqueen(board, 0, n) == True:
    for i in range(n):
        for j in range(n):
            print(board[i][j], end=' ')
        print()
else:
    print("Not possible")
```

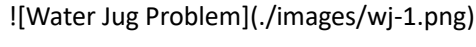
****Output****

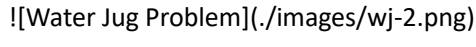
07. Water Jug Problem

****Algorithm****

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 gallon of water into the 4-gallons jug?

Solution:





****Implementation****

```
'''
Program to solve the water jug problem using state space search
'''

j1 = 0
j2 = 0
x = 4
y = 3
print("Initial state: (0, 0)")
print("Capacities: (4, 3)")
print("Goal state: (2, 0 or any number)")

while j1 != 2:
   r = int(input("Enter the rule: "))
   if (r == 1):
      j1 = x
   elif (r == 2):
      j2 = y
   elif (r == 3):
      j1 = 0
   elif (r == 4):
```

```
    j2 = 0
elif (r == 5):
    t = y-j2
    j2 = y
    j1 -= t
    if j1 < 0:
        j1 = 0
elif (r == 6):
    t = x-j1
    j1 = x
    j2 -= t
    if j2 < 0:
        j2 = 0
elif (r == 7):
    j2 += j1
    j1 = 0
    if j2 > y:
        j2 = y
elif (r == 8):
    j1 += j2
    j2 = 0
    if j1 > x:
        j1 = x
print(j1, j2)
```

Output

08. Tower of Hanoi

The Tower of Hanoi is a mathematical game or puzzle. It consists of three rods and a number of disks of different sizes, which can slide onto any rod. The puzzle starts with the disks stacked on one rod in order of decreasing size, the smallest at the top, thus approximating a conical shape. The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:

1. Only one disk can be moved at a time.
2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack or on an empty rod.
3. No larger disk may be placed on top of a smaller disk.

**Algorithm: **

1. Move top n-1 disks from the source to the auxiliary tower.
2. Move the nth disk from the source to the destination tower.
3. Move the n-1 disks from the auxiliary tower to the destination tower.

****Implementation****

```
'''
Program to implement the tower of hanoi
'''


def toh(n, source, destination, temp):
    if n == 1:
        print(f"Move disk 1 from {source} to {destination}")
        return
    toh(n - 1, source, temp, destination)
    print(f"Move disk {n} from {source} to {destination}")
    toh(n-1, temp, destination, source)


# Enter the number of rods
n = int(input("Enter the number of rods: "))
toh(n, 'Source', 'Destination', 'Temp')
```

****Output****

09. Alpha Beta Pruning

Alpha–beta pruning is a search algorithm that seeks to decrease the number of nodes that are evaluated by the minimax algorithm in its search tree. It is an adversarial search algorithm used commonly for machine playing of two-player games (Tic-tac-toe, Chess, Go, etc.). It stops evaluating a move when at least one possibility has been found that proves the move to be worse than a previously examined move. Such moves need not be evaluated further. When applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision.

****Algorithm: ****

1. Initialize alpha with -INFINITY and beta with +INFINITY.
2. Maximizer and Minimizer are two players.
3. Maximizer makes a move and calls Minimizer.
4. Minimizer makes a move and calls Maximizer.
5. Repeat the above two steps till any one of the player wins the game or the game ends.
6. At each level, compare the value of alpha and beta.
7. If alpha is greater than or equal to beta, stop further evaluation and return alpha.
8. Else, return beta.

****Implementation****

```
'''
Program to implement Alpha-Beta Pruning Algorithm
'''

tree = [
   [[5, 1, 2], [8, -8, -9]],
   [[9, 4, 5], [-3, 4, 3]]
]  # tree to search
root = 0  # root depth
pruned = 0  # times pruned

# function to search tree


def children(branch, depth, alpha, beta):
   global root  # global root depth to compare with current depth
   global pruned  # global times pruned to count times pruned
   i = 0  # index of child

   for child in branch:
      if type(child) is list:  # if child is a list, call children function recursively
         (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
  return (alpha, beta)

# function to call search


def alphabeta(branch=tree, depth=root, alpha=-15, beta=15):
  global pruned

  (alpha, beta) = children(branch, depth, alpha, beta)

  if depth == root:
    best_move = max(branch) if depth % 2 == 0 else min(branch)
    print("(alpha, beta): ", alpha, beta)
    print("Result: ", best_move)
    print("Times pruned: ", pruned)

  return (alpha, beta, branch, pruned)


if __name__ == "__main__":
  alphabeta()
```

****Output****

10. 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:****

1. Evaluate the initial state, if it is goal state then return success and Stop.
2. Loop Until a solution is found or there is no new operator left to apply.
3. Select and apply an operator to the current state.
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.
5. Exit.

****Implementation****

```
'''
Program to find local maxima using hill climbing search
'''

# This dictionary holds all the nodes with their successors and their corresponding heuristic value
adjList = {
   'A': [('B', 10), ('J', 8), ('F', 7)],
   'B': [('D', 4), ('C', 2)],
   'C': [('H', 0)],
   'E': [('I', 6)],
   'F': [('E', 5), ('G', 3)],
   'I': [('K', 0)],
   'J': [('K', 0)],
}

# root node
initial_node = str(input("Input initial node: ")).capitalize()
# holds heuristic value of root node
initial_value = eval(input(f"Input {initial_node}'s heuristic value: "))
```

```
# Function to sort the selected list in ascending order based on heuristic value


def sortList(new_list):
    new_list.sort(key=lambda x: x[1])
    return new_list


# Function to find shortest path using heuristic value
def hillClimbing_search(node, value):
    new_list = list()
    if node in adjList.keys():
        new_list = adjList[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"\nLocal maxima at node: '{node}'\nHeuristic value: {value}")
    else:
        print(f"\nLocal maxima at node: '{node}'\nHeuristic value: {value}")


if __name__ == "__main__":
    hillClimbing_search(initial_node, initial_value)
```

****Output****

11. tic-tac-toe

Tic-tac-toe is a game played by two players on a 3x3 grid. In this game, the first player marks move with a circle, the second with a cross. The player who has formed a horizontal, vertical, or diagonal sequence of three marks wins. Your task is to write a program that determines the status of a tic-tac-toe game.

****Algorithm: ****

1. Create a 3x3 grid with all values 0.
2. Take input from user for row and column.
3. Check if the position is already filled or not.
4. If not filled, then check if it is player 1 or player 2.
5. If player 1, then fill the position with 1.
6. If player 2, then fill the position with 2.
7. Check if any player has won the game or not.
8. If yes, then print the winner and exit.
9. If no, then continue from step 2.

****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.

****Implementation****

```
'''
Program to implement Tic Tac Toe Game
'''

# TODO retry if the position is already taken

board = [' ' for x in range(9)]
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[0], board[1], board[2]))
    print("___|___|___")
    print(" %c | %c | %c " % (board[3], board[4], board[5]))
    print("___|___|___")
    print(" %c | %c | %c " % (board[6], board[7], board[8]))
    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[0] == board[1] and board[1] == board[2] and board[0] != ' '):
        Game = Win
    elif (board[3] == board[4] and board[4] == board[5] and board[3] != ' '):
        Game = Win
    elif (board[6] == board[7] and board[7] == board[8] and board[6] != ' '):
        Game = Win

    # Vertical Winning Condition
    elif (board[0] == board[3] and board[3] == board[6] and board[0] != ' '):
        Game = Win
    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

    # Diagonal Winning Condition
    elif (board[0] == board[4] and board[4] == board[8] and board[4] != ' '):
        Game = Win
    elif (board[2] == board[4] and board[4] == board[6] and board[4] != ' '):
```

```
        Game = Win

    # Match Tie or Draw Condition
    elif (board[0] != ' ' and
          board[1] != ' ' and
          board[2] != ' ' and
          board[3] != ' ' and
          board[4] != ' ' and
          board[5] != ' ' and
          board[6] != ' ' and
          board[7] != ' ' and
          board[8] != ' '):
        Game = Draw
    else:
        Game = Running


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

while (Game == Running):
    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 [0-8] where you want to mark: "))
    if (CheckPosition(choice)):
        board[choice] = Mark
        player += 1
        CheckWin()

DrawBoard()
if (Game == Draw):
    print("Game is tied! 🥉 🏆 ")
elif (Game == Win):
    player -= 1

if (player % 2 != 0):
    print("Player 1 Wins! ❌ 🏆 ")
else:
```

```
print("Player 2 Wins!⭕🏆 ")
```

Output

12. Constraint Satisfaction Problem

Constraint satisfaction problems (CSPs) are mathematical questions defined as a set of objects whose state must satisfy several constraints or limitations. CSPs represent the entities in a problem as a homogeneous collection of finite constraints over variables, which is solved by constraint satisfaction methods. CSPs are the subject of intense research in both artificial intelligence and operations research since the regularity in their formulation provides a common basis to analyze and solve problems of many seemingly unrelated families. CSPs often exhibit high complexity, requiring a combination of heuristics and combinatorial search methods to be solved in a reasonable time. CSPs are applied in many areas including scheduling, planning, vehicle routing, configuration, and the design of communication networks.

****Algorithm:****

1. Initialize all the variables with their domain values.
2. Select a variable and assign it a value from its domain.
3. Check if the assignment is consistent with the constraints.
4. If yes, then select another variable and assign it a value from its domain.
5. If no, then backtrack and select another value from the domain of the previous variable.
6. Repeat steps 3 to 5 until all the variables are assigned with a value from their domain.

****Implementation****

```
'''
Program to implement the csp
'''

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)

# function that returns True if the neighbors of the variables have different values
def const_different(variables, values):
    # expect the value of the neighbors to be different
    return values[0] != values[1]

# constraints between neighbors (states) to have different colors (values)
constraints = [
    (('WA', 'NT'), const_different),
    (('WA', 'SA'), const_different),
    (('SA', 'NT'), const_different),
    (('SA', 'Q'), const_different),
```

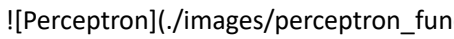
```
    (('NT', 'Q'), const_different),
    (('SA', 'NSW'), const_different),
    (('Q', 'NSW'), const_different),
    (('SA', 'V'), const_different),
    (('NSW', 'V'), const_different),
]

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****

13. NAND using Perceptron

A perceptron is a single-layer neural network. It is the simplest neural network model. It consists of a single layer of one or more perceptron (neurons).

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

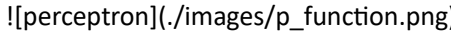


For a particular choice of the weight vector x and bias parameter b, the model predicts output y(cap) for the corresponding input vector x.

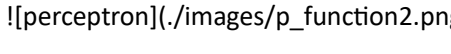
NAND logical function truth table for 2-bit binary variables, i.e, the input vector x: (x1,x2) and the corresponding output y –

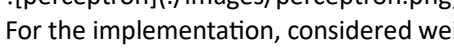
x1	x2	y
0	0	1
0	1	1
1	0	1
1	1	0

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:



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:





For the implementation, considered weight parameters are w1 = 1, w2 = 1, w_{not} = -1 and the bias parameters are band=-1.5, b_{not} = 0.5.

Algorithm:

1. Initialize the weights and bias with random values.
2. Take inputs from the user.
3. Calculate the weighted sum of inputs and weights.
4. Pass the weighted sum through the activation function.
5. Calculate the error.
6. Update the weights and bias.
7. Repeat steps 3 to 6 until the error is 0.

Implementation

```
'''
Program to implement NAND Logic Function using Perceptron Model
'''

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

if __name__ == '__main__':
    x, y = input("Enter the input x, y: ").split(' ')
    x = int(x)
```

```
    y = int(y)
    print(f"NAND({x}, {y}) = {NAND_logicFunction(np.array([x, y]))}")

# # 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****