*AIM* **: Implement an Informed (heuristic) A\* search strategy for optimal - 4 shortest path and traffic navigational system**

import heapq

graph = {

's': {'b': 4, 'c': 3},

'b': {'s': 4, 'f': 5, 'e': 12},

'f': {'b': 5, 'g': 16},

'c': {'s': 3, 'e': 10, 'd': 7},

'd': {'c': 7, 'e': 2},

'e': {'b': 12, 'c': 16, 'd': 2, 'g': 5},

'g': {'f': 16, 'e': 5}

}

heuristics = {

's': 14,

'b': 12,

'f': 11,

'c': 11,

'd': 6,

'e': 4,

'g': 0 # Add the heuristic value for 'g'

}

def astar(start, goal):

open\_list, closed\_list, parent = [(heuristics[start], 0, start)], set(), {}

while open\_list:

\_, cost, current = heapq.heappop(open\_list)

if current == goal:

path = [current]

while current in parent:

path.append(current := parent[current])

return path[::-1]

closed\_list.add(current)

for neighbor, edge\_cost in graph[current].items():

if neighbor not in closed\_list:

new\_cost = cost + edge\_cost

heapq.heappush(open\_list, (new\_cost + heuristics[neighbor], new\_cost, neighbor))

parent[neighbor] = current

return None

start\_node = 's'

goal\_node = 'g'

path = astar(start\_node, goal\_node)

if path:

print("Optimal Path:", " -> ".join(path))

else:

print("No path found.")

AIM : **To Implement the Water Jug Problem using an Uninformed Searching Technique**

from collections import deque

def water\_jug\_bfs(jug1\_capacity, jug2\_capacity, target):

visited = set()

queue = deque([(0, 0)])

while queue:

jug1, jug2 = queue.popleft()

if jug1 == target or jug2 == target:

return (jug1, jug2)

# Fill Jug 1

if jug1 < jug1\_capacity:

fill\_jug1 = (jug1\_capacity, jug2)

if fill\_jug1 not in visited:

visited.add(fill\_jug1)

queue.append(fill\_jug1)

# Fill Jug 2

if jug2 < jug2\_capacity:

fill\_jug2 = (jug1, jug2\_capacity)

if fill\_jug2 not in visited:

visited.add(fill\_jug2)

queue.append(fill\_jug2)

# Empty Jug 1

if jug1 > 0:

empty\_jug1 = (0, jug2)

if empty\_jug1 not in visited:

visited.add(empty\_jug1)

queue.append(empty\_jug1)

# Empty Jug 2

if jug2 > 0:

empty\_jug2 = (jug1, 0)

if empty\_jug2 not in visited:

visited.add(empty\_jug2)

queue.append(empty\_jug2)

# Pour from Jug 1 to Jug 2

if jug1 > 0 and jug2 < jug2\_capacity:

pour\_jug1\_to\_jug2 = (max(jug1 - (jug2\_capacity - jug2), 0), min(jug2 + jug1, jug2\_capacity))

if pour\_jug1\_to\_jug2 not in visited:

visited.add(pour\_jug1\_to\_jug2)

queue.append(pour\_jug1\_to\_jug2)

# Pour from Jug 2 to Jug 1

if jug2 > 0 and jug1 < jug1\_capacity:

pour\_jug2\_to\_jug1 = (min(jug1 + jug2, jug1\_capacity), max(jug2 - (jug1\_capacity - jug1), 0))

if pour\_jug2\_to\_jug1 not in visited:

visited.add(pour\_jug2\_to\_jug1)

queue.append(pour\_jug2\_to\_jug1)

return None

# Example usage:

jug1\_capacity = 4

jug2\_capacity = 3

target\_amount = 2

result = water\_jug\_bfs(jug1\_capacity, jug2\_capacity, target\_amount)

if result:

print("Solution found:")

print(f"Jug 1: {result[0]}, Jug 2: {result[1]}")

else:

print("No solution found.")

Aim **: *To implement a Hill climbing algorithm using Heuristic Searching Technique***

import random

def objective\_function(solution):

return sum(solution)

# Define the hill climbing algorithm

def hill\_climbing(max\_iterations, problem\_size):

current\_solution = [random.randint(0, 100) for \_ in range(problem\_size)]

current\_score = objective\_function(current\_solution)

for \_ in range(max\_iterations):

neighbor\_solution = current\_solution[:]

index\_to\_change = random.randint(0, problem\_size - 1)

neighbor\_solution[index\_to\_change] += random.randint(-10, 10)

neighbor\_score = objective\_function(neighbor\_solution)

if neighbor\_score > current\_score:

current\_solution = neighbor\_solution

current\_score = neighbor\_score

return current\_solution, current\_score

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

max\_iterations = 1000

problem\_size = 10

best\_solution, best\_score = hill\_climbing(max\_iterations, problem\_size)

print("Best Solution:", best\_solution)

print("Best Score:", best\_score)

AIM : **Implement minimax algorithm for a simple game like tic-tac-toe.**

board = {1: '', 2: '', 3: '', 4: '', 5: '', 6: '', 7: '', 8: '', 9: ''}

player = 'O'

computer = 'X'

def printBoard(board):

print(board[1] + "|" + board[2] + "|" + board[3])

print("-+-+-")

print(board[4] + "|" + board[5] + "|" + board[6])

print("-+-+-")

print(board[7] + "|" + board[8] + "|" + board[9] + "\n")

def spaceIsFree(position):

return board[position] == ''

def insertLetter(letter, position):

if spaceIsFree(position):

board[position] = letter

printBoard(board)

if checkDraw():

print("Draw!")

exit()

if checkWin():

if letter == 'X':

print("Bot wins!")

else:

print("Player wins!")

exit()

return

else:

print("Invalid position")

position = int(input("Please enter a new position: "))

insertLetter(letter, position)

return

def checkWin():

win\_conditions = [

[1, 2, 3], [4, 5, 6], [7, 8, 9], # rows

[1, 4, 7], [2, 5, 8], [3, 6, 9], # columns

[1, 5, 9], [3, 5, 7] # diagonals

]

for condition in win\_conditions:

if board[condition[0]] == board[condition[1]] == board[condition[2]] != '':

return True

return False

def checkWhichMarkWon(mark):

win\_conditions = [

[1, 2, 3], [4, 5, 6], [7, 8, 9], # rows

[1, 4, 7], [2, 5, 8], [3, 6, 9], # columns

[1, 5, 9], [3, 5, 7] # diagonals

]

for condition in win\_conditions:

if board[condition[0]] == board[condition[1]] == board[condition[2]] == mark:

return True

return False

def checkDraw():

for key in board.keys():

if board[key] == '':

return False

return True

def playerMove():

position = int(input("Enter a position for 'O': "))

insertLetter(player, position)

return

def compMove():

bestScore = -800

bestMove = 0

for key in board.keys():

if board[key] == '':

board[key] = computer

score = minimax(board, False)

board[key] = ''

if score > bestScore:

bestScore = score

bestMove = key

insertLetter(computer, bestMove)

return

def minimax(board, isMaximizing):

if checkWhichMarkWon(computer):

return 1

elif checkWhichMarkWon(player):

return -1

elif checkDraw():

return 0

if isMaximizing:

bestScore = -800

for key in board.keys():

if board[key] == '':

board[key] = computer

score = minimax(board, False)

board[key] = ''

bestScore = max(score, bestScore)

return bestScore

else:

bestScore = 800

for key in board.keys():

if board[key] == '':

board[key] = player

score = minimax(board, True)

board[key] = ''

bestScore = min(score, bestScore)

return bestScore

while not checkDraw():

compMove()

playerMove()

***Wumpus World Problem***

import random

class WumpusWorld:

def \_\_init\_\_(self, size):

self.size = size

self.wumpus\_location = self.generate\_random\_location()

self.gold\_location = self.generate\_random\_location()

self.pit\_locations = [self.generate\_random\_location() for \_ in range(size)]

self.agent\_location = (0, 0)

self.agent\_has\_gold = False

self.agent\_alive = True

def generate\_random\_location(self):

return (random.randint(0, self.size - 1), random.randint(0, self.size - 1))

def percept(self):

percept = {

'stench': self.is\_stench(),

'breeze': self.is\_breeze(),

'glitter': self.is\_glitter(),

'bump': self.is\_bump(),

'scream': self.is\_scream()

}

return percept

def is\_stench(self):

return self.is\_adjacent(self.wumpus\_location)

def is\_breeze(self):

return any(self.is\_adjacent(pit) for pit in self.pit\_locations)

def is\_glitter(self):

return self.agent\_location == self.gold\_location

def is\_bump(self):

x, y = self.agent\_location

return x < 0 or x >= self.size or y < 0 or y >= self.size

def is\_scream(self):

return self.agent\_has\_gold and self.agent\_location == self.wumpus\_location

def is\_adjacent(self, location):

x, y = location

ax, ay = self.agent\_location

return abs(x - ax) + abs(y - ay) == 1

def apply\_action(self, action):

if action == 'move':

self.move\_agent()

elif action == 'grab':

self.grab\_gold()

elif action == 'shoot':

self.shoot\_arrow()

def move\_agent(self):

if not self.agent\_alive:

return

x, y = self.agent\_location

self.agent\_location = (x + random.choice([-1, 1]), y + random.choice([-1, 1]))

if self.agent\_location == self.wumpus\_location:

self.agent\_alive = False

def grab\_gold(self):

if not self.agent\_alive:

return

if self.agent\_location == self.gold\_location:

self.agent\_has\_gold = True

def shoot\_arrow(self):

if not self.agent\_alive:

return

if self.agent\_has\_gold:

self.agent\_has\_gold = False

wx, wy = self.wumpus\_location

ax, ay = self.agent\_location

if ax == wx and abs(ay - wy) <= 1:

self.agent\_alive = False

elif ay == wy and abs(ax - wx) <= 1:

self.agent\_alive = False

# Example usage:

world = WumpusWorld(size=4)

while world.agent\_alive and not world.agent\_has\_gold:

print("Percept:", world.percept())

action = input("Enter action (move/grab/shoot): ")

world.apply\_action(action)

if world.agent\_has\_gold:

print("Agent has won!")

else:

print("Agent has been defeated.")

**Program on relationship in a given family tree:**

parent(john, mary).

parent(john, lisa).

parent(mary, anna).

parent(mary, david).

parent(lisa, peter).

parent(anna, susan).

parent(david, mike).

mother(X, Y):- parent(X, Y), female(X).

father(X, Y):- parent(X, Y), male(X).

sister(X, Y):- parent(Z, X), parent(Z, Y), female(X), X \= Y.

brother(X, Y):- parent(Z, X), parent(Z, Y), male(X), X \= Y.

male(john).

male(david).

male(peter).

male(mike).

female(mary).

female(lisa).

female(anna).

female(susan).

**Resolution theorem in prolog:**

**likes(john,jane).**

**likes(jane,john).**

**studies(Charlie,csc135).**

**studies(olivia,csc135).**