NETAJI SUBHAS UNIVERSITY OF TECHNOLOGY

ARTIFICIAL INTELLIGENCE

PROGRAME FILE

(CACSC11)

SUBMITTED BY: -

Name: *HARSH KUMAR*

Branch: CSAI

Roll no.:2021UCA1829

S NO.	EXPERIMENT
1	Experient: The Vacuum cleaner world example.
2	Design a program for the greedy best first search or A* search
3	Construct the simulated annealing algorithm over the travelling salesman problem.
4	Implement a basic binary genetic algorithm for a given problem.
5	Experiment: The Graph Colouring CSP or Cryptarithmetic Puzzle
6	Implement the Tic-Tac-Toe game using any adversarial searching algorithm

EXPERIMENT 1

AIM: Experiment the vacuum cleaner world example.

THEORY:

- Vacuum cleaner problem is a well-known search problem for an Goal based agent which works on Artificial Intelligence.
- Problem: In the classical vacuum cleaner problem, we have two rooms and one vacuum cleaner. There is dirt in both the rooms and it is to be cleaned. The vacuum cleaner is present in any one of these rooms. So, we have to reach a state in which both the rooms are clean and are dust free.

CODE:

```
# INSTRUCTIONS

# Enter LOCATION A/B in captial letters

# Enter Status O/1 accordingly where 0 means CLEAN and 1 means DIRTY


def vacuum_world():

    # initializing goal_state

    # 0 indicates Clean and 1 indicates Dirty

    goal_state = {'A': '0', 'B': '0'}

    cost = 0


    # user_input of location vacuum is placed

    location_input = input("Enter Location of Vacuum")

    # user_input if location is dirty or clean

    status_input = input("Enter status of " + location_input)

    status_input_complement = input("Enter status of other room")

    print("Initial Location Condition" + str(goal_state))


    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()
```

CODE:

```
PROBLEMS 2   OUTPUT   DEBUG CONSOLE   TERMINAL

harsh@DELLVOSTRO2H9 MINGW64 /f/AI
$ python -u "f:\AI\vaccume.py"
Enter Location of VacuumA
Enter status of A0
Enter status of other room1
Initial Location Condition{'A': '0', 'B': '0'}
Vacuum is placed in Location A
Location A is already clean
Location B is Dirty.
Moving RIGHT to the Location B.
COST for moving RIGHT 1
Cost for SUCK2
Location B has been Cleaned.
GOAL STATE:
{'A': '0', 'B': '0'}
Performance Measurement: 2
```

EXPERIMENT 2

AIM: Design a program for the greedy best first search or A* search.

THEORY:

- A* Search algorithm is one of the best and popular technique used in path-finding and graph traversals..
- What A* Search Algorithm does is that at each step it picks the node according to a value-'f' which is a parameter equal to the sum of two other parameters – 'g' and 'h'. At each step it picks the node/cell having the lowest 'f', and process that node/cell.
- We define 'g' and 'h' as simply as possible below
    - g = the movement cost to move from the starting point to a given square on the grid, following the path generated to get there.
    - h = the estimated movement cost to move from that given square on the grid to the final destination. This is often referred to as the heuristic, which is nothing but a kind of smart guess.

CODE:

```
def aStarAlgo(start_node, stop_node):

    open_set = set(start_node)
    closed_set = set()
    g = {}  # store distance from starting node
    parents = {}  # parents contains an adjacency map of all nodes
    # ditance of starting node from itself is zero
    g[start_node] = 0
    # start_node is root node i.e it has no parent nodes
    # so start_node is set to its own parent node
    parents[start_node] = start_node
    while len(open_set) > 0:
        n = None
    # node with lowest f() is found
    for v in open_set:
         if n == None or g[v] + heuristic(v) < g[n] + heuristic(n):
             n = v
        if n == stop_node or Graph_nodes[n] == None:
         pass
    else:
    for (m, weight) in get_neighbors(n):
```

```
    # nodes 'm' not in first and last set are added to first
    # n is set its parent
if m not in open_set and m not in closed_set:
open_set.add(m)
parents[m] = n
g[m] = g[n] + weight
# for each node m,compare its distance from start i.e g(m)
to the
# from start through n node
else:
if g[m] > g[n] + weight:
    # update g(m)
g[m] = g[n] + weight
# change parent of m to n
parents[m] = n
# if m in closed set,remove and add to open
if m in closed_set:
closed_set.remove(m)
open_set.add(m)
if n == None:
print('Path does not exist!')
return None
# if the current node is the stop_node
# then we begin reconstructin the path from it to the start_node
if n == stop_node:
path = []
while parents[n] != n:
path.append(n)
n = parents[n]
path.append(start_node)
path.reverse()
print('Path found: {}'.format(path))
return path
# remove n from the open_list, and add it to closed_list
# because all of his neighbors were inspected
open_set.remove(n)
closed_set.add(n)
print('Path does not exist!')
return None
```

```
    # define fuction to return neighbor and its distance
    # from the passed node
    def get_neighbors(v):

    if v in Graph_nodes:
    return Graph_nodes[v]
    else:
    return None
    # for simplicity we ll consider heuristic distances given
    # and this function returns heuristic distance for all nodes
    def heuristic(n):

    H_dist = {
        'A': 11,
        'B': 6,
        'C': 99,
        'D': 1,
        'E': 7,
        'G': 0,
    }
    return H_dist[n]
    # Describe your graph here
    Graph_nodes = {
        'A': [('B', 2), ('E', 3)],
        'B': [('C', 1), ('G', 9)],
        'C': None,
        'E': [('D', 6)],
        'D': [('G', 1)],
    }
    aStarAlgo('A', 'G')
```

OUTPUT:

```
PROBLEMS 2   OUTPUT   DEBUG CONSOLE   TERMINAL

harsh@DELLVOSTRO2H9 MINGW64 /f/AI
$ python -u "f:\AI\a star.py"
Path found: ['A', 'F', 'G', 'I', 'J']
```

EXPERIMENT 3

AIM: Construct the simulated annealing algorithm over the travelling salesman problem.

THEORY:

- The simulated annealing algorithm is a heuristic optimization algorithm that can be used to solve the traveling salesman problem (TSP).
- The TSP is a well-known combinatorial optimization problem where the goal is to find the shortest possible route that visits a given set of cities exactly once and returns to the starting city. The problem is NP-hard, which means that finding an exact solution for large instances of the problem is computationally infeasible.
- In this example, the distance matrix represents the distances between four cities, and the initial solution is a simple ordered list of the cities. The simulated annealing algorithm then generates new solutions by randomly swapping two cities, and determines whether to accept the new solution based on the acceptance probability function.
- The algorithm also cools down the temperature according to the cooling schedule parameters. The output of the algorithm is the best solution and distance found.

CODE:

```
import random
import math


# Define the distance matrix
distances = [
    [0, 2, 9, 10],
    [1, 0, 6, 4],
    [15, 7, 0, 8],
    [6, 3, 12, 0]
]


# Define the initial solution
initial_solution = [0, 1, 2, 3]


# Define the cooling schedule parameters
initial_temperature = 100
cooling_rate = 0.01
minimum_temperature = 0.1
```

```
# Define the energy function to calculate the total distance of a solution


def calculate_distance(solution):
    distance = 0
    for i in range(len(solution) - 1):
        distance += distances[solution[i]][solution[i+1]]
    distance += distances[solution[-1]][solution[0]]
    return distance

# Define the acceptance probability function


def acceptance_probability(old_energy, new_energy, temperature):
    if new_energy < old_energy:
        return 1.0
    else:
        return math.exp((old_energy - new_energy) / temperature)

# Implement the simulated annealing algorithm


def simulated_annealing(initial_solution, initial_temperature, cooling_rate,
minimum_temperature):
    current_solution = initial_solution
    current_energy = calculate_distance(current_solution)
    temperature = initial_temperature

    while temperature > minimum_temperature:
        # Generate a new solution by randomly swapping two cities
        new_solution = current_solution.copy()
        i, j = random.sample(range(len(new_solution)), 2)
        new_solution[i], new_solution[j] = new_solution[j], new_solution[i]
        new_energy = calculate_distance(new_solution)

        # Calculate the acceptance probability
        ap = acceptance_probability(current_energy, new_energy, temperature)
```

```
        # Determine whether to accept the new solution
        if random.uniform(0, 1) < ap:
            current_solution = new_solution
            current_energy = new_energy

        # Cool down the temperature
        temperature *= 1 - cooling_rate

    return current_solution, current_energy


# Run the simulated annealing algorithm
best_solution, best_distance = simulated_annealing(
    initial_solution, initial_temperature, cooling_rate, minimum_temperature)

# Print the best solution and distance found
print("Best solution:", best_solution)
print("Best distance:", best_distance)
```

OUTPUT:

EXPERIMENT 4

AIM: Implement a basic binary genetic algorithm for a given problem.

THEORY:

Example problem and solution using Genetic Algorithms :

Given a target string, the goal is to produce target string starting from a random string of the same length. In the following implementation, following analogies are made –

▪ Characters A-Z, a-z, 0-9, and other special symbols are considered as genes

▪ A string generated by these characters is considered as chromosome/solution/Individual

Fitness score is the number of characters which differ from characters in target string at a particular index. So individual having lower fitness value is given more preference.

.

CODE:

```
# Python3 program to create target string, starting from
# random string using Genetic Algorithm

import random

# Number of individuals in each generation
POPULATION_SIZE = 100

# Valid genes
GENES = '''abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOP
QRSTUVWXYZ 1234567890, .-;:_!"#%&/()=?@${[]}'''

# Target string to be generated
TARGET = "Harsh"



class Individual(object):
    '''
    Class representing individual in population
```

```
    '''

    def __init__(self, chromosome):
        self.chromosome = chromosome
        self.fitness = self.cal_fitness()

    @classmethod
    def mutated_genes(self):
        '''
        create random genes for mutation
        '''
        global GENES
        gene = random.choice(GENES)
        return gene

    @classmethod
    def create_gnome(self):
        '''
        create chromosome or string of genes
        '''
        global TARGET
        gnome_len = len(TARGET)
        return [self.mutated_genes() for _ in range(gnome_len)]

    def mate(self, par2):
        '''
        Perform mating and produce new offspring
        '''

        # chromosome for offspring
        child_chromosome = []
        for gp1, gp2 in zip(self.chromosome, par2.chromosome):

            # random probability
            prob = random.random()

            # if prob is less than 0.45, insert gene
            # from parent 1
```

```
            if prob < 0.45:
                child_chromosome.append(gp1)

            # if prob is between 0.45 and 0.90, insert
            # gene from parent 2
            elif prob < 0.90:
                child_chromosome.append(gp2)

            # otherwise insert random gene(mutate),
            # for maintaining diversity
            else:
                child_chromosome.append(self.mutated_genes())

        # create new Individual(offspring) using
        # generated chromosome for offspring
        return Individual(child_chromosome)

    def cal_fitness(self):
        '''
        Calculate fitness score, it is the number of
        characters in string which differ from target
        string.
        '''
        global TARGET
        fitness = 0
        for gs, gt in zip(self.chromosome, TARGET):
            if gs != gt:
                fitness += 1
        return fitness

# Driver code


def main():
    global POPULATION_SIZE

    # current generation
    generation = 1
```

```
found = False
population = []

# create initial population
for _ in range(POPULATION_SIZE):
    gnome = Individual.create_gnome()
    population.append(Individual(gnome))

while not found:

    # sort the population in increasing order of fitness score
    population = sorted(population, key=lambda x: x.fitness)

    # if the individual having lowest fitness score ie.
    # 0 then we know that we have reached to the target
    # and break the loop
    if population[0].fitness <= 0:
        found = True
        break

    # Otherwise generate new offsprings for new generation
    new_generation = []

    # Perform Elitism, that mean 10% of fittest population
    # goes to the next generation
    s = int((10*POPULATION_SIZE)/100)
    new_generation.extend(population[:s])

    # From 50% of fittest population, Individuals
    # will mate to produce offspring
    s = int((90*POPULATION_SIZE)/100)
    for _ in range(s):
        parent1 = random.choice(population[:50])
        parent2 = random.choice(population[:50])
        child = parent1.mate(parent2)
        new_generation.append(child)
```

```
        population = new_generation

        print("Generation: {}\tString: {}\tFitness: {}".
              format(generation,
                     "".join(population[0].chromosome),
                     population[0].fitness))

        generation += 1

    print("Generation: {}\tString: {}\tFitness: {}".
          format(generation,
                 "".join(population[0].chromosome),
                 population[0].fitness))


if __name__ == '__main__':
    main()
```

OUTPUT:

```
PROBLEMS 2   OUTPUT   DEBUG CONSOLE   TERMINAL

harsh@DELLVOSTRO2H9 MINGW64 /f/AI
$ python -u "f:\AI\generic.py"
Generation: 1   String: ea?Zh   Fitness: 3
Generation: 2   String: ea?Zh   Fitness: 3
Generation: 3   String: ea?Zh   Fitness: 3
Generation: 4   String: ea?Zh   Fitness: 3
Generation: 5   String: earZh   Fitness: 2
Generation: 6   String: Hars)   Fitness: 1
Generation: 7   String: Hars)   Fitness: 1
Generation: 8   String: Hars)   Fitness: 1
Generation: 9   String: Hars)   Fitness: 1
Generation: 10  String: Harsh   Fitness: 0
```

EXPERIMENT 5

AIM: Experiment: The Graph Colouring CSP or Cryptarithmetic Puzzle

THEORY:

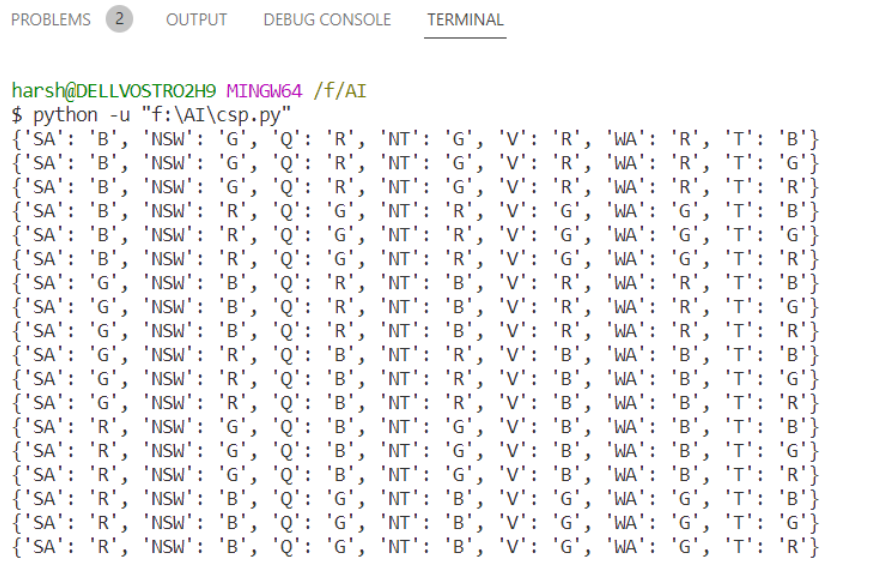
- The CSP (Constraint Satisfaction Problem) graph coloring problem involves assigning colors to the vertices of a graph such that no two adjacent vertices (connected by an edge) have the same color. The objective is to find a feasible solution that satisfies all the constraints.
- In the context of graph theory, a graph is a collection of vertices and edges, where each edge connects two vertices. A graph can be represented as G = (V, E), where V is the set of vertices and E is the set of edges.

CODE:

```
from constraint import *

# Define the CSP problem

problem = Problem()

# Define the variables and domains

variables = ["WA", "NT", "SA", "Q", "NSW", "V", "T"]

domains = {}

for var in variables:

    domains[var] = ["R", "G", "B"]


# Add the variables and domains to the problem

for var, domain in domains.items():

    problem.addVariable(var, domain)


# Define the constraints

def constraint_function(x, y):

    return x != y


problem.addConstraint(constraint_function, ["WA", "NT"])

problem.addConstraint(constraint_function, ["WA", "SA"])

problem.addConstraint(constraint_function, ["NT", "SA"])

problem.addConstraint(constraint_function, ["NT", "Q"])

problem.addConstraint(constraint_function, ["SA", "Q"])

problem.addConstraint(constraint_function, ["SA", "NSW"])

problem.addConstraint(constraint_function, ["SA", "V"])

problem.addConstraint(constraint_function, ["Q", "NSW"])

problem.addConstraint(constraint_function, ["NSW", "V"])
```

```
# Solve the problem
solutions = problem.getSolutions()

# Print the solutions
for solution in solutions:
    print(solution)
```

OUTPUT:



EXPERIMENT 6

AIM: Implement the Tic-Tac-Toe game using any adversarial searching algorithm.

THEORY:

- combine minimax and evaluation function to write a proper Tic-Tac-Toe AI (Artificial Intelligence) that plays a perfect game.
- We shall be introducing a new function called findBestMove(). This function evaluates all the available moves using minimax() and then returns the best move the maximizer can make.
- To check whether or not the current move is better than the best move we take the help of minimax() function which will consider all the possible ways the game can go and returns the best value for that move, assuming the opponent also plays optimally.

CODE:

```
#include <bits/stdc++.h>
# Python3 program to find the next optimal move for a player
player, opponent = 'x', 'o'

# This function returns true if there are moves
# remaining on the board. It returns false if
# there are no moves left to play.
def isMovesLeft(board) :

	for i in range(3) :
		for j in range(3) :
			if (board[i][j] == '_') :
				return True
	return False

# This is the evaluation function as discussed
def evaluate(b) :

	# Checking for Rows for X or O victory.
	for row in range(3) :
		if (b[row][0] == b[row][1] and b[row][1] == b[row][2]) :
			if (b[row][0] == player) :
				return 10
			elif (b[row][0] == opponent) :
				return -10
```

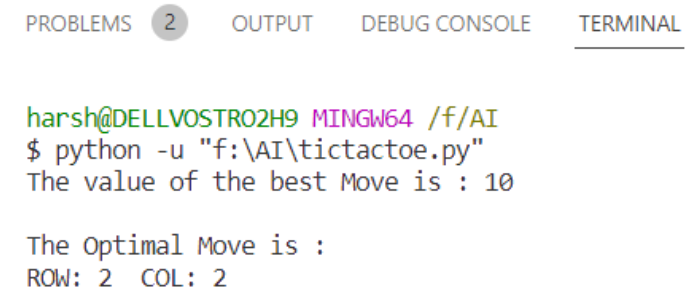
```
	# Checking for Columns for X or O victory.
	for col in range(3) :

		if (b[0][col] == b[1][col] and b[1][col] == b[2][col]) :

			if (b[0][col] == player) :
				return 10
			elif (b[0][col] == opponent) :
				return -10

	# Checking for Diagonals for X or O victory.
	if (b[0][0] == b[1][1] and b[1][1] == b[2][2]) :

		if (b[0][0] == player) :
			return 10
		elif (b[0][0] == opponent) :
			return -10

	if (b[0][2] == b[1][1] and b[1][1] == b[2][0]) :

		if (b[0][2] == player) :
			return 10
		elif (b[0][2] == opponent) :
			return -10

	# Else if none of them have won then return 0
	return 0

# This is the minimax function. It considers all
# the possible ways the game can go and returns
# the value of the board
def minimax(board, depth, isMax) :
	score = evaluate(board)

	# If Maximizer has won the game return his/her
	# evaluated score
	if (score == 10) :
		return score
```

```
# If Minimizer has won the game return his/her
# evaluated score
if (score == -10) :
	return score

# If there are no more moves and no winner then
# it is a tie
if (isMovesLeft(board) == False) :
	return 0

# If this maximizer's move
if (isMax) :
	best = -1000

	# Traverse all cells
	for i in range(3) :
		for j in range(3) :

			# Check if cell is empty
			if (board[i][j]=='_') :

				# Make the move
				board[i][j] = player

			# Call minimax recursively and choose
			# the maximum value
			best = max( best,minimax(board,depth + 1,not isMax) )

				# Undo the move
				board[i][j] = '_'
	return best

# If this minimizer's move
else :
	best = 1000

	# Traverse all cells
	for i in range(3) :
		for j in range(3) :
```

```
                        # Check if cell is empty
                        if (board[i][j] == '_') :

                                # Make the move
                                board[i][j] = opponent

                                # Call minimax recursively and choose
                                # the minimum value
                                best = min(best, minimax(board, depth + 1, not
isMax))

                                # Undo the move
                                board[i][j] = '_'
            return best

# This will return the best possible move for the player
def findBestMove(board) :
      bestVal = -1000
      bestMove = (-1, -1)

      # Traverse all cells, evaluate minimax function for
      # all empty cells. And return the cell with optimal
      # value.
      for i in range(3) :
            for j in range(3) :

                  # Check if cell is empty
                  if (board[i][j] == '_') :

                        # Make the move
                        board[i][j] = player

                        # compute evaluation function for this
                        # move.
                        moveVal = minimax(board, 0, False)

                        # Undo the move
                        board[i][j] = '_'
```

```
                        # If the value of the current move is
                        # more than the best value, then update
                        # best/
                        if (moveVal > bestVal) :
                                bestMove = (i, j)
                                bestVal = moveVal

        print("The value of the best Move is :", bestVal)
        print()
        return bestMove
# Driver code
board = [
        [ 'x', 'o', 'x' ],
        [ 'o', 'o', 'x' ],
        [ '_', '_', '_' ]
]

bestMove = findBestMove(board)

print("The Optimal Move is :")
print("ROW:", bestMove[0], " COL:", bestMove[1])
```

OUTPUT:



****