NETAJI SUBHAS UNIVERSITY OF TECHNOLOGY

Artificial Intelligence

(CACSE11))

LAB FILE

Name: SNEHA GUPTA

Roll No.: 2021UCA1859

Branch: COMPUTER SCIENCE WITH ARTIFICIAL INTELLIGENCE

Section: 1

INDEX

1. Experiment 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 Coloring CSP or Cryptarithmetic Puzzle.

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

Q1. Experiment the vacuum cleaner world example.

Problem-

Our vacuum cleaner is our agent(goal based) and the goal of this agent, is to clean up the whole area.

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.

So, there are eight possible states possible in our problem :

Here, states 1 and 2 are our initial states and state 7 and state 8 are our final states (goal states)

The vacuum cleaner can perform the following functions: move left, move right, move forward, move backward and to suck dust. But as there are only two rooms in our problem, the vacuum cleaner performs only the following functions here: move left, move right and suck

Program-

```
Welcome    prac11.py
prac11.py > ...

# Experiment 1 SNEHA GUPTA 2021UCA1859
import random

class Environment(object):

    def __init__(self):
        self.locationCondition = {'A': '0', 'B': '0'}
        self.locationCondition[ 'A'] =random.randint(0, 1)
        self.locationCondition [ 'B'] = random.randint(0, 1)

class SimpleReflexVacuumAgent (Environment):

    def __init__(self, Environment):

        print("sneha gupta 2021UCA1859 Output-1")
        print (Environment.locationCondition)
        Score = 0
        vacuumLocation = random.randint(0, 1)

        if vacuumLocation == 0:
            print ("Vacuum is randomly placed at Location A.")
            print ("Location A is Dirty.") if Environment.locationCondition [ 'A'] == 1 else print("Location A is clean.")

            if Environment.locationCondition [ 'A'] == 1:
                Environment.locationCondition['A'] = 0
                Score += 1
                print ("Location A has been Cleaned.")
            print ("Moving to Location B...")
            print ("Location B is Dirty.") if Environment.locationCondition[ 'B'] == 1 else print("Location B is Clean.")

            if Environment.locationCondition [ 'B'] == 1:
                Environment.locationCondition[ 'B'] = 0
                Score += 1
                print ("Location B has been cleaned.")
            print("Environment is Clean.")
```

```
        elif vacuumLocation == 1:

            print ("Vacuum randomly placed at Location B.")
            print ("Location B is Dirty.") if Environment.locationCondition[ 'B'] == 1 else print("Location B is Clean.")

            if Environment.locationCondition [ 'B'] == 1:
                Environment.locationCondition['B'] = 0
                Score += 1
                print ("Location B has been Cleaned.")
            print ("Moving to Location A...")
            print ("Location A is Dirty.") if Environment.locationCondition['A'] == 1 else print("Location A is Clean.")

            if Environment.locationCondition['A'] == 1:
                Environment.locationCondition['A'] = 0
                Score += 1
                print ("Location A has been cleaned.")
            print("Environment is Clean.")

        print (Environment.locationCondition)
        print ("Performance Measurement: " + str(Score))

theEnvironment =Environment()
theVacuum = SimpleReflexVacuumAgent(theEnvironment)
```

Output –

```
prac11.py"
sneha gupta 2021UCA1859 Output-1
{'A': 1, 'B': 1}
Vacuum is randomly placed at Location A.
Location A is Dirty.
Location A has been Cleaned.
Moving to Location B...
Location B is Dirty.
Location B has been cleaned.
Environment is Clean.
{'A': 0, 'B': 0}
Performance Measurement: 2
PS C:\sem 4\AI\AI PRAC>
```

Q2) Design a program for the greedy best first search or A* search

BFS-) It is of the most common search strategies. It generally starts from the root node and examines the neighbor nodes and then moves to the next level. It uses First-in First-out (FIFO) strategy as it gives the shortest path to achieving the solution.

Advantages -

BFS will never be trapped in any unwanted nodes.

If the graph has more than one solution, then BFS will return the optimal solution which provides the shortest path.

Disadvantages□BFS stores all the nodes in the current level and then go to the next level. It requires a lot of memory to store the nodes.

BFS takes more time to reach the goal state which is far away.

Program-

PRAC2.py > astar

```python
# Experiment 2 (sneha guptal 2021UCA18459)

import heapq

def astar(start, end, graph, heuristic):
    queue=[(heuristic (start, end), start)]
    costs = {start: 0}
    parents = {start: None}
    while queue:
        _, current = heapq.heappop (queue)
        if current == end:
            path = []
            while current:
                path.append(current)
                current =parents [current]
            return list (reversed (path))

    # Check each neighbor of the current node
    for neighbor, cost in graph[current].items():
        # Calculate the total cost of getting to the neighbor through the current node
        new_cost =costs[current] + cost
    # If we haven't seen this neighbor before, or if the new cost is lower than the existing cost
        if neighbor not in costs or new_cost < costs [neighbor]:
            # Update the cost and parent dictionaries
            costs[neighbor] = new_cost
            parents[neighbor]= current
            # Add the neighbor to the queue with its estimated cost
            priority= new_cost + heuristic(neighbor, end)
            heapq.heappush (queue, (priority, neighbor))

    # If we've explored the entire graph and haven't found the end node, return None
    return None

# Example usage with user input
graph = {
    1: {2: 1, 3: 2, 4: 5},
    2: {1: 1, 3: 3, 4: 3},
```

PRAC2.py > astar

```python
graph = {
    1: {2: 1, 3: 2, 4: 5},
    2: {1: 1, 3: 3, 4: 3},
    3: {1: 2, 2: 3, 4: 1},
    4: {1: 5, 2: 3, 3: 1}
}

start = int(input("Enter start node: "))
end= int(input("Enter end node: "))

path =astar (start, end, graph, lambda x, y: 0) # Use a trivial heuristic that always returns 0

if path:
    print("Path found: ", path)
else:
    print("No path found.")
```

Output:

```
PRAC2.py"
Enter start node: 1
Enter end node: 4
Path found: [1, 3, 4]
PS C:\sem 4\AI\AI PRAC>
```

Q3) Construct the simulated annealing algorithm over the travelling salesman problem.

The traveling salesman problem (TSP) is a classic optimization problem in computer science and mathematics. The problem can be stated as follows: Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city

The problem is often stated in the context of a traveling salesman who needs to visit a set of cities and return to the starting point while minimizing the total distance traveled. However, the problem has applications beyond just logistics and transportation, such as in circuit board design and DNA sequencing

The TSP is an NP-hard problem, which means that as the size of the problem grows, the time required to find the optimal solution grows exponentially. Therefore, exact algorithms for solving the TSP are only practical for small instances of the problem. Instead, heuristic and metaheuristic algorithms such as simulated annealing, genetic algorithms, and ant colony optimization are often used to find approximate solutions that are close to the optimal solution in a reasonable amount of time

Program

```python
# Experiment 3 (sneha guota 2021UCA1859)

import random
import math

def distance(city1, city2):
    return math.sqrt((city1[0] - city2[0]) ** 2 + (city1[1] -city2[1]) ** 2)

def tour_length(coords, tour):
    return sum (distance (coords[tour[i]], coords[tour[i+1]]) for i in range(len(tour)-1)) + distance (coords[tour[-1]], coords[tour[0]])

def simulated_annealing_tsp(coords, temp=10000, cool=0.99, stopping_temp=1e-8, stopping_iter=1000):

    tour = random.sample (range(len(coords)), len(coords))
    curr_length = tour_length (coords, tour)
    i = 0
    while temp >= stopping_temp and i < stopping_iter:
        c1,c2=sorted(random.sample(range(len(coords)),2))
        new_tour=tour[:c1]+tour[c1:c2][::-1]+tour[c2:]
        new_length=tour_length(coords,new_tour)
        delta=new_length-curr_length
        if delta<0 or math.exp(-delta/temp)>random.random():
            tour=new_tour
            curr_length=new_length
        temp*=cool
        i+=1
    return tour

random.seed(42)
cities=[(random.uniform (0, 1), random. uniform (0, 1)) for i in range(50)]
tour =simulated_annealing_tsp (cities)
print("Optimal tour:", tour)
print("Tour length:", tour_length(cities, tour))
```

Output-

```
prac3.py"
Optimal tour: [24, 31, 30, 48, 9, 28, 15, 12, 49, 19, 2
9, 47, 44, 0, 27, 32, 2, 46, 3, 1, 22, 36, 8, 35, 26, 2
5, 37, 42, 38, 4, 43, 18, 21, 45, 10, 16, 7, 14, 5, 33,
 11, 34, 39, 13, 23, 6, 20, 41, 17, 40]
Tour length: 21.061162982231966
PS C:\sem 4\AI\AI PRAC>
```

Q4) Implement a basic binary genetic algorithm for a given problem.

A genetic algorithm is a heuristic search and optimization technique inspired by the process of natural selection. It is often used to find approximate solutions to optimization and search problems. A binary genetic algorithm operates on a population of binary strings, also called chromosomes, where each bit in the string represents a decision variable

1)The problem to be solved is defined by the objective_function, which takes a binary string (or chromosome) as input and returns a score representing the quality of the solution.

2)The fitness_function is defined to calculate the fitness score of a chromosome based on its objective score

3)The genetic algorithm initializes a population of population_size chromosomes with random bits.

4)The algorithm then evaluates the fitness of each chromosome in the population by applying the fitness_function

5)The algorithm then applies crossover to the selected parent chromosomes based on the crossover_rate parameter. In this implementation, single-point crossover is used, where a random point in the chromosome is selected and the tail of one parent is swapped with the tail of the other parent to create two offspring.

6)The algorithm then applies mutation to the offspring chromosomes based on the mutation_rate parameter. In this implementation, a single bit in the chromosome is flipped with a small probability.

Program-

```
prac4.py > genetic_algorithm
# Experiment 4 (sneha gupta 2021UCA1859)

import random

def objective_function (chromosome):
    return sum(chromosome)

def fitness_function (chromosome):
    return objective_function(chromosome)

def genetic_algorithm (population_size, chromosome_length, crossover_rate, mutation_rate, max_iterations):
    population = [[random.randint(0, 1) for _ in range(chromosome_length)] for _ in range(population_size)]
    for iteration in range(max_iterations):
        fitness_values = [fitness_function(chromosome) for chromosome in population]
        parents = []
        for _ in range(population_size):
            candidatel=random.randint(0, population_size -1)
            candidate2 =random.randint(0, population_size - 1)
            parent=population[candidatel] if fitness_values[candidatel] > fitness_values[candidate2] else population[candidate2]
            parents.append(parent)
        offspring = []
        for i in range(0, population_size, 2):
            parent1 = parents[i]
            parent2 =parents[i+1]
            if random.random() < crossover_rate:
                crossover_point=random.randit(1,chromosome_length-1)
                child1 = parent1[:crossover_point] + parent2 [crossover_point:]
                child2 = parent2 [:crossover_point] + parent1 [crossover_point:]

            else:
                child1= parent1
                child2 = parent2
            offspring.append(child1)
            offspring.append(child2)
        for i in range(population_size):
            chromosome = offspring[i]
            for j in range (chromosome_length):
```

```python
        for i in range(population_size):
            chromosome = offspring[i]
            for j in range (chromosome_length):
                if random.random() < mutation_rate:
                    chromosome[j]=1-chromosome[j]
        offspring_fitness_values=[fitness_function(chromosome) for chromosome in offspring]
        population=[]
        for i in range(population_size):
            if fitness_values[i]>=offspring_fitness_values[i]:
                population.append(parents[i])
            else:
                population.append(offspring[i])
    best_chromosome=max(population,key=fitness_function)
    return best_chromosome,fitness_function(best_chromosome)
population_size=100
chromosome_length=20
crossover_rate=0.8
mutation_rate=0.01
max_iterations=1000
best_chromosome,best_fitness=genetic_algorithm(population_size,chromosome_length,crossover_rate,mutation_rate,max_iterations)
print("best chromosome:",best_chromosome)
print("best fitness",best_fitness)
```

Output-

```
Best chromosome: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
Best fitness: 20
```

Q5)Experiment the Graph Coloring CSP or Cryptarithmetic Puzzle

A function is_valid that checks whether a given vertex can be colored with a particular color without violating the constraint that no two adjacent vertices have the same color. The next function defined is backtrack, which is the main function that performs the backtracking search to find a valid coloring of the graph. The function takes in the current vertex to be colored, the current coloring of the vertices, and the number of colors to be used

The function first checks if all vertices have been colored. If so, it means we have found a valid coloring and the function returns True

If not, the function loops through all possible colors for the current vertex and checks if the color is valid using the is_valid function. If the color is valid, it sets the color for the current vertex and recursively calls backtrack for the next vertex

If the recursive call returns True, it means a valid coloring was found and the function returns True. If not, the color is removed for the current vertex and the function continues with the next color

Finally, the main program reads in the input graph and number of colors from the user, and calls the backtrack function to find a valid coloring of the graph. If a valid coloring is found, it is printed to the console. If not, a message indicating that no valid coloring was found is printed

Program-

prac5.py > ...

```python
# code by sneha gupta 2021UCA1859

class Graph:
    def __init__(self, vertices, edges):
        self.vertices = vertices
        self.edges =edges

    def is_valid_coloring (self, variable_assignments):
        for vertex, color in variable_assignments.items():
            if color is None:
                return False

            for neighbor in self.edges [vertex]:
                if variable_assignments [neighbor] == color:
                    return False

        return True

def backtrack (variable_assignments, domains, graph):
    if all (variable_assignments.values()):
        if graph.is_valid_coloring (variable_assignments):
            return variable_assignments
        else:
            return None

    unassigned_variables = [variable for variable, value in variable_assignments.items() if value is None]
    variable = unassigned_variables[0]

    for value in domains [variable]:
        variable_assignments [variable] = value
        domains_copy =domains.copy()
        for unassigned_variable in unassigned_variables:
            if unassigned_variable != variable:
                domains_copy [unassigned_variable] = {value}
        result = backtrack (variable_assignments, domains_copy, graph)
        if result is not None:
            return result
        variable_assignments[variable] = None
    return None

if __name__=='__main__':
    vertices = [1, 2, 3, 4]
    edges = {
        1: [2, 3],
        2: [1, 3, 4],
        3: [1, 2, 4],
        4: [2, 3]
    }
    graph =Graph (vertices, edges)

    variable_assignments = {vertex: None for vertex in vertices}
    domains={vertex: {1, 2, 3} for vertex in vertices}
    result = backtrack (variable_assignments, domains, graph)
    print(result)
```

Output-

```
C\prac5.py"
None
PS C:\sem 4\AI\AI PRAC>
```

Q6) Implement the Tic-Tac-Toe game using any adversarial searching algorithm.

This is an implementation of the Minimax algorithm

The code uses a few pre-defined constants:

1) HUMAN = -1 - this represents the human player

2) COMP = +1 - this represents the AI/Computer player

3) board = [[0, 0, 0], [0, 0, 0], [0, 0, 0],] - this is the initial game state. A two-dimensional array of size 3x3, representing the game board.

evaluate(state): returns the score of the game state. +1 if the computer wins, -1 if the human wins, and 0 for a tie

wins(state, player): tests if a specific player wins. Returns True if the player wins

game_over(state): tests if the game is over (either player wins or tie)

empty_cells(state): returns a list of empty cells (positions) on the board.

valid_move(x, y): checks if a move (x, y) is valid, i.e., if the cell is empty

set_move(x, y, player): sets the move on the board if the coordinates are valid

clean(): clears the console screen

<u>Program-</u>

```python
prac6.py > empty_cells
#code by sneha gupta 2021UCA1859
from math import inf as infinity
from random import choice
import platform
import time
from os import system


HUMAN = -1
COMP = +1
board = [
    [0, 0, 0],
    [0, 0, 0],
    [0, 0, 0],
]
def evaluate(state):
    if wins (state, COMP):
        score = +1
    elif wins (state, HUMAN):
        score=-1
    else:
        score = 0
    return score

def wins(state, player):
    win_state=[
        [state[0][0], state[0][1], state[0][2]], [state[1][0], state[1][1], state[1][2]], [state[2][0], state[2][1], state[2][2]],
        [state[0][0], state[1][0], state[2][0]], [state[0][1], state[1][1], state[2][1]], [state[0][2], state[1][2], state[2][2]], [state[0][0], state[1][1], state[2]
        [2]], [state[2][0], state[1][1], state[0][2]],
    ]
    if [player, player, player] in win_state:
        return True

def game_over (state):
    return wins (state, HUMAN) or wins (state, COMP)
```

prac6.py > empty_cells

```python
def empty_cells (state):
    cells = []
    for x, row in enumerate(state):
        for y, cell in enumerate (row):
            if cell== 0:
                cells.append([x, y])
    return cells

def valid_move(x, y):
    if [x, y] in empty_cells (board):
        return True
    else:
        return False

def set_move(x, y, player):
    if valid_move(x, y):
        board[x][y]= player
        return True
    else:
        return False

def minimax (state, depth, player):
    if player == COMP:
        best [-1, -1, -infinity]
    else:
        best =[-1, -1, +infinity]
    if depth == 0 or game_over(state):
        score = evaluate(state)
        return [-1, -1, score]
    for cell in empty_cells (state):
        x, y = cell[0], cell[1]
        state[x][y]= player
        score= minimax(state, depth-1,-player)
        state[x][y]= 0
        score[0], score[1] = x,y

        if player == COMP:
            if score[2]> best [2]:
                best=score # max value
            if score[2] < best[2]:
                best= score # min value
    return best

def clean():
    os_name = platform.system().lower()
    if 'windows' in os_name:
        system('cls')
    else:
        system('clear')


def render(state, c_choice, h_choice):
    chars = {
        -1: h_choice,
        +1: c_choice,
        0: ' '
    }
    str_line='_______________'

    print('\n' + str_line)
    for row in state:
        for cell in row:
            symbol=chars[cell]
            print(f' |{symbol}|',end=' ')
        print('\n'+str_line)

def ai_turn(c_choice, h_choice):
    depth = len(empty_cells (board))
    if depth==0 or game_over (board):
        return

    clean()
    print (f'Computer turn [{c_choice}]')
    render(board, c_choice, h_choice)

    if depth == 9:
        x = choice([0, 1, 2])
        y = choice([0, 1, 2])
    else:
        move=minimax (board, depth, COMP)
        x, y = move[0], move[1]

    set_move(x, y, COMP)
    time.sleep(1)


def human_turn (c_choice, h_choice):
    depth = len(empty_cells (board))
    if depth == 0 or game_over (board):
        return

# Dictionary of valid moves
    move = -1
    moves = {
        1: [0, 0], 2: [0, 1], 3: [0, 2],
        4: [1, 0], 5: [1, 1], 6: [1, 2],
        7: [2, 0], 8: [2, 1], 9: [2, 2],
    }

    clean()
    print(f'Human turn [{h_choice}]')
    render(board, c_choice, h_choice)

    while move < 1 or move> 9:
        try:
            move = int(input('Use numpad (1..9): '))
            coord= moves [move]
            can_move = set_move (coord[0], coord[1], HUMAN)

            if not can_move:
                print('Bad move')
                move = -1
        except (EOFError, KeyboardInterrupt):
            print('bye')
            exit()
        except (KeyError, ValueError):
            print('Bad choice')


def main():

    clean()
    h_choice = ' '
    c_choice =' '
    first =' '

    # Human chooses X or 0 to play
    while h_choice != 'O' and h_choice != 'X':
        try:
            print('')
            h_choice = input ('Choose X or O\nChosen: ').upper()
        except (EOFError, KeyboardInterrupt):
            print('Bye')
            exit()
        except (KeyError, ValueError):
            print('Bad choice')

    # Setting computer's choice
    if h_choice == 'X':
        c_choice = '0'
    else:
        c_choice= 'X'

    # Human may starts first
    clean()
```

prac6.py > empty_cells

```python
    while first != 'Y' and first != 'N':
        try:
            first=input('First to start? [y/n]: ').upper()
        except (EOFError, KeyboardInterrupt):
            print('Bye')
            exit()
        except (KeyError, ValueError):
            print('Bad choice')

    #Main loop of this game
    while len(empty_cells(board)) > 0 and not game_over(board):
        if first == 'N':
            ai_turn(c_choice, h_choice)
            first = ''

        human_turn(c_choice, h_choice)
        ai_turn(c_choice, h_choice)

    # Game over message
    if wins (board, HUMAN):
        clean()
        print (f'Human turn [{h_choice}]')
        render (board, c_choice, h_choice)
        print('YOU WIN!')
    elif wins (board, COMP):
        clean()
        print (f'Computer turn [{c_choice}]')
        render (board, c_choice, h_choice)
        print('YOU LOSE!')
    else:
        clean()
        render (board, c_choice, h_choice)
        print('DRAW!')

    exit()

if __name__ == '__main__':
```

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

Output:

```
Choose X or O
Chosen: X
First to start?[y/n]: y
Human turn [X]

---------------
|   ||   ||   |
---------------
|   ||   ||   |
---------------
|   ||   ||   |
---------------
Use numpad (1..9): 1
Computer turn [O]

---------------
| X ||   ||   |
---------------
|   ||   ||   |
---------------
|   ||   ||   |
---------------
Human turn [X]

---------------
| X ||   ||   |
---------------
|   || O ||   |
---------------
|   ||   ||   |
---------------
Use numpad (1..9): 3
Computer turn [O]

---------------
| X ||   || X |
---------------
|   || O ||   |
```

```
Use numpad (1..9): 3
Computer turn [O]

--------------
| X ||   || X |
--------------
|   || O ||   |
--------------
|   ||   ||   |
--------------
Human turn [X]

--------------
| X || O || X |
--------------
|   || O ||   |
--------------
|   ||   ||   |
--------------
Use numpad (1..9): 8
Computer turn [O]

--------------
| X || O || X |
--------------
|   || O ||   |
--------------
|   || X ||   |
--------------
Human turn [X]

--------------
| X || O || X |
--------------
| O || O ||   |
--------------
|   || X ||   |
--------------
Use numpad (1..9): 6
Computer turn [O]
```

```
Human turn [X]

--------------
| X || O || X |
--------------
| O || O ||   |
--------------
|   || X ||   |
--------------
Use numpad (1..9): 6
Computer turn [O]

--------------
| X || O || X |
--------------
| O || O || X |
--------------
|   || X ||   |
--------------
Human turn [X]

--------------
| X || O || X |
--------------
| O || O || X |
--------------
|   || X || O |
--------------
Use numpad (1..9): 7

--------------
| X || O || X |
--------------
| O || O || X |
--------------
| X || X || O |
--------------
DRAW!
```