

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 Cryptarithmic 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 > ...
1
2 # Experiment 1 SNEHA GUPTA 2021UCA1859
3 import random
4
5 class Environment(object):
6
7     def __init__(self):
8         self.locationCondition = {'A': '0', 'B': '0'}
9         self.locationCondition[ 'A'] =random.randint(0, 1)
10        self.locationCondition [ 'B'] = random.randint(0, 1)
11
12 class SimpleReflexVacuumAgent (Environment):
13
14     def __init__(self, Environment):
15
16         print("sneha gupta 2021UCA1859 Output-1")
17         print (Environment.locationCondition)
18         Score = 0
19         vacuumLocation = random.randint(0, 1)
20
21         if vacuumLocation == 0:
22             print ("Vacuum is randomly placed at Location A.")
23             print ("Location A is Dirty.") if Environment.locationCondition [ 'A'] == 1 else print("Location A is clean.")
24
25             if Environment.locationCondition [ 'A'] == 1:
26                 Environment.locationCondition['A'] = 0
27                 Score += 1
28                 print ("Location A has been Cleaned.")
29             print ("Moving to Location B...")
30             print ("Location B is Dirty.") if Environment.locationCondition[ 'B'] == 1 else print("Location B is Clean.")
31
32             if Environment.locationCondition [ 'B'] == 1:
33                 Environment.locationCondition[ 'B'] = 0
34                 Score += 1
35                 print ("Location B has been cleaned.")
36             print("Environment is Clean.")
37
```

```

38         elif vacuumLocation == 1:
39
40             print ("Vacuum randomly placed at Location B.")
41             print ("Location B is Dirty.") if Environment.locationCondition[ 'B'] == 1 else print("Location B is Clean.")
42
43             if Environment.locationCondition [ 'B'] == 1:
44                 Environment.locationCondition['B'] = 0
45                 Score += 1
46                 print ("Location B has been Cleaned.")
47             print ("Moving to Location A...")
48             print ("Location A is Dirty.") if Environment.locationCondition['A'] == 1 else print("Location A is Clean.")
49
50             if Environment.locationCondition['A'] == 1:
51                 Environment.locationCondition['A'] = 0
52                 Score += 1
53                 print ("Location A has been cleaned.")
54             print("Environment is Clean.")
55
56         print (Environment.locationCondition)
57         print ("Performance Measurement: " + str(Score))
58
59     theEnvironment =Environment()
60     theVacuum = SimpleReflexVacuumAgent(theEnvironment)
61

```

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
1  # Experiment 2 (sneha guptal 2021UCA18459)
2
3  import heapq
4
5  def astar(start, end, graph, heuristic):
6      queue=[(heuristic (start, end), start)]
7      costs = {start: 0}
8      parents = {start: None}
9      while queue:
10         _, current = heapq.heappop (queue)
11         if current == end:
12             path = []
13             while current:
14                 path.append(current)
15                 current =parents [current]
16             return list (reversed (path))
17
18         # Check each neighbor of the current node
19         for neighbor, cost in graph[current].items():
20             # Calculate the total cost of getting to the neighbor through the current node
21             new_cost =costs[current] + cost
22             # If we haven't seen this neighbor before, or if the new cost is lower than the existing cost
23             if neighbor not in costs or new_cost < costs [neighbor]:
24                 # Update the cost and parent dictionaries
25                 costs[neighbor] = new_cost
26                 parents[neighbor]= current
27                 # Add the neighbor to the queue with its estimated cost
28                 priority= new_cost + heuristic(neighbor, end)
29                 heapq.heappush ([queue, (priority, neighbor)])
30
31         # If we've explored the entire graph and haven't found the end node, return None
32         return None
33
34 # Example usage with user input
35 graph = {
36     1: {2: 1, 3: 2, 4: 5},
37     2: {1: 1, 3: 3, 4: 3},
```

```
PRAC2.py > astar
35 graph = {
36     1: {2: 1, 3: 2, 4: 5},
37     2: {1: 1, 3: 3, 4: 3},
38     3: {1: 2, 2: 3, 4: 1},
39     4: {1: 5, 2: 3, 3: 1}
40 }
41
42 start = int(input("Enter start node: "))
43 end= int(input("Enter end node: "))
44
45 path =astar (start, end, graph, lambda x, y: 0) # Use a trivial heuristic that always returns 0
46
47 if path:
48     print("Path found: ", path)
49 else:
50     print("No path found.")
51
```

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

```

1 # Experiment 3 (sneha gupta 2021UCA1859)
2
3 import random
4 import math
5
6 def distance(city1, city2):
7     return math.sqrt((city1[0] - city2[0]) ** 2 + (city1[1] - city2[1]) ** 2)
8
9 def tour_length(coords, tour):
10    return sum (distance (coords[tour[i]], coords[tour[i+1]]) for i in range(len(tour)-1)) + distance (coords[tour[-1]], coords[tour[0]])
11
12 def simulated_annealing_tsp(coords, temp=10000, cool=0.99, stopping_temp=1e-8, stopping_iter=1000):
13
14    tour = random.sample (range(len(coords)), len(coords))
15    curr_length = tour_length (coords, tour)
16    i = 0
17    while temp >= stopping_temp and i < stopping_iter:
18        c1,c2=sorted(random.sample(range(len(coords)),2))
19        new_tour=tour[:c1]+tour[c1:c2][::-1]+tour[c2:]
20        new_length=tour_length(coords,new_tour)
21        delta=new_length-curr_length
22        if delta<0 or math.exp(-delta/temp)>random.random():
23            tour=new_tour
24            curr_length=new_length
25        temp*=cool
26        i+=1
27    return tour
28
29 random.seed(42)
30 cities=[(random.uniform (0, 1), random. uniform (0, 1)) for i in range(50)]
31 tour =simulated_annealing_tsp (cities)
32 print("Optimal tour:", tour)
33 print("Tour length:", tour_length(cities, tour))
34

```

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
1  # Experiment 4 (sneha gupta 2021UCA1859)
2
3  import random
4
5  def objective_function (chromosome):
6      return sum(chromosome)
7
8  def fitness_function (chromosome):
9      return objective_function(chromosome)
10
11 def genetic_algorithm (population_size, chromosome_length, crossover_rate, mutation_rate, max_iterations):
12     population = [[random.randint(0, 1) for _ in range(chromosome_length)] for _ in range(population_size)]
13     for iteration in range(max_iterations):
14         fitness_values = [fitness_function(chromosome) for chromosome in population]
15         parents = []
16         for _ in range(population_size):
17             candidate1=random.randint(0, population_size -1)
18             candidate2 =random.randint(0, population_size - 1)
19             parent=population[candidate1] if fitness_values[candidate1] > fitness_values[candidate2] else population[candidate2]
20             parents.append(parent)
21         offspring = []
22         for i in range(0, population_size, 2):
23             parent1 = parents[i]
24             parent2 =parents[i+1]
25             if random.random() < crossover_rate:
26                 crossover_point=random.randint(1,chromosome_length-1)
27                 child1 = parent1[:crossover_point] + parent2 [crossover_point:]
28                 child2 = parent2 [:crossover_point] + parent1 [crossover_point:]
29
30             else:
31                 child1= parent1
32                 child2 = parent2
33             offspring.append(child1)
34             offspring.append(child2)
35         for i in range(population_size):
36             chromosome = offspring[i]
37             for j in range (chromosome_length):
```



```

35         for i in range(population_size):
36             chromosome = offspring[i]
37             for j in range (chromosome_length):
38                 if random.random() < mutation_rate:
39                     chromosome[j]=1-chromosome[j]
40         offspring_fitness_values=[fitness_function(chromosome) for chromosome in offspring]
41         population=[]
42         for i in range(population_size):
43             if fitness_values[i]>=offspring_fitness_values[i]:
44                 population.append(parents[i])
45             else:
46                 population.append(offspring[i])
47         best_chromosome=max(population,key=fitness_function)
48         return best_chromosome,fitness_function(best_chromosome)
49     population_size=100
50     chromosome_length=20
51     crossover_rate=0.8
52     mutation_rate=0.01
53     max_iterations=1000
54     best_chromosome,best_fitness=genetic_algorithm(population_size,chromosome_length,crossover_rate,mutation_rate,max_iterations)
55     print("best chromosome:",best_chromosome)
56     print("best fitness",best_fitness)
57

```

Output-

```

Best chromosome: [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 Cryptarithmic 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-

```
pracs.py > ...
1  # code by sneha gupta 2021UCA1859
2
3  class Graph:
4      def __init__(self, vertices, edges):
5          self.vertices = vertices
6          self.edges = edges
7
8      def is_valid_coloring (self, variable_assignments):
9          for vertex, color in variable_assignments.items():
10             if color is None:
11                 return False
12
13             for neighbor in self.edges [vertex]:
14                 if variable_assignments [neighbor] == color:
15                     return False
16
17             return True
18
19  def backtrack (variable_assignments, domains, graph):
20      if all (variable_assignments.values()):
21          if graph.is_valid_coloring (variable_assignments):
22              return variable_assignments
23          else:
24              return None
25
26      unassigned_variables = [variable for variable, value in variable_assignments.items() if value is None]
27      variable = unassigned_variables[0]
28
29      for value in domains [variable]:
30          variable_assignments [variable] = value
31          domains_copy = domains.copy()
32          for unassigned_variable in unassigned_variables:
33              if unassigned_variable != variable:
34                  domains_copy [unassigned_variable] = {value}
35          result = backtrack (variable_assignments, domains_copy, graph)
36          if result is not None:
37              return result
```

```
38      variable_assignments[variable] = None
39      return None
40
41  if __name__ == '__main__':
42      vertices = [1, 2, 3, 4]
43      edges = {
44          1: [2, 3],
45          2: [1, 3, 4],
46          3: [1, 2, 4],
47          4: [2, 3]
48      }
49      graph = Graph (vertices, edges)
50
51      variable_assignments = {vertex: None for vertex in vertices}
52      domains = {vertex: {1, 2, 3} for vertex in vertices}
53      result = backtrack (variable_assignments, domains, graph)
54      print(result)
55
```

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

Program-

```
pracs.py > empty_cells
1 #code by sneha gupta 2021UCA1859
2 from math import inf as infinity
3 from random import choice
4 import platform
5 import time
6 from os import system
7
8
9 HUMAN = -1
10 COMP = +1
11 board = [
12     [0, 0, 0],
13     [0, 0, 0],
14     [0, 0, 0],
15 ]
16
17 def evaluate(state):
18     if wins(state, COMP):
19         score = +1
20     elif wins(state, HUMAN):
21         score = -1
22     else:
23         score = 0
24     return score
25
26 def wins(state, player):
27     win_state = [
28         [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]],
29         [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]
30         [2]], [state[2][0], state[1][1], state[0][2]],
31     ]
32     if [player, player, player] in win_state:
33         return True
34
35 def game_over(state):
36     return wins(state, HUMAN) or wins(state, COMP)
```

```

35
36 def empty_cells (state):
37     cells = []
38     for x, row in enumerate(state):
39         for y, cell in enumerate (row):
40             if cell == 0:
41                 cells.append([x, y])
42     return cells
43
44 def valid_move(x, y):
45     if [x, y] in empty_cells (board):
46         return True
47     else:
48         return False
49
50 def set_move(x, y, player):
51     if valid_move(x, y):
52         board[x][y]= player
53         return True
54     else:
55         return False
56
57 def minimax (state, depth, player):
58     if player == COMP:
59         best [-1, -1, -infinity]
60     else:
61         best =[-1, -1, +infinity]
62     if depth == 0 or game_over(state):
63         score = evaluate(state)
64         return [-1, -1, score]
65     for cell in empty_cells (state):
66         x, y = cell[0], cell[1]
67         state[x][y]= player
68         score= minimax(state, depth-1,-player)
69         state[x][y]= 0
70         score[0], score[1] = x,y
71

```

```

71
72     if player == COMP:
73         if score[2]> best [2]:
74             best=score # max value
75         if score[2] < best[2]:
76             best= score # min value
77     return best
78
79 def clean():
80     os_name = platform.system().lower()
81     if 'windows' in os_name:
82         system('cls')
83     else:
84         system('clear')
85
86
87 def render(state, c_choice, h_choice):
88     chars = {
89         -1: h_choice,
90         +1: c_choice,
91         0: ' '
92     }
93     str_line='_____ '
94
95     print('\n' + str_line)
96     for row in state:
97         for cell in row:
98             symbol=chars[cell]
99             print(f' |{symbol}|',end=' ')
100         print('\n'+str_line)
101
102 def ai_turn(c_choice, h_choice):
103     depth = len(empty_cells (board))
104     if depth==0 or game_over (board):
105         return
106
107     clean()

```

```

108 print (f'Computer turn [{c_choice}]')
109 render(board, c_choice, h_choice)
110
111 if depth == 9:
112     x = choice([0, 1, 2])
113     y = choice([0, 1, 2])
114 else:
115     move=minimax (board, depth, COMP)
116     x, y = move[0], move[1]
117
118 set_move(x, y, COMP)
119 time.sleep(1)
120
121
122 def human_turn (c_choice, h_choice):
123     depth = len(empty_cells (board))
124     if depth == 0 or game_over (board):
125         return
126
127 # Dictionary of valid moves
128 move = -1
129 moves = {
130     1: [0, 0], 2: [0, 1], 3: [0, 2],
131     4: [1, 0], 5: [1, 1], 6: [1, 2],
132     7: [2, 0], 8: [2, 1], 9: [2, 2],
133 }
134
135 clean()
136 print(f'Human turn [{h_choice}]')
137 render(board, c_choice, h_choice)
138
139 while move < 1 or move> 9:
140     try:
141         move = int(input('Use numpad (1..9): '))
142         coord= moves [move]
143         can_move = set_move (coord[0], coord[1], HUMAN)
144

```

```

145         if not can_move:
146             print('Bad move')
147             move = -1
148     except (EOFError, KeyboardInterrupt):
149         print('bye')
150         exit()
151     except (KeyError, ValueError):
152         print('Bad choice')
153
154
155 def main():
156
157     clean()
158     h_choice = ' '
159     c_choice = ' '
160     first = ' '
161
162     # Human chooses X or O to play
163     while h_choice != 'O' and h_choice != 'X':
164         try:
165             print('')
166             h_choice = input ('Choose X or O\nChosen: ').upper()
167         except (EOFError, KeyboardInterrupt):
168             print('Bye')
169             exit()
170         except (KeyError, ValueError):
171             print('Bad choice')
172
173     # Setting computer's choice
174     if h_choice == 'X':
175         c_choice = 'O'
176     else:
177         c_choice = 'X'
178
179     # Human may starts first
180     clean()

```

```

prac6.py > empty_cells
181 while first != 'Y' and first != 'N':
182     try:
183         first=input('First to start? [y/n]: ').upper()
184     except (EOFError, KeyboardInterrupt):
185         print('Bye')
186         exit()
187     except (KeyError, ValueError):
188         print('Bad choice')
189
190 #Main loop of this game
191 while len(empty_cells(board)) > 0 and not game_over(board):
192     if first == 'N':
193         ai_turn(c_choice, h_choice)
194         first = ''
195
196     human_turn(c_choice, h_choice)
197     ai_turn(c_choice, h_choice)
198
199 # Game over message
200 if wins (board, HUMAN):
201     clean()
202     print (f'Human turn [{h_choice}]')
203     render (board, c_choice, h_choice)
204     print('YOU WIN!')
205 elif wins (board, COMP):
206     clean()
207     print (f'Computer turn [{c_choice}]')
208     render (board, c_choice, h_choice)
209     print('YOU LOSE!')
210 else:
211     clean()
212     render (board, c_choice, h_choice)
213     print('DRAW!')
214
215 exit()
216
217 if __name__ == '__main__':

```

```

210
217 if __name__ == '__main__':
218     main()
219

```

Output:

Choose X or O Chosen: X First to start?[y/n]: y Human turn [X]	Use numpad (1..9): 3 Computer turn [O]	Human turn [X]
----- ----- ----- ----- ----- Use numpad (1..9): 1 Computer turn [O]	----- x x ----- o ----- ----- Human turn [X]	----- x o x ----- o o ----- x ----- Use numpad (1..9): 6 Computer turn [O]
----- x ----- ----- ----- Human turn [X]	----- x o x ----- o ----- ----- Use numpad (1..9): 8 Computer turn [O]	----- x o x ----- o o x ----- x ----- Human turn [X]
----- x ----- o ----- ----- Use numpad (1..9): 3 Computer turn [O]	----- x o x ----- o o ----- x ----- Human turn [X]	----- x o x ----- o o x ----- x o ----- Use numpad (1..9): 7
----- x x ----- o -----	----- x o x ----- o o ----- x ----- Use numpad (1..9): 6 Computer turn [O]	----- x o x ----- o o x ----- x x o ----- DRAW!