	BELLARI INSTITUTE OF TECHNOLOGY & MANAGEMENT			
FORMS / FORMATS (ISO 9001:2015)	Doc. No: **FAF/L4**	Release No. **5.0** Date:**01/07/2017**	Revision No. **5.0** Date: **01/07/2017**	Section: **PP 04** Form No.: R/PP 04/04

Department of Artificial Intelligence and Machine Learning

Artificial Intelligence Lab Manual

(21AIL57)

(Academic Year – 2023-24)

Semester – V

Faculty Name: Dr. Renuka Sagar / Mr. Mohammad Thoseef D

Designation: Assoc Professor / Teaching Assistant

Prepared by: **Dr. T. Machappa**	Approved by: **Dr. Yashvanth Bhupal**
Signature:	Signature:
Designation: **ISO Coordinator**	Designation: **Director**

Artificial Intelligence Lab Manual

List of Experiments:

Part A

Practicing Problems in Python (Students can be encouraged to practice good number of practice problems, some practice problems are listed here)

1. (a) Write a python program to print the multiplication table for the given number

   (b) Write a python program to check whether the given number is prime or not?

   (c) Write a python program to find factorial of the given number?

2.(a) Write a python program to implement List operations (Nested List, Length, Concatenation, Membership, Iteration, Indexing and Slicing)

   (b) Write a python program to implement List methods (Add, Append, and Extend& Delete).

3. Write a python program to implement simple Chabot with minimum 10 conversations

4. Write a python program to Illustrate Different Set Operations

5. (a) Write a python program to implement a function that counts the number of times a string (s1) occurs in another string(s2)

   (b) Write a program to illustrate Dictionary operations ([], in, traversal)and methods: keys (),values().items()

Part B

Al Problems to be implemented in Python

1. Implement and Demonstrate Depth First Search Algorithm on Water Jug Problem

2. Implement and Demonstrate Best First Search Algorithm on any Al problem

3. Implement A* Search algorithm.

4. Solve 8-Queens Problem with suitable assumptions

5. Implementation of TSP using heuristic approach

6. Implementation of the problem-solving strategies: either using Forward Chaining or Backward Chaining

7. Implement resolution principle on FOPL related problems

8. Implement any Game and demonstrate the Game playing strategies

Part A

1. (a) Write a python program to print the multiplication table for the given number

Software used: Jupyter notebook

Code:

```
num = 12
for i in range(1,11):
    print(num,'x',i,'=',num*i)
```

Output:

12 x 1 = 12
12 x 2 = 24
12 x 3 = 36
12 x 4 = 48
12 x 5 = 60
12 x 6 = 72
12 x 7 = 84
12 x 8 = 96
12 x 9 = 108
12 x 10 = 120

(b) Write a python program to check whether the given number is prime or not?

Software used: Jupyter notebook

Code:

```
num=11
if num>1:
    for i in range(2,int(num/2)+1):
        if (num%i)==0:
            print(num,"is not a prime number")
            break
    else:
        print(num,"is a prime number")
```

```
else:
    print(num,"is not a prime number")
```

Output:

11 is a prime number

(c) Write a python program to find factorial of the given number?

Software used: Jupyter notebook

Code:

```
num=int(input('enter the number'))
#num=5
fact=1
for i in range(1,num+1):
    fact=fact*i
print("The Factorial of 23 is:",fact)
```

Output:

enter the number 5
The Factorial of 23 is: 120

2.(a) Write a python program to implement List operations (Nested List, Length, Concatenation, Membership, Iteration, Indexing and Slicing)

Software used: Jupyter notebook

Code:

```
# Creating a list
my_list = [1, 2, 3, 4, 5]
# Accessing elements
print("Original List:", my_list)
print("Element at index 2:", my_list[2])
# Slicing
print("Sliced List (index 1 to 3):", my_list[1:4])
# Modifying elements
my_list[2] = 6
print("Modified List:", my_list)
# Appending and extending
my_list.append(7)
print("List after appending 7:", my_list)
my_list.extend([8, 9])
print("List after extending with [8, 9]:", my_list)
# Removing elements
my_list.remove(6)
print("List after removing 6:", my_list)
popped_element = my_list.pop(2)
print(f"Popped element at index 2: {popped_element}, Updated List: {my_list}")
# Finding index of an element
index_of_5 = my_list.index(5)
print("Index of 5:", index_of_5)
# Length of the list
```

```
list_length = len(my_list)
print("Length of the list:", list_length)
# Check if an element is in the list
element_to_check = 8
if element_to_check in my_list:
 print(f"{element_to_check} is in the list.")
else:
 print(f"{element_to_check} is not in the list.")
```

Output:

```
Original List: [1, 2, 3, 4, 5]
Element at index 2: 3
Sliced List (index 1 to 3): [2, 3, 4]
Modified List: [1, 2, 6, 4, 5]
List after appending 7: [1, 2, 6, 4, 5, 7]
List after extending with [8, 9]: [1, 2, 6, 4, 5, 7, 8, 9]
List after removing 6: [1, 2, 4, 5, 7, 8, 9]
Popped element at index 2: 4, Updated List: [1, 2, 5, 7, 8, 9]
Index of 5: 2
Length of the list: 6
8 is in the list.
```

(b) Write a python program to implement List methods (Add, Append, and Extend& Delete).

Software used: Jupyter notebook

Code:

```
# Creating a list
my_list = [1, 2, 3, 4, 5]
# Method: append
my_list.append(6)
print("List after append(6):", my_list)
# Method: extend
my_list.extend([7, 8])
```

```
print("List after extend([7, 8]):", my_list)
# Method: insert
my_list.insert(2, 10)
print("List after insert(2, 10):", my_list)
# Method: remove
my_list.remove(4)
print("List after remove(4):", my_list)
# Method: pop
popped_element = my_list.pop(1)
print(f"Popped element at index 1: {popped_element}, Updated List: {my_list}")
# Method: index
index_of_3 = my_list.index(3)
print("Index of 3:", index_of_3)
# Method: count
count_of_6 = my_list.count(6)
print("Count of 6:", count_of_6)
# Method: reverse
my_list.reverse()
print("Reversed List:", my_list)
# Method: sort
my_list.sort()
print("Sorted List:", my_list)
# Method: copy
copied_list = my_list.copy()
print("Copied List:", copied_list)
# Method: clear
my_list.clear()
print("Cleared List:", my_list)
```

Output:

List after append(6): [1, 2, 3, 4, 5, 6]
List after extend([7, 8]): [1, 2, 3, 4, 5, 6, 7, 8]
List after insert(2, 10): [1, 2, 10, 3, 4, 5, 6, 7, 8]
List after remove(4): [1, 2, 10, 3, 5, 6, 7, 8]
Popped element at index 1: 2, Updated List: [1, 10, 3, 5, 6, 7, 8]
Index of 3: 2
Count of 6: 1
Reversed List: [8, 7, 6, 5, 3, 10, 1]
Sorted List: [1, 3, 5, 6, 7, 8, 10]
Copied List: [1, 3, 5, 6, 7, 8, 10]
Cleared List: []

3. Write a python program to implement simple Chabot with minimum 10 conversations

Software used: Jupyter notebook

Code:

```
import time
now = time.ctime()
def simple_chatbot(user_input):
    conversations = {
        "hi": "Hello! How can I help you?",
        "how are you": "I'm doing well, thank you. How about you?",
        "name": "I'm a chatbot. You can call me ChatPy!",
        "age": "I don't have an age. I'm just a program.",
        "bye": "Goodbye! Have a great day.",
        "python": "Python is a fantastic programming language!",
        "weather": "I'm sorry, I don't have real-time data. You can check a weather website for updates.",
        "help": "I'm here to assist you. Ask me anything!",
        "thanks": "You're welcome! If you have more questions, feel free to ask.",
        "default": "I'm not sure how to respond to that. You can ask me something else.",
        "what is the time now": now,
    }
    # Convert user input to lowercase for case-insensitive matching
    user_input_lower = user_input.lower()
    # Retrieve the response based on user input
    response = conversations.get(user_input_lower, conversations["default"])
    return response
# Chatbot interaction loop
print("Hello! I'm ChatPy, your friendly chatbot.")
```

```
print("You can start chatting. Type 'bye' to exit.")
while True:
    user_input = input("You: ")


    if user_input.lower() == 'bye':
        print("ChatPy: Goodbye! Have a great day.")
        break
    response = simple_chatbot(user_input)
    print("ChatPy:", response)
```

Output:

```
Hello! I'm ChatPy, your friendly chatbot.
You can start chatting. Type 'bye' to exit.
You:  what is the time now
ChatPy: Tue Mar 19 13:00:08 2024
You:  thanks
ChatPy: You're welcome! If you have more questions, feel free to ask.
You:  bye
ChatPy: Goodbye! Have a great day.
```

4. Write a python program to Illustrate Different Set Operations

Software used: Jupyter notebook

Code:

```
set1={1,2,3,4,5}
set2={3,4,5,6,7}
#union of set 1 and set2
union_set=set1.union(set2)
print(union_set,"union set")
#intersection of set1 and set2
intersection_set=set1.intersection(set2)
print(intersection_set,"intersection set")
#Difference of set1 and set2
difference_set1=set1.difference(set2)
print(difference_set1,"set1-set2")
#symmetric difference
symmetric_difference_set=set1.symmetric_difference(set2)
print(symmetric_difference_set,"symmetric difference set")
#check if sets have common elements
have_common_elements=set1.isdisjoint(set2)
print("Do set1 and set2 have any common elements?",not have_common_elements)
#Adding an element to a set
set1.add(6)
print("set1 after adding element:",set1)
#Removing an element from a set
set1.remove(3)
print("set1 after removing element:",set1)
```

Output:

{1, 2, 3, 4, 5, 6, 7} union set
{3, 4, 5} intersection set
{1, 2} set1-set2
{1, 2, 6, 7} symmetric difference set
Do set1 and set2 have any common elements? True
set1 after adding element: {1, 2, 3, 4, 5, 6}
set1 after removing element: {1, 2, 4, 5, 6}

5. (a) Write a python program to implement a function that counts the number of times a string (s1) occurs in another string(s2)

Software used: Jupyter notebook

Code:

```
def count_occurrences(main_string, substring):
    count = 0
    start_index = 0
    while start_index < len(main_string):
        index = main_string.find(substring, start_index)
        if index == -1:
            break
        count += 1
        start_index = index + 1
    return count


# Example usage:
main_string = "abababab ab ab"
substring = "ab"
result = count_occurrences(main_string, substring)


print(f"The substring '{substring}' occurs {result} times in the main string.")
print(main_string.count(substring))
```

Output:

```
The substring 'ab' occurs 7 times in the main string.
7
```

(b) Write a program to illustrate Dictionary operations ([], in, traversal)and methods: keys (),values().items()

Software used: Jupyter notebook

Code:

```
# Creating a sample dictionary
sample_dict = {'a': 1, 'b': 2, 'c': 3, 'd': 4}

# Checking if a key exists in the dictionary
key_to_check = 'b'
if key_to_check in sample_dict:
    print(f'The key "{key_to_check}" exists in the dictionary.')

# Traversing the dictionary using a loop
print("Traversing the dictionary:")
for key, value in sample_dict.items():
    print(f'Key: {key}, Value: {value}')

# Dictionary methods
keys_list = list(sample_dict.keys())
values_list = list(sample_dict.values())
items_list = list(sample_dict.items())

print("\nUsing dictionary methods:")
print(f'Keys: {keys_list}')
print(f'Values: {values_list}')
print(f'Items: {items_list}')
```

Output:

```
The key "b" exists in the dictionary.
Traversing the dictionary:
Key: a, Value: 1
Key: b, Value: 2
Key: c, Value: 3
```

Key: d, Value: 4

Using dictionary methods:
Keys: ['a', 'b', 'c', 'd']
Values: [1, 2, 3, 4]
Items: [('a', 1), ('b', 2), ('c', 3), ('d', 4)]

Part B

1. Implement and Demonstrate Depth First Search Algorithm on Water Jug Problem

Software used: Jupyter notebook

Theory:

In the Water Jug Problem, we have two jugs with finite capacities (let's denote them as (A) and (B)), and we aim to measure a specific volume of water (C) using these jugs. The problem involves determining whether it's possible to reach the target volume (C) using the given jugs, and if so, finding the sequence of pouring actions to achieve it.

When using Depth-First Search (DFS) to solve the Water Jug Problem, we systematically explore the search space by trying all possible pouring actions from the current state (i.e., the amount of water in each jug) and recursively exploring the resulting states until the target volume (C) is reached or until all possible states have been explored.

DFS follows the following steps:

1. Start with an initial state where both jugs are empty.

2. Mark the initial state as visited.

3. Generate all possible successor states by applying each pouring action (filling, emptying, or pouring water between jugs) to the current state.

4. For each successor state:

   - If it hasn't been visited before, recursively apply DFS to explore it.

   - If the target volume (C) is reached in any of the successor states, terminate the search and return True.

5. If the target volume (C) cannot be reached after exploring all possible states, return False.

DFS explores the search space in a depth-first manner, meaning it goes as deep as possible along each branch of the search tree before backtracking and exploring other branches. This approach ensures that all possible sequences of pouring actions are examined until a solution is found.

DFS is a complete and systematic approach to solve the Water Jug Problem. However, it may not always be the most efficient algorithm, especially for large problem instances, as it may explore a large number of states before finding a solution.

Code:

```
def water_jug_dfs(capacity_x, capacity_y, target):
    stack = [(0, 0, [])]  # (x, y, path)
    visited_states = set()
    while stack:
        x, y, path = stack.pop()
        if (x, y) in visited_states:
            continue
        visited_states.add((x, y))
        if x == target or y == target:
            return path + [(x, y)]
        # Define possible jug operations
        operations = [
            ("fill_x", capacity_x, y),
            ("fill_y", x, capacity_y),
            ("empty_x", 0, y),
            ("empt y_y", x, 0),
            ("pour_x_to_y", max(0, x - (capacity_y - y)), min(capacity_y, y + x)),
            ("pour_y_to_x", min(capacity_x, x + y), max(0, y - (capacity_x - x))),
        ]
        # print(operations)
        for operation, new_x, new_y in operations:
            if 0 <= new_x <= capacity_x and 0 <= new_y <= capacity_y:
                stack.append((new_x, new_y, path + [(x, y, operation)]))
    return None
# Example usage:
capacity_x = 4
capacity_y = 3
```

```
target = 2
solution_path = water_jug_dfs(capacity_x, capacity_y, target)
if solution_path:
    print("Solution found:")
    for state in solution_path:
        print(f"({state[0]}, {state[1]})")
else:
    print("No solution found.")
```

Output:

```
Solution found:
(0, 0)
(0, 3)
(3, 0)
(3, 3)
(4, 2)
```

2. Implement and Demonstrate Breadth First Search Algorithm on any AI problem

Software used: Jupyter notebook

Theory:

Breadth-first search (BFS) is a graph traversal algorithm used to explore all the nodes of a graph or tree systematically, starting from a specified root node and visiting its neighbors before moving on to the next level of nodes. BFS is particularly useful for finding the shortest path between two nodes in an unweighted graph or for exploring a graph without getting stuck in cycles.

Here's how BFS works:

1. Start with a queue data structure and enqueue the root node.

2. While the queue is not empty:

   a. Dequeue a node from the front of the queue. This node becomes the current node.

   b. Visit the current node and mark it as visited.

   c. Enqueue all the unvisited neighbors of the current node.

3. Repeat steps 2a-2c until the queue is empty.

BFS guarantees that all nodes at a given level will be visited before moving on to the next level. This ensures that the shortest path between the starting node and any other reachable node is found first.

BFS is often implemented using a queue data structure, which follows the First-In-First-Out (FIFO) principle. This ensures that nodes are visited in the order they were discovered, leading to a breadth-first traversal of the graph or tree.

BFS is commonly used in various applications, including shortest path algorithms, network analysis, and puzzle-solving algorithms. It has a time complexity of O(V + E), where V is the number of vertices and E is the number of edges in the graph.

Code:

```
#BFS

tree ={

1: [2,9,10],

2: [3,4],

3: [],
```

```
4: [5,6,7],
5: [8],
6: [],
7: [],
8: [],
9: [],
10: []
}
def breadth_first_search(tree,start):
   q=[start]
   visited=[]
   while q:
      print("before",q)
      node=q.pop(0)
      visited.append(node)
      for child in (tree[node]):
         if child not in visited and child not in q:
            q.append(child)
      print("after",q)
   return visited
result=breadth_first_search(tree,1)
print(result)
```

Output:

```
before [1]
after [2, 9, 10]
before [2, 9, 10]
after [9, 10, 3, 4]
before [9, 10, 3, 4]
after [10, 3, 4]
before [10, 3, 4]
```

```
after [3, 4]
before [3, 4]
after [4]
before [4]
after [5, 6, 7]
before [5, 6, 7]
after [6, 7, 8]
before [6, 7, 8]
after [7, 8]
before [7, 8]
after [8]
before [8]
after []
[1, 2, 9, 10, 3, 4, 5, 6, 7, 8]
```

3. Implement A* Search algorithm.

Software used: Jupyter notebook

Theory:

The A* search algorithm is a popular pathfinding algorithm used in artificial intelligence and graph traversal problems. It efficiently finds the shortest path from a starting node to a goal node, taking into account both the cost of reaching each node and an estimate of the remaining cost to reach the goal. A* is an informed search algorithm, meaning it uses heuristic information to guide its search.

A* is guaranteed to find the shortest path when the heuristic function is admissible, meaning it never overestimates the cost to reach the goal. The efficiency and effectiveness of A* depend on the quality of the heuristic function used.

Code:

```
import heapq

# Example road network graph

road_graph = {

'Arad': {'Zerind': 75, 'Timisoara': 118, 'Sibiu': 140},

'Zerind': {'Arad': 75, 'Oradea': 71},

'Timisoara': {'Arad': 118, 'Lugoj': 111},

'Sibiu': {'Arad': 140, 'Oradea': 151, 'Fagaras': 99, 'Rimnicu Vilcea': 80},

'Oradea': {'Zerind': 71, 'Sibiu': 151},

'Lugoj': {'Timisoara': 111, 'Mehadia': 70},

'Fagaras': {'Sibiu': 99, 'Bucharest': 211},

'Rimnicu Vilcea': {'Sibiu': 80, 'Pitesti': 97, 'Craiova': 146},

'Mehadia': {'Lugoj': 70, 'Drobeta': 75},

'Drobeta': {'Mehadia': 75, 'Craiova': 120},

'Craiova': {'Drobeta': 120, 'Rimnicu Vilcea': 146, 'Pitesti': 138},

'Pitesti': {'Rimnicu Vilcea': 97, 'Craiova': 138, 'Bucharest': 101},

'Bucharest': {'Fagaras': 211, 'Pitesti': 101}

}
```

```
heuristic_cost = {
    "Arad": {"Bucharest": 366},
    "Bucharest": {"Bucharest": 0},
    "Craiova": {"Bucharest": 160},
    "Dobreta": {"Bucharest": 242},
    "Eforie": {"Bucharest": 161},
    "Fagaras": {"Bucharest": 176},
    "Giurgiu": {"Bucharest": 77},
    "Hirsowa": {"Bucharest": 151},
    "Lasi": {"Bucharest": 226},
    "Lugoj": {"Bucharest": 244},
    "Mehadia": {"Bucharest": 241},
    "Neamt": {"Bucharest": 234},
    "Oradea": {"Bucharest": 380},
    "Pitesti": {"Bucharest": 100},
    "Rimnicu Vilcea": {"Bucharest": 193},
    "Sibiu": {"Bucharest": 253},
    "Timisoara": {"Bucharest": 329},
    "Urziceni": {"Bucharest": 80},
    "Vaslui": {"Bucharest": 199},
    "Zerind": {"Bucharest": 374}
}


def heuristic_cost_estimate(node, goal):
    return heuristic_cost[node][goal]


def a_star(graph, start, goal):
    open_set = [(0, start)] # Priority queue with initial node
```

```
    came_from = {}
    g_score = {city: float('inf') for city in graph}
    g_score[start] = 0
    while open_set:
        current_cost, current_city = heapq.heappop(open_set)
        if current_city == goal:
            path = reconstruct_path(came_from, goal)
            # print(graph)
            return path

        for neighbor, cost in graph[current_city].items():
            tentative_g_score = g_score[current_city] + cost
            if tentative_g_score < g_score[neighbor]:
                g_score[neighbor] = tentative_g_score
                f_score = tentative_g_score + heuristic_cost_estimate(neighbor, goal)
                heapq.heappush(open_set, (f_score, neighbor))
                came_from[neighbor] = current_city
    return None # No path found

def reconstruct_path(came_from, current_city):
    path = [current_city]
    while current_city in came_from:
        current_city = came_from[current_city]
        path.insert(0, current_city)
    return path

def calculate_distance(graph, path):
    total_distance = 0
```

```
    for i in range(len(path)-1):
        current_city = path[i]
        next_city = path[i+1]
        total_distance += graph[current_city][next_city]
    return total_distance

start_city = 'Arad'
goal_city = 'Bucharest'

path = a_star(road_graph, start_city, goal_city)
distance = calculate_distance(road_graph, path)

print("Shortest Path from {} to {}: {}".format(start_city, goal_city, path))
print("Total distance: {}".format(distance))
```

Output:

```
Shortest Path from Arad to Bucharest: ['Arad', 'Sibiu', 'Rimnicu Vilcea', 'Pitesti', 'Buchare
st']
Total distance: 418
```

4. Solve 8-Queens Problem with suitable assumptions

Software used: Jupyter notebook

Theory:

The N-Queens Problem is a classic problem in computer science and combinatorial optimization. It involves placing N queens on an N×N chessboard such that no two queens threaten each other. In other words, no two queens can share the same row, column, or diagonal.

The problem can be solved using various algorithms, including backtracking, recursion, and constraint satisfaction techniques. The most common approach is the backtracking algorithm, which systematically explores different configurations of queen placements until a valid solution is found or all possibilities are exhausted.

Here's a basic outline of the backtracking algorithm for the N-Queens Problem:

1. Start with an empty chessboard.

2. Place a queen in the first row, column by column, and recursively try to place queens in the subsequent rows.

3. If a queen can be placed in a column without threatening any other queens, move to the next row and repeat step 2.

4. If no queen can be placed in the current row without threatening others, backtrack to the previous row and try placing the queen in the next available column.

5. Repeat steps 2-4 until all queens are placed on the board or all possibilities are exhausted.

The algorithm terminates when a valid solution is found or when all possible configurations have been explored.

The N-Queens Problem has applications in various fields, including computer science, mathematics, and artificial intelligence, and serves as a benchmark for testing optimization algorithms and constraint satisfaction techniques.

Code:

```
#Taking number of queens as input from user

print ("Enter the number of queens")

N = int(input())

# here we create a chessboard

# NxN matrix with all elements set to 0

board = [[0]*N for _ in range(N)]
```

```
def attack(i, j):
    #checking vertically and horizontally
    for k in range(0,N):
        if board[i][k]==1 or board[k][j]==1:
            return True
    #checking diagonally
    for k in range(0,N):
        for l in range(0,N):
            if (k+l==i+j) or (k-l==i-j):
                if board[k][l]==1:
                    return True
    return False
def N_queens(n):
    if n==0:
        return True
    for i in range(0,N):
        for j in range(0,N):
            if (not(attack(i,j))) and (board[i][j]!=1):
                board[i][j] = 1
                if N_queens(n-1)==True:
                    return True
                board[i][j] = 0
    return False
N_queens(N)
for i in board:
    print (i)
```

Output:

Enter the number of queens

```
8
[1, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 1, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 1]
[0, 0, 0, 0, 0, 1, 0, 0]
[0, 0, 1, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 1, 0]
[0, 1, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 1, 0, 0, 0, 0]
```

5. Implementation of TSP using heuristic approach

Software used: Jupyter notebook

Theory:

The Traveling Salesman Problem (TSP) is a classic problem in computer science and optimization. It involves finding the shortest possible route that visits a given set of cities and returns to the original city, with each city visited exactly once. The problem is NP-hard, meaning that as the number of cities increases, finding the optimal solution becomes increasingly difficult.

Several approaches are used to tackle the TSP, including:

1. Exact Algorithms: These algorithms guarantee finding the optimal solution but are often computationally expensive and only feasible for small problem instances. Examples include dynamic programming and branch and bound.

2. Heuristic Algorithms: These algorithms aim to find a good solution in a reasonable amount of time but do not guarantee optimality. Examples include nearest neighbor, genetic algorithms, simulated annealing, and ant colony optimization.

3. Approximation Algorithms: These algorithms provide solutions that are guaranteed to be within a certain factor of the optimal solution. Examples include Christofides algorithm and the Lin-Kernighan heuristic.

The TSP has applications in various fields such as logistics, transportation, and manufacturing, where finding an efficient route is essential for cost-saving and resource optimization.

Code:

```
#TSP

from itertools import permutations

def calculate_total_distance(tour, distances):

    total_distance = 0

    for i in range(len(tour) - 1):

        total_distance += distances[tour[i]][tour[i + 1]]

    total_distance += distances[tour[-1]][tour[0]]  # Return to the starting city

    # print(total_distance)

    return total_distance
```

```
def traveling_salesman_bruteforce(distances):
    cities = range(len(distances))
    min_distance = float('inf')
    optimal_tour = None
    for tour in permutations(cities):
        # print(tour)
        distance = calculate_total_distance(tour, distances)
        if distance < min_distance:
            min_distance = distance
            optimal_tour = tour
        # print(tour, distance)
    return optimal_tour, min_distance
# Example usage:
# Replace the distances matrix with your own data
distances_matrix = [
    [0, 10, 15, 20],
    [10, 0, 35, 25],
    [15, 35, 0, 30],
    [20, 25, 30, 0]
]
optimal_tour, min_distance = traveling_salesman_bruteforce(distances_matrix)
print("Optimal Tour:", optimal_tour)
print("Minimum Distance:", min_distance)
```

Output:

Optimal Tour: (0, 1, 3, 2)
Minimum Distance: 80

6. Implementation of the problem-solving strategies: either using Forward Chaining or Backward Chaining

Software used: Jupyter notebook

Theory:

Forward chaining is a reasoning method used in expert systems and rule-based systems to derive conclusions from a set of rules and facts. It starts with the known facts and repeatedly applies inference rules to deduce new facts until no further conclusions can be drawn. This method is also known as data-driven reasoning because it relies on the available data to reach conclusions. Forward chaining is commonly used in systems like expert systems, decision support systems, and diagnostic systems.

Backward chaining is a reasoning method used in expert systems and rule-based systems to determine whether a given goal can be satisfied by working backward from the goal to the known facts and rules. It starts with the goal to be achieved and then searches for rules that could be applied to satisfy that goal. It recursively applies rules to subgoals until it reaches known facts or fails to find a solution. Backward chaining is particularly useful in systems where the number of possible goals is limited or where the problem-solving process can be naturally decomposed into a series of subgoals. It is commonly used in diagnostic systems, planning systems, and expert systems.

Code:

```
global facts
global rules
rules = True
facts =[["plant","mango"],["eating","mango"], ["seed","sprouts"]]
def assert_fact(fact):
    global facts
    global rules
    if not fact in facts:
        facts+=[fact]
        rules=True
while rules:
    rules=False
    for A1 in facts:
```

```
        if A1[0]=="seed":
            assert_fact(["plant",A1[1]])
        if A1[0]=="plant":
            assert_fact(["fruit",A1[1]])
        if A1[0]=="plant" and ["eating",A1[1]] in facts:
            assert_fact(["human",A1[1]])
print(facts)
```

Output:

[['plant', 'mango'], ['eating', 'mango'], ['seed', 'sprouts'], ['fruit', 'mango'], ['human', 'mango'], ['plant', 'sprouts'], ['fruit', 'sprouts']]

7. Implement resolution principle on FOPL related problems

Software used: Jupyter notebook

Theory:

The resolution principle is a fundamental inference rule in logic and automated reasoning. It is used to derive new clauses from existing ones by combining complementary literals and eliminating redundant terms. This process is crucial in resolution-based theorem proving, which is widely used in artificial intelligence and automated reasoning systems.

Code:

```
from sympy import symbols, Not, Or, simplify

def resolve(clause1, clause2):
    """
    Resolve two clauses and return the resulting resolvent.
    """
    resolvent = []
    for literal1 in clause1:
        for literal2 in clause2:
            if literal1 == Not(literal2) or literal2 == Not(literal1):
                resolvent.extend([l for l in (clause1 + clause2) if l != literal1 and l != literal2])
    return list(set(resolvent))


def resolution(clauses):
    """
    Apply resolution to a set of clauses until no new clauses can be generated.
    """
    new_clauses = list(clauses)
    while True:
        n = len(new_clauses)
        print(new_clauses)
        print("-------------------------------------")
```

```
        pairs = [(new_clauses[i], new_clauses[j]) for i in range(n) for j in range(i+1, n)]
        for (clause1, clause2) in pairs:
            print(clause1)
            print(clause2)
            resolvent = resolve(clause1, clause2)
            print(resolvent)
            print("---------------")
            if not resolvent:
                # Empty clause found, contradiction reached
                return True
            if resolvent not in new_clauses:
                new_clauses.append(resolvent)
        if n == len(new_clauses):
            # No new clauses can be generated, exit loop
            return False

# Example usage:
if __name__ == "__main__":
    # Example clauses in CNF (Conjunctive Normal Form)
    clause1 = [symbols('P'), Not(symbols('Q'))]
    clause2 = [Not(symbols('P')), symbols('Q')]
    clause3 = [Not(symbols('P')), Not(symbols('Q'))]

    # List of clauses
    clauses = [clause1, clause2, clause3]
    result = resolution(clauses)
    if result:
        print("The set of clauses is unsatisfiable (contradiction found).")
```

```
    else:
        print("The set of clauses is satisfiable.")
```

Output:

```
[[P, ~Q], [~P, Q], [~P, ~Q]]
-------------------------------------
[P, ~Q]
[~P, Q]
[~P, P, ~Q, Q]
---------------
[P, ~Q]
[~P, ~Q]
[~Q]
---------------
[~P, Q]
[~P, ~Q]
[~P]
---------------
[[P, ~Q], [~P, Q], [~P, ~Q], [~P, P, ~Q, Q], [~Q], [~P]]
-------------------------------------
[P, ~Q]
[~P, Q]
[~P, P, ~Q, Q]
---------------
[P, ~Q]
[~P, ~Q]
[~Q]
---------------
[P, ~Q]
[~P, P, ~Q, Q]
[~P, P, ~Q, Q]
---------------
[P, ~Q]
[~Q]
[]
---------------
The set of clauses is unsatisfiable (contradiction found).
```

8. Implement any Game and demonstrate the Game playing strategies

Software used: Jupyter notebook

Theory:

The game is to be played between two people One of the player chooses 'O' and the other 'X' to mark their respective cells. The game starts with one of the players and the game ends when one of the players has one whole row/ column/ diagonal filled with his/her respective character ('O' or 'X'). If no one wins, then the game is said to be draw.

Code:

```
# Tic-Tac-Toe game in Python
board = [" " for x in range(9)]
def print_board():
    row1 = "| {} | {} | {} |".format(board[0], board[1], board[2])
    row2 = "| {} | {} | {} |".format(board[3], board[4], board[5])
    row3 = "| {} | {} | {} |".format(board[6], board[7], board[8])
    print()
    print(row1)
    print(row2)
    print(row3)
    print()
def player_move(icon):
    if icon == "X":
        number = 1
    elif icon == "O":
        number = 2
    print("Your turn player {}".format(number))
    choice = int(input("Enter your move (1-9): ").strip())
    if board[choice - 1] == " ":
        board[choice - 1] = icon
```

```
        else:
            print()
            print("That space is taken!")
    def is_victory(icon):
        if (board[0] == icon and board[1] == icon and board[2] == icon) or \
            (board[3] == icon and board[4] == icon and board[5] == icon) or \
            (board[6] == icon and board[7] == icon and board[8] == icon) or \
            (board[0] == icon and board[3] == icon and board[6] == icon) or \
            (board[1] == icon and board[4] == icon and board[7] == icon) or \
            (board[2] == icon and board[5] == icon and board[8] == icon) or \
            (board[0] == icon and board[4] == icon and board[8] == icon) or \
            (board[2] == icon and board[4] == icon and board[6] == icon):
            return True
        else:
            return False
    def is_draw():
        if " " not in board:
            return True
        else:
            return False
    while True:
        print_board()
        player_move("X")
        print_board()
        if is_victory("X"):
            print("X wins! Congratulations!")
            break
        elif is_draw():
```

```
            print("It's a draw!")
            break
        player_move("O")
        if is_victory("O"):
            print_board()
            print("O wins! Congratulations!")
            break
        elif is_draw():
            print("It's a draw!")
            break
```

Output:

```
|   |   |   |
|   |   |   |
|   |   |   |

Your turn player 1
Enter your move (1-9):  1

| X |   |   |
|   |   |   |
|   |   |   |

Your turn player 2
Enter your move (1-9):  2

| X | O |   |
|   |   |   |
|   |   |   |

Your turn player 1
Enter your move (1-9):  3

| X | O | X |
|   |   |   |
|   |   |   |

Your turn player 2
Enter your move (1-9):  4
```

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

Your turn player 1
Enter your move (1-9):  5

| X | O | X |
| O | X |   |
|   |   |   |

Your turn player 2
Enter your move (1-9):  6

| X | O | X |
| O | X | O |
|   |   |   |

Your turn player 1
Enter your move (1-9):  7

| X | O | X |
| O | X | O |
| X |   |   |

X wins! Congratulations!
```