	<b>BELLARI INSTITUTE OF TECHNOLOGY &amp; MANAGEMENT</b>			
<b>FORMS / FORMATS</b> (ISO 9001:2015)	Doc. No: <b>FAF/L4</b>	Release No. <b>5.0</b> Date: <b>01/07/2017</b>	Revision No. <b>5.0</b> Date: <b>01/07/2017</b>	Section: <b>PP 04</b> Form No.: R/PP 04/03

**Department of Computer Science & Engineering**  
**(DATA SCIENCE)**

**Artificial Intelligence Lab Manual**  
**(22CDL54)**

(Academic Year – 2024-25)

**Semester – V**


**Faculty Name:** Dr. JAGADISH R M/Mrs. ANUSHYA V P

**Designation:** Professor /Assistant Professor

Staff  
Signature

HOD  
Signature

Principal  
Signature

Prepared by: **Dr. T. Machappa**  
 Signature:   
 Designation: **ISO Coordinator**

Approved by: **Dr. Yashvanth Bhupal**  
 Signature:   
 Designation: **Director**

**Semester: VI****Course Name: Artificial Intelligence Lab**

Course Code	<b>22CDL54</b>	CIE Marks	<b>50</b>
Teaching Hours/Week (L:T:P)	<b>0 : 0 : 2</b>	SEE Marks	<b>50</b>
Total Hours of Pedagogy	<b>20</b>	Total Marks	<b>100</b>
Credits	<b>01</b>	Exam Hours	<b>03</b>

**Course Objectives:**

1. To impart knowledge about Artificial Intelligence.
2. To give understanding of the main abstractions and reasoning for intelligent systems.
3. To enable the students to understand the basic principles of Artificial Intelligence in various applications.
4. To provide skills for designing and analyzing AI based algorithms.
5. To provide skills to work towards solution of real life problems.

**List of Experiments:**

SN	Experiments: Part-A
1.	a) Write a Python program to print the multiplication table of a given number. b) Write a Python program to check whether the given number is Prime or not. c) Write a Python program to find the factorial of the given number.
2.	a) Write a python program to implement list of operations (Nested list, Length, Concatenation, Membership, iteration, Indexing & Slicing). b) Write a Python program to implement List Methods ( Add, Append, and Extend & Delete
3.	Write a Python program to implement a simple chat bot with a minimum of 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 string (s2). b) Write a python program to illustrate Dictionary operations ([], traversal) and methods: keys(), values(), items().
Experiments: Part-B	
1.	implement and demonstrate the application of Depth first search on water jug problem.
2.	Implement and demonstrate the application of Breadth first search on any AI Problem.
3.	Implement A* search algorithm.
4.	Implement AO* search algorithm.
5.	Solve 8 Queen's problem with suitable assumptions.
6.	Implement Travelling salesman problem using heuristic approach.
7.	Implement problem solving strategies using: Forward chaining or Backward chaining.
8.	Implement resolution principle in FOPL
9.	Implement Game theory and demonstrate game playing strategies.

Prepared by: **Dr. T. Machappa**

Signature:

Designation: **ISO Coordinator**Approved by: **Dr. Yashvanth Bhupal**

Signature:

Designation: **Director**

**Course outcomes:**

The students should be able to:

1. Apply fundamental concepts of AI in solving real world problems.
2. Demonstrate the ability to solve problems using searching and backtracking.
3. Choose appropriate algorithm for solving given AI problems.
4. Apply state space representation to real life problems using searching and game playing.
5. Analyze the fundamental concepts of search algorithms for optimization engineering problem.

Prepared by: **Dr. T. Machappa**

Signature: 

Designation: **ISO Coordinator**

Approved by: **Dr. Yashvanth Bhupal**

Signature: 

Designation: **Director**



Prg. No.	Part A - Experiments
1.	<p><b>(a) Write a python program to print the multiplication table for the given number</b></p> <p><b>Code:</b></p> <pre>num = 12 for i in range(1,11):     print(num,'x',i,'=',num*i)</pre> <div> <div>OUTPUT</div> <div> 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 </div> </div>
	<p><b>(b) Write a python program to check whether the given number is prime or not?</b></p> <pre>num = 11 if num &gt; 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")</pre> <div> <div>OUTPUT:</div> <div>11 is a prime number</div> </div>
	<p><b>(C) Write a python program to find factorial of the given number?</b></p> <pre>num=int(input('enter the number ::')) #num=5 fact=1 for i in range(1,num+1):     fact=fact*i print("The Factorial of 5 is:",fact)</pre> <div> <div>OUTPUT</div> <div> enter the number :: 5  The Factorial of 5 is: 120 </div> </div>

Prepared by: **Dr. T. Machappa**

Signature:

Designation: **ISO Coordinator**Approved by: **Dr. Yashvanth Bhupal**

Signature:

Designation: **Director**



2.

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

# Nested List

```
list = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]  
print(list)
```

#Length of the List

```
length=len(list)  
print('length of the list',length)
```

# Concatenation of Lists

```
list1 = [1, 2, 3]  
list2 = [4, 5, 6]  
concatenated_list = list1 + list2  
print(concatenated_list)
```

### **Python Membership Operators**

The Python membership operators test for the membership of an object in a sequence, such as strings, lists, or tuples.

# Membership Check

```
print(2 in list1)  
print(3 not in list2)
```

### **# Iterate over a list**

```
for i in list1:  
    print(i)
```

The **index()** method returns the position at the first occurrence of the specified value.

```
fruits = ['apple', 'banana', 'cherry']
```

```
x = fruits.index("apple")
```

```
print(x)
```

**# Slicing**

Prepared by: **Dr. T. Machappa**

Signature: 

Designation: **ISO Coordinator**

Approved by: **Dr. Yashvanth Bhupal**

Signature: 

Designation: **Director**



List = [50, 70, 30, 20, 90, 10, 50]

# Display list

print(List[1:5])

**Output:**

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

length of the list 3

[1, 2, 3, 4, 5, 6]

True

True

1

2

3

0

[70, 30, 20, 90]

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

fruits=['Apple','Banana','Cherry','Grapes','Orange']

print(fruits)

fruits.insert(2,'watermelon')

print(fruits)

#Add function

fruits.append('mango')

print(fruits)

fruits.extend([4,5,6])

print(fruits)

fruits.remove('Banana')

print(fruits)

Prepared by: **Dr. T. Machappa**

Signature: 

Designation: **ISO Coordinator**

Approved by: **Dr. Yashvanth Bhupal**

Signature: 

Designation: **Director**

**Output**

['Apple', 'Banana', 'Cherry', 'Grapes', 'Orange']  
['Apple', 'Banana', 'watermelon', 'Cherry', 'Grapes', 'Orange']  
['Apple', 'Banana', 'watermelon', 'Cherry', 'Grapes', 'Orange', 'mango']  
['Apple', 'Banana', 'watermelon', 'Cherry', 'Grapes', 'Orange', 'mango', 4, 5, 6]  
['Apple', 'watermelon', 'Cherry', 'Grapes', 'Orange', 'mango', 4, 5, 6]

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

```
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':
```

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



```
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.

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



**4.****Write a python program to Illustrate Different Set Operations.**`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)`**OUTOUT:**`{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).**

```
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 = "ababababab 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**

Prepared by: **Dr. T. Machappa**

Signature: 

Designation: **ISO Coordinator**

Approved by: **Dr. Yashvanth Bhupal**

Signature: 

Designation: **Director**



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

**# 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 : Experiments****1.****Implement and Demonstrate Depth First Search Algorithm on Water Jug Problem.****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.

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

**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
```

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

**# 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:****(Jug1: 0, Jug2: 0)****(Jug1: 0, Jug2: 3)****(Jug1: 3, Jug2: 0)****(Jug1: 3, Jug2: 3)****(Jug1: 4, Jug2: 2)**Prepared by: **Dr. T. Machappa**Signature: Designation: **ISO Coordinator**Approved by: **Dr. Yashvanth Bhupal**Signature: Designation: **Director**

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

**after [2, 9]**

**after [2, 9, 10]**

**before [2, 9, 10]**

**after [9, 10, 3]**

**after [9, 10, 3, 4]**

**before [9, 10, 3, 4]**

**before [10, 3, 4]**

**before [3, 4]**

**before [4]**

**after [5]**

**after [5, 6]**

**after [5, 6, 7]**

**before [5, 6, 7]**

**after [6, 7, 8]**

**before [6, 7, 8]**

**before [7, 8]**

**before [8]**

**[1, 2, 9, 10, 3, 4, 5, 6, 7, 8]**

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



**3.****Implement A\* Search algorithm.****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:**

```

inf=99999
g=[
    [0,4,3,inf,inf,inf,inf],
    [inf,0,inf,inf,12,5,inf],
    [inf,inf,0,7,10,inf,inf],
    [inf,inf,inf,0,2,inf,inf],
    [inf,inf,inf,inf,0,inf,5],
    [inf,inf,inf,inf,inf,0,16],
    [inf,inf,inf,inf,inf,inf,0],
]
h=[14,12,11,6,4,11,0]
src=0
goal=6
class obj:
    def __init__(self,cost,path):
        self.cost=cost
        self.path=path
arr=[]
new_item=obj(h[src],[src])
arr.append(new_item)
while arr:
    cur_item=arr[0]
    cur_node=cur_item.path[-1]
    cur_cost=cur_item.cost
    cur_path=cur_item.path
    for i in range(0,len(h)):
        if g[cur_node][i]!=inf and g[cur_node][i]!=0:

```

Prepared by: **Dr. T. Machappa**

Signature:

Designation: **ISO Coordinator**Approved by: **Dr. Yashvanth Bhupal**

Signature:

Designation: **Director**



```

new_cost=cur_cost-h[cur_node]+h[i]+g[cur_node][i]
new_path=cur_path.copy()
new_path.append(i)
if i==goal:
    print("COST :", new_cost)
    print("PATH :", new_path)
    #sys.exit()
new_item=obj(new_cost,new_path)
arr.append(new_item)
arr.pop(0)
arr=sorted(arr,key=lambda item:item.cost)

```

**OUTPUT:****COST : 17****PATH : [0, 2, 3, 4, 6]****COST : 18****PATH : [0, 2, 4, 6]****COST : 21****PATH : [0, 1, 4, 6]****COST : 25****PATH : [0, 1, 5, 6]****4. Implement AO\* Search algorithm.**

```

# Cost to find the AND and OR path
def Cost(H, condition, weight = 1):
    cost = { }
    if 'AND' in condition:
        AND_nodes = condition['AND']
        Path_A = ' AND '.join(AND_nodes)
        PathA = sum(H[node]+weight for node in AND_nodes)
        cost[Path_A] = PathA

    if 'OR' in condition:
        OR_nodes = condition['OR']
        Path_B = ' OR '.join(OR_nodes)
        PathB = min(H[node]+weight for node in OR_nodes)
        cost[Path_B] = PathB
    return cost

# Update the cost
def update_cost(H, Conditions, weight=1):
    Main_nodes = list(Conditions.keys())
    Main_nodes.reverse()
    least_cost= { }
    for key in Main_nodes:
        condition = Conditions[key]
        print(key,':', Conditions[key], '>>>', Cost(H, condition, weight))

```

Prepared by: **Dr. T. Machappa**

Signature:

Designation: **ISO Coordinator**Approved by: **Dr. Yashvanth Bhupal**

Signature:

Designation: **Director**



```

c = Cost(H, condition, weight)
H[key] = min(c.values())
least_cost[key] = Cost(H, condition, weight)
return least_cost

# Print the shortest path
def shortest_path(Start, Updated_cost, H):
    Path = Start
    if Start in Updated_cost.keys():
        Min_cost = min(Updated_cost[Start].values())
        key = list(Updated_cost[Start].keys())
        values = list(Updated_cost[Start].values())
        Index = values.index(Min_cost)

        # FIND MINIMUM PATH KEY
        Next = key[Index].split()
        # ADD TO PATH FOR OR PATH
        if len(Next) == 1:

            Start = Next[0]
            Path += '<--' + shortest_path(Start, Updated_cost, H)
        # ADD TO PATH FOR AND PATH
        else:
            Path += '<--(' + key[Index] + ') '

            Start = Next[0]
            Path += '[' + shortest_path(Start, Updated_cost, H) + ' + '

            Start = Next[-1]
            Path += shortest_path(Start, Updated_cost, H) + ']'

    return Path

H = {'A': -1, 'B': 5, 'C': 2, 'D': 4, 'E': 7, 'F': 9, 'G': 3, 'H': 0, 'I': 0, 'J': 0}

Conditions = {
    'A': {'OR': ['B'], 'AND': ['C', 'D']},
    'B': {'OR': ['E', 'F']},
    'C': {'OR': ['G'], 'AND': ['H', 'I']},
    'D': {'OR': ['J']}
}
# weight
weight = 1
# Updated cost
print('Updated Cost :')
Updated_cost = update_cost(H, Conditions, weight=1)
print('*'*75)
print('Shortest Path :\n', shortest_path('A', Updated_cost, H))

```

Prepared by: **Dr. T. Machappa**

Signature:

Designation: **ISO Coordinator**Approved by: **Dr. Yashvanth Bhupal**

Signature:

Designation: **Director**

**output:****Updated Cost :****D : {'OR': ['J']} >>> {'J': 1}****C : {'OR': ['G'], 'AND': ['H', 'I']} >>> {'H AND I': 2, 'G': 4}****B : {'OR': ['E', 'F']} >>> {'E OR F': 8}****A : {'OR': ['B'], 'AND': ['C', 'D']} >>> {'C AND D': 5, 'B': 9}**

\*\*\*\*\*

**Shortest Path :****A<--(C AND D) [C<--(H AND I) [H + I] + D<--J]**Prepared by: **Dr. T. Machappa**Signature: Designation: **ISO Coordinator**Approved by: **Dr. Yashvanth Bhupal**Signature: Designation: **Director**

**5. Solve 8-Queens Problem with suitable assumptions****Theory:**

The N-Queens Problem is a classic problem in computer science and combinatorial optimization. It involves placing N queens on an  $N \times N$  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

5

[1, 0, 0, 0, 0]

[0, 0, 1, 0, 0]

[0, 1, 0, 0, 0]

[0, 0, 0, 0, 1]

[0, 0, 0, 1, 0]

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

**6. Implementation of TSP using a heuristic approach****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:**

```
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
```

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

**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  
04/03

```
# 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, 2, 3, 1)  
Minimum Distance : 100

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



**7. Implementation of the problem-solving strategies: either using Forward Chaining or Backward Chaining.****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']]
```

Prepared by: **Dr. T. Machappa**

Signature:

Designation: **ISO Coordinator**Approved by: **Dr. Yashvanth Bhupal**

Signature:

Designation: **Director**

**8. Implement resolution principle on FOPL related problems****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.

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\_\_":

Prepared by: **Dr. T. Machappa**

Signature: 

Designation: **ISO Coordinator**

Approved by: **Dr. Yashvanth Bhupal**

Signature: 

Designation: **Director**



```
# 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]

[~Q, ~P, Q, P]

[P, ~Q]

[~P, ~Q]

[~Q]

[~P, Q]

[~P, ~Q]

[~P]

[[P, ~Q], [~P, Q], [~P, ~Q], [~Q, ~P, Q, P], [~Q], [~P]]

[P, ~Q]

[~P, Q]

[~Q, ~P, Q, P]

[P, ~Q]

[~P, ~Q]

[~Q]

[P, ~Q]

[~Q, ~P, Q, P]

[~Q, ~P, Q, P]

[P, ~Q]

[~Q]

[]

Prepared by: **Dr. T. Machappa**

Signature:

Designation: **ISO Coordinator**Approved by: **Dr. Yashvanth Bhupal**

Signature:

Designation: **Director**

**9. Implement any Game and demonstrate the Game playing strategies.****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):
```

Prepared by: **Dr. T. Machappa**

Signature:

Designation: **ISO Coordinator**Approved by: **Dr. Yashvanth Bhupal**

Signature:

Designation: **Director**



```
        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 1:**

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

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

```
| | | |
| | X | |
| | | |
```

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



Your turn player 2

Enter your move (1-9): 3

			O	
		X		

Your turn player 1

Enter your move (1-9): 7

			O	
		X		
	X			

Your turn player 2

Enter your move (1-9): 4

			O	
	O	X		
	X			

Your turn player 1

Enter your move (1-9): 5

That space is taken!

			O	
	O	X		
	X			

Your turn player 2

Enter your move (1-9): 6

			O	
	O	X	O	
	X			

Prepared by: **Dr. T. Machappa**

Signature:

Designation: **ISO Coordinator**Approved by: **Dr. Yashvanth Bhupal**

Signature:

Designation: **Director**



Your turn player 1

Enter your move (1-9): 2

		X		O	
	O		X		O
	X				

Your turn player 2

Enter your move (1-9): 8

		X		O	
	O		X		O
	X		O		

Your turn player 1

Enter your move (1-9): 9

		X		O	
	O		X		O
	X		O		X

Your turn player 2

Enter your move (1-9): 1

It's a draw!

**OUTPUT 2:**


Your turn player 1

Enter your move (1-9): 1

	X			

Prepared by: **Dr. T. Machappa**

Signature:

Designation: **ISO Coordinator**Approved by: **Dr. Yashvanth Bhupal**

Signature:

Designation: **Director**



Your turn player 2

Enter your move (1-9): 2

	X		O		

Your turn player 1

Enter your move (1-9): 5

	X		O		
		X			

Your turn player 2

Enter your move (1-9): 6

	X		O		
		X		O	

Your turn player 1

Enter your move (1-9): 9

	X		O		
		X		O	
			X		

X wins! Congratulations!

Prepared by: **Dr. T. Machappa**

Signature:

Designation: **ISO Coordinator**Approved by: **Dr. Yashvanth Bhupal**

Signature:

Designation: **Director**