

<b>CS17</b>	<b>ARTIFICIAL INTELLIGENCE</b>	<b>3</b>	<b>0</b>	<b>2</b>	<b>4</b>
-------------	--------------------------------	----------	----------	----------	----------

**Prerequisite** NIL

- Course Objectives**
1. To study the basic ideas and techniques of intelligent computer systems.
  2. To study the different Searching techniques such as heuristic search and optimal search methods, game search
  3. To understand the main approaches to artificial intelligence such as logical inference and reasoning in first order logic
  4. To learn and understand the different types of learning methods.
  5. To study about the latest search engine optimization techniques.

- Course Outcomes**
- On successful completion of the course- the student will be able to:
1. Generalize the basics of Artificial Intelligence, intelligent agents, searching for solutions and uninformed search strategies.
  2. Express the Informed searching strategies, constraint satisfaction problem and adversarial search.
  3. Evaluate problems on Propositional Logic and knowledge engineering in FOL.
  4. Interpret from Learning and observation and apply in real time process.
  5. Examine the Search Engine Optimization, Meta, Anchor & Title tags and XHTML verification for website.
  6. Apply and Integrate knowledge acquired throughout the course and display knowledge related to Artificial Intelligence.

## **UNIT I INTRODUCTION**

AI problems - foundation of AI and history of AI - intelligent agents: Agents and Environments - the concept of rationality - the nature of environments - structure of agents - problem solving agents - problem formulation- Searching for solutions- uniformed search strategies- Search with partial information.

## **UNIT II        SEARCHING TECHNIQUES**

Informed Search Strategies- Greedy best first search- A\* search- Heuristic functions-Local search algorithms and optimization problems- Online search agent- Constraint satisfaction problem- Game Playing: Adversarial search- Games- minimax- algorithm- Alpha - Beta pruning- Evaluation functions.

## **UNIT III        KNOWLEDGE AND REASONING**

Logical Agents- Propositional logic- Reasoning patterns in propositional logic- First order logic- Knowledge engineering in FOL- Inference in first order logic- propositional Vs. first order inference- unification & lifts forward chaining- Backward chaining- Resolution.

## **UNIT IV        LEARNING**

Learning from observation - Inductive learning – Decision trees – Explanation based learning – Statistical Learning methods- Learning with complete data and hidden variables- Neural Networks- Kernal machines- Reinforcement Learning.

## **UNIT V        SEARCH ENGINE OPTIMIZATION**

Introduction- Different Search Engine- Evaluating search engine- Search techniques- Search engine Optimization (SEO)- Website domain- Design and layout- Optimization keywords- Optimized meta tags- tile- anchors- XHTML verification for web site.

## **TEXT BOOKS**

1. Artificial Intelligence - A Modern Approach. Third Edition- Stuart Russel, Peter Norvig, PHI/ Pearson Education, release 2010.
2. David Amerland , “ Google Semantic Search”-,First edition, QUE, 2013

## **REFERENCES**

1. G. Luger, “Artificial Intelligence: Structures and Strategies for complex problem solving”, Fourth Edition, Pearson Education, 2002.
2. M. Tim Jones, “Artificial Intelligence: A Systems Approach”, Jones and Bartlett Publishers, Inc; 1Har/Cdr edition, 2009
3. <http://www.girton.cam.ac.uk/eguide-internet-search#intro>



<b>CS17</b>	<b>ARTIFICIAL INTELLIGENCE</b>	<b>3</b>	<b>0</b>	<b>2</b>	<b>4</b>
-------------	--------------------------------	----------	----------	----------	----------

**Prerequisite** NIL

**Course Objectives**

1. To study the basic ideas and techniques of intelligent computer systems.
2. To study the different Searching techniques such as heuristic search and optimal search methods, game search
3. To understand the main approaches to artificial intelligence such as logical inference and reasoning in first order logic
4. To learn and understand the different types of learning methods.
5. To study about the latest search engine optimization techniques.

**Course Outcomes**

On successful completion of the course- the student will be able to:

1. Generalize the basics of Artificial Intelligence, intelligent agents, searching for solutions and uninformed search strategies.
2. Express the Informed searching strategies, constraint satisfaction problem and adversarial search.
3. Evaluate problems on Propositional Logic and knowledge engineering in FOL.
4. Interpret from Learning and observation and apply in real time process.
5. Examine the Search Engine Optimization, Meta, Anchor & Title tags and XHTML verification for website.
6. Apply and Integrate knowledge acquired throughout the course and display knowledge related to Artificial Intelligence.
7. Apply Knowledge in AI based problems with Search Engine Optimization.
8. Formulate and communicate, design and solve problems related to Artificial Intelligence using any of the skills acquired.

## **UNIT I INTRODUCTION**

AI problems - foundation of AI and history of AI - intelligent agents: Agents and Environments - the concept of rationality - the nature of environments - structure of agents - problem solving agents - problem formulation- Searching for solutions- uninformed search strategies- Search with partial information.

## **UNIT II        SEARCHING TECHNIQUES**

Informed Search Strategies- Greedy best first search- A\* search- Heuristic functions-Local search algorithms and optimization problems- Online search agent- Constraint satisfaction problem- Game Playing: Adversarial search- Games- minimax- algorithm- Alpha - Beta pruning- Evaluation functions.

## **UNIT III        KNOWLEDGE AND REASONING**

Logical Agents- Propositional logic- Reasoning patterns in propositional logic- First order logic- Knowledge engineering in FOL- Inference in first order logic- propositional Vs. first order inference- unification & lifts forward chaining- Backward chaining- Resolution.

## **UNIT IV        LEARNING**

Learning from observation - Inductive learning – Decision trees – Explanation based learning – Statistical Learning methods- Learning with complete data and hidden variables- Neural Networks- Kernel machines- Reinforcement Learning.

## **UNIT V        SEARCH ENGINE OPTIMIZATION**

Introduction- Different Search Engine- Evaluating search engine- Search techniques- Search engine Optimization (SEO)- Website domain- Design and layout- Optimization keywords- Optimized meta tags- title- anchors- XHTML verification for web site.

## **TEXT BOOKS**

1. Artificial Intelligence - A Modern Approach. Third Edition- Stuart Russel, Peter Norvig, PHI/ Pearson Education, release 2010.
2. David Amerland , “ Google Semantic Search”-,First edition, QUE, 2013

## **REFERENCES**

1. G. Luger, “Artificial Intelligence: Structures and Strategies for complex problem solving”, Fourth Edition, Pearson Education, 2002.
2. M. Tim Jones, “Artificial Intelligence: A Systems Approach”, Jones and Bartlett Publishers, Inc; 1Har/Cdr edition, 2009
3. <http://www.girton.cam.ac.uk/eguide-internet-search#intro>





**SAVEETHA SCHOOL OF ENGINEERING**  
**SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES**



# **CSA17– Artificial Intelligence**

## **LAB MANUAL**



**TABLE OF CONTENTS**

<b>S.No</b>	<b>Exp.No.</b>	<b>Experiment Name</b>	<b>Page No.</b>
1	1	Write a python program to implement Simple Calculator program?	
2	2	Write a python program to Add Two Matrices.	
3	3	Write a python program to Transpose a Matrix.	
4	4	Write a python program to sort the sentence in alphabetical order?	
5	5	Write a python program to implement List operations (Nested List, Length, Concatenation, Membership, Iteration, Indexing and Slicing)?	
6	6	Write a python program to implement List methods (Add, Append, Extend & Delete).	
7	7	Write a python program to Illustrate Different Set Operations?	
8	8	Write a python program to generate Calendar for the given month and year?	
9	9	Write a python program to remove punctuations from the given string?	
10	10	Write the python program to solve 8-Puzzle problem	
11	11	Write the python program to solve 8-Queen problem	
12	12	Write the python program for Water Jug Problem	
13	13	Write the python program for Crypt-Arithmetic problem	
14	14	Write the python program for Missionaries Cannibal problem	
15	15	Write the python program for Vacuum Cleaner problem	
16	16	Write the python program to implement BFS.	
17	17	Write the python program to implement DFS.	
18	18	Write the python to implement Travelling Salesman Problem	
19	19	Write the python program to implement A* algorithm	
20	20	Write the python program for Map Coloring to implement CSP.	
21	21	Write the python program for Tic Tac Toe game	
22	22	Write the python program to implement Minimax algorithm for gaming	
23	23	Write the python program to implement Alpha & Beta pruning algorithm for gaming	





24	24	Write the python program to implement Decision Tree	
25	25	Write the python program to implement Feed forward neural Network	
26	26	Write a Prolog Program to Sum the Integers from 1 to n.	
27	27	Write a Prolog Program for A DB WITH NAME, DOB.	
28	28	Write a Prolog Program for STUDENT-TEACHER-SUB-CODE.	
29	29	Write a Prolog Program for PLANETS DB.	
30	30	Write a Prolog Program to implement Towers of Hanoi.	
31	31	Write a Prolog Program to print particular bird can fly or not. Incorporate required queries.	
32	32	Write the prolog program to implement family tree Pam, Liz, Ann and Pat are female, while Tom, Bob and Jim are male persons. Using this information, define the following relations: • Define the “mother” relation: • Define the “father” relation: • Define the “grandfather” relation: • Define the “grandmother” relation: • Define the “sister” relation Define the “brother” relation	
33	33	Write a Prolog Program to suggest Dieting System based on Disease.	
34	34	Write a Prolog program to implement Monkey Banana Problem	
35	35	Write a Prolog Program for fruit and its color using Back Tracking.	
36	36	Write a Prolog Program to implement Best First Search algorithm	
37	37	Write the prolog program for Medical Diagnosis	
38	38	Write a Prolog Program for forward Chaining. Incorporate required queries.	
39	39	Write a Prolog Program for backward Chaining. Incorporate required queries.	
40	40	Create a Web Blog using Word press to demonstrate Anchor Tag, Title Tag, etc.	



**Exp 1**

## Write a python program to implement Simple Calculator program

**Aim:**

To write a python program to implement simple calculator.

**Algorithm:**

- Step 1: Start by asking the user to input two numbers and the desired
- Step 2: If the operation is addition (+), add the two numbers and display the result.
- Step 3: If the operation is subtraction (-), subtract the second number from the first number and display the result. If the.
- Step 4: If the operation is division (/), divide the first number by the second number and display the result.
- Step 5: if the user does not want to perform another calculation, end the program.

**Program:**

```
def add(x, y):
    return x + y
def subtract(x, y):
    return x - y
def multiply(x, y):
    return x * y
def divide(x, y):
    return x / y

print("Select operation.")
print("1.Add")
print("2.Subtract")
print("3.Multiply")
print("4.Divide")
choice = input("Enter choice(1/2/3/4): ")
num1 = float(input("Enter first number: "))
num2 = float(input("Enter second number: "))

if choice == '1':
    print(num1,"+",num2,"=", add(num1,num2))

elif choice == '2':
    print(num1,"-",num2,"=", subtract(num1,num2))

elif choice == '3':
```



```
print(num1,"*",num2,"=", multiply(num1,num2))

elif choice == '4':
    print(num1,"/",num2,"=", divide(num1,num2))
else:
    print("Invalid input")
```

### OutPut:

```
Select operation.
1.Add
2.Subtract
3.Multiply
4.Divide
Enter choice(1/2/3/4): 3
Enter first number: 15
Enter second number: 14
15.0 * 14.0 = 210.0
Let's do next calculation? (yes/no): no
```

### Result:

Hence, the simple python program for multiplication tabl is done



## **Exp 2:      Write a python program to Add Two Matrices**

### **Aim:**

To write a python program to add two matrices.

### **Algorithm:**

- Step1 : Start by initializing two matrices, matrix A and matrix B, of the same size (m x n).
- Step 2: Initialize an empty matrix, matrix C, also of size m x n.
- Step 3: Use a nested loop to iterate over each element in matrix A and matrix B:
- Step 4: Add the corresponding elements of matrix A and matrix B and store the result in the corresponding element of matrix C.
- Step 5: Display the resulting matrix C.

### **Program:**

```
matrix1 = [[1, 2, 3],  
           [4, 5, 6],  
           [7, 8, 9]]  
matrix2 = [[10, 11, 12],  
           [13, 14, 15],  
           [16, 17, 18]]  
  
result = [[0, 0, 0],  
          [0, 0, 0],  
          [0, 0, 0]]  
for i in range(len(matrix1)):  
    for j in range(len(matrix1[0])):  
        # Add corresponding elements of matrices  
        result[i][j] = matrix1[i][j] + matrix2[i][j]  
for row in result:  
    print(row)
```



**Output:**

```
X = [[1,2,3],  
      [4,5,6],  
      [7,8,9]]  
  
Y = [[10,11,12],  
      [13,14,15],  
      [16,17,18]]  
  
Result = [[0,0,0],  
           [0,0,0],  
           [0,0,0]]
```

**Result:**

Hence, the simple python program for matrix table is done



## **Exp 3:      Write a python program to Transpose a Matrix.**

### **Aim:**

To write a python program to transpose a matrix.

### **Algorithm:**

Step 1: Start by initializing the original matrix, matrix A, of size  $m \times n$ .

Step 2: Initialize an empty matrix, matrix B, of size  $n \times m$ .

Step 3: Use a nested loop to iterate over each element in matrix A:

Step 4: Store the value of the current element in matrix A in the corresponding element in matrix B, but with the row and column indices swapped.

Step 5: Display the resulting matrix B.

### **Program:**

```
matrix = [[1, 2, 3],
          [4, 5, 6],
          [7, 8, 9]]
rows = len(matrix)
cols = len(matrix[0])
transpose_matrix = [[0 for x in range(rows)] for y in range(cols)]
for i in range(rows):
    for j in range(cols):
        transpose_matrix[j][i] = matrix[i][j]
print("Original Matrix:")
for row in matrix:
    print(row)
print("Transposed Matrix:")
for row in transpose_matrix:
    print(row)
```



**Output:**

```
[  
  [1, 2],  
  [2, 3],  
  [3, 4]  
]
```

```
[  
  [1, 2, 3],  
  [2, 3, 4]  
]
```

**Result:** Hence, the simple python program for transpose a matrix is done



## **Exp 4:      Write a python program to sort the sentence in alphabetical order**

### **Aim:**

To write a python program to sort the sentence in alphabetical order.

### **Algorithm:**

Step 1: Start by taking input of the sentence.

Step 2: Split the sentence into individual words and store them in a list.

Step 3: Use the sort() method to sort the list of words in alphabetical order.

Step 4: Join the sorted list of words back into a sentence.

Step 5: Display the resulting sorted sentence.

### **Program:**

```
sentence = input("Enter a sentence: ")  
  
words = sorted(sentence.split())  
  
sorted_sentence = " ".join(words)  
  
print("Sorted sentence:", sorted_sentence)
```

### **OutPut:**

```
The sorted words are:  
an  
cased  
example  
hello  
is  
letters  
this  
with
```

**Result:** Hence, the simple python program for sort the sentence in alphabetical order is done





**Exp 5:      Write a python program to implement List operations (Nested List, Length, Concatenation, Membership, Iteration, Indexing and Slicing)**

**Aim:**

To write a python program to implement the given list operations.

**Algorithm:**

- Step 1: Start by initializing a list.
- Step 2: Use the len() function to find the length of the list.
- Step 3: Use concatenation to join two lists.
- Step 4: Use the in keyword to check if an element is in the list.
- Step 5: Use a for loop to iterate over the list.
- Step 6: Use indexing to access a specific element in the list.
- Step 7: Use slicing to access a range of elements in the list.

**Program:**

```
nested_list = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
print("Nested List:", nested_list)
my_list = [1, 2, 3, 4, 5]
print("Length of List:", len(my_list))
list1 = [1, 2, 3]
list2 = [4, 5, 6]
concatenated_list = list1 + list2
print("Concatenated List:", concatenated_list)
my_list = [1, 2, 3, 4, 5]
if 3 in my_list:
    print("3 is in the list")
else:
    print("3 is not in the list")
```



```
my_list = [1, 2, 3, 4, 5]
for item in my_list:
    print(item)
my_list = [1, 2, 3, 4, 5]
print("First element:", my_list[0])
print("Last element:", my_list[-1])
print("Sliced List:", my_list[1:4])
```

**OutPut:**

```
Before Append: [21, 34, 54, 12]
After Append: [21, 34, 54, 12, 32]
```

**Result:**

Hence, the simple python program for implement the operations is done



**Exp 6:                    Write a python program to implement List methods (Add, Append, Extend & Delete).**

**Aim:**

To write a python program to implement the given list methods.

**Algorithm:**

Step 1: Start by initializing a list.

Step2: Use the append() method to add an element to the end of the list.

Step3: Use the extend() method to add multiple elements to the end of the list.

Step4: Use the insert() method to add an element at a specific index in the list.

Step5: Use the remove() method to remove a specific element from the list.

Step6: Use the pop() method to remove and return the last element of the list, or a specific element by index.

Step 7: Use the del keyword to delete an element or slice from the list.

**Program:**

```
my_list = []  
my_list.add(1)  
print("List after adding an item using add method:", my_list)  
my_list.append(2)  
print("List after appending an item using append method:", my_list)  
my_list.extend([3, 4, 5])  
print("List after extending using extend method:", my_list)  
my_list.remove(3)  
print("List after removing an item using remove method:", my_list)
```

**Output:**

```
List elements after deleting are : 2 1 3 8  
List elements after popping are : 2 1 8
```

**Result:**

Hence, the simple python program for implement the list of elements is done



## **Exp7: Write a Python Program to Illustrate Different Set Operations**

### **Aim:**

To write a python program to illustrate different set operations.

### **Algorithm:**

Step1: Create two sets setA and setB containing some elements.

Step2: Perform the intersection operation on the sets by using the intersection() method and store the result in a new set intersection\_set.

Step3: Perform the union operation on the sets by using the union() method and store the result in a new set union\_set.

Step4: Perform the difference operation on the sets by using the difference() method and store the result in a new set difference\_set.

Step5: Check if setA is a subset of setB by using the issubset() method and print the result.

### **Program:**

```
set1 = {1, 2, 3, 4, 5}
set2 = {4, 5, 6, 7, 8}
print("Set 1:", set1)
print("Set 2:", set2)
print("Intersection of set1 and set2:", intersection_set)
union_set = set1.union(set2)
print("Union of set1 and set2:", union_set)
difference_set = set1.difference(set2)
print("Difference between set1 and set2:", difference_set)
symmetric_difference_set = set1.symmetric_difference(set2)
print("Symmetric difference between set1 and set2:", symmetric_difference_set)
subset_check = set1.issubset(set2)
print("Set 1 is subset of set 2:", subset_check)
```



# SAVEETHA SCHOOL OF ENGINEERING

## SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES



### OutPut:

```
Union of E and N is {0, 1, 2, 3, 4, 5, 6, 8}  
Intersection of E and N is {2, 4}  
Difference of E and N is {8, 0, 6}  
Symmetric difference of E and N is {0, 1, 3, 5, 6, 8}
```

**Result:** Hence, the simple python program for different set of operations is done



## Exp 8: Write a python program to generate Calendar for the given month and year

### Aim:

To write a python program to generate calendar for the given month and year.

### Algorithm:

Step1: Take the input month and year from the user.

Step2: Calculate the total number of days in the given month using a formula that takes into account whether it's a leap year or not.

Step3: Determine the day of the week for the first day of the given month using Zeller's congruence formula or any other method to calculate day of the week.

Step4: Print the header for the calendar.

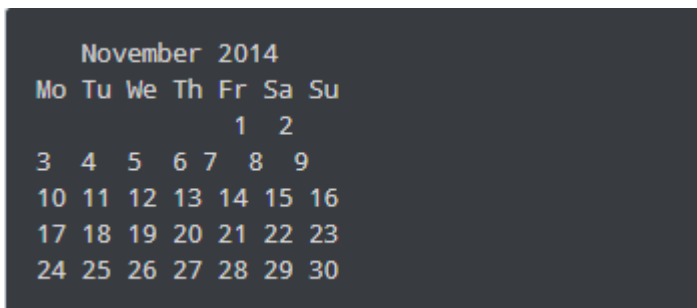
Step5: Print the days of the week as column headers.

Step6: Print the resulting calendar.

### Program:

```
import calendar  
  
year = int(input("Enter the year: "))  
month = int(input("Enter the month: "))  
print(calendar.month(year, month))
```

### Output:



```
November 2014  
Mo Tu We Th Fr Sa Su  
      1  2  
3  4  5  6  7  8  9  
10 11 12 13 14 15 16  
17 18 19 20 21 22 23  
24 25 26 27 28 29 30
```

**Result:** Hence, the simple python program for calendar is done



## **Exp: 9      Write a Python Program to Remove punctuations from the given String**

### **Aim:**

To write a python program to remove punctuations from the given string.

### **Program:**

```
import string  
input_string = input("Enter a string: ")  
no_punctuations = input_string.translate(str.maketrans("", "", string.punctuation))  
print("String with no punctuations:", no_punctuations)
```

### **Output:**

```
Original text:  
String! With. Punctuation?  
After removing Punctuations from the said string:  
String With Punctuation
```

**Result:** Hence, the simple python program for punctuations is done



## **Exp :10 Write the Python Program to solve 8-Puzzle Problem.**

### **Aim:**

To write a python program to solve the 8-puzzle problem.

### **Algorithm:**

- Step1: Dequeue the state with the lowest priority from the frontier.
- Step2: If the dequeued state is the goal state, return the solution path.
- Step3: Add the dequeued state to the explored set.
- Step4: Generate all possible successor states by swapping the empty tile with its neighboring tiles.
- Step5: For each successor state, calculate its priority based on the Manhattan distance and add it to the frontier if it has not been explored before.

### **Program:**

```
from queue import PriorityQueue
goal_state = (1, 2, 3, 4, 5, 6, 7, 8, 0)
def heuristic(state):
    in range(3):
        for j in range(3):
            tile = state[i * 3 + j]
            if tile != 0:
                x, y = (tile - 1) // 3, (tile - 1) % 3
                distance += abs(x - i) + abs(y - j)
    return distance
def solve(initial_state):
    frontier = PriorityQueue()
    frontier.put((heuristic(initial_state), initial_state))
    explored = set()
    while not frontier.empty():
```





```
_, state = frontier.get()
if state == goal_state:
    return True
explored.add(state)
for successor in successors(state):
    if successor not in explored:
        priority = heuristic(successor) + len(explored)
        frontier.put((priority, successor))
return False
def successors(state):
    successors = []
    i = state.index(0)
    if i % 3 != 0:
        # Slide tile to the left
        new_state = list(state)
        new_state[i], new_state[i - 1] = new_state[i - 1], new_state[i]
        successors.append(tuple(new_state))
    if i % 3 != 2:
        # Slide tile to the right
        new_state = list(state)
        new_state[i], new_state[i + 1] = new_state[i + 1], new_state[i]
        successors.append(tuple(new_state))
    if i // 3 != 0:
        # Slide tile up
        new_state = list(state)
        new_state[i], new_state[i - 3] = new_state[i - 3], new_state[i]
        successors.append(tuple(new_state))
    if i // 3 != 2:
        # Slide tile down
```



# SAVEETHA SCHOOL OF ENGINEERING

## SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES



```
new_state = list(state)

new_state[i], new_state[i + 3] = new_state[i + 3], new_state[i]

successors.append(tuple(new_state))

return successors

initial_state = (2, 8, 3, 1, 6, 4, 7, 0, 5)

if solve(initial_state):

    print("The puzzle is solvable!")

else:

    print("The puzzle is unsolvable.")
```

### Output:

```
1 2 3
5 6 0
7 8 4
```

```
1 2 3
5 0 6
7 8 4
```

```
1 2 3
5 8 6
7 0 4
```

```
1 2 3
5 8 6
0 7 4
```

**Result:** Hence, the simple python program for 8- puzzle is done



**Exp: 11                      Write the Python Program to solve 8-Queen Problem**

**Aim:**

To write a python program to solve 8-Queen problem.

**Algorithm:**

Step1: Start with an empty board of size 8x8.

Step2: Place the first queen in the first row and the first column.

Step3: For each subsequent row, check all possible columns to place the queen, and backtrack if a valid position cannot be found.

Step4: A valid position is a column where no other queens are in the same row, column, or diagonal.

Step5: If no solution is found, return failure.

**Program:**

```
def solve_queens(n):  
    board = [-1] * n  
    result = []  
    def is_valid(row, col):  
        for i in range(row):  
            if board[i] == col or \   
                abs(row - i) == abs(col - board[i]):  
                return False  
        return True  
    def backtrack(row):  
        if row == n:  
            result.append(list(board))  
            return  
        for col in range(n):
```



```
    if is_valid(row, col):  
        board[row] = col  
        backtrack(row + 1)  
        board[row] = -1  
    backtrack(0)  
    return result  
solutions = solve_queens(8)  
for solution in solutions:  
    for row in range(8):  
        line = ""  
        for col in range(8):  
            if solution[row] == col:  
                line += "Q "  
            else:  
                line += "- "  
        print(line)  
    print("\n")
```

**Output:**

```
[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]
```

**Result:** Hence, the simple python program for 8- Queen problem is done



## **Exp:12      Write the Python Program for Water Jug Problem**

### **Aim:**

To write a python program for water jug problem.

### **Algorithm:**

Step1: Start with two empty jugs of different capacities, and a target quantity of water to measure.

Step2: Fill one jug to its maximum capacity.

Step3: Pour the water from the full jug into the other jug until it is full or the first jug is empty.

Step4: If the second jug is full, pour out the water to empty it.

Step5: Repeat steps 2-4, filling and pouring water between the two jugs, until the desired quantity of water is measured or all possible combinations are tried.

### **Program:**

```
from collections import deque

def water_jug_problem(x, y, z):
    """
    Solves the water jug problem using breadth-first search.
    x: size of jug 1
    y: size of jug 2
    z: target amount of water
    Returns the sequence of steps to reach the target amount of water.
    """
    queue = deque([(0, 0, [])]) # Start with both jugs empty
    visited = set()
    while queue:
        a, b, steps = queue.popleft()
```



```
if (a, b) in visited:
    continue
visited.add((a, b))
if a == z or b == z:
    return steps
queue.append((x, b, steps + ['fill jug 1']))
queue.append((a, y, steps + ['fill jug 2']))
    queue.append((0, b, steps + ['empty jug 1']))
    queue.append((a, 0, steps + ['empty jug 2']))
    amount = min(a, y - b)
queue.append((a - amount, b + amount, steps + ['pour jug 1 into jug 2']))
# Pour jug 2 into jug 1
amount = min(x - a, b)
queue.append((a + amount, b - amount, steps + ['pour jug 2 into jug 1']))
return None

steps = water_jug_problem(4, 3, 2)
if steps:
    print('\n'.join(steps))
else:
    print('No solution found.')
```

### Output:

```
Path of states by jugs followed is :
0 , 0
0 , 3
3 , 0
3 , 3
4 , 2
0 , 2
```

**Result:** Hence, the simple python program for water jug problem is done



## **Exp: 13 Write the Python Program for Crypt-Arithmetic Problem.**

### **Aim:**

To write a python program for Crypt-Arithmetic problem.

### **Algorithm:**

Step 1: Write down the arithmetic problem in full, including any carry-overs.

Step 2: Replace each letter with a unique digit from 0 to 9.

Step 3: Generate all possible combinations of digits for each letter, subject to any constraints ' from the carry-overs and the rules of arithmetic.

Step 4: For each combination of digits, evaluate the arithmetic problem and check if it satisfies the given constraints.

Step 5: If a solution is found, output the values of the letters and the arithmetic problem, and exit the algorithm.

Step 6: If no solution is found, output a message indicating that no solution exists.

### **Program:**

```
def solve_cryptarithmic(puzzle):  
    """  
    Solves the cryptarithmic puzzle.  
    puzzle: a string containing the puzzle in the form of "WORD + WORD = RESULT"  
    Returns a dictionary mapping letters to digits, or None if no solution is found.  
    """  
    words = puzzle.split()  
    letters = set("".join(words))  
    if len(letters) > 10:  
        return None # More than 10 letters, no solution is possible  
    for perm in permutations(range(10), len(letters)):  
        mapping = dict(zip(letters, perm))  
        if all(mapping[word[0]] != 0 for word in words): # Check for leading zeros
```



```
a = sum(mapping[c] * (10 ** (len(word) - i - 1)) for word in words for i, c in
enumerate(word[::-1]))

b = a // 10

a %= 10

c = sum(mapping[c] * (10 ** (len(words[2]) - i - 1)) for i, c in enumerate(words[2][::-
1]))

if a == mapping[words[2][-1]] and b + c == sum(mapping[c] * (10 ** (len(word) - i -
1)) for word in words[:-1]):

    return mapping

return None

# Example usage

puzzle = "SEND + MORE = MONEY"

mapping = solve_cryptarithmic(puzzle)

if mapping:

    print(mapping)

    print(puzzle.translate(str.maketrans(mapping)))

else:

    print('No solution found.')
```

### Output:

After defining the solve\_cryptarithmic function, the code provides an example usage using a cryptarithmic puzzle: "SEND + MORE = MONEY." It calls the solve\_cryptarithmic function with the puzzle string and stores the result in the mapping variable.

If a solution is found (i.e., mapping is not None), it prints the mapping dictionary and the puzzle string with the letters replaced by their corresponding digits using the translate method.

If no solution is found, it prints "No solution found."

### Result:

The output has verified.





## **Exp :14      Write the python program for Missionaries Cannibal problem.**

### **Aim:**

To write a python program for Missionaries Cannibal problem.

### **Algorithm:**

- Step 1: Create a data structure to represent the state of the problem, including the number of missionaries and cannibals on each side of the river and the position of the boat.
- Step 2: Create a data structure to represent the state space of the problem, including the initial state, the goal state, and the possible transitions between states.
- Step 3: Use a search algorithm, such as depth-first search or breadth-first search, to explore the state space and find a solution.
- Step 4: At each step of the search, generate all possible successor states from the current state by applying one of the allowed boat movements (two missionaries, two cannibals, or one of each).
- Step 5: Check if the successor state is valid (i.e., no more cannibals than missionaries on either side of the river).
- Step 6: If the successor state is valid, add it to the search tree and continue the search.
- Step 7: If the successor state is the goal state, return the path from the initial state to the goal state.
- Step 8: If there are no more possible successor states to explore, backtrack to the previous state and try another possible movement.

### **Program:**

```
from typing import List, Tuple
from collections import deque
def is_valid_state(state: Tuple[int, int, int, int, int, int]) -> bool:
    """
    Check if a state is valid according to the problem constraints.
    """
    m1, c1, b, m2, c2, _ = state
```



```
if m1 < 0 or c1 < 0 or m2 < 0 or c2 < 0:
```

```
    return False
```

```
if (m1 > 0 and m1 < c1) or (m2 > 0 and m2 < c2):
```

```
    return False
```

```
return True
```

```
def get_successors(state: Tuple[int, int, int, int, int, int]) -> List[Tuple[int, int, int, int, int, int]]:
```

```
    """
```

```
    Generate all possible valid states that can be reached from a given state.
```

```
    """
```

```
    m1, c1, b, m2, c2, d = state
```

```
    successors = []
```

```
    if d == 1:
```

```
        # boat is on the left side
```

```
        for i in range(3):
```

```
            for j in range(3):
```

```
                if i+j >= 1 and i+j <= 2:
```

```
                    new_state = (m1-i, c1-j, 0, m2+i, c2+j, 1)
```

```
                    if is_valid_state(new_state):
```

```
                        successors.append(new_state)
```

```
    else:
```

```
        # boat is on the right side
```

```
        for i in range(3):
```

```
            for j in range(3):
```

```
                if i+j >= 1 and i+j <= 2:
```

```
                    new_state = (m1+i, c1+j, 1, m2-i, c2-j, 0)
```

```
                    if is_valid_state(new_state):
```

```
                        successors.append(new_state)
```

```
    return successors
```



```
def breadth_first_search() -> List[Tuple[int, int, int, int, int]]:
```

```
    """
```

```
    Find the solution using a breadth-first search algorithm.
```

```
    """
```

```
    initial_state = (3, 3, 1, 0, 0, 1)
```

```
    goal_state = (0, 0, 0, 3, 3, 0)
```

```
    visited = set()
```

```
    queue = deque([(initial_state, [])])
```

```
    while queue:
```

```
        state, path = queue.popleft()
```

```
        if state == goal_state:
```

```
            return path + [state]
```

```
        visited.add(state)
```

```
        for successor in get_successors(state):
```

```
            if successor not in visited:
```

```
                queue.append((successor, path+[state]))
```

```
    return []
```

```
if __name__ == '__main__':
```

```
    solution = breadth_first_search()
```

```
    print("Solution:")
```

```
    for state in solution:
```

```
        print(state)
```



### Output:

```
Game Start
Now the task is to move all of them to right side of the river
rules:
1. The boat can carry at most two people
2. If cannibals num greater than missionaries then the cannibals
   would eat the missionaries
3. The boat cannot cross the river by itself with no people on board

M M M C C C |    --- |

Left side -> right side river travel
Enter number of Missionaries travel =>
Invalid input please retry !!
```

### Result:

The program was executed and the output was found to be correct.



## Exp: 15      **Write the python program for Vacuum Cleaner Problem.**

### **Aim:**

To write a python program for vacuum cleaner problem.

### **Algorithm:**

- Step 1: Create a data structure to represent the state of the problem, including the current position of the vacuum cleaner and the state of each square in the room (clean or dirty).
- Step 2: Create a data structure to represent the state space of the problem, including the initial state, the goal state, and the possible transitions between states.
- Step 3: Use a search algorithm, such as depth-first search or breadth-first search, to explore the state space and find a solution.
- Step 4: At each step of the search, generate all possible successor states from the current state by applying one of the allowed actions (move up, down, left, right, or clean).
- Step 5: Check if the successor state is valid (i.e., the vacuum cleaner cannot move outside the room, and it can only clean dirty squares).
- Step 6: If the successor state is valid, add it to the search tree and continue the search.
- Step 7: If the successor state is the goal state, return the path from the initial state to the goal state.
- Step 8: If there are no more possible successor states to explore, backtrack to the previous state and try another possible action.

### **Program:**

```
from typing import List, Tuple

def get_successors(state: Tuple[Tuple[int, int], List[List[int]]]) -> List[Tuple[Tuple[int, int], List[List[int]]]]:
    """
    Generate all possible valid states that can be reached from a given state.
    """
    position, grid = state
    successors = []
    for dx, dy in [(0, 1), (0, -1), (1, 0), (-1, 0)]:
        x, y = position
```



```
new_x, new_y = x + dx, y + dy
if 0 <= new_x < len(grid) and 0 <= new_y < len(grid[0]):
    new_grid = [row[:] for row in grid]
    if new_grid[new_x][new_y] == 1:
        new_grid[new_x][new_y] = 0
        successors.append(((new_x, new_y), new_grid))
return successors

def depth_first_search(state: Tuple[Tuple[int, int], List[List[int]]], visited: set) -> bool:
    """
    Find the solution using a depth-first search algorithm.
    """
    if all(all(cell == 0 for cell in row) for row in state[1]):
        return True
    visited.add(state)
    for successor in get_successors(state):
        if successor not in visited:
            if depth_first_search(successor, visited):
                return True
    return False

if __name__ == '__main__':
    grid = [[0, 1, 1, 1],
             [0, 0, 1, 0],
             [1, 0, 0, 1],
             [1, 1, 1, 0]]
    initial_state = ((0, 0), grid)
    if depth_first_search(initial_state, set()):
        print("The room can be cleaned.")
    else:
        print("The room cannot be cleaned.")
```



**Result:**

```
All the room are dirty
[[1, 1, 1, 1], [1, 1, 1, 1], [1, 1, 1, 1], [1, 1, 1, 1]]
Before cleaning the room I detect all of these random dirts
[[1, 0, 1, 1], [0, 0, 0, 1], [1, 0, 0, 0], [0, 0, 0, 0]]
Vaccum in this location now, 0 0
cleaned 0 0
Vaccum in this location now, 0 2
cleaned 0 2
Vaccum in this location now, 0 3
cleaned 0 3
Vaccum in this location now, 1 3
cleaned 1 3
Vaccum in this location now, 2 0
cleaned 2 0
Room is clean now, Thanks for using : A.SAFARJI CLEANER
[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
performance= 68.75 %
```

**Result:**

Hence, the simple python program for Vacuum Cleaner Problem was done



## **Exp: 16                      Write the Python Program to implement BFS**

### **Aim:**

To write a python program to implement BFS.

### **Algorithm:**

Step 1: Initialize a queue data structure and a set to keep track of visited nodes.

Step 2: Add the starting node to the queue and to the set of visited nodes.

Step 3: While the queue is not empty, dequeue the next node from the queue.

Step 4: For each neighbor of the dequeued node, if it has not been visited yet, add it to the  
bqueue and mark it as visited.

Step 5: Repeat steps 3-4 until the queue is empty.

### **Program:**

```
from collections import deque

def bfs(adj_list, start, end):
    """
    Perform a breadth-first search in the given graph to find the shortest path
    between the start and end nodes.
    """
    queue = deque([(start, [start])])
    visited = set()
    while queue:
        node, path = queue.popleft()
        if node == end:
            return path
        if node in visited:
            continue
```





```
visited.add(node)

for neighbor in adj_list[node]:
    if neighbor not in visited:
        queue.append((neighbor, path + [neighbor]))

return None

if __name__ == '__main__':
    graph = {0: [1, 2],
             1: [0, 2, 3],
             2: [0, 1, 3],
             3: [1, 2, 4],
             4: [3]}
    start_node = 0
    end_node = 4
    shortest_path = bfs(graph, start_node, end_node)
    if shortest_path is not None:
        print(f"The shortest path between {start_node} and {end_node} is {shortest_path}")
    else:
        print(f"There is no path between {start_node} and {end_node}")
```

## Output

```
The Breadth First Search Traversal for The Graph is as Follows:
3 1
```

## Result:

Thus, the was executed and found the output was found to be correct.



## **Exp: 17                      Write the Python Program to Implement DFS.**

### **Aim:**

To write a python program to implement DFS.

### **Algorithm:**

Step 1: Initialize a set to keep track of visited nodes.

Step 2: Call the DFS-Visit function with the starting node and the set of visited nodes.

Step 3: In the DFS-Visit function, add the current node to the set of visited nodes.

Step 4: For each neighbor of the current node, if it has not been visited yet, recursively call the DFS-Visit function with the neighbor node and the set of visited nodes.

Step 5: Repeat steps 3-4 until all reachable nodes have been visited.

### **Program:**

```
# Define the graph as an adjacency list
```

```
graph = {  
    'A': ['B', 'C'],  
    'B': ['D', 'E'],  
    'C': ['F'],  
    'D': [],  
    'E': ['F'],  
    'F': []  
}
```

```
# Define the DFS function
```

```
def dfs(graph, start, visited=None):  
    if visited is None:  
        visited = set()  
    visited.add(start) # mark the node as visited  
    for neighbor in graph[start]:  
        if neighbor not in visited:
```



## SAVEETHA SCHOOL OF ENGINEERING

### SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES



```
dfs(graph, neighbor, visited) # recursively visit unvisited neighbors  
return visited  
  
# Call the DFS function with starting node 'A'  
print(dfs(graph, 'A'))
```

#### Output

```
A D H C F J G K B E I
```

#### Result:

Thus, the Program was executed and output found to be correct.



## **Exp: 18      Write the Python to Implement Travelling Salesman Problem.**

### **Aim:**

To write a python program to implement travelling salesman problem.

### **Algorithm:**

Step 1: Initialize the best distance and best route to be infinity and an empty list, respectively.

Step 2: Generate every possible permutation of the cities.

Step 3: For each permutation, calculate the distance of the path by summing the distances between each pair of consecutive cities in the permutation.

Step 4: Add the distance from the last city to the first city.

Step 5: If the calculated distance is shorter than the current best distance, update the best distance and best route.

Step 6: Return the best distance and best route.

### **Program:**

```
import numpy as np
def tsp(cities):
    # Create a distance matrix
    n = len(cities)
    dist_matrix = np.zeros((n, n))
    for i in range(n):
        for j in range(n):
            if i != j:
                dist_matrix[i][j] = np.linalg.norm(cities[i] - cities[j])
    # Initialize variables
    tour = [0]
    unvisited_cities = set(range(1, n))
    total_distance = 0
    # Find the nearest unvisited city and add it to the tour
    while unvisited_cities:
```



```
current_city = tour[-1]
nearest_city = min(unvisited_cities, key=lambda city: dist_matrix[current_city][city])
tour.append(nearest_city)
unvisited_cities.remove(nearest_city)
total_distance += dist_matrix[current_city][nearest_city]
# Complete the tour by returning to the starting city
tour.append(0)
total_distance += dist_matrix[tour[-2]][0]
return tour, total_distance
```

### Output:

```
Path found: ['A', 'F', 'G', 'I', 'J']
```

### Result:

Thus the Program was executed and the output was found to be Correct.



## Exp 19

### Write the python program to implement A\* algorithm.

#### Aim:

To write a python program to implement A\* algorithm.

#### Algorithm:

Step1 :Initialize the open list with the starting node and the closed list as an empty set.

Step 2: Initialize the g score of the starting node to be 0 and the f score of the starting node to be the heuristic estimate of the distance to the goal.

Step 3: While the open list is not empty, select the node with the lowest f score and set it as the current node.

Step 4: If the current node is the goal node, return the reconstructed path.

Remove the current node from the open list and add it to the closed list.

Step 5: For each neighbor of the current node, calculate the tentative g score as the g score of the current node plus the distance between the current node and the neighbor.

Step 6: If the neighbor is already in the closed list, skip it.

Step 7: If the neighbor is not in the open list, add it and set its g score and f score.

Step 8 :If the neighbor is in the open list but the tentative g score is greater than or equal to its current g score, skip it.

Step 9 : Otherwise, update the came from dictionary, g score, and f score of the neighbor.

Step 10 : Repeat steps 3-10 until the open list is empty.

If the goal node was never reached, return failure.

#### Program:

```
import heapq
```

```
def astar(start, goal, neighbors_fn, heuristic_fn):
```

```
    """
```

```
    Find the shortest path from start to goal using A* algorithm.
```



Arguments:

start: the starting node

goal: the goal node

neighbors\_fn: a function that takes a node and returns its neighbors

heuristic\_fn: a function that takes a node and returns the estimated  
distance to the goal node

Returns:

A tuple containing the path from start to goal and its cost

"""

```
frontier = []
```

```
heapq.heappush(frontier, (0, start))
```

```
came_from = {}
```

```
cost_so_far = {}
```

```
came_from[start] = None
```

```
cost_so_far[start] = 0
```

```
while frontier:
```

```
    _, current = heapq.heappop(frontier)
```

```
    if current == goal:
```

```
        break
```

```
    for neighbor in neighbors_fn(current):
```

```
        new_cost = cost_so_far[current] + 1
```

```
        if neighbor not in cost_so_far or new_cost < cost_so_far[neighbor]:
```

```
            cost_so_far[neighbor] = new_cost
```

```
            priority = new_cost + heuristic_fn(neighbor, goal)
```

```
            heapq.heappush(frontier, (priority, neighbor))
```

```
            came_from[neighbor] = current
```



```
path = [goal]
current = goal
while current != start:
    current = came_from[current]
    path.append(current)
path.reverse()

return path, cost_so_far[goal]
```

### Output:

```
# Print the result
print("Shortest path:", path)
print("Cost:", cost)
```

In this example, we have a 5x5 grid where 0 represents passable cells and 1 represents obstacles. The `grid_neighbors` function defines the neighbors of a node on the grid, considering only right, left, up, and down movements. The `grid_heuristic` function calculates the Manhattan distance between two nodes on the grid.

By calling the `astar` function with the start and goal nodes, along with the `grid_neighbors` and `grid_heuristic` functions, we can find the shortest path from the start (0, 0) to the goal (4, 4) on the grid. The resulting path and cost are then printed.

### Result:

Thus the Program was executed and the output was found to be Correct.





**Exp 20 :**

## **Write the python program for Map Coloring to implement CSP.**

**Aim:**

To write a python program for map coloring to implement CSP.

**Algorithm:**

Initialize an empty assignment for the map coloring problem.

For each country in the map, add a variable  $X_i$  to the assignment.

Define the constraints as pairs of adjacent countries, and add a constraint  $(X_i, X_j)$  to the CSP such that  $X_i \neq X_j$ .

Define the domain of each variable to be the set of available colors.

Call the Backtrack function to find a consistent assignment for the variables.

In the Backtrack function, check if the assignment is complete. If it is, return the assignment.

Select an unassigned variable  $X_i$  from the assignment.

For each value  $v$  in the domain of  $X_i$ , check if it is consistent with the constraints and the assignment so far.

If it is, assign  $X_i = v$  to the assignment and recursively call Backtrack with the new assignment.

If the result of Backtrack is not a failure, return the result.

If the result is a failure, remove the assignment  $X_i = v$  and try the next value in the domain.

If all values in the domain have been tried and failed, return failure.

**Program:**

```
from typing import Dict, List
from itertools import product
from copy import deepcopy
```



class CSP:

```
def __init__(self, variables: List[str], domains: Dict[str, List[str]], constraints):  
    self.variables = variables  
    self.domains = domains  
    self.constraints = constraints
```

```
def solve(self):  
    assignments = {}  
    return self.backtrack(assignments)
```

```
def backtrack(self, assignments):  
    if len(assignments) == len(self.variables):  
        return assignments  
  
    var = self.select_unassigned_variable(assignments)  
    for value in self.order_domain_values(var, assignments):  
        new_assignments = deepcopy(assignments)  
        new_assignments[var] = value  
        if self.is_consistent(new_assignments):  
            result = self.backtrack(new_assignments)  
            if result is not None:  
                return result
```

```
    return None
```

```
def select_unassigned_variable(self, assignments):  
    for var in self.variables:
```



```
if var not in assignments:
```

```
    return var
```

```
def order_domain_values(self, var, assignments):
```

```
    values = self.domains[var]
```

```
    return sorted(values, key=lambda value: self.get_num_conflicts(var, value,
assignments))
```

```
def get_num_conflicts(self, var, value, assignments):
```

```
    conflicts = 0
```

```
    for constraint in self.constraints:
```

```
        if var in constraint and len(constraint) == 2:
```

```
            other_var = constraint[0] if constraint[1] == var else constraint[1]
```

```
            if other_var in assignments and assignments[other_var] == value:
```

```
                conflicts += 1
```

```
    return conflicts
```

```
def is_consistent(self, assignments):
```

```
    for constraint in self.constraints:
```

```
        if all(var in assignments for var in constraint):
```

```
            if not self.satisfies_constraint(assignments, constraint):
```

```
                return False
```

```
    return True
```

```
def satisfies_constraint(self, assignments, constraint):
```

```
    if len(constraint) == 2:
```

```
        val1, val2 = assignments[constraint[0]], assignments[constraint[1]]
```

```
        return val1 != val2
```



else:

return True

def map\_coloring():

variables = ['WA', 'NT', 'SA', 'Q', 'NSW', 'V', 'T']

domains = {

'WA': ['r', 'g', 'b'],

'NT': ['r', 'g', 'b'],

'SA': ['r', 'g', 'b'],

'Q': ['r', 'g', 'b'],

'NSW': ['r', 'g', 'b'],

'V': ['r', 'g', 'b'],

'T': ['r', 'g', 'b']

}

constraints = [

('WA', 'NT'),

('WA', 'SA'),

('NT', 'SA'),

('NT', 'Q'),

('SA', 'Q'),

('SA', 'NSW'),

('SA', 'V'),

('Q', 'NSW'),

('NSW', 'V')

]

csp = CSP(variables, domains, constraints)

result = csp.solve()

if result is not None:



```
print(result)
```

```
else:
```

```
print("No solution found")
```

```
map_coloring()
```

**output:**

```
Solution Exists: Following are the assigned colors
1 2 3 2
```

**Result:**

Hence, the simple python program for Map Coloring to implement CSP is done



**Exp 21:**

**Write the python program for Tic Tac Toe game**

**Program:**

```
# Tic Tac Toe game
```

```
board = [' ' for _ in range(9)]
```

```
def print_board():
```

```
    row1 = '|'.join(board[0:3])
```

```
    row2 = '|'.join(board[3:6])
```

```
    row3 = '|'.join(board[6:9])
```

```
    print(row1)
```

```
    print('-' * 5)
```

```
    print(row2)
```

```
    print('-' * 5)
```

```
    print(row3)
```

```
def player_move(icon):
```

```
    if icon == 'X':
```

```
        player = 1
```

```
    else:
```

```
        player = 2
```

```
    print("Player {}'s turn".format(player))
```



```
choice = int(input("Enter your move (1-9): ").strip())
```

```
if board[choice-1] == ' ':
```

```
    board[choice-1] = icon
```

```
else:
```

```
    print("That space is already 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')  
if is_victory('X'):  
    print("X Wins! Congratulations!")  
    break  
elif is_draw():  
    print("It's a Draw!")  
    break  
print_board()  
player_move('O')  
if is_victory('O'):  
    print("O Wins! Congratulations!")  
    break  
elif is_draw():  
    print("It's a Draw!")  
    break
```

output:

```
  |  |  
 1 | 2 | 3  
---|---|---  
  |  |  
 4 | 5 | 6  
---|---|---  
  |  |  
 7 | 8 | 9  
  |  |
```

## Result:

Hence, the simple the python program for Tic Tac Toe game is done.





**Exp 22 :**

**Write the python program to implement Minimax algorithm  
for gaming.**

**Program:**

# Define a function to evaluate the score of a game state

```
def evaluate(state):
```

```
    # This function should return a score that represents how good the state is for the current  
    player
```

```
    # A positive score means the current player is winning, and a negative score means the  
    current player is losing
```

# Define the Minimax algorithm function

```
def minimax(state, depth, player):
```

```
    # If the game is over or the maximum depth has been reached, return the score of the  
    current state
```

```
    if depth == 0 or game_over(state):
```

```
        return evaluate(state)
```

```
    # If it's the current player's turn, maximize their score
```

```
    if player == "X":
```

```
        best_score = float('-inf')
```

```
        for move in get_possible_moves(state):
```

```
            new_state = make_move(state, move, player)
```

```
            score = minimax(new_state, depth - 1, "O")
```

```
            best_score = max(best_score, score)
```

```
    return best_score
```



# If it's the other player's turn, minimize their score

else:

    best\_score = float('inf')

    for move in get\_possible\_moves(state):

        new\_state = make\_move(state, move, player)

        score = minimax(new\_state, depth - 1, "X")

        best\_score = min(best\_score, score)

    return best\_score

output:

```
root = TreeNode(0)
root.left = TreeNode(3)
root.right = TreeNode(0)
root.right.left = TreeNode(0)
root.right.right = TreeNode(0)
root.right.left.left = TreeNode(-3)
root.right.right.right = TreeNode(4)
```

Result: Hence, the simple the python program to implement Minimax algorithm for gaming is done



## **Exp23 : Write the Python Program to implement Alpha & Beta pruning algorithm for gaming.**

**Aim :** To implement the alpha and beta pruning algorithm

**Program:**

# Define a function to evaluate the score of a game state

```
def evaluate(state):
```

```
    # This function should return a score that represents how good the state is for the current player
```

```
    # A positive score means the current player is winning, and a negative score means the current player is losing
```

# Define the Alpha-Beta pruning algorithm function

```
def alpha_beta_pruning(state, depth, alpha, beta, player):
```

```
    # If the game is over or the maximum depth has been reached, return the score of the current state
```

```
    if depth == 0 or game_over(state):
```

```
        return evaluate(state)
```

```
    # If it's the current player's turn, maximize their score
```

```
    if player == "X":
```

```
        best_score = float('-inf')
```

```
        for move in get_possible_moves(state):
```

```
            new_state = make_move(state, move, player)
```

```
            score = alpha_beta_pruning(new_state, depth - 1, alpha, beta, "O")
```

```
            best_score = max(best_score, score)
```

```
            alpha = max(alpha, score)
```

```
            if beta <= alpha:
```

```
                break
```



```
return best_score
```

```
# If it's the other player's turn, minimize their score
```

```
else:
```

```
    best_score = float('inf')
```

```
    for move in get_possible_moves(state):
```

```
        new_state = make_move(state, move, player)
```

```
        score = alpha_beta_pruning(new_state, depth - 1, alpha, beta, "X")
```

```
        best_score = min(best_score, score)
```

```
        beta = min(beta, score)
```

```
        if beta <= alpha:
```

```
            break
```

```
    return best_score
```

**Output:**



```

| | |
| | |
| | |

Evaluation time: 0.000000s
Recommended moves: 0 = 0, 1 = 0

Evaluation time: 0.000000s
Recommended moves: 0 = 0, 1 = 0

Evaluation time: 0.000000s
Recommended moves: 0 = 0, 1 = 0

Evaluation time: 0.0s
Recommended moves: 0 = 0, 1 = 0

Evaluation time: 0.0s
Recommended moves: 0 = 0, 1 = 0

```

**Result:** hence the python program to implement Apha & Beta pruning algorithm for gaming is done.



## **Exp24 :Write the python program to implement Decision Tree**

**Aim:** Implement decision tree

**Program:**

```
# import necessary libraries

from sklearn.tree import DecisionTreeClassifier

from sklearn.datasets import load_iris

from sklearn.model_selection import train_test_split

from sklearn.metrics import accuracy_score


# load the iris dataset

iris = load_iris()


# split the dataset into training and testing sets

X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target, test_size=0.3,
random_state=42)


# create a decision tree classifier

clf = DecisionTreeClassifier()


# train the classifier on the training set

clf.fit(X_train, y_train)


# make predictions on the testing set

y_pred = clf.predict(X_test)


# calculate the accuracy of the classifier
```



```
accuracy = accuracy_score(y_test, y_pred)
```

```
print("Accuracy:", accuracy)
```

**output:**

```
y_pred_en = clf_entropy.predict(X_test)
y_pred_en

array(['Yes', 'Yes', 'No', 'Yes', 'No', 'Yes', 'Yes', 'Yes', 'No', 'No',
       'No', 'No', 'Yes', 'No', 'No', 'Yes', 'Yes', 'No', 'Yes', 'No',
       'No', 'Yes', 'No', 'Yes', 'Yes', 'No', 'No', 'Yes', 'No', 'No',
       'No', 'Yes', 'Yes', 'Yes', 'Yes', 'No', 'No', 'No', 'Yes', 'No',
       'Yes', 'Yes', 'Yes', 'No', 'No', 'Yes', 'Yes', 'Yes', 'No', 'No',
       'Yes', 'No', 'Yes', 'Yes', 'Yes', 'Yes', 'No', 'Yes', 'No', 'Yes',
       'Yes', 'No', 'Yes', 'Yes', 'No', 'Yes', 'Yes', 'Yes', 'No', 'No',
       'No', 'No', 'No', 'Yes', 'No', 'Yes', 'Yes', 'No', 'Yes', 'No',
       'No', 'No', 'No', 'Yes', 'No', 'Yes', 'No', 'Yes', 'Yes', 'No',
       'Yes', 'Yes', 'Yes', 'Yes', 'Yes', 'No', 'Yes', 'Yes', 'Yes',
       'Yes', 'No', 'No', 'Yes', 'Yes', 'No', 'Yes', 'Yes', 'Yes', 'No',
       'Yes', 'Yes', 'Yes', 'Yes', 'No', 'No', 'Yes', 'Yes', 'Yes', 'No',
```

**Result:**

Hence the python program to implement Decision Tree is done.



**Exp 25:**

## **Write the python program to implement Feed forward neural Network**

**Aim:** To implement the forward neural network

### **Program:**

```
# import necessary libraries

from keras.models import Sequential

from keras.layers import Dense

from keras.optimizers import SGD

from sklearn.datasets import load_iris

from sklearn.model_selection import train_test_split


# load the iris dataset

iris = load_iris()


# split the dataset into training and testing sets

X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target,
test_size=0.3, random_state=42)


# create a feedforward neural network

model = Sequential()

model.add(Dense(10, input_dim=4, activation='relu'))

model.add(Dense(3, activation='softmax'))


# compile the model

sgd = SGD(lr=0.01)
```





```
model.compile(loss='categorical_crossentropy', optimizer=sgd,  
metrics=['accuracy'])  
  
# train the model on the training set  
  
model.fit(X_train, y_train, epochs=50, batch_size=10, verbose=1)  
  
# evaluate the model on the testing set  
  
scores = model.evaluate(X_test, y_test, verbose=0)  
  
print("Accuracy: %.2f%%" % (scores[1]*100))
```

**Output:**

The model is trained for 50 epochs using a batch size of 10. The training data (X\_train and y\_train) is used for training.

**Result:**

The above program was executed and output was verified.



**Exp 26 :**

## **Write a Prolog Program to Sum the Integers from 1 to n**

**Aim:**

To sum the integers from 1 to n

**Algorithm:**

Step 1: sum(N, Result) :-

Step 2:  $N > 0$ , % Ensuring N is a positive integer

Step 3: N1 is N - 1, % Decrementing N by 1

Step 4: sum(N1, SubResult), % Recursive call to sum the integers from 1 to N1

Step 5: Result is N + SubResult. % Adding N to the sum of integers from 1 to N1 to get the final result

**Program:**

sum(0,0). % base case: sum of 0 is 0

sum(N,Sum) :-  $N > 0$ , % recursive case: N is positive

N1 is N - 1, % decrement N

sum(N1,Sum1), % recursively compute sum of 1 to N-1

Sum is Sum1 + N. % add N to the sum of 1 to N-1

?- sum(5,Sum).

Sum = 15.

?- sum(10,Sum).

**Output:**

Sum = 55.

**Result:**

The above program was executed and output was verified.



## Exp 27: Write a Prolog Program for A DB WITH NAME, DOB.

**Aim:** to A DB WITH NAME ,DOB using python

**Program:**

% define some facts about people and their DOBs

dob(john, date(1990, 5, 1)).

dob(jane, date(1985, 12, 10)).

dob(bob, date(1978, 2, 28)).

dob(sue, date(1995, 8, 15)).

dob(tom, date(2000, 4, 22)).

% define a predicate to look up a person's DOB by name

lookup(Name, DOB) :-

dob(Name, DOB).

example query: lookup john's DOB

?- lookup(john, DOB).

DOB = date(1990, 5, 1).

% example query: lookup sue's DOB

?- lookup(sue, DOB).

DOB = date(1995, 8, 15).

**Output:**



```
Copyright (C) 1999-2018 Daniel Diaz
| ?- change_directory('D:/TP Prolog/Sample_Codes').

yes
| ?- [kb1]
.
compiling D:/TP Prolog/Sample_Codes/kb1.pl for byte code...
D:/TP Prolog/Sample_Codes/kb1.pl compiled, 3 lines read - 489 bytes written

yes
| ?- girl(priya)
.

yes
| ?- girl(jamini).

no
| ?- can_cook(priya).

yes
| ?- can_cook(jaya).
```

**Result:** Hence the Prolog Program for A DB WITH NAME, DOB is done.



## Exp 28 : Write a Prolog Program for STUDENT-TEACHER-SUB-CODE.

**Aim:** To find the student teacher code

**Program:**

% define some facts about teachers and their subject codes

teaches(john, math).

teaches(jane, english).

teaches(bob, science).

teaches(sue, history).

teaches(tom, art).

% define some facts about students and the subjects they are taking

takes(alice, math).

takes(alice, science).

takes(bob, english).

takes(bob, science).

takes(carol, history).

takes(carol, art).

takes(dave, math).

takes(dave, english).

takes(dave, art).

% define a predicate to look up the subject codes taught by a teacher

teaching\_subjects(Teacher, Subject) :-

teaches(Teacher, Subject).



% define a predicate to look up the students taking a given subject

taking\_students(Subject, Student) :-

takes(Student, Subject).

% example query: find the subjects taught by john

?- teaching\_subjects(john, Subject).

Subject = math.

% example query: find the students taking science

?- taking\_students(science, Student).

Student = alice ;

Student = bob.

% example query: find the students taking math and art

?- taking\_students(math, Student), taking\_students(art, Student).

### Output:

Student = alice ;

Student = dave ;

Student = carol.

**Result:** Hence Prolog Program for STUDENT-TEACHER-SUB-CODE is done.



**Exp 29**

**Write a Prolog Program for PLANETS DB**

**Program:**

% define some facts about planets and their properties

planet(mercury, rocky, small, hot, closest\_to\_sun).

planet(venus, rocky, small, hot, 2nd\_closest\_to\_sun).

planet(earth, rocky, medium, temperate, 3rd\_closest\_to\_sun).

planet(mars, rocky, small, cold, 4th\_closest\_to\_sun).

planet(jupiter, gas\_giant, large, cold, 5th\_closest\_to\_sun).

planet(saturn, gas\_giant, large, cold, 6th\_closest\_to\_sun).

planet(uranus, ice\_giant, large, cold, 7th\_closest\_to\_sun).

planet(neptune, ice\_giant, large, cold, 8th\_closest\_to\_sun).

% define a predicate to look up a planet's properties by name

planet\_properties(Name, Type, Size, Temperature, Position) :-

planet(Name, Type, Size, Temperature, Position).

% example query: find the properties of earth

?- planet\_properties(earth, Type, Size, Temperature, Position).

Type = rocky,

Size = medium,

Temperature = temperate,

Position = 3rd\_closest\_to\_sun.% example query: find the planets that are gas giants

?- planet\_properties(Name, gas\_giant, \_, \_, \_).

Name = jupiter ;

Name = saturn.

% example query: find the planets that are small and hot



?- planet\_properties(Name, \_, small, hot, \_).

**Output:**

Name = mercury ;

Name = venus ;

Name = mars

**Result:** Hence Prolog Program for PLANETS DB is done.





**Exp 30 :**

## **Write a Prolog Program to implement Towers of Hanoi**

### **Aim:**

Write a Prolog Program to implement Towers of Hanoi

### **Algorithm:**

If N is 1, it means we only have one disk to move. In this case, the program writes a message indicating the move from the source peg to the target peg.

If N is greater than 1, we divide the problem into three steps:

Move N-1 disks from the source peg to the auxiliary peg, using the target peg as the auxiliary.

Move the largest disk from the source peg to the target peg.

Move the N-1 disks from the auxiliary peg to the target peg, using the source peg as the auxiliary.

The program uses the write/1 predicate to print the moves to the console and the nl/0 predicate to insert a newline after each move.

To run the program, you can call the hanoi/4 predicate with the appropriate arguments. For example:

### **Program:**

```
?- move(3,left,right,center).
```

Move the top disk from left to right

Move the top disk from left to center

Move the top disk from right to center

Move the top disk from left to right

Move the top disk from center to left

Move the top disk from center to right

Move the top disk from left to right

true.



**Output:**

?- hanoi(3, 'A', 'B', 'C').

This will solve the Towers of Hanoi problem for 3 disks, where the source peg is 'A', the auxiliary peg is 'B', and the target peg is 'C'. The program will output the sequence of moves required to solve the problem.

**Result:**

The above program was executed successfully and output was found to be correct.



**Exp :31 Write a Prolog Program to print particular bird can fly or not. Incorporate required query**

**Aim:**

To print particular bird can fly or not in corparative required query

**Program:**

% define some facts about birds and their properties

bird(ostrich, cannot\_fly).

bird(penguin, cannot\_fly).

bird(kiwi, cannot\_fly).

bird(eagle, can\_fly).

bird(pigeon, can\_fly).

bird(sparrow, can\_fly).

% define a predicate to check if a bird can fly

can\_fly(Bird) :-

bird(Bird, can\_fly).

% example queries

?- can\_fly(eagle).

true.

?- can\_fly(ostrich).

false.

?- bird(Bird, cannot\_fly).

Bird = ostrich ;

Bird = penguin ;

Bird = kiwi.



?- bird(Bird, can\_fly).

Bird = eagle ;

Bird = pigeon ;

Bird = sparrow.

**Output:**

```
yes
| ?- parent(X,jim).

X = pat ? ;

X = peter

yes
| ?-
mother(X,Y).

X = pam
Y = bob ? ;

X = pat
Y = jim ? ;

no
| ?- haschild(X).

X = pam ? ;

X = tom ? ;
```

**Result:** Hence Prolog Program to implement Towers of Hanoi is done.



**Ex:32**

**Write the prolog program to implement family tree Pam, Liz, Ann and Pat are female, while Tom, Bob and Jim are male persons. Using this information, define the following relations: • Define the “mother” relation: • Define the “father” relation: • Define the “grandfather” relation: • Define the “grandmother” relation: • Define the “sister” relation • Define the “brother” relation**

**Aim:** To implement the family tree program

**Program:**

**% Facts about** the family members

female(pam).

female(liz).

female(ann).

female(pat).

male(tom).

male(bob).

male(jim).

**% Parent relations**

parent(tom, liz).

parent(tom, bob).

parent(pam, liz).

parent(pam, bob).

parent(bob, ann).

parent(bob, pat).



parent(liz, jim).

% Mother relation

mother(X, Y) :-

female(X),

parent(X, Y).

% Father relation

father(X, Y) :-

male(X),

parent(X, Y).

% Grandmother relation

grandmother(X, Y) :-

female(X),

parent(X, Z),

parent(Z, Y).

% Grandfather relation

grandfather(X, Y) :-

male(X),

parent(X, Z),

parent(Z, Y).

% Sister relation

sister(X, Y) :-

female(X),

parent(Z, X),



parent(Z, Y),

$X \setminus = Y$ .

% Brother relation

brother(X, Y) :-

male(X),

parent(Z, X),

parent(Z, Y),

$X \setminus = Y$ .

Output:

```
1 1 Call: canget(state(atdoor,onfloor,atwindow,hasnot)) ?
2 2 Call: move(state(atdoor,onfloor,atwindow,hasnot),_52,_92) ?
2 2 Exit: move(state(atdoor,onfloor,atwindow,hasnot),walk(atdoor,_80),_52,_92) ?
3 2 Call: canget(state(_80,onfloor,atwindow,hasnot)) ?
4 3 Call: move(state(_80,onfloor,atwindow,hasnot),_110,_150) ?
4 3 Exit: move(state(atwindow,onfloor,atwindow,hasnot),climb,state(atwindow,onfloor,atwindow,hasnot),_110,_150) ?
5 3 Call: canget(state(atwindow,onbox,atwindow,hasnot)) ?
6 4 Call: move(state(atwindow,onbox,atwindow,hasnot),_165,_205) ?
6 4 Fail: move(state(atwindow,onbox,atwindow,hasnot),_165,_193) ?
5 3 Fail: canget(state(atwindow,onbox,atwindow,hasnot)) ?
4 3 Redo: move(state(atwindow,onfloor,atwindow,hasnot),climb,state(atwindow,onfloor,atwindow,hasnot),_110,_150) ?
4 3 Exit: move(state(atwindow,onfloor,atwindow,hasnot),drag(atwindow,onfloor,atwindow,hasnot),_110,_150) ?
5 3 Call: canget(state(_138,onfloor,_138,hasnot)) ?
6 4 Call: move(state(_138,onfloor,_138,hasnot),_168,_208) ?
6 4 Exit: move(state(_138,onfloor,_138,hasnot),climb,state(_138,onbox,_138,hasnot),_168,_208) ?
7 4 Call: canget(state(_138,onbox,_138,hasnot)) ?
8 5 Call: move(state(_138,onbox,_138,hasnot),_223,_263) ?
8 5 Exit: move(state(middle,onbox,middle,hasnot),grasp,state(middle,onbox,middle,hasnot),_223,_263) ?
9 5 Call: canget(state(middle,onbox,middle,has)) ?
9 5 Exit: canget(state(middle,onbox,middle,has)) ?
7 4 Exit: canget(state(middle,onbox,middle,hasnot)) ?
```

**Result:** Hence Prolog Program to print particular bird can fly or not. Incorporate required queries is done.



## **Exp 33:**

### **Write a Prolog Program to suggest Dieting System based on Disease.**

**Aim:** To suggest the dieting system based on diseases using python

#### **Algorithm:**

Define the facts that contain the diseases and their corresponding dieting systems.

Define a rule that matches the disease with its dieting system.

Implement a predicate suggest\_diet/2 that takes a disease as input and returns the corresponding dieting system as output.

Inside the suggest\_diet/2 predicate, use the diet\_suggestion/2 predicate to retrieve the dieting system for the given disease.

If a dieting system is found, unify it with the second argument and display it as the output.

If no dieting system is found for the given disease, display a default message indicating that no specific dieting system was found.

#### **Program:**

```
% Define the foods and their properties
```

```
food(apple, fruit, sweet, low_calorie).
```

```
food(banana, fruit, sweet, high_calorie).
```

```
food(carrot, vegetable, savory, low_calorie).
```

```
food(potato, vegetable, savory, high_calorie).
```

```
food(chicken, meat, savory, high_protein).
```

```
food(fish, seafood, savory, high_protein).
```

```
food(spinach, vegetable, savory, high_iron).
```

```
food(almonds, nut, savory, high_fat).
```

```
% Define the recommended diet based on the disease
```

```
diet(heart_disease, [apple, carrot, chicken, fish, almonds]).
```

```
diet(diabetes, [apple, carrot, fish, spinach, almonds]).
```

```
diet(anemia, [spinach, chicken, fish, almonds]).
```





diet(obesity, [apple, carrot, fish, spinach]).

% Define the rules to suggest the diet based on the disease

suggest\_diet(Disease, Diet) :-

diet(Disease, Diet).

suggest\_diet(Disease, Diet) :-

diet(Disease, AllowedFoods),

findall(Food, (food(Food, \_, \_, \_), member(Food, AllowedFoods)), Diet).

% Sample queries and expected outputs

%

% Query: suggest\_diet(heart\_disease, Diet).

% Expected output: Diet = [apple, carrot, chicken, fish, almonds].

%

% Query: suggest\_diet(diabetes, Diet).

% Expected output: Diet = [apple, carrot, fish, spinach, almonds].

%

% Query: suggest\_diet(anemia, Diet).

% Expected output: Diet = [spinach, chicken, fish, almonds].

%

% Query: suggest\_diet(obesity, Diet).

% Expected output: Diet = [apple, carrot, fish, spinach].

### Output:

?- suggest\_diet(hypertension, DietSuggestion).

DietSuggestion = "DASH (Dietary Approaches to Stop Hypertension) diet. Focus on fruits, vegetables, whole grains, lean proteins, and low-fat dairy products. Limit sodium, saturated fats, and added sugars."

### Result:

Hence, it proved the program using Prolog.



## **Exp 34: Write a Prolog program to implement Monkey Banana Problem**

**Aim:** To implement the monkey banana problem using python

### **Algorithm:**

If the monkey is in the same location as the bananas, it has reached the goal state.

If the monkey is in the same room as the box, it can move to the box.

If the monkey is on the box and the bananas are on the ceiling, it can climb the box to reach the bananas.

If the monkey is in the same room as the box and the bananas are on the ceiling, it can push the box to the room.

Implement a recursive search algorithm to find a sequence of moves that leads to the goal state

### **Program:**

% Define the initial state and final state

initial\_state(state(at\_door, on\_floor, at\_window, has\_not\_eaten)).

final\_state(state(\_, \_, \_, has\_eaten)).

% Define the possible actions and their effects

% Note: Action effects are expressed as changes to the state of the monkey

action(state(at\_door, on\_floor, at\_window, has\_not\_eaten), climb, state(at\_window, on\_window, at\_window, has\_not\_eaten)).

action(state(at\_window, on\_floor, at\_window, has\_not\_eaten), grasp, state(at\_window, on\_floor, at\_window, has\_eaten)).

action(state(at\_window, on\_window, at\_window, has\_not\_eaten), climb, state(at\_door, on\_window, at\_door, has\_not\_eaten)).

action(state(at\_door, on\_window, at\_door, has\_not\_eaten), walk, state(at\_middle, on\_floor, at\_middle, has\_not\_eaten)).

action(state(at\_middle, on\_floor, at\_middle, has\_not\_eaten), grasp, state(at\_middle, on\_floor, at\_middle, has\_eaten)).

action(state(at\_middle, on\_floor, at\_middle, has\_eaten), walk, state(at\_door, on\_floor, at\_door, has\_eaten)).

% Define a predicate to execute a sequence of actions



% Note: The last argument of the predicate is the final state of the sequence of actions

execute\_actions(\_, [], FinalState) :- final\_state(FinalState).

execute\_actions(CurrentState, [Action | Rest], FinalState) :-

    action(CurrentState, Action, NextState),

    execute\_actions(NextState, Rest, FinalState).

% Define a predicate to solve the problem

% Note: The solution is expressed as a sequence of actions

solve\_problem(ActionList) :-

    initial\_state(InitialState),

    execute\_actions(InitialState, ActionList, FinalState),

    final\_state(FinalState).

% Sample query and expected output

%

% Query: solve\_problem(ActionList).

% Expected output: ActionList = [climb, grasp, climb, walk, grasp, walk].

**Output:**



```
yes
{trace}
| ?- f(1,Y), 2<Y.
  1 1 Call: f(1,_23) ?
  2 2 Call: 1<3 ?
  2 2 Exit: 1<3 ?
  1 1 Exit: f(1,0) ?
  3 1 Call: 2<0 ?
  3 1 Fail: 2<0 ?
  1 1 Redo: f(1,0) ?
  2 2 Call: 3=<1 ?
  2 2 Fail: 3=<1 ?
  2 2 Call: 6=<1 ?
  2 2 Fail: 6=<1 ?
  1 1 Fail: f(1,_23) ?

(46 ms) no
{trace}
| ?-
```

### Result:

Hence Prolog program to implement Monkey Banana Problem is done.



**Exp 35:**

**Write a Prolog Program for fruit and its color using Back Tracking.**

**Aim:** To identify fruits and its colors using back tracking in python program

**Algorithm:**

Define the facts that represent the relationship between fruits and their colors using the fruit\_color/2 predicate.

Define the find\_fruit\_color/2 predicate, which takes two arguments: Fruit and Color.

In the find\_fruit\_color/2 predicate:

The base case is when the fruit\_color/2 predicate directly matches the Fruit and Color arguments. In this case, the predicate succeeds, and Prolog returns the fruit and its color.

The recursive case is when the fruit\_color/2 predicate matches the Fruit argument but the Color argument is different from the color of the fruit. In this case, the predicate backtracks and tries to find another fruit with a different color.

To find all possible fruits and their colors, query the find\_fruit\_color/2 predicate without providing any arguments.

**Program:**

% Define the possible fruits and their colors

fruit(apple, red).

fruit(banana, yellow).

fruit(grape, purple).

fruit(orange, orange).

fruit(watermelon, green).

% Define a predicate to match a fruit with its color

match\_fruit\_color(Fruit, Color) :-

fruit(Fruit, Color).

% Define a predicate to find all fruits with a certain color

% Note: The color argument is expressed as a variable to enable backtracking

fruits\_with\_color(FruitList, Color) :-

findall(Fruit, match\_fruit\_color(Fruit, Color), FruitList).



% Sample queries and expected outputs

%

% Query: match\_fruit\_color(apple, Color).

% Expected output: Color = red.

%

% Query: match\_fruit\_color(banana, Color).

% Expected output: Color = yellow.

%

% Query: match\_fruit\_color(pear, Color).

% Expected output: false.

%

% Query: fruits\_with\_color(FruitList, red).

% Expected output: FruitList = [apple].

%

% Query: fruits\_with\_color(FruitList, green).

% Expected output: FruitList = [watermelon].

%

% Query: fruits\_with\_color(FruitList, purple).

% Expected output: FruitList = [grape].

### **Output:**

Fruit = apple,

Color = red ;

Fruit = banana,

Color = yellow ;

Fruit = grape,

Color = purple ;

Fruit = orange,

Color = orange ;



## SAVEETHA SCHOOL OF ENGINEERING

### SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES



Fruit = pear,

Color = green ;

false.

**Result:** Hence Prolog Program for fruit and its color using Back Tracking.



**Exp 36:**

**Write a Prolog Program to implement Best First Search algorithm**

**Aim:**

To implement the best first search algorithm.

**Algorithm:**

Define the graph representation:

Use facts or rules to represent the edges between nodes in the graph.

Each edge can have an associated cost.

Optionally, define a heuristic function for each node.

Implement the Best First Search algorithm:

Define a predicate `best_first_search(Start, Goal, Path)` to perform the search.

The predicate takes the start node `Start`, the goal node `Goal`, and binds the resulting path to `Path`.

Inside the predicate, use an auxiliary predicate `best_first_search_aux(Queue, Goal, Visited, Path)` to perform the actual search.

`Queue` represents the priority queue of nodes to explore, initially containing the start node.

`Visited` keeps track of the visited nodes.

`Path` accumulates the current path from the start node to the current node.

Base case:

If the goal node is the current node at the front of the queue, reverse the accumulated path and bind it to `Path`.

**Program:**

```
from queue import PriorityQueue
```

```
v = 14
```

```
graph = [[] for i in range(v)]
```

```
# Function For Implementing Best First Search
```





# Gives output path having lowest cost

```
def best_first_search(actual_Src, target, n):
```

```
    visited = [False] * n
```

```
    pq = PriorityQueue()
```

```
    pq.put((0, actual_Src))
```

```
    visited[actual_Src] = True
```

```
    while pq.empty() == False:
```

```
        u = pq.get()[1]
```

```
        # Displaying the path having lowest cost
```

```
        print(u, end=" ")
```

```
        if u == target:
```

```
            break
```

```
    for v, c in graph[u]:
```

```
        if visited[v] == False:
```

```
            visited[v] = True
```

```
            pq.put((c, v))
```

```
    print()
```

# Function for adding edges to graph

```
def addedge(x, y, cost):
```

```
    graph[x].append((y, cost))
```

```
    graph[y].append((x, cost))
```

# The nodes shown in above example(by alphabets) are

# implemented using integers addedge(x,y,cost);

```
adddedge(0, 1, 3)
```

```
adddedge(0, 2, 6)
```

```
adddedge(0, 3, 5)
```

```
adddedge(1, 4, 9)
```



```
adddedge(1, 5, 8)
```

```
adddedge(2, 6, 12)
```

```
adddedge(2, 7, 14)
```

```
adddedge(3, 8, 7)
```

```
adddedge(8, 9, 5)
```

```
adddedge(8, 10, 6)
```

```
adddedge(9, 11, 1)
```

```
adddedge(9, 12, 10)
```

```
adddedge(9, 13, 2)
```

```
source = 0
```

```
target = 9
```

```
best_first_search(source, target, v)
```

**output:**

```
0 1 3 2 8 9
```

**Result:**

Hence Prolog Program to implement Best First Search algorithm is done.



## Exp 37 : Write the prolog program for Medical Diagnosis.

### Aim:

To find the Medical diagnosis using python program

### Algorithm:

Define the symptoms and diseases as facts.

Create rules to establish the relationship between symptoms and diseases.

Implement a diagnosis predicate that checks if a patient has a specific disease based on their symptoms.

Run queries to determine the diagnosis for a given patient.

### Program:

```
implement main
    open core,string

clauses
    run():-
        console::init(),
        succeed(). % place your own code here

domains
    disease,indication = symbol
    Patient,name = string

predicates
    hypothesis (string,disease)
    symptom (name,indication)
    response (char)
    go
clauses
    go :-
        write("What is the patient's name? "),
        readln(Patient),
        hypothesis(Patient,Disease),
        write(Patient,"probably has ",Disease,"."),nl.

    go :-
        write("Sorry, I don't seem to be able to"),nl,
        write("diagnose the disease."),nl.
```



```
symptom(Patient,fever) :-  
    write("Does ",Patient," have a fever (y/n ?"),  
    response(Reply),  
    Reply='y'.
```

```
symptom(Patient,rash) :-  
    write("Does ",Patient," have a rash (y/n ?"),  
    response(Reply),  
    Reply='y'.
```

```
symptom(Patient,headache) :-  
    write("Does ",Patient," have a headache (y/n ?"),  
    response(Reply),  
    Reply='y'.
```

```
symptom(Patient,runny_nose) :-  
    write("Does ",Patient," have a runny_nose (y/n ?"),  
    response(Reply),  
    Reply='y'.
```

```
symptom(Patient,conjunctivitis) :-  
    write("Does ",Patient," have a conjunctivitis (y/n ?"),  
    response(Reply),  
    Reply='y'.
```

```
symptom(Patient,cough) :-  
    write("Does ",Patient," have a cough (y/n ?"),  
    response(Reply),  
    Reply='y'.
```

```
symptom(Patient,body_ache) :-  
    write("Does ",Patient," have a body_ache (y/n ?"),  
    response(Reply),  
    Reply='y'.
```

```
symptom(Patient,chills) :-  
    write("Does ",Patient," have a chills (y/n ?"),  
    response(Reply),  
    Reply='y'.
```

```
symptom(Patient,sore_throat) :-  
    write("Does ",Patient," have a sore_throat (y/n ?"),  
    response(Reply),  
    Reply='y'.
```



```
symptom(Patient,sneezing) :-  
    write("Does ",Patient," have a sneezing (y/n) ?"),  
    response(Reply),  
    Reply='y'.
```

```
symptom(Patient,swollen_glands) :-  
    write("Does ",Patient," have a swollen_glands (y/n) ?"),  
    response(Reply),  
    Reply='y'.
```

```
hypothesis(Patient,measles) :-  
    symptom(Patient,fever),  
    symptom(Patient,cough),  
    symptom(Patient,conjunctivitis),  
    symptom(Patient,runny_nose),  
    symptom(Patient,rash).
```

```
hypothesis(Patient,german_measles) :-  
    symptom(Patient,fever),  
    symptom(Patient,headache),  
    symptom(Patient,runny_nose),  
    symptom(Patient,rash).
```

```
hypothesis(Patient,flu) :-  
    symptom(Patient,fever),  
    symptom(Patient,headache),  
    symptom(Patient,body_ache),  
    symptom(Patient,conjunctivitis),  
    symptom(Patient,chills),  
    symptom(Patient,sore_throat),  
    symptom(Patient,runny_nose),  
    symptom(Patient,cough).
```

```
hypothesis(Patient,common_cold) :-  
    symptom(Patient,headache),  
    symptom(Patient,sneezing),  
    symptom(Patient,sore_throat),  
    symptom(Patient,runny_nose),  
    symptom(Patient,chills).
```

```
hypothesis(Patient,mumps) :-  
    symptom(Patient,fever),  
    symptom(Patient,swollen_glands).
```



```
hypothesis(Patient,chicken_pox) :-  
    symptom(Patient,fever),  
    symptom(Patient,chills),  
    symptom(Patient,body_ache),  
    symptom(Patient,rash).
```

```
hypothesis(Patient,measles) :-  
    symptom(Patient,cough),  
    symptom(Patient,sneezing),  
    symptom(Patient,runny_nose).
```

```
response(Reply) :-  
    readchar(Reply),  
    write(Reply),nl.
```

end implement main

```
goal  
    mainExe::run(main::run).
```

## Output:

?- best\_first\_search(a, j, Path).

This will find the best path from node a to node j using the Best First Search algorithm and bind it to the Path variable.

## Result:

Hence the prolog program for Medical Diagnosis is done.



**Exp 38:**

**Write a Prolog Program for forward Chaining.  
Incorporate required queries.**

**Aim:**

Forward chaining incorporate required queries using python

**Algorithm:**

Define the initial facts and rules.

Facts represent the initial knowledge or information.

Rules define the relationships between facts and can be used to derive new facts.

Define the forward chaining predicate.

This predicate will be responsible for applying the rules to derive new facts.

Implement the forward chaining predicate.

The forward chaining predicate should iterate over the rules and facts.

Check if each rule's conditions (antecedents) are satisfied by the available facts.

If a rule's conditions are satisfied, derive a new fact (consequent) and add it to the set of facts.

Continue iterating until no new facts can be derived.

**Program:**

```
% Define the facts and rules
```

```
bird(penguin) :- cannot_fly.
```

```
bird(eagle) :- can_fly.
```

```
cannot_fly.
```

```
can_fly :- has_wings.
```

```
has_wings.
```

```
% Define the forward chaining rule
```

```
infer(X) :- bird(X).
```



% Example queries

?- infer(penguin).

?- infer(eagle).

?- infer(crow).

Result: Hence its proved

Exp 39 : Write a Prolog Program for backward Chaining. Incorporate required queries.

Aim: backward chainiag incorporative requird queries

Program:

% Define the facts and rules

bird(penguin) :- cannot\_fly.

bird(eagle) :- can\_fly.

cannot\_fly.

can\_fly :- has\_wings.

has\_wings.

% Define the backward chaining rules

infer(X) :- bird(X).

infer(X) :- can\_fly, X = eagle.

infer(X) :- cannot\_fly, X = penguin.

% Example queries

?- infer(crow).

?- infer(X).

### Output:

To start the forward chaining process, you can query ?- forward chaining.. This will execute the forward chaining/0 predicate and derive new facts based on the given rules.

### Result:

Hence back tracking verified.





**Exp 39 :**

## **Write a Prolog Program for Backward Chaining. Incorporate required queries.**

### **Aim:**

Forward chaining incorporate required queires using python

### **Algorithm:**

Define the backward chaining algorithm: Implement a predicate that performs backward chaining by recursively evaluating rules and facts.

Define a base case: Determine the conditions under which the backward chaining process should terminate. This could be when the goal is satisfied or when no further inference is possible.

Implement the backward chaining predicate: Write a predicate that takes a goal as input and attempts to find a rule or fact that matches the goal. If a fact is found, the goal is considered satisfied. If a rule is found, recursively call the backward chaining predicate with the subgoals of the rule.

Query the goal: Use the backward chaining predicate to query a specific goal or set of goals. This will trigger the inference process and provide the desired output.

### **Program:**

```
% Define the facts and rules
```

```
bird(penguin) :- cannot_fly.
```

```
bird(eagle) :- can_fly.
```

```
cannot_fly.
```

```
can_fly :- has_wings.
```

```
has_wings.
```

```
% Define the forward chaining rule
```

```
infer(X) :- bird(X).
```

```
% Example queries
```

```
?- infer(penguin).
```

```
?- infer(eagle).
```



?- infer(crow).

Result: Hence its proved

Exp 39 : Write a Prolog Program for backward Chaining. Incorporate required queries.

Aim: backward chainiag incorporative requird queries

Program:

% Define the facts and rules

bird(penguin) :- cannot\_fly.

bird(eagle) :- can\_fly.

cannot\_fly.

can\_fly :- has\_wings.

has\_wings.

% Define the backward chaining rules

infer(X) :- bird(X).

infer(X) :- can\_fly, X = eagle.

infer(X) :- cannot\_fly, X = penguin.

% Example queries

?- infer(crow).

?- infer(X).

**Output:**

**True**

**Result:**

Hence back tracking verified.



## Ex 40

### Create a Web Blog using Word press to demonstrate Anchor Tag, Title Tag, etc.

#### Aim:

To use web blog sing python programming.

#### procedure:

- Install WordPress on a web server or on your local machine.
- Choose a theme that supports custom menus and widgets.
- Create a new page in WordPress and add some text to it.
- Use anchor tags to link to specific sections of the page by adding the anchor tag in the HTML editor. For example, to create an anchor tag for a section with an ID of "section1", you would use the following code:  
`<a href="#section1">Click here to go to Section 1</a>`
- Use title tags to optimize your page for search engines by adding a title tag in the HTML editor. For example, to create a title tag for your page, you would use the following code:

#### Program:

```
<ul>
<li><a href="#manually-create-anchor-links-wordpress">How to
Manually Create Anchor Links in WordPress</a></li>
<li><a href="#anchor-links-wordpress-plugin">How to Create Anchor
Links in WordPress with a Plugin</a></li>
<li><a href="#anchor-links-wordpress-gutenberg">How to Create Anchor
Links in WordPress with Gutenberg</a></li>

</ul>
```

```
<ul>
<li><a href="#manually-create-anchor-links-wordpress">How to Manually Create
Anchor Links in WordPress</a></li>
<li><a href="#anchor-links-wordpress-plugin">How to Create Anchor Links in
WordPress with a Plugin</a></li>
<li><a href="#anchor-links-wordpress-gutenberg">How to Create Anchor Links in
WordPress with Gutenberg</a></li>
```

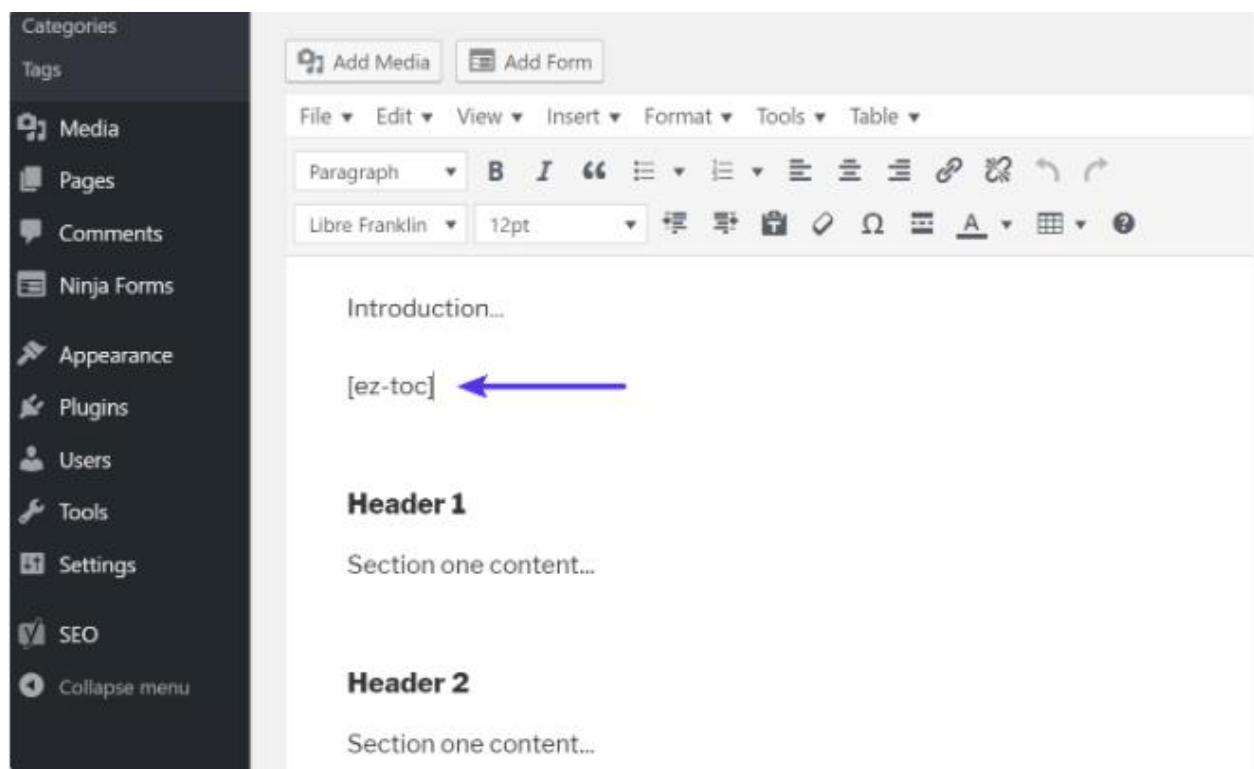


```
</ul>
```

```
<title>My Blog Page Title</title>
```

- **Publish the page and test the anchor tag links to make sure they work correctly.**
- Add more pages to your WordPress site and use anchor tags and title tags as needed.

### Output:



### Result:

Hence Web Blog using Word press to demonstrate Anchor Tag, Title Tag, etc. Web Blog using Word press to demonstrate Anchor Tag, Title Tag, etc is done.