Department of Computer Science & Engineering

IIMT College of Engineering

Greater Noida, Uttar Pradesh

Artificial Intelligence (KCS 751A)

Lab File (2024 – 2025)

For

7th Semester

Submitted To:

Ms. Akanksha Singh

Submitted By:

Raju

B.Tech CSE (7th Sem)

(2102160100088)

Department Of CSE

S. No.	Program Name	Date	Remarks
01	Study of Prolog language and its function.		
02	Write simple fact for the statements using PROLOG.		
03	Write predicates One converts centigrade temperatures to Fahrenheit, the other checks if a temperature is below freezing.		
04	Write a program to solve the Monkey Banana problem.		
05	WAP in turbo prolog for medical diagnosis and show the advantage and disadvantage of green and red cuts.		
06	WAP to implement factorial, Fibonacci of a given number.		
07	Write a program to solve 4-Queen problem.		
08	Write a program to solve traveling salesman problem.		
09	Write a program to solve water jug problem using LISP		
10	Write a program to implement Hill Climbing Algorithm		

Lab No: - 01

AIM: - Study of Prolog programming language and its function.

Prolog (short for "Programming in Logic") is a high-level programming language associated primarily with artificial intelligence and computational linguistics. It was created in the early 1970s by Alain Colmerauer and Philippe Roussel. Prolog is distinctive in its foundation on logic programming, which is a style of programming where the programmer declares what the program should accomplish, leaving how it accomplishes it largely to the system's inference engine.

Core Concepts of Prolog

Logic-Based Paradigm: Prolog is declarative, meaning that you define facts and rules rather than step-by-step procedures. Instead of writing explicit instructions, you describe relationships and let Prolog infer solutions.

Facts, Rules, and Queries:

Facts represent basic assertions about data. For example:

Prolog

```
parent(john, mary).
```

This states that "John is a parent of Mary."

Rules define logical relationships between facts, allowing Prolog to infer new information. For example:

Prolog

```
grandparent(X, Y) :- parent(X, Z), parent(Z, Y).
```

This rule defines that if X is a parent of Z and Z is a parent of Y, then X is a grandparent of Y.

Queries allow the user to ask questions about the facts and rules, and Prolog will attempt to satisfy the query. For example:

Prolog

```
?- grandparent(john, Who).
```

This query asks, "Who is the grandchild of John?"

Backtracking: Prolog uses a mechanism called *backtracking* to find all possible solutions to a query. When Prolog encounters a query, it tries to satisfy it by exploring available facts and rules. If a solution fails, Prolog backtracks to find alternative paths until all solutions are exhausted.

Unification: Unification is the process of pattern matching in Prolog. It attempts to make different expressions identical by finding common values for variables.

Applications of Prolog

Artificial Intelligence: Prolog is widely used in AI because of its ability to represent complex relationships and its support for logical reasoning.

Natural Language Processing: Prolog's structure allows it to be effective in representing and manipulating linguistic structures.
Expert Systems: With its rules and inference mechanism, Prolog is ideal for building expert systems that simulate decision-making processes.
Knowledge Representation: Prolog can efficiently represent knowledge through facts and rules, making it suitable for systems requiring complex relational data.
Sample Prolog Program
A simple Prolog program could represent a family tree and answer queries about family relationships:

Prolog
% Facts parent(john, mary). parent(mary, anne). parent(mary, tom). parent(tom, lisa). % Rules grandparent(X, Y) :- parent(X, Z), parent(Z, Y). % Query examples ?- grandparent(john, Who). % Expected output: Who = anne; Who = tom.
In this program:
The facts define direct parent-child relationships.
The rule for grandparent specifies how to infer grandparent relationships based on the parent facts.

Strengths and Limitations
Strengths:
Ideal for tasks that involve symbolic reasoning and manipulation.
Highly expressive for representing knowledge and logical relationships.

Limitations:
Not suited for numerical computation or tasks requiring extensive data processing.
Can be less efficient for large-scale applications without careful optimization.

Lab No: - 04

AIM: - Write a program to solve the Monkey Banana problem.

The “Monkey and Banana” problem is a classic puzzle often used to demonstrate problem-solving capabilities in various programming languages, including Prolog. In this problem, a monkey is placed in a room containing a banana, a chair, and a box. The monkey’s objective is to reach the banana and eat it. To do so, the monkey can perform the following actions:

Move: The monkey can move around the room freely.

Climb: The monkey can climb the chair or the box to reach higher places.

Push: The monkey can push or drag the chair or the box.

The goal is to find a sequence of actions that allows the monkey to reach the banana.

Program:

```
% Initial state: monkey is at the door, chair is at the window, banana and box are in the middle of the room.
at(monkey, door).
at(chair, window).
at(box, center).
at(banana, center).

% Actions to move the monkey
move(monkey, door, box) :- % Move to the box
    at(monkey, door),
    at(box, center).
move(monkey, box, door) :- % Move back to the door
    at(monkey, box),
    at(box, center).
move(monkey, box, window) :- % Climb the box and move to the window
    at(monkey, box),
    at(box, center),
    at(chair, window).
move(monkey, window, box) :- % Move back to the box from the window
```

```
    at(monkey, window),
    at(box, center),
    at(chair, window).

% Actions to push the box
push(box, window) :- % Push the box to the window
    at(box, center),
    at(chair, window).
push(box, center) :- % Push the box back to the center
    at(box, window),
    at(chair, window).

% Define the goal state: monkey is holding the banana
goal_state(State) :-
    at(monkey, door),
    at(banana, door),
    State = [at(monkey, door), at(chair, _), at(box, _), at(banana, door)].

% Define a predicate to solve the problem using a sequence of actions
solve(State, Actions) :-
    goal_state(State), % If the goal state is reached, no actions are needed
    Actions = [].
solve(State, [Action|Rest]) :-
    move_object(Action), % Attempt to move an object (monkey or box)
    update_state(State, Action, NewState), % Update the state
    solve(NewState, Rest). % Recursively solve the rest of the problem

% Helper predicate to move an object
move_object(Action) :-
    move(Action, _, _).
```

```
move_object(Action) :-
    push(Action, _).

% Helper predicate to update the state after an action
update_state([OldState|Rest], Action, [NewState|Rest]) :-
    apply_action(OldState, Action, NewState).

% Helper predicate to apply an action to a state
apply_action([at(monkey, M), at(chair, C), at(box, B), at(banana, Ba)],
    Action, [at(monkey, M1), at(chair, C1), at(box, B1), at(banana, Ba1)]) :-
    update_object_position(Action, M, C, B, Ba, M1, C1, B1, Ba1).

% Helper predicate to update the position of objects based on an action
update_object_position(move(monkey, From, To), From, C, B, Ba, To, C, B, Ba).
update_object_position(move(box, From, To), M, From, B, Ba, M, To, B, Ba).
update_object_position(push(box, To), M, C, To, Ba, M, C, center, Ba).
update_object_position(push(box, From), M, C, From, Ba, M, C, center, Ba).

% Example query to find a sequence of actions to solve the problem
% ?- solve([at(monkey, door), at(chair, window), at(box, center), at(banana, center)],
Actions).
```

Lab No: - 05

AIM: - WAP in turbo prolog for medical diagnosis and show the advantage and disadvantage of green and red cuts.

Turbo Prolog Program for Medical Diagnosis

```
DOMAINS
   symptom = string
   disease = string

PREDICATES
   has_symptom(symptom)
   diagnose(disease)

CLAUSES
   has_symptom(fever).
   has_symptom(cough).
   has_symptom(headache).
   has_symptom(rash).

   diagnose(flu) :-
      has_symptom(fever),
      has_symptom(cough),
      !.

   diagnose(measles) :-
      has_symptom(fever),
      has_symptom(rash),
      !.

   diagnose(migraine) :-
      has_symptom(headache),
      !.

   diagnose(unknown) :-
      write("Diagnosis could not be determined."), nl.

GOAL
   diagnose(Disease),
   write("The diagnosis is: "), write(Disease), nl.
```

Green Cuts vs. Red Cuts

In Prolog, **cuts** are a way to control backtracking, which is Prolog's built-in mechanism for exploring different possible solutions. Cuts are represented by the ! symbol and are used to limit the search space.

Green Cut: Used to improve efficiency without changing the logical correctness of the program.

- **Advantage**: Reduces unnecessary backtracking, thus speeding up the program.

- **Disadvantage**: If misused, a green cut can reduce readability and make debugging harder.

Red Cut: Changes the logical meaning of the program by altering the way solutions are derived.

- **Advantage**: Allows more control over program flow, which can be useful in cases requiring specific logical outcomes.
- **Disadvantage**: Alters the logic, making the program harder to understand and potentially incorrect if not carefully managed.

Lab No: - 06

AIM: - WAP to implement factorial, Fibonacci of a given number.

Factorial: -

The factorial of a number n*n* is defined as:

- n!=n×(n−1)×(n−2)×…×1*n*!=*n*×(*n*−1)×(*n*−2)×…×1
- 0!=10!=1 (by definition)

Code: -

```
% Base case: factorial of 0 is 1
factorial(0, 1).

% Recursive case: N! = N * (N-1)!
factorial(N, Result) :-
    N > 0,
    N1 is N - 1,
    factorial(N1, SubResult),
    Result is N * SubResult.
```

Explanation:

- The **base case** states that the factorial of 0 is 1.
- The **recursive case** calculates N!*N*! by multiplying N*N* with the factorial of N−1*N*−1, using N1 is N - 1 to decrement N*N* and compute recursively.

Output: -

```
?- factorial(5, Result).
Result = 120.
```

Fibonacci: -

The Fibonacci sequence is defined as:

- F(0)=0*F*(0)=0
- F(1)=1*F*(1)=1
- F(n)=F(n−1)+F(n−2)*F*(*n*)=*F*(*n*−1)+*F*(*n*−2) for n>1*n*>1

Code: -

```
% Base cases
fibonacci(0, 0).
fibonacci(1, 1).

% Recursive case: F(N) = F(N-1) + F(N-2)
fibonacci(N, Result) :-
    N > 1,
    N1 is N - 1,
    N2 is N - 2,
    fibonacci(N1, Result1),
    fibonacci(N2, Result2),
    Result is Result1 + Result2.
```

Explanation:

- The **base cases** define the Fibonacci values for 0 and 1.

- The **recursive case** calculates F(N)$F(N)$ by summing up F(N−1)$F(N-1)$ and F(N−2)$F(N-2)$, using N1 is N - 1 and N2 is N - 2 to decrement NN and compute recursively.

Output: -

?- fibonacci(6, Result).

Result = 8.

Lab No: - 07

AIM: - Write a program to solve 4-Queen problem.

The **4-Queen problem** is a specific case of the **N-Queen problem**, where the task is to place 4 queens on a 4x4 chessboard such that no two queens can attack each other. This means no two queens should be in the same row, column, or diagonal.

Code: -

```
% Main predicate to solve the 4-Queen problem
solve_4_queens(Solution) :-
    Solution = [Q1, Q2, Q3, Q4],
    % Each queen is in a different row
    permutation([1, 2, 3, 4], Solution),
    % Check that no queens are in the same diagonal
    safe(Q1, Q2, 1),
    safe(Q1, Q3, 2),
    safe(Q1, Q4, 3),
    safe(Q2, Q3, 1),
    safe(Q2, Q4, 2),
    safe(Q3, Q4, 1).

% Predicate to ensure no two queens are on the same diagonal
safe(Q1, Q2, Distance) :-
    Diff is abs(Q1 - Q2),
    Diff =\= Distance.

% Find all solutions to the 4-Queen problem
all_solutions(Solutions) :-
    findall(Solution, solve_4_queens(Solution), Solutions).
```

Explanation of the Code

1. **Representation**: The list Solution = [Q1, Q2, Q3, Q4] represents the positions of queens in each row. Each element in the list corresponds to the column position of the queen in that row. For example, Q1 represents the column of the queen in the first row, Q2 for the second row, etc.
2. **Permutations**: The permutation ([1, 2, 3, 4], Solution) predicate generates all possible permutations of positions for queens across rows, ensuring that each row has exactly one queen.
3. **Diagonal Check**: The safe predicate ensures that no two queens are on the same diagonal. The safe(Q1, Q2, Distance) rule checks that the absolute difference between two queen positions is not equal to their row difference (Distance), thereby ensuring they are not on the same diagonal.
4. **Finding Solutions**: all_solutions(Solutions) uses findall to collect all possible solutions.

Output: -

```
?- all_solutions(Solutions).
Solutions = [[2, 4, 1, 3], [3, 1, 4, 2]].
```

Lab No: - 08

AIM: - Write a program to solve traveling salesman problem.

The **Traveling Salesman Problem (TSP)** is a classic optimization problem where the goal is to find the shortest possible route for a salesman to visit a set of cities exactly once and return to the starting city. This problem is computationally challenging due to the factorial growth in the number of possible routes as the number of cities increases.

Code: -

```
% Define the distances between cities
distance(a, b, 10).
distance(a, c, 15).
distance(a, d, 20).
distance(b, a, 10).
distance(b, c, 35).
distance(b, d, 25).
distance(c, a, 15).
distance(c, b, 35).
distance(c, d, 30).
distance(d, a, 20).
distance(d, b, 25).
distance(d, c, 30).

% Calculate the distance of a round trip path
calculate_path_distance([City1, City2 | Rest], Distance) :-
    distance(City1, City2, D),
    calculate_path_distance([City2 | Rest], D1),
    Distance is D + D1.

% Base case for recursion: distance from last city back to the starting city
calculate_path_distance([LastCity, StartCity], Distance) :-
    distance(LastCity, StartCity, Distance).

% Find all possible paths and calculate the distances
find_tsp(StartCity, Path, MinDistance) :-
    % Define the list of cities
    Cities = [a, b, c, d],
    % Remove the starting city from the list
    select(StartCity, Cities, RemainingCities),
    % Find all permutations of remaining cities
    permutation(RemainingCities, PermutedCities),
    % Create a round trip path starting and ending with StartCity
    append([StartCity | PermutedCities], [StartCity], Path),
    % Calculate the distance of this path
    calculate_path_distance(Path, MinDistance).

% Find the shortest path by finding all paths and selecting the one with the minimum distance
tsp(StartCity, ShortestPath, MinDistance) :-
    setof((Distance, Path), find_tsp(StartCity, Path, Distance), Set),
```

Set = [(MinDistance, ShortestPath) | _].

Explanation of the Code

1. **Defining Distances**: The distance/3 predicate defines the distances between each pair of cities. This example uses four cities: a, b, c, and d.
2. **Calculating Path Distance**:
    - The predicate calculate_path_distance/2 recursively calculates the total distance for a given path.
    - The base case calculates the distance from the last city back to the starting city to complete the round trip.
3. **Finding All Paths**:
    - find_tsp/3 generates all possible round-trip paths starting from StartCity.
    - permutation/2 generates permutations of the cities excluding the starting city to ensure each city is visited once.
4. **Finding the Optimal Solution**:
    - tsp/3 uses setof/3 to find all paths with their distances, sorted in ascending order.
    - The shortest path is the first element in the sorted set (MinDistance, ShortestPath).

Output: -
?- tsp(a, ShortestPath, MinDistance).
ShortestPath = [a, b, d, c, a],
MinDistance = 80.

Lab No: - 09

AIM: - Write a program to solve water jug problem using LISP.

The **Water Jug Problem** is a classic problem in which you have two jugs with fixed capacities and an infinite water supply. The goal is to measure a specific amount of water using the jugs and various operations such as filling, emptying, and transferring water between the jugs.

Code: -

```
; Define the capacities of the jugs and the target amount
(defparameter *jug1-capacity* 4)
(defparameter *jug2-capacity* 3)
(defparameter *target* 2)

; Define the start state as both jugs being empty
(defparameter *initial-state* '(0 0))

; Helper function to check if the target amount is reached in either jug
(defun is-goal-state (state)
  (or (= (first state) *target*)
      (= (second state) *target*)))

; Generate all possible moves from a given state
(defun generate-next-states (state)
  (let* ((jug1 (first state))
         (jug2 (second state))
         (next-states
          (list
           ; Fill Jug1
           (list *jug1-capacity* jug2)
           ; Fill Jug2
           (list jug1 *jug2-capacity*)
           ; Empty Jug1
           (list 0 jug2)
           ; Empty Jug2
           (list jug1 0)
           ; Pour Jug1 into Jug2
           (if (<= (+ jug1 jug2) *jug2-capacity*)
               (list 0 (+ jug1 jug2))
               (list (- jug1 (- *jug2-capacity* jug2)) *jug2-capacity*))
           ; Pour Jug2 into Jug1
           (if (<= (+ jug1 jug2) *jug1-capacity*)
               (list (+ jug1 jug2) 0)
               (list *jug1-capacity* (- jug2 (- *jug1-capacity* jug1))))))
    next-states))

; Breadth-first search algorithm to find the solution
(defun bfs (initial-state)
  (let ((queue (list (list initial-state)))
```

```
      (visited (make-hash-table :test 'equal)))
    (loop
      ; If the queue is empty, return no solution found
      (if (null queue)
          (return "No solution found.")
          (let* ((path (pop queue))
                 (current-state (car (last path))))
            ; Check if the goal state is reached
            (if (is-goal-state current-state)
                (return (reverse path))
                (progn
                  ; Mark the current state as visited
                  (setf (gethash current-state visited) t)
                  ; Expand the state by generating possible moves
                  (dolist (next-state (generate-next-states current-state))
                    (unless (gethash next-state visited)
                      (push (append path (list next-state)) queue)))))))))

; Solve the water jug problem
(defun solve-water-jug-problem ()
  (bfs *initial-state*))
```

Explanation of the Code

1. **Define Capacities and Target**: The parameters *jug1-capacity*, *jug2-capacity*, and *target* define the capacities of the jugs and the target amount of water to measure.
2. **Initial State**: *initial-state* is defined as (0 0), representing both jugs being empty initially.
3. **Goal Check**: The function is-goal-state checks if either jug contains the target amount of water.
4. **Generating Moves**:
    - The function generate-next-states takes a state and generates all possible next states by performing actions:
        - Fill Jug1 or Jug2.
        - Empty Jug1 or Jug2.
        - Pour water from Jug1 to Jug2 and vice versa until one jug is full or the other is empty.
5. **Breadth-First Search (BFS)**:
    - The function bfs performs BFS to find the shortest sequence of moves to reach the target.
    - It keeps track of visited states using a hash table to avoid revisiting states.
    - For each state, it expands possible moves and adds unvisited states to the queue.
    - When a state reaches the goal, it returns the sequence of moves leading to that state.
6. **Solving the Problem**: The solve-water-jug-problem function runs the BFS starting from the initial state.

Lab No: - 10

AIM: - Write a program to implement Hill Climbing Algorithm.

The **Hill Climbing Algorithm** is a local search algorithm often used for optimization problems. It begins with an arbitrary solution and iteratively makes small changes to improve it, with the aim of finding a local maximum or minimum. The algorithm stops when it reaches a peak, where no neighboring state has a better value than the current state.

Suppose we want to maximize the function f(x)=−x2+4x$f(x)=-x2+4x$, which has a peak at x=2$x=2$.

Code: -

```
; Define the function to maximize: f(x) = -x^2 + 4x
(defun objective-function (x)
  (- (* -1 (expt x 2)) (* -4 x)))

; Define a small step size for incrementing or decrementing x
(defparameter *step-size* 0.1)

; Define the hill climbing function
(defun hill-climb (current-solution)
  (let ((current-value (objective-function current-solution))
        (left-solution (- current-solution *step-size*))
        (right-solution (+ current-solution *step-size*)))
    (let ((left-value (objective-function left-solution))
          (right-value (objective-function right-solution)))
      (cond
       ; Check if we have reached a peak (no better neighbors)
       ((and (<= left-value current-value)
             (<= right-value current-value))
        (list :solution current-solution :value current-value))
       ; Move towards the left if it has a higher value
       ((> left-value current-value)
        (hill-climb left-solution))
       ; Move towards the right if it has a higher value
       ((> right-value current-value)
        (hill-climb right-solution))))))

; Starting point for the hill climb
(defun find-max (start)
  (hill-climb start))
```

Explanation of the Code

1. **Objective Function**:
   - objective-function defines the mathematical function f(x)=−x2+4x$f(x)=-x2+4x$ that we want to maximize.
   - This function returns the value of f(x)$f(x)$ for a given xx.
2. **Step Size**:

- The *step-size* parameter defines the incremental step size used for checking neighboring values.
- In this example, it is set to 0.1 to allow a fine-grained search around the current solution.

3. **Hill Climbing Function**:
    - The hill-climb function takes a current-solution as input.
    - It calculates the function's value at the current solution and at two neighboring points: left-solution and right-solution.
    - Based on the values:
        - If neither neighbor has a higher value than the current point, it returns the current solution as a peak (local maximum).
        - If the left or right neighbor has a higher value, it recursively calls hill-climb on that direction to move towards a better solution.
4. **Starting the Search**:
    - find-max initiates the hill climbing from a specified starting point start.

Output: -
(find-max 0)
(:solution 2.0 :value 4.0)