Sri Eshwar College of Engineering

Kinathukadavu, Coimbatore-641202

(Approved By AICTE, New Delhi & Affiliated to Anna University, Chennai)

Department of Information Technology

R19AD211 – ARTIFICIAL INTELLIGENCE LABORATORY

Sri Eshwar College of Engineering

Kinathukadavu, Coimbatore-641202

Sri Eshwar College of Engineering
An Autonomous Institution
Affiliated to Anna University, Chennai

DEPARTMENT OF INFORMATION TECHNOLOGY

BONAFIDE CERTIFICATE

Certified that this is the bonafide record of work done by

Name:Mr./Ms...……

Register No: ……..……. of Third Year B.Tech – Information Technology in the **R19AD211 – ARTIFICIAL INTELLIGENCE LABORATORY** during the 5th Semester of the academic year 2024 – 2025 (Odd Semester).

Signature of faculty In-charge **Head of the Department**

Submitted for the practical examinations of Anna University, held on…….

Internal Examiner **External Examiner**

TABLE OF CONTENTS

S.No	Title	Page No	Aim and procedure (30 Marks)	Coding (40 Marks)	Output and Result (20 Marks)	Viva Voce (10Marks)	Total Marks (100 Marks)	Signature of faculty
1.	PROLOG Programming Introduction							
2.	Four queens on a 4x4 chessboard Problem							
3.	Traveling Salesman Problem							
4.	Breadth-First Search (BFS) for Pac-Man navigation							
5.	A* search algorithm For Pathfinding in Age of Empires							
6.	Tic-Tac-Toe Game Tree using Alpha-Beta Pruning algorithm							
7.	AI-driven Robotic System for package delivery							
8.	Telecommunication Network issue solving Simulated Annealing Algorithm							
9.	Smart Home Automation System							
10	Medical Diagnosis System							
	CONTENT BEYOND SYLLABUS							
11	Food Serving in Hotel							

AVERAGE:

MARKS IN WORDS:

SIGNATURE OF THE FACULTY

Ex.No: 1	**PROLOG Programming- Introduction**

Aim:
To study about prolog programming.

PROLOG (Programming in Logic) is a declarative programming language that stands out due to its unique approach to problem-solving and its emphasis on logical reasoning. Here are the distinctive features that set PROLOG apart, along with an exploration of its essential elements: facts, rules, and queries.

Distinctive Features of PROLOG

1. **Declarative Nature**:
   - In PROLOG, you specify *what* you want to achieve rather than *how* to achieve it. This contrasts with imperative languages, where the programmer provides a step-by-step algorithm.
2. **Logical Foundations**:
   - PROLOG is based on first-order logic, enabling it to express complex relationships and perform reasoning through logical inference.
3. **Backtracking Mechanism**:
   - PROLOG uses a backtracking algorithm to explore possible solutions. If a certain path fails, it automatically retraces its steps to try alternative routes, allowing for comprehensive search strategies.
4. **Pattern Matching**:
   - PROLOG heavily relies on pattern matching to unify queries with facts and rules, enabling a more natural expression of relationships.

Essential Components of PROLOG

1. Facts

- **Definition**: Facts are basic assertions about objects and their relationships. They represent known information in the domain.

Syntax: A fact is expressed in the form of predicates, for example:
prolog
Copy code
parent(john, mary).
parent(mary, alice).

-
- **Role**: Facts serve as the foundational data that the system can query and build upon. They establish a base from which rules can infer additional information.

2. Rules

- **Definition**: Rules define relationships and infer new facts based on existing ones. They often include conditions that must be satisfied for the rule to hold.

Syntax: A rule typically takes the form of:
prolog
Copy code
grandparent(X, Y) :- parent(X, Z), parent(Z, Y).

- This states that X is a grandparent of Y if X is a parent of Z and Z is a parent of Y.
- **Role**: Rules encapsulate logical relationships, allowing PROLOG to derive new information from known facts. They enable reasoning by establishing conditions under which new conclusions can be reached.

3. Queries

- **Definition**: Queries are questions posed to the PROLOG system, requesting information based on the facts and rules defined in the program.

Syntax: A query is typically written as a goal, for example:
prolog
Copy code
?- grandparent(john, X).

- **Role**: Queries trigger the reasoning process in PROLOG. The system attempts to satisfy the query using the available facts and rules, returning any solutions it finds.

Interaction of Components

The interplay between facts, rules, and queries enables logical reasoning in PROLOG:

- **Knowledge Base**: The combination of facts and rules forms a knowledge base, representing the entire domain of knowledge.
- **Inference**: When a query is made, PROLOG uses its inference engine to search through the knowledge base. It attempts to unify the query with existing facts and apply rules to derive new facts.
- **Backtracking**: If the first attempt to satisfy a query fails, PROLOG's backtracking mechanism allows it to explore alternative paths, ensuring that all possible solutions are considered.

Result:

Thus we study about Prolog programming

Ex.No: 2	Four queens on a 4x4 chessboard Problem

Aim:
To solve a problem of placing four queens on a 4x4 chessboard in such a way that no two queens threaten each other, using PROLOG

Procedure:

Constraints:

- No two queens can be in the same row.
- No two queens can be in the same column.
- No two queens can be on the same diagonal (both main diagonal and anti-diagonal).

Goal:

Find a configuration of 4 queens that satisfies all these constraints.

Program:

```
:- use_module(library(clpfd)).  % Import the CLP(FD) library for constraint logic programming over finite domains

% Define the main predicate to find a solution place_queens(Queens) :-
   length(Queens, 4),           % There will be 4 queens
   Queens ins 1..4,             % Each queen must be in a valid row
   all_different(Queens),       % No two queens in the same column
   no_attack(Queens),           % Ensure queens do not attack each other
   label(Queens).               % Generate solutions

% Ensure that no two queens attack each other diagonally
no_attack([]).
no_attack([Q|Rest]) :-
   no_attack(Rest),
   length(Rest, N),
   findall(D, (between(1, N, I), member(Q2, Rest), D #= abs(Q - Q2) - I), Diffs),
   all_different(Diffs).        % Diagonal attack condition

% To display the solution in a 4x4 grid format
display_board(Queens) :-
   format('  1 2 3 4~n'),
   format('  --------~n'),
   forall(between(1, 4, Row),
         (   format('~d |', Row),  % Corrected to use Row as a digit
            (   nth1(Row, Queens, Col), Col > 0 ->
               format(' Q |')
            ;   format('   |')
            )
         )),
   nl.
```

```
% Query to run the program
solve :-
    place_queens(Queens),
    display_board(Queens).

% Entry point
:- solve.
```

To Run:
Enter the following command in Terminal
?-solve.

Output:

```
?-   1 2 3 4
  --------
1 | Q |2 | Q |3 | Q |4 | Q |
```

Result:

Thus the problem of placing four queens on a 4x4 chessboard , using PROLOG has been executed successfully

Ex.No: 3	**Traveling Salesman Problem**

Aim:

The goal is to design a Prolog program that solves the Traveling Salesman Problem for the character, finding the optimal route to visit each city exactly once and return to the starting point while minimizing the total distance traveled

Procedure:

1. Define the cities and distances between them.
2. Calculate the total distance for a given route.
3. Generate all possible routes and compute their distances
4. Find the minimum distance and route.

Program:

```
% Define cities and distances
distance(city_a, city_b, 10).
distance(city_a, city_c, 15).
distance(city_a, city_d, 20).
distance(city_b, city_c, 35).
distance(city_b, city_d, 25).
distance(city_c, city_d, 30).

% Get the distance between two cities (bidirectional)
distance(X, Y, D) :- distance(Y, X, D).

% Find all cities
cities([city_a, city_b, city_c, city_d]).

% Dynamic programming table to store computed costs
:- dynamic memo/3.

% TSP using dynamic programming
tsp(Start, Cities, Cost, Path) :-
    tsp(Start, Cities, [Start], 0, Cost, Path).

tsp(_, [], Cost, Cost, Path) :-
    Path = [Start | _].  % Completed path

tsp(Start, [Next | Rest], Visited, AccumulatedCost, Cost, Path) :-
    distance(Start, Next, D),
    \+ member(Next, Visited),  % Check if already visited
    NewCost is AccumulatedCost + D,
```

```
    tsp(Next, Rest, [Next | Visited], NewCost, Cost, Path).

tsp(Start, [Next | Rest], Visited, AccumulatedCost, Cost, Path) :-
    distance(Start, Next, D),
    \+ member(Next, Visited),  % Check if already visited
    (   memo(Visited, Cost, Path) ->
        true  % Use cached result if available
    ;   NewCost is AccumulatedCost + D,
        tsp(Next, Rest, [Next | Visited], NewCost, NewCost, NewPath),
        (   NewCost < Cost ->
            assertz(memo(Visited, NewCost, NewPath))  % Cache result
        ;   true
        )
    ).

% Find the optimal route and distance
find_optimal_route(OptimalRoute, MinDistance) :-
    cities(Cities),
    findall((Route, Distance), tsp(city_a, Cities, 0, Route, Distance), Results),
    sort(2, @=<, Results, SortedResults),
    SortedResults = [(OptimalRoute, MinDistance) | _].
```

To Run:

Enter the following command in Terminal
?- find_optimal_route(OptimalRoute, MinDistance).

Output:
False

Results:

Thus the Traveling Salesman Problem for the character was solved successfully.

Ex.No: 4	**Breadth-First Search (BFS) for Pac-Man navigation**

Aim:

To play Pac-Man Maze game by implementing Breadth-First Search (BFS) for Navigation

Procedure:

- Initialize a queue with Pac-Man's starting position.
- For each position in the queue:
    - Check the 4 neighboring cells (up, down, left, right).
    - If a neighboring cell is valid (empty space or goal) and hasn't been visited, mark it as visited and add it to the queue.
    - If a neighboring cell contains a ghost or a wall, it's ignored.
- Stop when the goal is reached or when the queue is empty (i.e., no path exists).

Program:

```
% Define grid dimensions (rows, columns)
grid_dimensions(5, 5).  % For a 5x5 grid
% Maze representation (using lists of lists)
% 'empty' is a walkable space, 'wall' is an impassable space,
% 'pacman' is the starting position, 'ghost' is an obstacle.
% Example maze layout
% pacman is at (0, 0), ghosts at (2, 2) and (3, 3), walls at (1, 1) and (4, 4)
maze([
    [pacman, empty, empty, empty, empty],
    [empty, wall, empty, empty, empty],
    [empty, empty, ghost, empty, empty],
    [empty, empty, empty, ghost, empty],
    [empty, empty, empty, empty, wall]
]).

% Directions for BFS (up, down, left, right)
directions([
    (0, 1),  % Move right
    (1, 0),  % Move down
    (0, -1), % Move left
    (-1, 0)  % Move up
]).

% Check if a position is within the bounds of the grid
valid_position(X, Y) :-
    grid_dimensions(Rows, Cols),
    X >= 0, X < Rows,
    Y >= 0, Y < Cols.
```

```
% Check if the position is walkable (either empty or pacman)
walkable(X, Y) :-
    maze(Maze),
    nth0(X, Maze, Row),
    nth0(Y, Row, Cell),
    (Cell = empty; Cell = pacman).
% Check if a position contains a ghost (not walkable)
blocked_by_ghost(X, Y) :-
    maze(Maze),
    nth0(X, Maze, Row),
    nth0(Y, Row, ghost).
% BFS Algorithm to find the shortest path from Pac-Man to a target
bfs(Start, Target, Path) :-
    bfs([Start], [], Target, Path).
bfs([], _, _, no_path). % No path found
bfs([Current | Rest], Visited, Target, [Current | Path]) :-
    Current = Target, !, % Found the target
    reverse([Current | Rest], Path).
bfs([Current | Rest], Visited, Target, Path) :-
    Current \= Target, % Not the target, continue exploring
    Current = (X, Y), % Extract the coordinates from the current position
    valid_position(X, Y), % Pass X, Y to valid_position/2
    \+ member(Current, Visited), % Avoid revisiting nodes
    walkable(X, Y), % Check if the current position is walkable
    findall(Next, (move((X, Y), Next), \+ member(Next, Visited), walkable(Next)),
NextPositions),
    append(Rest, NextPositions, Queue),
    bfs(Queue, [Current | Visited], Target, Path).
% Movement logic: calculate next possible positions
move((X, Y), (NewX, NewY)) :-
    directions(Directions),
    member((DX, DY), Directions),
    NewX is X + DX,
    NewY is Y + DY,
    valid_position(NewX, NewY),
    \+ blocked_by_ghost(NewX, NewY). % Avoid ghosts
```

To Run:

```
?-bfs((0, 0), (4, 4), Path).
```

Output:

If there is a valid path,
Path = [(0, 0), (1, 0), (1, 1), (2, 1), (2, 2), ...].

If no path exists,
Path = no_path.

Results:

Thus the Pac-Man Maze game was played and executed successfully.

Ex.No: 5	**A* search algorithm For Pathfinding in Age of Empires**

Aim:

To implement an A* search algorithm for pathfinding in a dynamic environment.

Procedure:

1. Create a Prolog fact for each terrain type, associating it with a cost.
2. Create a Prolog fact to store unit positions and their types (friendly or enemy).
3. Implement a simple Manhattan distance heuristic, which sums the absolute differences of the X and Y coordinates between two points.
4. Define the starting point and goal.
5. Expand the frontier and recursively explore until the goal is reached.
6. Create a predicate to find neighbours, considering movement rules, terrain types, and obstacle avoidance.
7. Create a predicate to add neighbours to the frontier.
8. Sort the frontier (open list) by the F value, so that the node with the lowest F value is expanded first.

Program:

```
% Facts about terrain types with their movement costs

terrain_cost(open_field, 1).

terrain_cost(mountain, 3).

terrain_cost(urban_area, 2).

% Define the positions of the units on the battlefield

% unit(Position, Type), where Type can be 'friendly' or 'enemy'

unit((X, Y), friendly).

unit((X, Y), enemy).

% Define dynamic obstacles (e.g., moving enemy units)

obstacle((X, Y)) :-

unit((X, Y), enemy).

% A* search components:

% 1. Heuristic function: simple Manhattan distance

heuristic((X1, Y1), (X2, Y2), H) :-

H is abs(X2 - X1) + abs(Y2 - Y1).

% 2. A* search algorithm

a_star(Start, Goal, Path) :-
```

```
a_star_search([start(Start, 0, 0, Start, [])], Goal, Path).
% 3. Search function that expands nodes
a_star_search([start(Goal, _, G, Goal, Path)|_], Goal, Path).
a_star_search([start(Current, F, G, Current, Path)|Rest], Goal, FinalPath) :-
find_neighbors(Current, Goal, G, Path, Neighbors),
add_neighbors_to_frontier(Neighbors, Rest, Frontier),
sort_by_f_value(Frontier, SortedFrontier),
a_star_search(SortedFrontier, Goal, FinalPath).
% 4. Finding neighbors based on terrain and obstacles
find_neighbors((X, Y), Goal, G, Path, Neighbors) :-
findall(
start((X2, Y2), F, G2, (X2, Y2), Path),
(   (X2, Y2) \= (X, Y), % Avoid revisiting the same position
valid_move((X, Y), (X2, Y2), Terrain),
terrain_cost(Terrain, Cost),
\+ obstacle((X2, Y2)), % Avoid obstacles
G2 is G + Cost,
heuristic((X2, Y2), Goal, H),
F is G2 + H,
\+ member((X2, Y2), Path) % Avoid loops
),
Neighbors
).
% 5. Valid move check based on terrain type (can be extended for diagonal or more complex
rules)
valid_move((X, Y), (X2, Y2), open_field) :- X2 is X + 1, Y2 is Y.  % Example: Rightward
movement
valid_move((X, Y), (X2, Y2), open_field) :- X2 is X - 1, Y2 is Y.  % Example: Leftward
movement
valid_move((X, Y), (X2, Y2), open_field) :- X2 is X, Y2 is Y + 1.  % Example: Downward
movement
valid_move((X, Y), (X2, Y2), open_field) :- X2 is X, Y2 is Y - 1.  % Example: Upward
movement
```

```
% 6. Add neighbors to frontier and avoid cycles
add_neighbors_to_frontier([], Frontier, Frontier).
add_neighbors_to_frontier([Neighbor|Rest], Frontier, [Neighbor|NewFrontier]) :-
add_neighbors_to_frontier(Rest, Frontier, NewFrontier).
% 7. Sort the frontier list based on F values (ascending order)
sort_by_f_value([], []).
sort_by_f_value([start(Pos1, F1, G1, Pos1, Path1)|Rest], Sorted) :-
sort_by_f_value(Rest, SortedRest),
insert_sorted(start(Pos1, F1, G1, Pos1, Path1), SortedRest, Sorted).
% Insert node into sorted list based on F value
insert_sorted(X, [], [X]).
insert_sorted(start(Pos1, F1, G1, Pos1, Path1), [start(Pos2, F2, G2, Pos2, Path2)|Rest],
[start(Pos1, F1, G1, Pos1, Path1), start(Pos2, F2, G2, Pos2, Path2)|Rest]) :-
F1 =< F2.
insert_sorted(start(Pos1, F1, G1, Pos1, Path1), [start(Pos2, F2, G2, Pos2, Path2)|Rest],
[start(Pos2, F2, G2, Pos2, Path2)|SortedRest]) :-
F1 > F2,
insert_sorted(start(Pos1, F1, G1, Pos1, Path1), Rest, SortedRest).
```

To Run:

```
a_star((0, 0), (5, 5), Path).
```

Output:

```
false
```

Results:

Thus A* search algorithm for pathfinding in a dynamic environment was executed successfully.

Ex.No: 6	**Tic-Tac-Toe Game Tree using Alpha-Beta Pruning algorithm**

Aim:

To implement the Alpha-Beta Pruning algorithm to determine the best move for a player in a Tic-Tac-Toe game tree.

Procedure:

1. Initialize the Game Tree
2. Provide the inputs to the terminal node.
3. Check for the terminal state repeatedly
4. Generate the available moves
5. Often recursively evaluate each move
6. Prune the unnecessary branches if condition satisfies.
7. Update the Alpha and Beta values

Program:

```
% Define possible win conditions for Tic-Tac-Toe

win_condition([1, 2, 3]).

win_condition([4, 5, 6]).

win_condition([7, 8, 9]).

win_condition([1, 4, 7]).

win_condition([2, 5, 8]).

win_condition([3, 6, 9]).

win_condition([1, 5, 9]).

win_condition([3, 5, 7]).

% Check if a player has won

has_won(Board, Player) :-

win_condition([A, B, C]),

nth1(A, Board, Player),

nth1(B, Board, Player),

nth1(C, Board, Player).

% Check if the board is full

is_full(Board) :-

\+ member(empty, Board).

% Evaluation function for the board: 1 for X win, -1 for O win, 0 for draw
```

```
evaluate(Board, 1) :- has_won(Board, x). % X wins
evaluate(Board, -1) :- has_won(Board, o). % O wins
evaluate(Board, 0) :- is_full(Board), \+ has_won(Board, x), \+ has_won(Board, o). % Draw
% Generate possible moves for the current player
possible_moves(Board, Player, NewBoards) :-
findall(NewBoard, make_move(Board, Player, NewBoard), NewBoards).
% Make a move by placing the current player's symbol in an empty spot
make_move(Board, Player, NewBoard) :-
select(empty, Board, Player, NewBoard).
% The main Alpha-Beta Pruning algorithm
alpha_beta(Board, Depth, Alpha, Beta, Player, BestMove, Value) :-
Depth > 0,
possible_moves(Board, Player, NewBoards),
evaluate_moves(NewBoards, Depth, Alpha, Beta, Player, BestMove, Value).
alpha_beta(Board, 0, Alpha, Beta, Player, _, Value) :-
evaluate(Board, Value),
Value >= Alpha,
Value =< Beta.
% Evaluate all possible moves for the current player
evaluate_moves([Board | RestBoards], Depth, Alpha, Beta, Player, BestMove, BestValue) :-
next_player(Player, NextPlayer),
alpha_beta(Board, Depth - 1, Alpha, Beta, NextPlayer, _, Value),
update_alpha_beta(Value, Alpha, Beta, BestMove, BestValue, RestBoards, Depth, Player).

% Update Alpha-Beta values and prune if necessary
update_alpha_beta(Value, Alpha, Beta, Board, Value, [], _, _) :-
Value >= Beta, !. % Prune
update_alpha_beta(Value, Alpha, Beta, Board, Value, [NextBoard | RestBoards], Depth, Player)
:-
Value > Alpha, !,
evaluate_moves(RestBoards, Depth, Value, Beta, Player, _, BestValue).
```

% Switch players between 'x' and 'o'

next_player(x, o).

next_player(o, x).

To Run:

?- % Example of a fresh empty board:

% [empty, empty, empty, empty, empty, empty, empty, empty, empty]

| | alpha_beta([empty, empty, empty, empty, empty, empty, empty, empty, empty], 9, -inf, inf, x, BestMove, BestValue).|

Output:

False

Results:

Thus the Alpha-Beta Pruning algorithm to determine the best move for a player in a Tic-Tac-Toe game tree is executed successfully.

Ex.No: 7	AI-driven Robotic System for package delivery

Aim:

The goal is to optimize how the robotic fleet navigates through the urban environment while considering traffic, pedestrian density, and delivery deadlines.

Theory:

The **genetic operators**, specifically **crossover** and **mutation**, play a critical role in the search for the optimal parameters by balancing **exploration** (discovering new areas of the solution space) and **exploitation** (refining and optimizing already discovered good solutions). Below, I'll describe how these operators are designed and how they contribute to solving the route planning problem.

1. Crossover Operator

Crossover is a genetic operator used to combine parts of two parent chromosomes (solutions) to produce offspring that may inherit the strengths of both parents. The purpose of crossover is to **explore the solution space** by combining different features of good solutions and hopefully creating new, better solutions.

In the case of the **robotic delivery system**, each chromosome encodes a set of parameters for route planning. These parameters might include:

- **Speed limits**: How fast the robot can move on different types of streets.
- **Time buffer**: Extra time allowed for potential delays (e.g., traffic, pedestrian density).
- **Traffic sensitivity**: The robot's responsiveness to varying traffic conditions.
- **Pedestrian sensitivity**: The robot's behavior when encountering pedestrians (e.g., slow down, reroute).

Crossover Design

For the crossover operator, I would implement a **two-point crossover** for this problem, where two parents (route parameters) are combined at two crossover points. This is appropriate because route planning often involves multiple components (speed, time buffer, traffic sensitivity, etc.), and combining these components in different ways could lead to better performance.

Crossover Mechanism:

1. **Select two parents**: Each parent will have a list of parameters [SpeedLimit, TimeBuffer, TrafficSensitivity, PedestrianSensitivity].

2. **Choose two crossover points**: Randomly select two points in the list of parameters. These points will divide the chromosome into three segments.
3. **Exchange the segments between parents**: Combine the first segment from Parent 1, the second segment from Parent 2, and the third segment from Parent 1 (or vice versa).

Example:

Parent 1: [50, 10, 8, 5]

Parent 2: [60, 15, 5, 3]

If we choose the first and third points for crossover:

- Segment 1: [50] (from Parent 1)
- Segment 2: [15] (from Parent 2)
- Segment 3: [5] (from Parent 1)

The offspring might look like this:

- **Offspring 1**: [50, 15, 8, 3]
- **Offspring 2**: [60, 10, 5, 5]

Impact on Exploration and Exploitation:

- **Exploration**: The two-point crossover allows the solution space to be explored by combining different segments of each parent, which may lead to new configurations of parameters.
- **Exploitation**: By preserving good segments of parents, the offspring retain some beneficial features (e.g., if one parent had a good time buffer and the other had good traffic sensitivity, the offspring can inherit both features).

  This enables the algorithm to explore various combinations of parameters and find more optimal configurations for routing.

2. Mutation Operator

Mutation is another important genetic operator that introduces **random changes** to a chromosome, helping to maintain **diversity** within the population and preventing premature convergence to suboptimal solutions.

For this problem, mutation is crucial because route planning in urban environments involves highly dynamic and complex factors (e.g., traffic patterns, pedestrian density, etc.). **Mutating** one of the parameters could simulate environmental changes or unexpected factors, pushing the algorithm to discover better solutions over time.

Mutation Design

The mutation operator will randomly alter one or more parameters in a chromosome, which could represent:

- **Speed limits**: Modify the speed of the robot.
- **Time buffer**: Increase or decrease the buffer time.
- **Traffic sensitivity**: Change the robot's response to traffic conditions.
- **Pedestrian sensitivity**: Adjust how sensitive the robot is to pedestrians.

Mutation Mechanism:

1. **Select a chromosome**: Randomly choose a chromosome from the population.
2. **Choose a parameter to mutate**: Randomly select one of the parameters (speed limit, time buffer, etc.).
3. **Alter the parameter value**: Introduce a small random change, either increasing or decreasing the parameter within its feasible range.

Example:

Chromosome: [50, 10, 8, 5]

If the mutation is applied to the **TimeBuffer** (2nd parameter), the new value might become 12 (mutated).

Resulting chromosome: [50, 12, 8, 5]

Alternatively, if the mutation happens on the **PedestrianSensitivity**, it could become 6 (mutated).

Resulting chromosome: [50, 10, 8, 6]

Impact on Exploration and Exploitation:

- **Exploration**: Mutation allows the algorithm to explore parts of the solution space that may not be reachable through crossover alone. By introducing small, random changes, the mutation ensures that the population does not get stuck in local optima.
- **Exploitation**: By occasionally modifying parameters that are not working well, the mutation operator enables fine-tuning of solutions that were already performing well, improving them over time.

3. Exploration and Exploitation in Urban Route Planning

In the context of **route planning for a robotic fleet** in urban environments, the crossover and mutation operators help address specific challenges like:

- **Dynamic Environments**: Traffic and pedestrian density are not static. Crossover and mutation ensure the algorithm can adapt to changing conditions by exploring different parameter configurations and adjusting them based on new insights.
- **Multiple Objectives**: The robot needs to optimize for various conflicting factors (e.g., minimizing travel time while avoiding pedestrians and traffic). Crossover allows for a

flexible combination of different strategies, while mutation introduces the randomness needed to search for solutions that balance these competing objectives.

- **Complexity of the Problem**: Urban environments are highly complex, with unpredictable variables. Mutation helps maintain diversity and avoids the algorithm getting stuck in a local optimum, while crossover enables the combination of potentially diverse strategies to yield better solutions.

Conclusion

The **crossover** and **mutation** operators in this genetic algorithm are designed to effectively balance **exploration** and **exploitation** of the solution space for route planning in an urban environment.

- **Crossover** helps explore combinations of good features from multiple solutions, allowing the algorithm to discover new, potentially better solutions.
- **Mutation** introduces random changes to maintain diversity and avoid local optima, ensuring that the search space is continuously explored and refined.

By using these operators in tandem, the genetic algorithm can evolve an optimal set of parameters that enable the robotic fleet to navigate efficiently, respond to traffic and pedestrian density, and meet delivery deadlines in a dynamic urban environment.

Ex.No: 8	**Telecommunication Network issue solving Simulated Annealing Algorithm**

Aim:

The aim is to create an efficient layout of network nodes and connections that minimizes latency, maximizes data throughput, and ensures resource utilization is optimized.

Procedure:

1. Generates a random float between 0 and 1.
2. Calculates the total cost of a given network layout
3. Calculate latency based on distance between nodes and hop count
4. Calculate throughput based on link bandwidth
5. Calculate resource utilization based on node load and link usage.
6. Simulated Annealing algorithm for network layout optimization.
7. Ensure costs are instantiated before calculating DeltaCost
8. Accept the neighbour if it improves the cost or based on a probability.
9. Reduce the temperature and iterate.
10. Determines the probability of accepting a worse solution based on the temperature and cost change.
11. Generates a neighboring solution by making a small change in the network layout.

Program:

```
% random_float/1: Generates a random float between 0 and 1.
random_float(Rand) :-
random(Rand).
DeltaCost is NeighborCost - CurrentCost,
% cost/2 calculates the total cost of a given network layout.
cost(NetworkLayout, TotalCost) :-
calculate_latency(NetworkLayout, Latency),
calculate_throughput(NetworkLayout, Throughput),
calculate_resource_utilization(NetworkLayout, Utilization),
TotalCost is Latency + (1 / Throughput) + Utilization.
```

```
% Calculate latency based on distance between nodes and hop count.
calculate_latency(NetworkLayout, Latency) :-
% Placeholder for actual latency calculation logic.
% In a real scenario, this would take into account the distances and number of hops.
Latency = 10.
% Calculate throughput based on link bandwidth.
calculate_throughput(NetworkLayout, Throughput) :-
% Placeholder for throughput calculation logic.
Throughput = 1000.  % Example value.
% Calculate resource utilization based on node load and link usage.
calculate_resource_utilization(NetworkLayout, Utilization) :-
% Placeholder for utilization calculation logic.
Utilization = 5.
% Simulated Annealing algorithm for network layout optimization.
simulated_annealing(InitialLayout, InitialTemperature, CoolingRate, MaxIterations,
BestLayout) :-
simulate_annealing(InitialLayout, InitialTemperature, CoolingRate, MaxIterations, BestLayout,
BestCost).
simulate_annealing(CurrentLayout, Temperature, CoolingRate, 0, BestLayout, BestCost) :-
% If we reach 0 iterations, return the best layout.
cost(CurrentLayout, BestCost),
BestLayout = CurrentLayout.
simulate_annealing(CurrentLayout, Temperature, CoolingRate, IterationsLeft, BestLayout,
BestCost) :-
IterationsLeft > 0,
% Generate a neighboring layout.
generate_neighbor(CurrentLayout, NeighborLayout),
% Ensure costs are instantiated before calculating DeltaCost
cost(CurrentLayout, CurrentCost),    % Calculate current cost.
cost(NeighborLayout, NeighborCost),  % Calculate neighbor cost.
% Now DeltaCost can be computed safely
DeltaCost is NeighborCost - CurrentCost,

% Accept the neighbor if it improves the cost or based on a probability.
```

```
(NeighborCost < CurrentCost ->
NewLayout = NeighborLayout, NewCost = NeighborCost
;
accept_probability(DeltaCost, Temperature, Probability),
random_float(Rand),
(Rand < Probability ->
NewLayout = NeighborLayout, NewCost = NeighborCost
;
NewLayout = CurrentLayout, NewCost = CurrentCost
)
),
% Update the best layout if needed.
(NewCost < BestCost ->
BestLayout = NewLayout, BestCost = NewCost
;
BestLayout = BestLayout, BestCost = BestCost
),
% Reduce the temperature and iterate.
NewTemperature is Temperature * CoolingRate,
NewIterationsLeft is IterationsLeft - 1,
simulate_annealing(NewLayout, NewTemperature, CoolingRate, NewIterationsLeft,
BestLayout, BestCost).
% accept_probability/3: Determines the probability of accepting a worse solution based on the
temperature and cost change.
accept_probability(DeltaCost, Temperature, Probability) :-
(DeltaCost < 0 -> Probability = 1 ; Probability is exp(-DeltaCost / Temperature)).
% generate_neighbor/2: Generates a neighboring solution by making a small change in the
network layout.
% This could involve moving a node, rewiring a link, or adjusting the link bandwidth.
generate_neighbor(CurrentLayout, NeighborLayout) :-
% Placeholder for generating a neighbor.
% This could involve changing node positions or adjusting link parameters.
% Example: swap two nodes or change link bandwidth.
NeighborLayout = CurrentLayout.  % No change for simplicity.
```

To Run:

% Initial network layout (a simple example).

InitialLayout = [node1, node2, node3],

InitialTemperature = 100,

CoolingRate = 0.95,

MaxIterations = 1000,

% Start the simulated annealing process.

simulated_annealing(InitialLayout, InitialTemperature, CoolingRate, MaxIterations, BestLayout).

Results:

Thus prolog program is executed successfully.

Ex.No: 9	Smart Home Automation System

Aim:

To implement propositional logic inferences to make decisions and take actions based on the information gathered by these sensors.

Theory:

In a smart home automation system that utilizes sensors like motion, door/window, and temperature sensors, propositional logic can be used to make inferences and drive automated actions based on sensor readings. To handle dynamic situations, the system would need to continuously monitor changes in the environment, recognize patterns, and adapt its behavior accordingly.

1. Propositional Logic Representation

Let's start by representing the sensor inputs as propositional variables. For simplicity, we'll define the following:

- M: Motion detected (True if motion is detected, False otherwise)
- D: Door/window is open (True if a door/window is open, False otherwise)
- T: Temperature is below a certain threshold (True if temperature is below threshold, False otherwise)
- P: Person is at home (True if a person is detected in the house, False otherwise)

2. Basic Inferences and Actions

2.1. Heating/AC Control (Temperature)

- If temperature drops below a threshold (e.g., 18°C), turn on the heating system.
- If temperature rises above a threshold (e.g., 24°C), turn on the air conditioning.

Propositional Logic for Heating/AC control:

- $T \rightarrow$ Turn on heating
- $\neg T$ (temperature above threshold) $\rightarrow$ Turn off heating
- $\neg T \rightarrow$ Turn on air conditioning
- $T \rightarrow$ Turn off air conditioning

Action:

- If T = True, heating system is activated.
- If T = False and temperature exceeds the upper threshold, the AC system is activated.

2.2. Lighting Control Based on Motion

- If motion is detected in a room (i.e., someone is in the room), turn on the lights.
- If no motion is detected for a certain period, turn off the lights.

Propositional Logic:

- M → Turn on lights
- ¬M (no motion for some time) → Turn off lights

Action:

- When M = True, lights are turned on.
- After a predefined time period with M = False, lights are turned off.

2.3. Security System (Door/Window)

- If a door or window is opened, alert the homeowner.
- If motion is detected and no one is supposed to be home, trigger an alert.

Propositional Logic:

- D → Trigger security alert
- M ∧ ¬P (motion detected and no person at home) → Trigger security alert

Action:

- When D = True, a security alert is triggered.
- If M = True and P = False, a security alert is triggered.

2.4. Presence Detection

- If motion is detected and no one is at home, update the occupancy status to "Person present."
- If no motion is detected for a longer period, update to "Person not present."

Propositional Logic:

- M ∧ ¬P → P (A person is likely present if motion is detected and no one was previously registered as home)
- ¬M for a prolonged period → ¬P (No motion leads to "Person not present" status)

Action:

- When M = True, update P = True.
- After a period of ¬M, update P = False.

3. Adapting to Dynamic Situations

To handle dynamic situations, the system needs to adjust its behavior based on changes in occupancy patterns, temperature fluctuations, or new devices being added to the system. This can be achieved through inference rules that automatically adjust based on detected conditions.

3.1. Handling Sudden Temperature Drops

The system should detect if the temperature has dropped suddenly and respond accordingly, especially if this happens outside of the usual seasonal pattern.

- If temperature drops rapidly below a predefined threshold (e.g., 5°C within 30 minutes), it could indicate a heating failure or an extreme weather event. The system should turn on auxiliary heating or alert the user.

Propositional Logic:

- ΔT (sudden drop in temperature) $\rightarrow$ Alert or activate backup heating.

Action:

- The system should check for ΔT (a significant temperature drop over a short period).
- If ΔT is true, trigger an alert and turn on auxiliary heating.

3.2. Changes in Occupancy Patterns

Over time, the system can learn occupancy patterns (e.g., presence or absence at certain times of day) and adjust its logic accordingly.

- If a person is detected in the house more frequently at certain hours, the system may anticipate their presence and adjust settings (e.g., lighting, heating) in advance.

Propositional Logic:

- P detected in the house at hour X $\rightarrow$ Turn on lights or adjust heating for hour X.

Action:

- Track occupancy patterns (e.g., using a time-series database).
- If occupancy patterns suggest a person is more likely to be home at certain times, adjust the system's settings in advance.

3.3. Introduction of New Devices

When a new device is added to the system (e.g., a new smart light, thermostat, or sensor), the system should adapt by integrating the new device into its decision-making process.

- If a new device is added, update the logic to include the new sensor's state in existing inferences (e.g., if a new motion sensor is added in another room, extend the logic for motion detection and lighting control to that room).

Propositional Logic:

- New Sensor (S) added $\rightarrow$ Incorporate sensor into existing rules.
- $S \land \neg P$ (motion detected by the new sensor but no person detected) $\rightarrow$ Trigger alert.

Action:

- New sensors are automatically incorporated into the logic.
- If the new sensor detects motion, the system can trigger actions (e.g., lights, security alerts).

4. Adapting the System Over Time

As the system learns and adapts, it can refine its logic. For example:

- If a motion sensor consistently detects movement in a room at certain times, the system can anticipate the person's behavior and automatically adjust the lighting, temperature, or security settings based on learned patterns.
- If the temperature fluctuates significantly at certain times of day (e.g., due to the sun heating up certain rooms), the system could learn to adjust the heating or cooling more proactively.

Conclusion

Using propositional logic in this way enables a smart home system to make intelligent, rule-based decisions based on real-time sensor data. The system can also be designed to adapt dynamically by learning from patterns in sensor data, adjusting actions to handle changes like temperature drops, shifts in occupancy, and the introduction of new devices.

Ex.No: 10	**Medical Diagnosis System**

Aim:

The main aim is to design a knowledge base for the medical diagnosis system, including predicates representing symptoms, diseases, and treatments.

Procedure:

1. Define a Predicate class, Knowledge base class, Inference Engine class
2. Initialize the Predicate class, Knowledge base class, Inference Engine class
3. Add facts to the Knowledgebase
4. Perform inference.

Program:

Python code

```
class Predicate:
def __init__(self, name, args):
self.name = name
self.args = args
def __str__(self):
return f"{self.name}({', '.join(map(str, self.args))})"
class KnowledgeBase:
def __init__(self):
self.facts = []
def add_fact(self, fact):
self.facts.append(fact)
def has_fact(self, fact):
return fact in self.facts
class InferenceEngine:
def __init__(self, knowledge_base):
self.kb = knowledge_base
def resolve(self, predicate):
```

```
for fact in self.kb.facts:
if fact.name == predicate.name and len(fact.args) == len(predicate.args):
match = True
for f, p in zip(fact.args, predicate.args):
if f != p:
match = False
break
if match:
return True
return False
# Initialize Knowledge Base and add facts
kb = KnowledgeBase()
kb.add_fact(Predicate("has_symptom", ["john", "cough"]))
kb.add_fact(Predicate("has_symptom", ["john", "fever"]))
kb.add_fact(Predicate("has_symptom", ["mary", "cough"]))
kb.add_fact(Predicate("has_symptom", ["mary", "shortness_of_breath"]))
kb.add_fact(Predicate("has_disease", ["john", "flu"]))
kb.add_fact(Predicate("has_disease", ["mary", "pneumonia"]))
kb.add_fact(Predicate("recommended_treatment", ["flu", "rest"]))
kb.add_fact(Predicate("recommended_treatment", ["pneumonia", "antibiotics"]))
# Initialize inference engine
inference_engine = InferenceEngine(kb)
# Example: Diagnosing disease based on symptoms
if inference_engine.resolve(Predicate("has_symptom", ["john", "cough"])) and
inference_engine.resolve(Predicate("has_symptom", ["john", "fever"])):
print("Diagnosing: John might have the flu.")
# Example: Recommending treatment
if inference_engine.resolve(Predicate("has_disease", ["john", "flu"])):
print("Treatment for flu: Rest.")
```

Output:

Diagnosing: John might have the flu.

Treatment for flu: Rest.

Result:

Thus, we executed a medical diagnosis system successfully.

Ex.No: 11	**Food Serving in Hotel**

Aim:

To wite a prolog program to design a robot that serve food in a hotel.

Procedure:

Program:

```
% Facts about room status (can be dynamic in real scenarios)
room_status(room101, occupied).
room_status(room102, vacant).
room_status(room103, occupied).
room_status(room104, vacant).
% Facts about food orders
order(room101, pizza).
order(room102, sushi).
order(room103, pasta).
order(room104, salad).
% Facts about robot's actions
robot_delivers(room101, pizza).
robot_delivers(room103, pasta).
% Rule to check if a room is occupied
is_occupied(Room) :-
room_status(Room, occupied).
% Rule to check if a room is vacant
is_vacant(Room) :-
room_status(Room, vacant).
% Rule for robot to deliver food to an occupied room
deliver_food(Room) :-
is_occupied(Room),
order(Room, Food),
\+ robot_delivers(Room, Food),  % Prevent double delivery
```

```
assert(robot_delivers(Room, Food)),
write('Robot delivering '), write(Food), write(' to '), write(Room), nl.
% Rule to handle the robot when a room is vacant
handle_vacant_room(Room) :-
is_vacant(Room),
write('Room '), write(Room), write(' is vacant. No delivery needed.'), nl.
% Rule to check if food has been delivered to a room
food_delivered(Room) :-
robot_delivers(Room, Food),
write('Food '), write(Food), write(' has been delivered to '), write(Room), nl.
% Rule to check for any pending food deliveries in all rooms
deliver_all :-
order(Room, Food),
\+ robot_delivers(Room, Food),
deliver_food(Room),
fail.  % Force backtracking to check all orders
deliver_all.  % Success case to stop backtracking
% Example: Handle all rooms and deliver food where needed
deliver_all_rooms :-
write('Starting food delivery...'), nl,
deliver_all,
write('All food deliveries are complete!'), nl.
% Example query to check if the robot has delivered food to a room
% food_delivered(room101).
```

To Run:

1. ?-deliver_all_rooms.
2. food_delivered(room101).

Output:

1. Starting food delivery…

All food deliveries are complete!

2. Starting food delivery…

All food deliveries are complete!

true

Result:

Thus, the prlog program for designing a robot to serve a food in hotel was executed successfully.