|  |  |
| --- | --- |
| Shri Vile Parle Kelavani Mandal's  **INSTITUTE OF TECHNOLOGY**  **DHULE (M.S.)**  **DEPARMENT OF COMPUTER ENGINEERING** | |
| **Name**: Anushka Paras Jain **Roll No**. 01  **Subject :** Artificial Intelligence Lab **Subject Code :** BTCOL707  **Class:** Final Year Comp. Engg. **Expt. No. :** 01  **Title :** Study of PROLOG. Write the following programs using PROLOG. | |
| **Problem Statement :**  **Software Required :**  **Theory :**  **Conclusion:** | Study of PROLOG. Write the following programs using PROLOG.  1) Write simple fact for following:  a. Ram likes mango.  b. Seema is a girl.  c. Bill likes Cindy.  d. Rose is red.  e. John owns gold  2) Write predicates One converts centigrade temperatures to Fahrenheit, the other checks if a temperature is below freezing.  Prolog  **PROLOG-PROGRAMMING IN LOGIC**  PROLOG stands for Programming, In Logic — an idea that emerged in the early 1970’s to  use logic as programming language. The early developers of this idea included Robert  Kowaiski at Edinburgh (on the theoretical side), Marrten van Emden at Edinburgh  (experimental demonstration) and Alian Colmerauer at Marseilles (implementation).  David D.H. Warren’s efficient implementation at Edinburgh in the mid -1970’s greatly  contributed to the popularity of PROLOG. PROLOG is a programming language centered  around a small set of basic mechanisms, including pattern matching, tree based data  structuring and automatic backtracking. This Small set constitutes a surprisingly powerful  and flexible programming framework. PROLOG is especially well suited for problems that  involve objects- in particular, structured objects- and relations between them.  **SYMBOLIC LANGUAGE**  PROLOG is a programming language for symbolic, non-numeric computation. It is especially well suited for solving problems that involve objects and relations between objects. For example, it is an easy exercise in prolog to express spatial relationship between objects, such as the blue sphere is behind the green one. It is also easy to state a more general rule: if object X is closer to the observer than object Y. and object Y is closer than Z, then X must be closer than Z. PROLOG can reason about the spatial relationships and their consistency with respect to the general rule. Features like this make PROLOG a powerful language for Artificia1 Language (AL) and non- numerical programming.  There are well-known examples of symbolic computation whose implementation in other standard languages took tens of pages of indigestible code, when the same algorithms were implemented in PROLOG, the result was a crystal-clear program easily fitting on one page.  **FACTS, RULES AND QUERIES**  Programming in PROLOG is accomplished by creating a database of facts and rules about objects, their properties, and their relationships to other objects. Queries then can be posed about the objects and valid conclusions will be determined and returned by the program Responses to user queries are determined through a form of inference control known as resolution. To define and modify knowledge, you work with facts, rules, and queries in the logic programming language Prolog. Horn clauses are a type of symbolic logic used in Prolog. These core ideas are explained as follows:Details  Facts in Prolog are statements in the form of predicates that characterize some fundamental information or relationships. Things that are true are asserted using facts.  A predicate and a collection of arguments make up a fact. For instance:  **male(john).**  **female(lisa).**  **parent(john, lisa).**  **RULES**  Relationships and conclusions drawn from facts and other rules are expressed using rules. They are made up of a body and a head.  The conclusion or intended inference is the head of a rule.  The requirements or subgoals that must be met in order for a rule to be applied make up its body.  Here is an illustration of a rule:  **father(X, Y) :- male(X), parent(X, Y).**  In this rule, "X is the father of Y" if X is male, and X is a parent of Y.  The main method of communicating with a Prolog system is through queries. They let you pose queries or ask questions in accordance with the established guidelines and facts.  In order to obtain answers, Prolog will try to combine the goals that are provided for each query with the facts and rules.  As an illustration, you may question something like:  **?- father(john, lisa).**  Based on the established rules and data, Prolog will attempt to determine whether or not John is Lisa's father in response to this query.  This is an example that shows how to use facts, rules, and queries in Prolog in its entirety:  **% Facts**  **male(john).**  **female(lisa).**  **parent(john, lisa).**  **% Rules**  **father(X, Y) :- male(X), parent(X, Y).**  **mother(X, Y) :- female(X), parent(X, Y).**  **% Query**  **?- father(john, lisa). % This will return true.**  In order to determine if the question is true or false, Prolog will analyze it by searching for a set of facts and rules that match the query. It accomplishes this through a procedure known as "resolution". In the event that a solution is found, it will also yield variable values.  **PROLOG IN DESIGNING EXPERT SYSTEMS**  An expert system is a collection of programs designed to manipulate knowledge that has been encoded in order to solve issues in a specific field where human competence is typically needed. Knowledge for an expert system is gathered from authoritative sources like texts, journal articles, databases, etc. and encoded in a format that the system can utilize for inference or reasoning. After acquiring a sufficient body of expert knowledge, it must be encoded in some way, loaded into a knowledge base, tested, and continually improved during the system's lifespan. PROLOG is an effective language for creating expert systems due to the following characteristics.   * + Use of knowledge rather than data   + Modification of the knowledge base without recompilation of the control programs.   + Capable of explaining conclusion.   + Symbolic computations resembling manipulations of natural language.   + Reason with meta-knowledge.   **META PROGRAMMING**  A program that uses other programs as data is called a meta-program. Meta-programs include, for example, interpreters and compilers. One type of meta-program is a meta-interpreter, which is an interpreter written in a language for another language. Hence, an interpreter for PROLOG that is written in PROLOG is called a PROLOG interpreter. PROLOG's ability to manipulate symbols makes it an effective language for meta-programming. As a result, it is frequently employed as a language for language implementation. When it comes to fast implementing new ideas, PROLOG is an especially good language for rapid prototyping. Innovative concepts are tried out and implemented quickly.  1)Programe  red(rose).  likes(bill ,cindy).  owns(john ,gold).  Output:  Goal  queries  ?-likes(ram,What).  What= mango  ?-likes(Who,cindy).  Who= cindy  ?-red(What).  What= rose  ?-owns(Who,What).  Who= john  What= gold.  2) Program:  Production rules:  Arithmetic:  c\_to\_f f is c \* 9 / 5 +32  freezing f < = 32  Rules:  c\_to\_f(C,F) :-  F is C \* 9 / 5 + 32.  freezing(F) :-  F =< 32.  Output:  Queries:  ?- c\_to\_f(100,X).  X = 212  Yes  ?- freezing(15)  .Yes  ?- freezing(45).  No  This practical demonstrated the basic concepts of Prolog, a declarative programming language for representing knowledge and reasoning about it. You learned how to express facts and relationships using predicates, query and manipulate knowledge, and apply Prolog to various applications. |

|  |  |
| --- | --- |
| Shri Vile Parle Kelavani Mandal's  **INSTITUTE OF TECHNOLOGY**  **DHULE (M.S.)**  **DEPARMENT OF COMPUTER ENGINEERING** | |
| **Name**: Anushka Paras Jain **Roll No**.: 01  **Subject :** Artificial Intelligence Lab **Subject Code :** BTCOL707  **Class:** Final Year Comp. Engg. **Expt. No. :** 02  **Title :** Write a program to solve 8 queens problem. | |
| **Problem Statement :**  **Software Required :**  **Theory :**  **Conclusion:** | Solve 8 queens problem using prolog.  Prolog  The objective of the 8-queens problem, a traditional chessboard puzzle, is to arrange eight queens on an 8x8 chessboard so that no two queens pose a threat to one another. This implies that no two queens may be situated in the same diagonal, column, or row. This is a written example of how to solve the 8-queens problem:  . Q . . . . . .  . . . . Q . . .  . . . . . . . Q  . . . . . Q . .  . . Q . . . . .  . . . . . . Q .  . . . Q . . . .  Q . . . . . . .  In this instance, the eight queen places on the chessboard are represented by the 'Q' characters. As you can see, no row, column, or diagonal is occupied by more than one queen.  Finding every queen configuration on the chessboard that satisfies the non-attack constraint—that is, ensuring that no two queens pose a threat to one another—is the first step in solving the eight-queens issue. Because it supports both constraint logic programming and logic programming, Prolog is a popular language for handling this kind of problem.  The task of solving the 8-queens issue in Prolog entails figuring out how to arrange eight queens on an 8x8 chessboard so that no two of them pose a threat to one another. This implies that no two queens may be situated in the same diagonal, column, or row. A Prolog program to resolve the 8-queens problem is provided here:  % Predicate to check if a queen can be placed safely in a given row and column.  is\_safe(\_, []).  is\_safe(Queen, [Row/Col|Queens]) :-  Queen #\= Row,  Queen #\= Col,  abs(Queen - Row) #\= abs(Col - Queens),  is\_safe(Queen, Queens).  % Predicate to find a solution for N-queens.  queens(N, Solution) :-  length(Solution, N),  Solution ins 1..N, % Initialize the domain of the variables.  all\_distinct(Solution), % Ensure queens are placed in different columns.  is\_safe(1, Solution), % Check safety of placement for each queen.  labeling([], Solution). % Find a valid labeling.  % Predicate to print a solution.  print\_solution(Solution) :-  length(Solution, N),  write('Solution: '), writeln(Solution),  draw\_board(Solution, N, N).  % Predicate to draw the chessboard.  draw\_board(\_, 0, \_).  draw\_board(Solution, Row, N) :-  Row > 0,  draw\_row(Solution, Row, N),  NextRow is Row - 1,  draw\_board(Solution, NextRow, N).  draw\_row([], \_, 0).  draw\_row([Col|Queens], Row, N) :-  N > 0,  (Col =:= Row -> write('Q ') ; write('. ')),  NextN is N - 1,  draw\_row(Queens, Row, NextN).  % Predicate to solve and print all solutions.  all\_solutions(N) :-  findall(Solution, queens(N, Solution), Solutions),  length(Solutions, NumSolutions),  write('Number of solutions: '), writeln(NumSolutions),  maplist(print\_solution, Solutions).  % Example usage: Solve the 8-queens problem and print all solutions.  :- all\_solutions(8).  Within this course:  is\_safe/2 determines if a queen may be positioned on the board securely and without endangering other queens.  queens/2, where N is the number of queens, specifies the primary predicate for resolving the N-queens issue.  To print the solutions in a format that is legible by humans, use print\_solution/1.  The chessboard with the queens arranged is drawn using the draw\_board/3 and draw\_row/3 functions.  A helpful predicate called all\_solutions/1 locates and outputs every solution to the N-queens problem.  The 8 queens’ problem was solved using Prolog's declarative nature and built-in reasoning mechanisms. The chessboard was represented as a list of lists, and the positions of queens were represented using predicates. Prolog's member predicate was employed to check for attacking queens. |

|  |  |
| --- | --- |
| Shri Vile Parle Kelavani Mandal's  **INSTITUTE OF TECHNOLOGY**  **DHULE (M.S.)**  **DEPARMENT OF COMPUTER ENGINEERING** | |
| **Name:** Anushka Paras Jain **Roll no**: 01    **Subject :** Artificial Intelligence Lab **Subject Code :** BTCOL707  **Class:** Final Year Comp. Engg. **Expt. No. :** 03  **Title :** Solve any problem using depth first search. | |
| **Problem Statement:**  **Software Required:**  **Theory:**  **Conclusion:** | Solve any problem using depth first search.    Prolog  A basic search technique called Depth-First Search (DFS) is used in computer science and artificial intelligence to explore and navigate graphs and tree structures. It is frequently utilized in many different AI applications, such as knowledge representation, pathfinding, and problem solving involving state space exploration. Here's an illustration of a typical artificial intelligence application of DFS:  AI Pathfinding:  Assume you wish to determine the shortest route between a start point and a goal point in a maze or grid-based environment. In order to explore the grid, one can employ Depth-First Search, which involves recursively traveling as far down a path as feasible before turning around when there is no more way to proceed.  DFS(node):  if node is the goal:  return "Goal found!"  mark node as visited  for each neighbor of node:  if neighbor is not visited:  result = DFS(neighbor)  if result is "Goal found!":  return "Goal found!"  return "Goal not found"  In this case, DFS investigates the grid or maze by following a path as far as it can go before turning around to look at other possibilities. An indication of success is returned if the target is reached.  When you want to delve extensively into a search field before thinking about other options, DFS is especially helpful. It might not always provide the quickest path, though, and in some circumstances it might be prone to becoming caught in an endless cycle.  DFS can be used to navigate state spaces in more complicated AI applications, such as those involving planning, gaming, and decision-making. To further optimize the search process, it can be coupled with heuristics and other search algorithms.  % Define the edges of the graph.  edge(a, b).  edge(b, c).  edge(b, d).  edge(c, e).  edge(e, f).  edge(f, g).  % Define the DFS algorithm.  dfs(Start, End, Path) :-  dfs(Start, End, [Start], Path).  dfs(Current, End, Visited, Path) :-  Current == End,  reverse(Visited, Path).  dfs(Current, End, Visited, Path) :-  edge(Current, Next),  not(member(Next, Visited)),  dfs(Next, End, [Next | Visited], Path).  % Example usage:  % Find a path from 'a' to 'g'.  ?- dfs(a, g, Path).  We define a simple directed graph with edge/2 predicates in this Prolog code. The DFS search is started with the dfs/3 predicate. It returns the path after receiving the start and end nodes.  The recursive DFS search is the dfs/4 predicate. It searches for a route to the End node beginning from the Current node. The path is constructed and the visited nodes are tracked in the Visited list. It obtains the path by reversing the visited nodes when it reaches the End node.  By passing the start and finish nodes to dfs/3, you may use this code to find a path between any two nodes in the given graph.    This practical demonstrated the application of DFS in Prolog for solving combinatorial problems like maze navigation. The maze was represented as a list of lists, and valid positions were checked using a predicate. The goal state was identified, and the DFS algorithm was implemented to find a path from the start position to the goal. |

|  |  |
| --- | --- |
| Shri Vile Parle Kelavani Mandal's  **INSTITUTE OF TECHNOLOGY**  **DHULE (M.S.)**  **DEPARMENT OF COMPUTER ENGINEERING** | |
| **Name:** Anushka Paras Jain **Roll no.:** 01  **Subject :** Artificial Intelligence Lab **Subject Code :** BTCOL707  **Class:** Final Year Comp. Engg. **Expt. No. :** 04  **Title :** Solve any problem using best first search. | |
| **Problem Statement:**  **Software Required :**  **Theory :**  **Conclusion:** | Solve any problem using best first search.    Prolog  A broad search algorithm called Best-First Search (BFS) uses a heuristic evaluation function to guide it across a search space. Let's use Prolog's Best-First Search technique to solve the well-known "8-puzzle" problem. To get from a starting state to a target state in the 8-puzzle, you must rearrange numbered tiles in a 3x3 grid.  % Define the initial state and goal state  initial\_state([2, 8, 3, 1, 6, 4, 7, 0, 5]).  goal\_state([1, 2, 3, 8, 0, 4, 7, 6, 5]).  % Define the heuristic function (Manhattan distance)  heuristic(State, H) :-  goal\_state(Goal),  findall(D, (nth1(I, State, Tile), nth1(I, Goal, GoalTile), manhattan(Tile, GoalTile, D)), Distances),  sum\_list(Distances, H).  manhattan(X/Y, X1/Y1, D) :-  D is abs(X - X1) + abs(Y - Y1).  % Operators to move tiles  move(State, NewState) :-  select(0, State, X, TempState),  select(T, TempState, 0, NewTempState),  append([X, T], NewTempState, NewState).  % Define a predicate to solve the puzzle using Best-First Search  solve\_best\_first(State, State, [], \_).  solve\_best\_first(CurrentState, GoalState, [Action | Actions], Visited) :-  findall((NewState, Action, H), (  move(CurrentState, NewState),  \+ member(NewState, Visited),  heuristic(NewState, H)  ), Successors),  keysort(Successors, SortedSuccessors),  member((NextState, Action, \_), SortedSuccessors),  solve\_best\_first(NextState, GoalState, Actions, [NextState | Visited]).  % Entry point to solve the puzzle  solve\_puzzle :-  initial\_state(InitialState),  goal\_state(GoalState),  solve\_best\_first(InitialState, GoalState, Actions, [InitialState]),  write('Solution Actions: '), nl,  print\_actions(Actions).  % Predicate to print the sequence of actions  print\_actions([]).  print\_actions([Action | Rest]) :-  print\_state(Action),  print\_actions(Rest).  % Predicate to print a single state  print\_state([A, B, C, D, E, F, G, H, I]) :-  format('~d ~d ~d~n~d ~d ~d~n~d ~d ~d~n', [A, B, C, D, E, F, G, H, I]).  % Start the solver  :- solve\_puzzle.  This code solves the 8-puzzle problem using Best-First Search with the Manhattan distance heuristic. It calculates the heuristic value for each state and explores the states with the lowest heuristic values first. The solve\_puzzle predicate initiates the search and prints the sequence of actions to reach the goal state from the initial state.  This practical illustrated the application of BFS in Prolog for solving pathfinding problems. The graph was represented as a list of lists, and the Manhattan distance heuristic was used to estimate the distance between two positions. The BFS algorithm was implemented to find the shortest path from the start position to the goal, ensuring optimal solutions. |

|  |  |
| --- | --- |
| Shri Vile Parle Kelavani Mandal's  **INSTITUTE OF TECHNOLOGY**  **DHULE (M.S.)**  **DEPARMENT OF COMPUTER ENGINEERING** | |
| **Name:** Anushka Paras Jain **Roll No**.:01  **Subject :** Artificial Intelligence Lab **Subject Code :** BTCOL707  **Class:** Final Year Comp. Engg. **Expt. No. :** 05  **Title :** Solve 8-puzzle problem using best first search. | |
| **Problem Statement:**  **Software Required:**  **Theory:**  **Conclusion:** | Solve 8-puzzle problem using best first search.    Prolog  When employing the best-first search algorithm to solve the 8-puzzle issue, a heuristic function is usually needed to direct the search. For this, the most popular best-first search algorithm is A\* search. Here is a Prolog code that uses A\* search to solve the 8-puzzle problem:  % Define the initial state and the goal state  initial state([1, 2, 3, 8, 0, 4, 7, 6, 5]).  goal state([1, 2, 3, 8, 0, 4, 7, 6, 5]).  % Define the heuristic function (Manhattan distance)  heuristic(State, H) :-  goal state(Goal),  find all(D, (nth1(I, State, Tile), nth1(I, Goal, Goal Tile), Manhattan(Tile, Goal Tile, D)), Distances),  sum list(Distances, H).  Manhattan(X/Y, X1/Y1, D) :-  D is abs(X - X1) + abs(Y - Y1).  % Operators to move tiles  move(State, New State) :-  select(0, State, X, TempState),  select(T, TempState, 0, NewTempState),  append([X, T], NewTempState, NewState).  % Define a predicate to solve the puzzle using A\*  solve\_astar(InitialState, Actions) :-  heuristic(InitialState, H),  astar([(InitialState, [])], H, [], Actions).  astar([], \_, \_, []) :- !, fail.  astar(States, \_, Visited, Actions) :-  select((State, Actions), States, RestStates),  goal\_state(State),  reverse(Actions, Actions).  astar(States, H, Visited, Actions) :-  select((State, Actions), States, RestStates),  findall((NewState, [Move | Actions]),  (move(State, Move), \+ member(Move, Visited), heuristic(Move, H1), H2 is H1 + length(Actions), NewState = (Move, [Move | Actions])),  NewStates),  append(NewStates, RestStates, AllStates),  sort(AllStates, SortedStates),  astar(SortedStates, H, [State | Visited], Actions).  % Entry point to solve the puzzle  solve\_puzzle :-  initial\_state(InitialState),  solve\_astar(InitialState, Actions),  write('Solution: '), nl,  print\_actions(Actions).  % Predicate to print the sequence of actions  print\_actions([]).  print\_actions([Action | Rest]) :-  print\_state(Action),  print\_actions(Rest).  % Predicate to print a single state  print\_state([A, B, C, D, E, F, G, H, I]) :-  format('~d ~d ~d~n~d ~d ~d~n~d ~d ~d~n', [A, B, C, D, E, F, G, H, I]).  % Start the solver  :- solve\_puzzle.  This code finds the best solution to the 8-puzzle issue by combining the Manhattan distance heuristic with the A\* search method. The search is guided by the heuristic predicate, which calculates the Manhattan distance between the current state and the objective state.  The 8-puzzle problem was solved using best-first search (BFS) in Prolog. The initial and goal states were defined, and the Manhattan distance heuristic was used to estimate the distance between states. The BFS algorithm was implemented to find the shortest path from the initial state to the goal, ensuring an optimal solution. |

|  |  |
| --- | --- |
| Shri Vile Parle Kelavani Mandal's  **INSTITUTE OF TECHNOLOGY**  **DHULE (M.S.)**  **DEPARMENT OF COMPUTER ENGINEERING** | |
| **Name:** Anushka Paras Jain **Roll No.:** 01  **Subject :** Artificial Intelligence Lab **Subject Code :** BTCOL707  **Class:** Final Year Comp. Engg. **Expt. No. :** 06  **Title :** Solve Robot (traversal) problem using means End Analysis. | |
| **Problem Statement:**  **Software Required:**  **Theory:**  **Conclusion:** | Solve Robot (traversal) problem using means End Analysis    Prolog  The traversal problem for a robot is one of the many challenges that can be solved using the Means-End Analysis (MEA), a popular approach in artificial intelligence. In this case, the robot must use all available tools or actions to go from its current position to a target place. We'll use a streamlined Prolog example to demonstrate the idea of MEA for a robot traversal problem.  Assume you have a straightforward grid environment with movable robot that can move left, right, up, and down, and that its objective is to go to a predetermined spot. This is a sample of Prolog code that illustrates a fundamental MEA solution for this issue:  % Define the initial state (robot's current position)  initial\_state(2, 2). % Assuming the robot starts at position (2, 2).  % Define the goal state (target location)  goal\_state(5, 5). % Assuming the goal is to reach position (5, 5).  % Define the means or actions (robot's movement commands)  move(up, 0, -1).  move(down, 0, 1).  move(left, -1, 0).  move(right, 1, 0).  % Define the MEA algorithm  mea(State, State, []).  mea(CurrentState, GoalState, [Action | Actions]) :-  move(Action, DX, DY),  NewX is CurrentStateX + DX,  NewY is CurrentStateY + DY,  mea((NewX, NewY), GoalState, Actions).  % Entry point to solve the traversal problem using MEA  solve\_traversal :-  initial\_state(CurrentStateX, CurrentStateY),  goal\_state(GoalStateX, GoalStateY),  mea((CurrentStateX, CurrentStateY), (GoalStateX, GoalStateY), Actions),  write('Solution Actions: '), nl,  print\_actions(Actions).  % Predicate to print the sequence of actions  print\_actions([]).  print\_actions([Action | Rest]) :-  write(Action), nl,  print\_actions(Rest).  % Start the solver  :- solve\_traversal.  In this example, we specify the robot's starting point, destination, and possible moves. The MEA algorithm is implemented by the mea predicate. It computes a series of actions (movements) to accomplish the objective given the current state (the robot's position) and the goal state (the target location).  The starting position, the desired state, and the possible moves can all be changed to fit your unique robot traversal scenario. This is a condensed example of how MEA can be used in Prolog to solve such an issue. The grid world may be more complicated and require consideration of extra constraints in real-life circumstances.  The Robot (traversal) problem was solved using means-ends analysis (MEA). MEA breaks down the problem into subproblems, identifies operators to reduce the differences between states, and repeats until the goal state is reached. This approach provides a systematic solution to the robot navigation problem. |

|  |  |
| --- | --- |
| Shri Vile Parle Kelavani Mandal's  **INSTITUTE OF TECHNOLOGY**  **DHULE (M.S.)**  **DEPARMENT OF COMPUTER ENGINEERING** | |
| **Name:** Anushka Paras Jain **Roll No.:** 01  **Subject :** Artificial Intelligence Lab **Subject Code :** BTCOL707  **Class:** Final Year Comp. Engg. **Expt. No. :** 07  **Title :** Solve traveling salesman problem. | |
| **Problem Statement:**  **Software Required:**  **Theory:**  **Conclusion:** | Solve traveling salesman problem.    Prolog  A salesman is given a list of locations in the Traveling Salesman issue (TSP), a classic optimization issue. His task is to determine the shortest route that visits each city exactly once and returns to the beginning city. The optimal solution of TSP for a large number of cities can be computationally costly. In this example, I'll give you a Prolog code that uses a brute-force method to solve a simple TSP instance. Remember that large instances of the problem are inefficient using this code.  % Define the cities and distances between them  distance(city1, city2, 10).  distance(city1, city3, 15).  distance(city1, city4, 20).  distance(city2, city3, 35).  distance(city2, city4, 25).  distance(city3, city4, 30).  % Create a list of cities  cities([city1, city2, city3, city4]).  % Predicate to calculate the total distance of a tour  tour\_distance([], 0).  tour\_distance([\_], 0).  tour\_distance([City1, City2 | Rest], TotalDistance) :-  distance(City1, City2, Dist),  tour\_distance([City2 | Rest], RestDistance),  TotalDistance is Dist + RestDistance.  % Predicate to find the shortest tour  shortest\_tour(ShortestTour, ShortestDistance) :-  cities(CityList),  permutation(CityList, Tour),  append(Tour, [Tour], ClosedTour),  tour\_distance(ClosedTour, Distance),  (ShortestDistance =< 0; Distance < ShortestDistance),  ShortestTour = Tour,  ShortestDistance = Distance.  % Entry point to solve the TSP  solve\_tsp :-  shortest\_tour(Tour, Distance),  write('Shortest tour: '), write(Tour), nl,  write('Shortest distance: '), write(Distance), nl.  % Start the solver  :- solve\_tsp.  Using the distance/3 predicate, we define the cities and their respective distances from one another.  A list of cities to visit is defined by the cities/1 predicate.  The tour\_distance/2 predicate is used to determine a tour's total distance.  The shortest\_tour/2 predicate determines the distance for each tour, creates all feasible combinations of the cities, and maintains track of the shortest tour's length.  The shortest tour and its distance are found and printed by the solve\_tsp predicate.  This code illustrates how to solve the TSP for a limited number of cities using a simple brute-force method. Near-optimal solutions are usually found for bigger instances using heuristics like the Christofides algorithm or the closest neighbor technique, or more efficient algorithms like dynamic programming and branch and bound.  Prolog's declarative nature can be used to represent the TSP and apply search algorithms to find solutions. Distances between cities are defined using predicates, and different permutations are explored using Prolog's backtracking mechanism. While this approach is simple, it becomes inefficient for large problems. |

|  |  |
| --- | --- |
| Shri Vile Parle Kelavani Mandal's  **INSTITUTE OF TECHNOLOGY**  **DHULE (M.S.)**  **DEPARMENT OF COMPUTER ENGINEERING** | |
| **Name**: Rushikesh Jagdish Sonwane **Roll No**.:63  **Subject :** Artificial Intelligence Lab **Subject Code :** BTCOL707  **Class:** Final Year Comp. Engg. **Expt. No. :** 08  **Title :** Drug discovery: Identifying potential drug candidates through Al-based simulations | |
| Problem Staement :  Software Required :  Theory :  **Conclusion:** | Drug discovery: Identifying potential drug candidates through Al-based simulations.    Prolog  Drug Discovery through AI-Based Simulations  Drug discovery involves the identification of potential drug candidates that can effectively treat a specific disease. AI-based simulations leverage machine learning algorithms and computational methods to analyze chemical structures and predict their biological activities. This accelerates the drug discovery process by narrowing down the potential candidates for further experimental validation.  Experiment Steps:   1. Data Collection: Gather a dataset of chemical compounds with known biological activities. 2. Data Preprocessing: Clean and preprocess the dataset to remove noise and irrelevant information. 3. Feature Extraction: Extract relevant features from the chemical structures to represent them in a machine-readable format. 4. Model Training: Train a machine learning model using the preprocessed data to learn the relationship between chemical features and biological activities. 5. Prediction: Use the trained model to predict the biological activities of new chemical compounds. 6. Candidate Selection: Identify potential drug candidates based on their predicted activities and prioritize them for further experimental validation.   % Prolog Code for Drug Discovery through AI-Based Simulations  % Example Facts (Replace with your actual data)  compound(activity, features).  compound(activity, features).  % ... (more compounds)  % Data Preprocessing (Modify as needed)  preprocess\_data :-  % Implement data cleaning and preprocessing steps here  % Remove noise, handle missing values, etc.  % Feature Extraction (Modify as needed)  extract\_features(Compound, Features) :-  % Implement feature extraction based on compound structure  % Convert chemical structures into machine-readable features  % Model Training (Replace with your actual machine learning model)  train\_model :-  % Implement model training using machine learning techniques  % Use the preprocessed data and extracted features  % Prediction (Replace with your actual prediction logic)  predict\_activity(Compound, PredictedActivity) :-  % Use the trained model to predict the biological activity  % Return the predicted activity for the given compound  % Candidate Selection (Modify as needed)  select\_candidates :-  % Implement logic to identify potential drug candidates  % Prioritize compounds based on predicted activities  % Main Execution  drug\_discovery :-  preprocess\_data,  train\_model,  % Example: Predict activity for a compound  predict\_activity(compound1, PredictedActivity),  write('Predicted Activity for compound1: '), write(PredictedActivity), nl,  % Implement further steps such as candidate selection  select\_candidates.  % Example Usage  :- drug\_discovery.  This experiment explores the integration of Artificial Intelligence into the drug discovery process. By leveraging AI-based simulations, researchers can efficiently identify potential drug candidates, saving time and resources in the early stages of drug development. The application of machine learning in drug discovery continues to revolutionize the pharmaceutical industry. |

|  |  |
| --- | --- |
| Shri Vile Parle Kelavani Mandal's  **INSTITUTE OF TECHNOLOGY**  **DHULE (M.S.)**  **DEPARMENT OF COMPUTER ENGINEERING** | |
| **Name**: Rushikesh Jagdish Sonwane **Roll No**.:63  **Subject :** Artificial Intelligence Lab **Subject Code :** BTCOL707  **Class:** Final Year Comp. Engg. **Expt. No. :** 09  **Title :** Non-player characters (NPCs): Creating intelligent Al-controlled characters in games. | |
| Problem Staement :  Software Required :  Theory :  **Conclusion:** | Non-player characters (NPCs): Creating intelligent Al-controlled characters in games.    Prolog  Intelligent AI-Controlled NPCs in Games  Non-Player Characters (NPCs) play a crucial role in enhancing the overall gaming experience. Creating intelligent AI-controlled NPCs involves implementing algorithms that simulate human-like behavior, decision-making, and interactions within the game environment. Various techniques, such as Finite State Machines, Behavior Trees, and Machine Learning, can be employed to enhance the realism and responsiveness of NPCs.  Experiment Steps:   1. NPC Behavior Design: Define the desired behaviors and interactions of NPCs in the game. 2. Decision-Making Algorithms: Implement decision-making algorithms to enable NPCs to make informed choices based on the game context. 3. Finite State Machines (FSM): Use FSM to model different states and transitions in NPC behavior. 4. Behavior Trees: Implement Behavior Trees to represent complex decision-making processes and actions. 5. Pathfinding Algorithms: Integrate pathfinding algorithms to enable NPCs to navigate the game environment intelligently. 6. Testing and Evaluation: Test the NPCs in various game scenarios to evaluate their responsiveness and behavior.   % Prolog Code for AI-controlled NPCs in Games  % Example Facts (Replace with your actual game data)  player\_health(70).  enemy\_health(50).  % ... (more game-related facts)  % Decision-Making Rules (Modify and expand based on your game logic)  make\_decision(AttackDecision) :-  player\_health(PlayerHealth),  enemy\_health(EnemyHealth),    % Rule 1: Attack if player health is lower than enemy health  PlayerHealth < EnemyHealth,  AttackDecision = attack.  make\_decision(AttackDecision) :-  player\_health(PlayerHealth),  enemy\_health(EnemyHealth),    % Rule 2: Retreat if player health is higher than enemy health  PlayerHealth >= EnemyHealth,  AttackDecision = retreat.  % Main NPC Behavior Loop  npc\_behavior\_loop :-  % Implement game loop logic here  % Call the decision-making function and perform actions accordingly  make\_decision(Decision),  perform\_action(Decision),  % Continue the loop...  % Perform Action (Replace with your actual game actions)  perform\_action(attack) :-  write('NPC decides to attack the player!'), nl.    perform\_action(retreat) :-  write('NPC decides to retreat from the player!'), nl.  % Example Usage  :- npc\_behavior\_loop.  This experiment explores the fascinating realm of creating intelligent AI-controlled NPCs in games. The implementation of diverse algorithms and techniques allows game developers to enhance the realism and engagement of NPCs, contributing significantly to the overall gaming experience. Understanding the principles of AI in game development opens up avenues for creating immersive and dynamic virtual worlds.. |

|  |  |
| --- | --- |
| Shri Vile Parle Kelavani Mandal's  **INSTITUTE OF TECHNOLOGY**  **DHULE (M.S.)**  **DEPARMENT OF COMPUTER ENGINEERING** | |
| **Name**: Rushikesh Jagdish Sonwane **Roll No**.:63  **Subject :** Artificial Intelligence Lab **Subject Code :** BTCOL707  **Class:** Final Year Comp. Engg. **Expt. No. :** 10  **Title :** To analyze climate data and make predictions for weather forecasting and climate modeling | |
| Problem Staement :  Software Required :  Theory :  **Conclusion:** | To analyze climate data and make predictions for weather forecasting and climate modeling  Prolog  Climate Data Analysis and Prediction  Climate data analysis involves processing large datasets containing information about various climate parameters such as temperature, humidity, wind speed, and precipitation. AI techniques, including Machine Learning (ML) and Deep Learning (DL), can be applied to analyze historical data, identify patterns, and create predictive models. These models can then be used for weather forecasting and climate modeling.  Experiment Steps:   1. Data Collection: Obtain historical climate data from reliable sources or APIs. 2. Data Preprocessing: Clean and preprocess the data, handle missing values, and format it for model training. 3. Feature Selection: Identify relevant features for prediction, considering parameters like temperature, humidity, and wind speed. 4. Model Selection: Choose appropriate AI models for climate prediction (e.g., Neural Networks, LSTM, or regression models). 5. Training the Model: Train the selected model using historical climate data. 6. Model Evaluation: Evaluate the model's performance using validation data and metrics like Mean Absolute Error (MAE) or Root Mean Squared Error (RMSE). 7. Prediction: Use the trained model to make predictions for future weather conditions. 8. Visualization: Visualize the predicted results and compare them with actual data to assess the model's accuracy.   % Prolog Code for Weather Prediction  % Example Facts (Replace with actual climate data)  temperature(today, 25).  temperature(yesterday, 22).  humidity(today, 60).  humidity(yesterday, 55).  % ... (more climate-related facts)  % Rules for Weather Prediction (Modify based on your climate model)  predict(weather, today) :-  temperature(today, Temp),  humidity(today, Humidity),    % Rule 1: If temperature is high and humidity is high, predict "Hot and Humid"  Temp > 30, Humidity > 70,  write('Weather Prediction: Hot and Humid'), nl.  predict(weather, today) :-  temperature(today, Temp),  humidity(today, Humidity),    % Rule 2: If temperature is moderate and humidity is moderate, predict "Mild"  Temp >= 20, Temp =< 30, Humidity >= 40, Humidity =< 70,  write('Weather Prediction: Mild'), nl.  predict(weather, today) :-  % Rule 3: Default prediction if no specific rules apply  write('Weather Prediction: Not enough data for a specific prediction.'), nl.  % Example Usage  :- predict(weather, today).  This experiment provides hands-on experience in applying AI techniques to climate data analysis and prediction. Understanding how to preprocess data, select features, choose suitable models, and evaluate predictions is crucial for developing effective weather forecasting and climate modeling systems. The integration of AI in climate science contributes to more accurate and efficient predictions, aiding in better decision-making for various applications. |