
The program mainly consists of 9 predicates:

- agents_distance(+Agent1, +Agent2, -Distance): Calculates the Manhattan distance between two agents.
- number_of_agents(+State, -NumberOfAgents): Counts the number of agents in the farm.
- value_of_farm(+State, -NumberOfAgents): Calculates the value of the farm given the value of each agent and object in the farm.pro file.
- find_food_coordinates(+State, +AgentId -Coordinates): Finds the coordinates of the foods that the agent can eat and instantiates the Coordinates variable with the list of coordinates.
- find_nearest_agent(+State, +AgentId, -Coordinates, -NearestAgent): Finds the nearest agent to the given agent.
- find_nearest_food(+State, +AgentId, -Coordinates, -FoodType, -Distance): Finds the nearest food to the given agent. In this predicate it doesn't matter if the food is reachable or not.
- move_to_coordinate(+State, +AgentId, +X, +Y, -ActionList, +DepthLimit): Moves the agent to the given coordinate if possible. Other agents act as obstacles. (Wolf can move through other agents except other wolves).
- move to nearest food(+State, +AgentId, -ActionList, +DepthLimit): Moves the agent to the nearest food if possible.
- consume all(+State, +AgentId, -NumberOfMovements, -Value, -NumberOfChildren, +DepthLimit): Consumes all the foods that the agent can eat. Tries the nearest food first. If at any point the food is not reachable, tries to move to the next nearest food. If the agent has enough food, it reproduces. Implemented breadth-first search algorithm to find the shortest path to the food. The algorithm is limited by the DepthLimit parameter. If the food is not reachable within the DepthLimit, the agent doesn't move.

---

1. Ideation Process

Problem Understanding

The farm simulation project required implementing a Prolog-based system to model agent interactions in a grid-based environment.

Design Philosophy

My approach focused on breaking down complex behaviors into smaller, manageable predicates. I chose to implement:

- **Modular Design**: Each predicate handles a specific aspect of the simulation

- **Helper Functions**: Custom implementations for list operations (quicksort, partition, etc.)
- **BFS Algorithm**: For optimal pathfinding in the consume_all predicate
- **State Representation**: Using Prolog dictionaries for clean agent and object management

Key Design Decisions

- **Distance Calculation**: Manhattan distance for grid-based movement
- **Search Strategy**: Breadth-first search for shortest path finding
- **Depth Limiting**: Preventing infinite recursion with DepthLimit parameters
- **Cut Operations**: Strategic use of cuts to prevent unwanted backtracking

---

2. Implementation Steps

Phase 1: Basic Helper Functions

Started with fundamental list operations:

- quicksort/2: For sorting distances when finding nearest elements
- get_nth_element/3, get_min_element/2: For accessing sorted results
- append_list/3: Custom list concatenation for compatibility

Phase 2: Core Distance and Counting Functions

Implemented basic farm analysis predicates:

- agents_distance/3: Manhattan distance calculation
- number_of_agents/2: Agent counting using dictionary operations
- value_of_farm/2: Summing values while filtering wolves

Phase 3: Search and Discovery Functions

Developed food and agent finding capabilities:

- find_food_coordinates/3: Comprehensive food location discovery
- find_nearest_agent/2: Distance-based nearest neighbor finding
- find_nearest_food/4: Multi-criteria food selection with backtracking

Phase 4: Movement and Navigation

Implemented pathfinding and movement logic:

- move_to_coordinate/6: Recursive movement with depth limiting
- move_to_nearest_food/4: Integration of finding and movement
- BFS algorithm for optimal pathfinding in complex scenarios

Phase 5: Advanced Consumption Logic

The most complex predicate consume_all/6:

- Iterative food consumption with state updates
- Shortest path calculation using BFS
- Resource management with move counting and value tracking

---

3. Challenges Encountered

Backtracking Control Managing Prolog's automatic backtracking behavior, especially in find_nearest_food/4, required strategic use of cuts. The challenge was maintaining the ability to find alternative solutions while preventing infinite loops.

State Management Maintaining consistent state throughout the simulation, particularly in consume_all/6, required careful parameter passing and state updates between recursive calls.

Complex Predicate Dependencies Some predicates relied heavily on others (move_to_nearest_food depends on find_nearest_food and move_to_coordinate), requiring careful testing to ensure all components worked together correctly.

Edge Case Handling Handling scenarios where no valid moves exist, unreachable foods, or empty lists required extensive conditional logic and error prevention.

---

4. Lessons Learned

Prolog Programming Insights

Declarative vs. Imperative Thinking This project reinforced the importance of thinking declaratively in Prolog. Instead of describing "how" to solve a problem, I learned to focus on "what" the solution should look like through logical relationships.

Cut Operator Usage Understanding when and where to use cuts became crucial. Overuse can eliminate useful backtracking, while underuse can lead to inefficient or incorrect solutions.

Built-in vs. Custom Predicates While Prolog provides many built-in predicates, implementing custom versions (like quicksort, append_list) provided deeper understanding of the underlying logic and better control over the behavior.

---

5. Conclusion

This project provided valuable experience in Prolog programming and logical problem-solving. The implementation successfully handles all required functionality while maintaining code clarity and efficiency. The modular approach allows for easy extension and modification, and the comprehensive commenting ensures maintainability.The most significant learning outcome was developing intuition for Prolog's unique programming paradigm and understanding how to leverage its strengths for complex problem-solving tasks.
