DAIIA Assignment Report

Course: Distributed Artificial Intelligence and Intelligent Agents

Assignment: Final Project - Mars Colony: The Evolution of Trust

Lorenzo Deflorian, Riccardo Fragale, Juozas Skarbalius

KTH Royal Institute of Technology

January 4, 2026

Contents

1. Running Instructions

- **How to run the program:**
  - Import the project files into GAMA platform
  - Navigate to the folder `src/project/models`
  - Open the file `MarsColony.gaml`
  - From the GAMA interface, click the play button to start the simulation
  - Use the simulation controls to adjust speed, pause, or reset as needed
- **Expected inputs:** No manual inputs are required. The simulation runs automatically with predefined parameters that can be modified in the global section if desired (e.g., desired number of agents per type, map dimensions, learning parameters).

2. General Overview

Solution Summary

This project implements a Mars Colony simulation where five different types of agents (Engineer, Medic, Scavenger, Parasite, and Commander) must survive in a hostile environment while managing biological needs (oxygen, energy, health) and learning to identify malicious actors through reinforcement learning.

The simulation demonstrates how a population can evolve a "social immune system" using Q-Learning to distinguish between cooperative agents and parasites that steal resources. Agents use BDI (Belief-Desire-Intention) architecture for survival behaviors and Q-Learning for social interaction decisions.

Key Features:

- Five distinct agent types with unique roles and behaviors
- Continuous simulation with supply shuttle system maintaining population
- BDI architecture for survival-oriented decision making
- Q-Learning for trust-based parasite detection
- FIPA communication for long-distance messaging (storm warnings)
- Real-time learning metrics visualization
- Behavioral metrics visualization (sociability, happiness, generosity)

Assumptions and Limitations:

- Death is currently disabled to allow learning to occur (can be re-enabled)

- Oxygen and energy refill plans are disabled to prioritize learning interactions
- Maximum colony size is capped at 60 agents
- Agents spawn at the habitat dome location, not the landing pad
- The variable `patience` is defined but not currently used in decision-making

3. Agent Types and Interactions

3.1 Explanation. The simulation features five distinct agent types, each with unique roles, capabilities, and interaction rules. All agents share three core biological traits: `oxygen_level`, `energy_level`, and `health_level`, which decrease over time and must be managed for survival. Each agent type has specific interaction rules that determine how they trade resources with other agents.

Note: In the current implementation, the BDI plans `get_oxygen` and `get_energy` are intentionally disabled to prioritize learning through social interactions. Oxygen/energy can still change due to environmental drain, role-specific trades (e.g., Engineers/Commanders), and the Med-Bay proximity "immunity" behavior.

Agent Types:

1. **Engineer:** Repairs broken oxygen generators. Provides oxygen (+10) or repairs generators during trades.
2. **Medic:** Heals other agents at the med-bay via a queue system. (Healing is performed at the Med-Bay, not through trading.)
3. **Scavenger:** Ventures into wasteland to mine resources. Provides raw materials (converted to +20 energy) during trades.
4. **Parasite:** Antagonist agent that steals resources without reciprocating. Uses `presented_role` to disguise as other types.
5. **Commander:** Broadcasts storm warnings via FIPA. Provides both oxygen (+10) and energy (+10) during trades.

Each agent type has at least one unique set of interaction rules. For example, Engineers repair generators, Medics heal at the med-bay, Scavengers mine resources, Parasites steal, and Commanders broadcast alerts.

3.2 Code. The agent types are defined as species inheriting from the `Human` base species. Each agent initializes with its role and presented role:

```
species Engineer parent: Human {
    init {
        do add_desire(wander_desire);
```

```
        role <- "Engineer";
        presented_role <- "Engineer";
    }
    // ... Engineer-specific behaviors
}

species Parasite parent: Human {
    init {
        do add_desire(wander_desire);
        role <- "Parasite";
        presented_role <- one_of(["Engineer", "Medic", "Scavenger", "Commander"]);
    }
}
```

The trading mechanism is implemented in the `attempt_trade` action, which handles different behaviors based on agent types:

```
action attempt_trade(Human partner) {
    bool i_gave <- false;
    bool partner_gave <- false;

    if (!(self is Parasite)) {
        if (self is Scavenger and raw_amount > 0) {
            raw_amount <- raw_amount - 1;
            i_gave <- true;
            ask partner { energy_level <- min(max_energy_level, energy_level + 20.0);
        }
        // ... other agent type behaviors
    }

    if (partner is Parasite) {
        partner_gave <- false;  // Parasites never give back
    }

    if (self is Parasite) {
        // Parasite performs harmful, one-sided interaction
        i_gave <- true;
        partner_gave <- false;
        // ... stealing logic
    }

    if (i_gave and partner_gave) { return 20.0; }
    if (i_gave and !partner_gave) { return ((partner is Parasite) ? -20.0 : -1.0); }
    return 0.0;
}
```

3.3 Demonstration.

1. **Input:** Simulation starts with all five agent types present in the habitat dome.
2. **Screenshot:** [PLACEHOLDER: Screenshot showing the main simulation view with all five agent types visible (Engineer in green, Medic in red, Scavenger in blue, Parasite in yellow, Commander in purple) within the habitat dome. The agent state inspector table should be visible showing different roles and states.]
3. **Interpretation:** The simulation successfully initializes with all required agent types. Each type is visually distinct and has appropriate starting locations and attributes.
4. **Input:** Agents engage in trading interactions within the habitat dome.
5. **Screenshot:** [PLACEHOLDER: Screenshot showing agents in close proximity within the dome, with the agent state inspector showing trade cooldown values, happiness levels, and trust_memory entries. The console should show trade log messages indicating successful trades between different agent types.]
6. **Interpretation:** Agents successfully interact with each other following their type-specific rules. Scavengers provide energy via raw resources, Engineers provide oxygen/repairs, Commanders provide oxygen+energy, and Parasites attempt to steal/drain. Medic contributions occur through Med-Bay queue healing (not via trade).

4. Environment and Places

4.1 Explanation. The simulation features multiple distinct places where agents can interact and perform activities. The environment includes safe zones (habitat dome) and danger zones (wasteland), each with different properties affecting agent survival.

Places:

1. **Habitat Dome:** Large safe zone (250x250 units) containing Greenhouse, Oxygen Generator, Med-Bay, and the two meeting places used for social interaction.
2. **Common Area:** Meeting/trading area (circle) inside the dome.
3. **Recreation Area:** Meeting/trading area (circle) inside the dome.
4. **Wasteland:** Danger zone (100x100 units) where Scavengers mine. Features 1.2x faster oxygen depletion and random dust storms that increase danger.
5. **Med-Bay:** Facility within the dome where injured agents queue for treatment. Features visual feedback (orange color when queue has patients).
6. **Landing Pad:** Location where agents retire and despawn.
7. **Rock Mine:** Location in wasteland where Scavengers collect raw materials.

4.2 Code. The places are defined as species with specific geometries and behaviors:

```
species HabitatDome {
    geometry shape <- rectangle(250, 250);
    Greenhouse greenhouse;
    OxygenGenerator oxygen_generator;
    MedBay med_bay;
    CommonArea common_area;
    RecreationArea recreation_area;

    init {
        location <- point(200, 200);
        shape <- shape at_location location;
        // Create and position facilities
        create Greenhouse number: 1 returns: greenhouses;
        greenhouse <- greenhouses[0];
        // ... initialize other facilities
    }
}

species Wasteland {
    geometry shape <- rectangle(100, 100);
    bool dust_storm <- false;
    int storm_timer <- 0;

    reflex manage_storm {
        if (dust_storm) {
            storm_timer <- storm_timer - 1;
            if (storm_timer <= 0) {
                dust_storm <- false;
            }
        } else {
            if (flip(0.01)) {
                dust_storm <- true;
                storm_timer <- 15;
            }
        }
    }
}
```

Oxygen depletion rates differ based on location:

```
reflex update_oxygen{
    if (habitat_dome.shape covers location) {
        oxygen_level <- max(0, oxygen_level - oxygen_decrease_rate);
    } else {
        float decrease <- oxygen_decrease_rate * oxygen_decrease_factor_in_wasteland;
```

```
        if (wasteland.dust_storm and (wasteland.shape covers location)) {
             decrease <- decrease * 2.0;
        }
        oxygen_level <- max(0, oxygen_level - decrease);
    }
}
```

4.3 Demonstration.

1. **Input:** Simulation displays the full map with all places visible.
2. **Screenshot:** [PLACEHOLDER: Screenshot showing the complete 400x400 map with habitat dome (green rectangle) at center, wasteland (red/brown rectangle) visible, landing pad (gray square), and all facilities within the dome clearly labeled. Agents should be visible within appropriate zones.]
3. **Interpretation:** All required places are correctly initialized and positioned. The habitat dome serves as the primary safe zone, while the wasteland represents a danger zone with different environmental properties.
4. **Input:** A dust storm occurs in the wasteland.
5. **Screenshot:** [PLACEHOLDER: Screenshot showing the wasteland area turned orange with "Wasteland (STORM)" label visible. Agents in the wasteland should be visible, and the console may show FIPA messages from the Commander.]
6. **Interpretation:** The wasteland correctly displays visual feedback during dust storms. The storm system affects oxygen depletion rates and triggers FIPA communication from Commanders.

5. Continuous Running System

5.1 Explanation. The simulation runs continuously by maintaining a target population through a supply shuttle system. When agents retire (after 2000 cycles) or die, new agents are spawned to maintain desired population levels. The system maintains at least 50 agents total across all types, with a maximum cap of 60 agents.

The supply shuttle operates in "deficit mode," spawning new agents when any agent type falls below its desired count:

- Engineers: 16 desired
- Medics: 10 desired
- Scavengers: 8 desired
- Parasites: 12 desired
- Commanders: 4 desired

Additionally, the supply shuttle aggregates Q-tables and trust memory from all survivors to calculate averages (preparing for potential knowledge inheritance by new agents).

5.2 Code. The supply shuttle reflex checks population counts and spawns new agents:

```
reflex supply_shuttle when: enable_supply_shuttle {
    int total_colonists <- length(list(Engineer) + list(Medic) +
                                  list(Scavenger) + list(Parasite) +
                                  list(Commander));

    bool deficit_mode <- (current_number_of_engineers < desired_number_of_engineers) o
                         (current_number_of_medics < desired_number_of_medics) or
                         (current_number_of_scavengers < desired_number_of_scavengers
                         (current_number_of_parasites < desired_number_of_parasites)
                         (current_number_of_commanders < desired_number_of_commanders

    if (deficit_mode) {
        int max_to_spawn <- max_colony_size - total_colonists;

        // Spawn engineers if needed
        if (delta_engineers > 0 and max_to_spawn > 0) {
            int to_spawn <- min(delta_engineers, max_to_spawn);
            create Engineer number: to_spawn returns: new_engineers;
            ask new_engineers {
                location <- habitat_dome.location;
                oxygen_level <- max_oxygen_level;
                energy_level <- max_energy_level;
            }
            engineers <- engineers + new_engineers;
            current_number_of_engineers <- current_number_of_engineers + to_spawn;
            max_to_spawn <- max_to_spawn - to_spawn;
        }
        // ... similar logic for other agent types
    }
}
```

Retirement is handled through BDI:

```
reflex update_eta {
    if (not enable_retirement) { return; }
    eta <- eta + eta_increment;
    if (eta >= retirement_age) {
        do add_belief(should_retire_belief);
    }
}

plan do_retire intention: retire_desire {
```

```
    state <- "retiring";
    do goto target: landing_pad.location speed: movement_speed;

    if ((location distance_to landing_pad.location) <= facility_proximity) {
        do die_and_update_counter;
    }
}
```

5.3 Demonstration.

1. **Input:** Simulation runs for an extended period, allowing agents to reach retirement age (2000 cycles).
2. **Screenshot:** [PLACEHOLDER: Screenshot showing agent state inspector with ETA values approaching or exceeding 2000. Some agents should have state "retiring" and be moving toward the landing pad. Console should show retirement-related messages.]
3. **Interpretation:** The retirement system correctly tracks agent age and triggers retirement behavior when the threshold is reached. Agents successfully navigate to the landing pad to despawn.
4. **Input:** Agents retire, causing population counts to drop below desired levels.
5. **Screenshot:** [PLACEHOLDER: Screenshot showing the simulation after agents have retired. The console should show supply shuttle messages indicating new agents are being spawned. The agent state inspector should show newly created agents with default attributes at the habitat dome location.]
6. **Interpretation:** The supply shuttle system successfully detects population deficits and spawns new agents to maintain the desired population levels. The simulation continues running indefinitely.

6. FIPA Communication

6.1 Explanation. FIPA (Foundation for Intelligent Physical Agents) protocol is used for long-distance messaging between agents. In this simulation, Commanders send storm warning messages to all agents in the wasteland when a dust storm is detected. This demonstrates asynchronous, long-range communication that triggers behavioral changes in receiving agents.

When a dust storm occurs in the wasteland, Commanders detect it and broadcast a "Return to Base" message using the FIPA propose protocol. Agents receiving this message add a `storm_warning_belief` to their BDI system, which triggers the highest-priority desire (`escape_storm`) to return to the safe habitat dome.

6.2 Code. Commanders monitor for storms and send FIPA messages:

```
species Commander parent: Human {
    reflex check_storm {
        if (wasteland.dust_storm) {
            list<Human> agents_in_wasteland <- Human where
                (wasteland.shape covers each.location);
            if (!empty(agents_in_wasteland)) {
                do start_conversation to: agents_in_wasteland
                   protocol: "fipa-propose"
                   performative: "propose"
                   contents: ["Return to Base"];
            }
        }
    }
}
```

Agents receive and process FIPA messages:

```
reflex receive_message when: !empty(mailbox) {
    message msg <- first(mailbox);
    mailbox <- mailbox - msg;
    string msg_contents <- string(msg.contents);
    if (msg_contents contains "Return to Base") {
        if (not has_belief(storm_warning_belief)) {
            do add_belief(storm_warning_belief);
        }
    }
}
```

The BDI system processes the belief:

```
rule belief: storm_warning_belief new_desire: escape_storm_desire strength: 200.0;

plan escape_storm intention: escape_storm_desire {
    state <- "escaping_storm";
    do goto target: habitat_dome.location speed: movement_speed;

    if (habitat_dome.shape covers location) {
        do remove_belief(storm_warning_belief);
        do remove_intention(escape_storm_desire, true);
        state <- "idle";
    }
}
```

6.3 Demonstration.

1. **Input:** A dust storm occurs in the wasteland while agents (particularly Scavengers) are present there.

2. **Screenshot:** [PLACEHOLDER: Screenshot showing wasteland in storm state (orange color), with agents visible in the wasteland. The console should show FIPA messages from Commanders with "Return to Base" content. Agent state inspector should show some agents with state `"escaping_storm"`.]

3. **Interpretation:** Commanders successfully detect the storm and send FIPA messages to agents in the wasteland. Receiving agents correctly process the messages and add the storm warning belief.

4. **Input:** Agents receive storm warning messages and begin moving toward the habitat dome.

5. **Screenshot:** [PLACEHOLDER: Screenshot showing agents moving from wasteland toward the habitat dome. Agent state inspector should show agents with state `"escaping_storm"` and locations gradually approaching the dome. Console should show agents removing the storm warning belief upon reaching safety.]

6. **Interpretation:** The FIPA communication successfully triggers behavioral changes. Agents prioritize storm escape (strength 200.0, highest priority) and navigate to safety, demonstrating the integration between FIPA messaging and BDI architecture.

7. Charts and Monitoring

7.1 Explanation. The simulation includes real-time monitoring of learning metrics through charts and global values. The primary chart tracks the evolution of trust and parasite detection accuracy over time, demonstrating how the population learns to identify malicious actors.

Global Values and Metrics:

- `avg_trust_to_parasites`: Average trust value agents have toward parasites
- `avg_trust_to_non_parasites`: Average trust value agents have toward cooperative agents
- `precision`: True positives / (True positives + False positives) for parasite detection
- `recall`: True positives / (True positives + False negatives) for parasite detection
- `total_trades`: Counter of total trade interactions
- `avg_sociability`: Average learning-rate parameter across agents
- `avg_happiness`: Average accumulated reward from social interactions
- `avg_generosity`: Average giving/receiving balance tracked during trades

The chart displays these metrics over time, showing the learning evolution. The expectation is that trust toward parasites decreases over time while trust toward non-parasites remains positive, and precision/recall improve as agents learn.

Interesting Conclusion: The simulation demonstrates that a population utilizing Reinforcement Learning can develop a "Social Immune System," effectively identifying and isolating malicious actors (Parasites) without centralized police control, simply through individual negative experiences and learning.

7.2 Code. Learning metrics are updated each cycle:

```
reflex update_learning_metrics {
    int total_agents <- length(list(Engineer) + list(Medic) +
                               list(Scavenger) + list(Parasite) +
                               list(Commander));

    map<string, bool> is_parasite_by_id <- map([]);
    loop h over: (list(Engineer) + list(Medic) + list(Scavenger) +
                  list(Parasite) + list(Commander)) {
        is_parasite_by_id["id:" + h.name] <- (h is Parasite);
    }

    float sum_par <- 0.0; int cnt_par <- 0;
    float sum_non <- 0.0; int cnt_non <- 0;
    int tp0 <- 0; int fp0 <- 0; int fn0 <- 0; int tn0 <- 0;

    float sum_soc <- 0.0;
    float sum_hap <- 0.0;
    float sum_gen <- 0.0;

    loop h over: (list(Engineer) + list(Medic) + list(Scavenger) +
                  list(Parasite) + list(Commander)) {
        sum_soc <- sum_soc + h.sociability;
        sum_hap <- sum_hap + h.happiness;
        sum_gen <- sum_gen + h.generosity;

        loop k over: keys(h.trust_memory) {
            float v <- h.trust_memory[k];
            bool gt_par <- ((is_parasite_by_id contains_key k) ?
                            is_parasite_by_id[k] : false);
            if (gt_par) {
                sum_par <- sum_par + v;
                cnt_par <- cnt_par + 1;
            } else {
                sum_non <- sum_non + v;
                cnt_non <- cnt_non + 1;
            }

            bool pred_par <- v < 0.0;
            if (pred_par and gt_par) { tp0 <- tp0 + 1; }
            else if (pred_par and not gt_par) { fp0 <- fp0 + 1; }
```

```
            else if ((not pred_par) and gt_par) { fn0 <- fn0 + 1; }
            else { tn0 <- tn0 + 1; }
        }
    }

    avg_trust_to_parasites <- (cnt_par > 0 ? sum_par / float(cnt_par) : 0.0);
    avg_trust_to_non_parasites <- (cnt_non > 0 ? sum_non / float(cnt_non) : 0.0);
    precision <- ((tp0 + fp0) > 0 ? float(tp0) / float(tp0 + fp0) : 0.0);
    recall <- ((tp0 + fn0) > 0 ? float(tp0) / float(tp0 + fn0) : 0.0);

    avg_sociability <- (total_agents > 0 ? sum_soc / float(total_agents) : 0.0);
    avg_happiness <- (total_agents > 0 ? sum_hap / float(total_agents) : 0.0);
    avg_generosity <- (total_agents > 0 ? sum_gen / float(total_agents) : 0.0);
}
```

The chart is defined in the experiment:

```
display LearningMetrics {
    chart "Avg Trust & Detection" type: series {
        data 'Avg trust (parasites)' value: avg_trust_to_parasites;
        data 'Avg trust (non-parasites)' value: avg_trust_to_non_parasites;
        data 'Precision' value: precision;
        data 'Recall' value: recall;
    }
}

display BehavioralMetrics {
    chart "Sociability, Happiness & Generosity" type: series {
        data 'Avg sociability' value: avg_sociability;
        data 'Avg happiness' value: avg_happiness;
        data 'Avg generosity' value: avg_generosity;
    }
}
```

7.3 Demonstration.

1. **Input:** Simulation runs for an extended period to allow learning to occur.
2. **Screenshot:** [PLACEHOLDER: Screenshot showing the "Avg Trust & Detection" chart with four lines visible: average trust to parasites (expected to trend downward), average trust to non-parasites (expected to remain positive), precision (expected to increase), and recall (expected to increase). The chart should show evolution over time with clear trends visible.]
3. **Interpretation:** The learning metrics chart successfully tracks the evolution of trust and detection accuracy. Over time, agents learn to distinguish parasites from co-operative agents, as evidenced by decreasing trust toward parasites and improving precision/recall metrics.

4. **Input:** Agent state inspector shows trust memory values for different agents.
5. **Screenshot:** [PLACEHOLDER: Screenshot showing the agent state inspector table with trust_memory column visible. The trust memory should show negative values for parasite IDs and positive values for cooperative agent IDs. Console should show trade interactions and trust updates.]
6. **Interpretation:** Individual agents maintain trust memory maps that correctly distinguish between parasites (negative trust) and cooperative agents (positive trust). The learning system successfully updates trust values based on trade outcomes.

8. Challenge 1: BDI Agents

8.1 Explanation. The simulation implements a full BDI (Belief-Desire-Intention) architecture using GAMA's Simple BDI control. This architecture allows agents to make decisions based on their beliefs about the world, desires (goals) they want to achieve, and intentions (plans) to fulfill those desires.

Beliefs (Sensors): Agents maintain beliefs about their internal state and external environment:

- `suffocating_belief`: Triggered when oxygen level $< 20\%$
- `starving_belief`: Triggered when energy level $< 20\%$
- `injured_belief`: Triggered when health level $< 50\%$
- `should_retire_belief`: Triggered when ETA ≥ 2000 cycles
- `storm_warning_belief`: Triggered by FIPA messages from Commander
- `generator_broken_belief`: Engineer-specific, triggered when oxygen generator is broken
- `patients_waiting_belief`: Medic-specific, triggered when med-bay queue has patients
- `mission_time_belief`: Scavenger-specific, triggered with 20% probability per cycle

Desires (Goals): Desires are ranked by rule strengths, determining priority:

- `escape_storm` (200.0) - Highest priority
- `has_oxygen` (100.0) - High priority (plan currently disabled)
- `has_energy` (25.0) - Medium priority (plan currently disabled)
- `be_healthy` (12.0) - Medium-low priority
- `fix_generator` (9.0) - Engineer only

- `heal_patients` (7.0) - Medic only
- `retire` (6.0) - Low priority
- `mine_resources` (5.0) - Scavenger only
- `wander` (default) - Lowest priority

Intentions (Plans): Plans define the actions agents take to fulfill desires. Each plan is associated with a specific desire and executes until the desire is satisfied or the plan is completed.

8.2 Code. Beliefs are managed through perception reflexes:

```
reflex perception {
    if (oxygen_level < oxygen_level_threshold) {
        if (not has_belief(suffocating_belief)) {
            do add_belief(suffocating_belief);
        }
    } else {
        if (has_belief(suffocating_belief)) {
            do remove_belief(suffocating_belief);
        }
    }
    // Similar logic for starving_belief and injured_belief
}
```

Desires are created through rules that link beliefs to desires:

```
rule belief: storm_warning_belief new_desire: escape_storm_desire strength: 200.0;
rule belief: suffocating_belief new_desire: has_oxygen_desire strength: 100.0;
rule belief: starving_belief new_desire: has_energy_desire strength: 25.0;
rule belief: injured_belief new_desire: be_healthy_desire strength: 12.0;
rule belief: should_retire_belief new_desire: retire_desire strength: 6.0;
```

Plans execute intentions:

```
plan get_health intention: be_healthy_desire
      finished_when: health_level >= max_health_level {
    if ((location distance_to habitat_dome.med_bay.location) <= facility_proximity) {
        state <- "waiting_at_med_bay";
        ask habitat_dome.med_bay { do add_to_queue(myself); }
    } else {
        state <- "going_to_med_bay";
        do goto target: habitat_dome.med_bay.location speed: movement_speed;
    }
```

```
    if (health_level >= max_health_level) {
        ask habitat_dome.med_bay { do remove_from_queue(myself); }
        do remove_belief(injured_belief);
    }
}
```

Engineer-specific BDI behavior:

```
species Engineer parent: Human {
    predicate generator_broken_belief <- new_predicate("generator_broken");
    predicate fix_generator_desire <- new_predicate("fix_generator");

    reflex check_generator {
        if (habitat_dome.oxygen_generator.is_broken) {
            if (not has_belief(generator_broken_belief)) {
                do add_belief(generator_broken_belief);
            }
        }
    }

    rule belief: generator_broken_belief new_desire: fix_generator_desire strength: 9

    plan fix_generator intention: fix_generator_desire {
        if ((location distance_to habitat_dome.oxygen_generator.location) <= facility_
            habitat_dome.oxygen_generator.is_broken <- false;
            do remove_belief(generator_broken_belief);
            do remove_intention(fix_generator_desire, true);
            state <- "idle";
        } else {
            state <- "going_to_oxygen_generator";
            do goto target: habitat_dome.oxygen_generator.location speed: movement_spe
        }
    }
}
```

8.3 Demonstration.

1. **Input:** An agent's oxygen level drops below 20% threshold.
2. **Screenshot:** [PLACEHOLDER: Screenshot showing agent state inspector with an agent having `oxygen_level` < 20. Console should show the `suffocating_belief` being added and the `has_oxygen_desire` being created.]
3. **Interpretation:** The perception reflex correctly detects low oxygen and adds the `suffocating_belief`. The BDI system creates the `has_oxygen_desire` with high priority (100.0). In the current implementation, the corresponding plan `get_oxygen` is disabled to prioritize learning interactions.

4. **Input:** An agent's health drops below 50% threshold.

5. **Screenshot:** [PLACEHOLDER: Screenshot showing agent state inspector with an agent having `health_level` < 50. The agent should have state `"going_to_med_bay"` or `"waiting_at_med_bay"`. The med-bay should show the agent in its queue (orange color, queue count > 0). Console should show the agent adding itself to the med-bay queue.]

6. **Interpretation:** The agent correctly adds the `injured_belief`, creates the `be_healthy_desire`, and executes the `get_health` plan. The agent navigates to the med-bay and queues for treatment, demonstrating BDI-driven behavior with proper plan execution.

7. **Input:** The oxygen generator breaks (5% probability per cycle).

8. **Screenshot:** [PLACEHOLDER: Screenshot showing the oxygen generator displayed in red with "O2 Gen (BROKEN)" label. Engineer agents should have state `"going_to_oxygen_genera` or be at the generator location. Console should show engineers detecting the broken generator and adding the `generator_broken_belief`. After repair, the generator should return to blue color.]

9. **Interpretation:** Engineers correctly detect the broken generator through their `check_generator` reflex, add the `generator_broken_belief`, create the `fix_generator_desire`, and execute the repair plan. This demonstrates role-specific BDI behavior where only Engineers respond to generator failures.

10. **Input:** Multiple agents have different priority desires simultaneously (e.g., one suffocating, one injured, one with storm warning).

11. **Screenshot:** [PLACEHOLDER: Screenshot showing multiple agents with different states: one `"escaping_storm"` (highest priority), one `"going_to_med_bay"` (medium priority), one `"going_to_oxygen"` (high priority). The agent state inspector should show different agents prioritizing different tasks. Console should show the priority system in action.]

12. **Interpretation:** The BDI system correctly prioritizes desires based on rule strengths. The agent with storm warning (strength 200.0) prioritizes escape over other tasks, while agents with suffocating (strength 100.0) prioritize oxygen over injured agents (strength 12.0). This demonstrates the priority system working correctly across the population.

9. Challenge 2: Reinforcement Learning

9.1 Explanation. The simulation implements Q-Learning, a model-free reinforcement learning algorithm, to enable agents to learn which other agents are trustworthy through experience. Agents learn to identify parasites by tracking the outcomes of trade interactions and updating Q-values accordingly.

Q-Learning Implementation:

- **State:** Agent ID (e.g., "id:agent_name") - Each agent tracks trust per individual, not per type
- **Actions:** TRADE or IGNORE (selected via curiosity-greedy: 20% random exploration, 80% exploitation)
- **Rewards:**
  - Successful mutual trade: +20.0
  - Scammed by Parasite: -20.0
  - One-sided trade (non-parasite): -1.0
  - Ignore action: 0.0
- **Q-Value Update:** $Q(s,a) = Q(s,a) + sociability \times (reward - Q(s,a))$ where $sociability = 0.2$
- **Trust Calculation:** $trust = (Q[TRADE] - Q[IGNORE])/10.0$, clamped to $[-1.0, 1.0]$
- **Detection:** Trust < 0 indicates predicted parasite (used for precision/recall metrics)

Learning Evolution: Initially, agents interact randomly (curiosity = 20%). Parasites thrive and get rich through scams. Over time, Q-values for parasite interactions decrease due to negative rewards. Trust values drop toward -1.0 for parasites, and agents learn to IGNORE agents with negative trust values. The learning metrics track average trust to parasites vs non-parasites, plus precision/recall for parasite detection.

9.2 Code. Q-Learning state and action selection:

```
map<string, list<float>> Q <- map([]);
map<string, float> trust_memory <- map([]);

float sociability <- 0.2;
float patience <- 0.0;
float curiosity <- 0.20;

action ensure_state_exists(string s) {
    if (!(Q contains_key s)) {
        Q[s] <- [0.0, 0.0];
    }
    if (!(trust_memory contains_key s)) {
        trust_memory[s] <- 0.0;
    }
}

action choose_action(string s) {
    if (flip(curiosity)) {
```

```
        return (flip(0.5) ? "TRADE" : "IGNORE");
    }
    list<float> qs <- Q[s];
    return (qs[0] >= qs[1] ? "TRADE" : "IGNORE");
}
```

Q-value update and trust calculation:

```
action update_q(string s, string a, float r) {
    do ensure_state_exists(s);
    int idx <- (a = "TRADE" ? 0 : 1);

    list<float> qs <- Q[s];
    float old <- qs[idx];
    float updated <- old + sociability * (r - old);
    qs[idx] <- updated;
    Q[s] <- qs;

    float pref <- Q[s][0] - Q[s][1];
    trust_memory[s] <- max(-1.0, min(1.0, pref / 10.0));
}
```

Trade interaction with learning:

```
reflex learn_and_trade when:
    trade_cooldown = 0
    and (habitat_dome.common_area.shape covers location or habitat_dome.recreation_are
    and not has_belief(storm_warning_belief)
{
    list<Human> all_humans <- list(Engineer) + list(Medic) +
                              list(Scavenger) + list(Parasite) +
                              list(Commander);
    list<Human> nearby <- [];

    loop h over: all_humans {
        if (h != self) {
            float dist <- h.location distance_to location;
            if (dist <= meet_distance) {
                nearby <- nearby + [h];
            }
        }
    }

    if (empty(nearby)) { return; }

    Human partner <- one_of(nearby);
    string s <- "id:" + partner.name;
```

```
    do ensure_state_exists(s);

    string a <- choose_action(s);
    if (a = "IGNORE") {
        do update_q(s, a, 0.0);
        trade_cooldown <- trade_cooldown_max;
        return;
    }

    float reward <- attempt_trade(partner);
    do update_q(s, a, reward);
    happiness <- happiness + reward;
    total_trades <- total_trades + 1;
    trade_cooldown <- trade_cooldown_max;
}
```

9.3 Demonstration.

1. **Input:** Simulation starts with agents having empty Q-tables and trust memory. Agents begin trading with each other.
2. **Screenshot:** [PLACEHOLDER: Screenshot showing agent state inspector with trust_memory column showing initial values (mostly 0.0 or empty). Console should show trade interactions with rewards. The learning metrics chart should show initial trust values (around 0) for both parasites and non-parasites. Some trades should show positive rewards (+20) and some negative rewards (-20 from parasites).]
3. **Interpretation:** Agents initially interact randomly (20% exploration rate). Q-tables are being populated as agents encounter each other. Both positive (mutual trades) and negative (parasite scams) rewards are being recorded, starting the learning process.
4. **Input:** Simulation runs for several hundred cycles, allowing multiple trade interactions with the same agents.
5. **Screenshot:** [PLACEHOLDER: Screenshot showing agent state inspector with trust_memory containing negative values for parasite IDs and positive values for cooperative agent IDs. Console should show agents choosing IGNORE actions for agents with negative trust. The learning metrics chart should show average trust to parasites trending downward while average trust to non-parasites remains positive or increases.]
6. **Interpretation:** After multiple interactions, agents have learned to distinguish parasites from cooperative agents. Q-values for parasite interactions have decreased (negative rewards accumulate), resulting in negative trust values. Agents begin using exploitation (80%) more than exploration, making informed decisions based on learned Q-values.
7. **Input:** Simulation runs for an extended period (1000+ cycles) with learning metrics being tracked.

8. **Screenshot:** [PLACEHOLDER: Screenshot showing the learning metrics chart with clear trends: average trust to parasites significantly negative (approaching -1.0), average trust to non-parasites positive, precision increasing (more true positives, fewer false positives), recall increasing (more parasites correctly identified). Console should show decreasing number of trades with parasites as agents learn to ignore them.]

9. **Interpretation:** The learning system has successfully evolved. Agents have developed strong negative trust toward parasites (trust values near -1.0) and positive trust toward cooperative agents. Precision and recall metrics improve as agents correctly identify parasites (trust < 0) and avoid trading with them, while maintaining positive relationships with cooperative agents.

10. **Input:** A new agent is spawned by the supply shuttle and begins interacting with existing agents.

11. **Screenshot:** [PLACEHOLDER: Screenshot showing a newly spawned agent with empty Q-table and trust memory. The agent begins trading with existing agents. Console should show the new agent learning from scratch, while existing agents maintain their learned trust values. Over time, the new agent should develop similar trust patterns to existing agents after sufficient interactions.]

12. **Interpretation:** New agents start with empty Q-tables and must learn from scratch. However, existing agents have already learned to avoid parasites, so the new agent can observe patterns (though direct observation isn't implemented, the new agent learns through its own interactions). This demonstrates that learning is individual but the population as a whole develops immunity to parasites over time.

10. Final Remarks

This project successfully demonstrates how a distributed multi-agent system can evolve collective intelligence through individual learning. The Mars Colony simulation combines BDI architecture for survival behaviors with Q-Learning for social interaction, creating a complex adaptive system where agents learn to identify and avoid malicious actors.

Key Achievements:

- Successfully implemented a full BDI architecture with multiple belief types, prioritized desires, and executable plans
- Implemented Q-Learning with state-action-reward framework for trust-based decision making
- Created a continuous simulation system that maintains population through supply shuttles
- Integrated FIPA communication for long-distance messaging
- Demonstrated learning evolution through metrics showing improved parasite detection over time

Limitations:

- Death system is currently disabled to allow learning to occur (can be re-enabled)
- Oxygen and energy refill plans are disabled, prioritizing learning interactions over full survival simulation
- Q-table inheritance from supply shuttle averages is calculated but not yet applied to new agents
- Learning is individual - agents don't directly share knowledge (though they could observe patterns)
- Parasites use simple disguise mechanism - could be enhanced with more sophisticated deception

Potential Improvements:

- Apply Q-table inheritance so new agents start with averaged knowledge from survivors
- Implement knowledge sharing mechanisms (e.g., agents can share trust information)
- Re-enable death system with proper handling to test learning under population pressure
- Add more sophisticated parasite strategies (e.g., temporary cooperation, selective stealing)
- Implement reputation systems where agents can observe others' interactions
- Add more complex reward structures (e.g., reputation-based rewards, long-term relationship benefits)
- Enhance visualization with additional charts (population over time, trade success rates, etc.)

Conclusion: The simulation successfully demonstrates the evolution of a "Social Immune System" where agents learn to identify and avoid parasites through reinforcement learning. Without centralized control, the population develops collective immunity through individual negative experiences, showing how distributed learning can lead to emergent protective behaviors. This has implications for understanding trust formation in multi-agent systems, reputation mechanisms, and the evolution of cooperation in distributed environments.