Agent-Based Simulation

Muneeb Azfar Nafees

March 2025

Contents

1 Abstract

In this project, I explored the dynamics of self-organizing systems through an agent-based simulation inspired by real-world social interactions where individuals cluster or disperse based on their neighbors. To investigate this phenomenon, I created a 2D environment populated by two types of agents: SocialAgents and AntiSocialAgents. These agents move according to the conditions in their local neighborhood.

I utilized key computer science concepts, such as linked lists for storing and managing the agents and object-oriented design to handle agent behaviors through inheritance. This approach allowed for efficient updates and flexible experimentation. My findings revealed that higher agent densities and larger interaction radii could delay or even prevent system convergence, while moderate values tend to enable the system to stabilize more quickly.

2 Results

In this study, I conducted two sets of experiments to evaluate how different parameters affect the dynamics of the agent simulation. I designed the experiments to investigate the impact of varying both the number of **social agents** and the **radius of interaction** of the agents on the number of iterations required for the simulation to stabilize (meaning no further movement by any of the agents). To streamline the process, I modified the simulation file to allow the input parameters to represent different variables: **the number of agents** for Experiment 1 and **the agent radius** for Experiment 2. This change made it easier to conduct multiple tests without requiring constant modifications to the code.

2.1 Experiment 1: Varying the Number of Agents

For the first experiment, I kept the grid size constant at 500×500 and fixed the radius of each agent at 25. I ran the simulation with 50, 100, 150, 200, and 250 social agents. I recorded the number of iterations required for the simulation to reach a stable state or time out at 5000 iterations. I used the command:

```
java AgentSimulation <Number of Social Agents>
```

to carry out my experiments. The results are summarized in Table 1 (below).

Number of Agents	Iterations Until Stop
50	209
100	590
150	786
200	1620
250	Timed Out

Table 1: Iterations Until Simulation Stop for Varying Number of Agents

The data clearly indicate that as the number of agents increases, the simulation takes more iterations to stabilize. With 250 agents, the simulation did not stabilize within the 5000-iteration limit, suggesting that higher agent densities lead to prolonged movements.

2.2 Experiment 2: Varying the Agent Radius

For the second experiment, I maintained the grid size at 500×500 and kept the number of social agents constant at 150. However, I varied the radius of these agents. To facilitate this experiment, I modified the simulation file so that the input parameter would be interpreted as the agent radius. This change allowed for quick testing of multiple radii without needing to alter the code repeatedly. The experiments were conducted with radii of 5, 10, 15, 20, 25, 30, and 35. I used the command:

```
java AgentSimulation <Radius of Agents>
```

to carry out my experiments. The results are summarized in Table 2 (below).

Agent Radius	**Iterations Until Stop**
5	1381
10	1010
15	779
20	608
25	794
30	1867
35	Timed Out

Table 2: Iterations Until Simulation Stop for Varying Agent Radius

The results suggest that a moderate radius ($15 - 25$) leads to quicker stabilization of the simulation. Radii that are too small or too large tend to increase the number of iterations required for stabilization, or in the case of a large radius (35), prevent the simulation from stabilizing within the 5000-iteration limit. This outcome may be due to the fact that a larger radius increases the likelihood of agents detecting many neighbors, thus causing more frequent movement decisions.

Overall, these experiments demonstrate that both the density of agents and the extent of their interaction range play crucial roles in the behavior of the simulation.

3 Extensions

3.1 GUI: Dynamic Simulation Controls

3.1.1 What Did I Do and Why?

In this extension, I enhanced the simulation by developing a graphical user interface (GUI) using Java Swing. The GUI allows users to dynamically input simulation parameters, including the dimensions of the landscape, the number of Social Agents, the number of Anti-Social Agents, the interaction radius for each agent, and the maximum number of iterations. Additionally, the interface provides control buttons to start, pause, and resume the simulation, along with a slider to adjust the simulation speed (i.e., the timer delay) in real time. This extension was implemented to make the simulation more interactive and user-friendly, eliminating the need for code modifications when testing different parameters.

3.1.2 What Was the Outcome?

The resulting GUI not only visualizes the simulation but also displays real-time information such as the current iteration count, the number of Social and Anti-Social Agents, and the number of agents that moved in the last iteration. The inclusion of a speed slider allows users to easily modify the simulation's pace during execution, offering a finer level of control over the simulation process. Overall, this extension significantly improves the user experience by providing immediate visual feedback and interactive controls.

3.1.3 How Can You Replicate My Outcome in My Code-Base?

To replicate this extension, follow these steps:

1. Ensure that the following files are compiled and available in your project directory (preferably in an `extension` folder):
   - `Landscape.java` (the simulation model)
   - `LandscapeDisplayGUI.java` (the custom GUI display class)
   - `AgentSimulationGUI.java` (the main GUI class that prompts for parameters and provides control buttons)
   - `SocialAgent.java` and `AntiSocialAgent.java` (the agent classes)
2. Compile the code-base, ensuring that all Swing libraries and dependencies are properly resolved.
3. Run the application by executing:

```
java AgentSimulationGUI
```

4. Upon startup, the program will display dialog boxes asking for the landscape dimensions, the initial number of Social and Anti-Social Agents, each agent's interaction radius, and the maximum number of iterations. Provide the requested values and click *OK*.

5. After entering these parameters, the simulation will open in a single window. At the top, you will see a panel displaying the current iteration count, the number of Social and Anti-Social Agents, and the number of agents that moved in the last update.

6. Below the simulation canvas, you will find:

   - Control buttons (*Start/Resume* and *Pause*) to manage the simulation.
   - A slider to adjust the simulation speed (i.e., the timer delay in milliseconds).

7. Use the control buttons to start, pause, or resume the simulation. Adjust the slider to speed up or slow down the simulation. The simulation will stop automatically if no agents move in an iteration or once it reaches the specified maximum number of iterations.

3.2 LeaderAgent Influence on Simulation Convergence

3.2.1 What Did I Do and Why?

For this extension, I introduced a new agent type,the **LeaderAgent**, into the simulation. Unlike SocialAgents and AntiSocialAgents, the LeaderAgent is designed to remain static (its `updateState` method is empty) and serves as an anchor that influences the movement of nearby agents. Specifically, if a SocialAgent or AntiSocialAgent detects a LeaderAgent within its radius, its candidate movement is constrained so that it remains within the leader's influence zone.

The motivation behind this extension was to explore how an external "leadership" constraint affects the self-organizing behavior of agents. I hypothesized that the presence of leader agents would alter the convergence dynamics of the simulation (measured by the number of iterations until stabilization).

3.2.2 What Was the Outcome?

I conducted a series of experiments while keeping the agent radius fixed at 25 and the number of SocialAgents constant at 150. I varied the number of LeaderAgents (L) and observed the following results:

Number of LeaderAgents (L)	**SocialAgents (S)**	**Iterations Until Stabilization**
0	150	680
1	150	725
2	150	1020
4	150	1352
8	150	1023
16	150	930
32	150	1072
64	150	1179
128	150	Timed Out (5000 iterations)
150	150	Timed Out (5000 iterations)

Table 3: Simulation results with varying numbers of LeaderAgents and 150 SocialAgents.

The data indicate that even a small number of LeaderAgents can increase the number of iterations required for the simulation to stabilize. The relationship is non-linear: while a moderate number (like 4 leaders) greatly slows convergence (1352 iterations), increasing the number to 16 brings the iterations down (930 iterations), suggesting that overlapping influence zones might mitigate the leader effect to some extent. When the number of LeaderAgents becomes too high (128 or 150), the simulation fails to stabilize within the iteration limit.

3.2.3 How Can You Replicate My Outcome in My Code-Base?

To replicate this experiment in my code-base:

1. **Ensure all relevant files are compiled:**

   - `LeaderAgent.java`: Implements a static agent (with an empty `updateState` method).
   - `SocialAgentExt.java / AntiSocialAgentExt.java`: Their `updateState` methods include the logic to constrain movement when a LeaderAgent is detected.
   - `Landscape.java` and `LandscapeDisplay.java`: Manage agent storage, updating, and drawing.
   - `AgentSimulationExt.java`: The experiment driver that spawns agents based on command-line inputs.

2. **Run the simulation with different numbers of LeaderAgents:** Use the command line to run your simulation. For example:

   - `java AgentSimulationExt 0 150` for a baseline with no leaders.
   - `java AgentSimulationExt 16 150` to test with 16 LeaderAgents.
   - Similarly, run `java AgentSimulationExt 1 150`, `java AgentSimulationExt 2 150`, etc.

3. **Compare Results:** The simulation outputs the number of iterations until stabilization. Use these results to compare how varying the number of LeaderAgents impacts convergence.

4 Acknowledgments

1. Stack Overflow - Get random numbers in a specific range in java
2. Stack Overflow - Java Timers with jSliders
3. Java JFrame - GeeksForGeeks
4. Java Swing – JPanel With Examples - GeekForGeeks
5. Class JFrame - Oracle
6. Class JPanel - Oracle
7. Java ActionListener in AWT - GeekForGeeks
8. Class Timer - Oracle