Modeling Evolving Social Behavior

Stephen J. Hilber

June 14, 2005

1 Abstract

With the creation of Epstein and Axtell's Sugarscape environment, increasing emphasis has been placed on the creation of "root" agents - agents that can each independently act and interact to establish patterns identifiable in our everyday world. Models created for traffic patterns and flocking patterns confirm that these conditions are caused by each participating agent trying to achieve the best possible outcome for itself. The purpose of this project is to attempt to model evolutionary behavior in agents in an envionment by introducing traits and characteristics that change with the different generations of agents. Using the modeling package MASON programmed in Java, I will be able to create an environment where agents will pass down their genetic traits through different generations. By adding certain behavioral traits and a common resource to the agents, I hope to create an environment where certain agents will prosper and reproduce while others will have traits that negatively affect their performance. In the end, a single basic agent will evolve into numerous subspecies of the original agent and demonstrate evolutionary behavior. This project will show that agents which possess the capability to change will change to better fit their environment.

2 Background and Design

2.1 Introduction

Computer modeling, simply defined, is the process of programming the conditions of an environment into a computer and adjusting parameters of the model to see how they affect the results of the model. Today, though, many computer models are merely used to verify behavior that we already suspect is accurate. Many models created today focus on topics such as disease, population growth, and traffic, where the results of the models were already well known. The models are used primarily to see how adding, subtracting, or otherwise changing any parameters in the environment affect the outcome of the groups in the environment or the environment as a whole. This opens the doors for many scientists and researchers to simple, cost-efficient experimentation. For example, analyzing traffic patterns requires a large committment to observation and analysis over months and even years. Such research is not undertaken lightly. With computer modeling, this information can be programmed into a model and used to estimate how the real-world system will react to changes in its stimuli. Also, computer modeling allows us to perform "experiments" in areas of science where experimentation is not normally possible. This is especially true in astronomy, where the sheer vastness of space and our relative insignificance to the universe means that experimentation is simply not possible. You can't reset the universe and watch over five billion years of history. However, computer models of planetary orbits and systems allows us to try and find explanations for phenonoma we have observed. In short, the majority of computer modeling is used to verify existing theories.

Social behavioral patterns, although well-researched in general, have thus far been used to demonstrate how one individual interacts with his social environment as a whole. This project, based off of such personality research projects as the Dr. John A. Johnson's IPIP-NEO and using evolutionary behaviors first established in Epstein and Axtell's famous Sugarscape models, attempts to use computer modeling as a form of research in and of itself into the evolutionary aspects of social behavior. In particular, my project attempts to analyze the behavioral patterns of introverted agents and extroverted agents over a long period of time.

Interactions betwen introverted and extroverted people are not well documented. Society as a whole functions as if every member was an extravert. This is to be expected, as all interactions between people require some sort of personal conduct. Our lives are based on the idea that other people will become our friends, our coworkers, our lovers, or our enemies. This is why 75 percent of the entire population of the world is clearly extroverted - such people succeed in in a society where interactions between people are commonplace. However, introversion and extroversion are not necessarily measured of social tact or social ineptitude. Extroverts are simply people who recharge their energy by interactions with other people, while introverts recharge their energies when left to think to themselves. Introverts are also invaluable in our society - they provide us with feedback and thought that help drive our science, technology, and philisophy. Although we know a great deal about introverts and extroverts as well as their individual behaviors in society, we know relatively little about how they interact with each other.

2.2 Background

Conway's Game of Life was the first prominent agent-based model. Each cell was an "agent" that contained either a 0 or a 1 (alive or dead) depending on how many neighbors it had, and acted independently of the environment. Conway's Game of Life didn't lead to any profound insights, but it did pave the way for future agent-based modeling. The advantage of agent-based modeling, as many found out, is that it did not assume prior conditions. It was a method of building worlds "from the bottom up", where independent agents were able to create complex worlds without any overseers. One popular psychological game, Prisoner's Dilemma, spawned a series of games where agents tried to maximize their outcome, often at the expense of other agents. Eventually, these agent-based models were incorporated in studies of flocking. The models created to show flocking behavior in birds did not incorporate flock leaders, as many presumed. Instead, the birds all acted for their own best interests, and directions and resting points were chosen as compromises of sorts.

Using the theory that independent agents can create organized structures such as flocks, Epstein and Axtell created the Sugarscape world in an effort to discover if social behaviors and human characteristics could emerge through independent actions. The Sugarscape model had agents able to breed, fight, trade, and die, and the core of the model was the resource sugar. Each agent had a metabolism rate which burned off its sugar; if it ran out of sugar, the agent died. Instead of isolated behavior, however, the agents soon used their resources to work together. Agents shared sugar, sent "scouts" to gather sugar for the benefit of all, engaged in wars, and in general performed a startingly large amount of human behavioral characteristics. When spice, a second fresource with its own metabolism rate, was introduced into the world of Sugarscape, trade emerges as agents tried to meet their needs as best they could - and tried to get the best deal as a result. These behaviors are surprising, but ultimately show the value of agent-based modeling and the useful insight it can provide.

2.3 Theory

In order to simulate evolutionary behavior in an agent-based system, the agents need to simulate the real world as much as possible. In actual evolution (as described in computer science terms), two agents of opposing sex combine their genetic information at random to generate offspring with half of each parent's traits. Genetic mutations also happen at random, causing new traits that neither parent had in their genetic code. This gradual evolution creates swarms of different agents, and those agents that are best suited for their environments will be best able to survive and reproduce. Of course, several different portions of the environment could be home to dffer-ent "breeds" of agents, and these different breeds could live alongside each other in seperate societies. It is this phenomonon that this project is trying to recreate. By closely following the rules of genetics, the project should be able to show several different breeds of agents thriving, having only been created by a single agent type. Instead of passing on dominant and recessive genes, however, this project opts for a higher-level approach by using characteristics such as extraversion as the "genetic currency". While the human genome has tens of thousands of genes to determine these characteristics, such attention to detail is impossible and unnecessary for this project. By changing characteristics such as cooperation and extraversion on a slider, agent's traits will gradually change as they are passed down from generation to generation. This effectively simulates actual genetic activity, and is thus effective for this project. Agents will breed, die, and interact, eventually changing the genetic code of their societies to suit their needs.

The main change in the program will revolve around the gradual changes between extraversion and introversion. Over time, each agent type will breed with other agents. The population will then evolve to identify the most effective agent type. Extraverted agents can band together and reproduce en mass, while introverted agents can focus on preserving self above any community good. In the end, the program will randomly create worlds of extraverted and introverted agents attempting to establish control over their environment.

2.4 Design Criteria and Procedures

The program is built according to a three-level structural system; the environment, the agent, and the graphical interface. The environment creates the world that the agents live and interact on, tells the agents when to act, and takes care of all background processes along the way. The agents are able to interact with each other and move around the environment, and form the basis of the research of this project. The graphical interface takes the environment and the agents and uses that information to display everything graphically so that users can observe the simulation.

In the first stages of the project, I created an environment based off of Mason's Particle tutorial. My first step was to create an environment where agents used random movement to interact with each other. To this end, I created a "move" method which provided the foundations for many aspects of my project. The move method took into account the boundaries of the environment, and using Java's standard Math.random class generated sequences of random numbers by which the agents could move in the environment. This proved to be only somewhat effective, so I looked through various sites to try and find better generators. I eventually found the Mersenne Twister, a reliable and popular random number generator. By incorporating the Mersenne Twister into my project, I was able to create a more reliable random movement scheme. At the end of the first step of the project, I had an environment capable of supporting thousands of agents moving randomly.

The second step of my project was to create introverted and extroverted agents. To do this, I at first created a scale from one to five. Agents were then given numbers on a gradient from one to five. Agents who scored high were considered to be extroverted, wanting to be around as many other agents as possible. Agents who scored lower on the scale were considered to be introverted, wanting to avoid contact with agents as much as possible. At this time, the input of the introversion or extraversion was the same for all agents involved. When I ran the model with inrtoverted agents, the agents would be spread throughout the environment like a fine mist. When the extroverted agents were inputted into the model, however, i was surprised that there seemed to be fewer agents than the 500 standard I had included into my program. It turns out that although my extraverted agents were grouping together correctly, the module supported multiple agents sharing the same space in the environment. Because of this, the extraverted agents were sharing the same space as other extraverted agents, causing the total amount of agents onscreen to be reduced.

Knowing this, I spent time rewriting the core code supporting my project. I revisited the move method and introduced substantial new code designed to limit agents to sharing one space apiece. The mechanics of agents were redesigned so that I could have both extraverted and introverted agents interacting in the environment together - the core goal of my project. The distribution of extraverted and introverted traits was handled on a scaling basis, so there would be fewer straight introverts and extroverts and more agents with median tendencies. I focused my energies on adding to the move method, which became the basis for the interactions between introverts and extroverts. The move method soon had numerous levels of motion, where agents would check their surroundings, evauluate all nearby squares in terms of nearby agents, and finally input all of this data into a unique algorithm for each numerical score on the scale of introverts and extraverts. These algorithms would figure out which neighboring cell would be the best place for agents to move. After all of this had been completed, my project was able to show the interactions between extraverted and introverted agents.

In the later stages of development, I decided to change the scaling system of introverts and extraverts to a static class system. The reason for this was twofold: first, specific introverts and extraverts allowed the user to more easily understand the mechanics of the model; secondly, using specific classes for each agent that extended the basic Agent class allowed for different agents to have different coloring schemes, a necessity to the project that was hindered by the restrictions of the MASON package. Instead of completely recreating my move method, I decided to extend the original Agent class and create two sub-classes of Agent. AgentE represented extraverts, and AgentI represented the introverts. By this time, most of the work on the model had been completed. The interactions between the two types of agents on a social level were working flawlessly. Although I didn't implement any evolutionary behavior as I originally intended to do, I did manage to create a social simulation that can set the groundwork for a more detailed project.

3 Results

When I originally started to build this model, the results I envisioned for my project were simple. Introverts would stay away from communities of agents in general, and extraverts would group together mainly by themselves. Occasionally an extravert might chase after an introvert, but for the most part extraverts would be in groups and introverts would be drifters. The actual results proved differently. Introverts would group together with other introverts, and extraverts would group with other extraverts. This occured regardless of the environment, and this seperation of introverts and extraverts was surprising. It seems that although the introverts are normally adverse to being too close to other agents, they prefer to interact with like kinds instead of being trapped in a sphere of different kinds of agents.

The fact that my findings are different from my initial expectations only show the advantages of model-based experimentation. Using a computer system with preexisting equations, I was able to find that although introverts tends to stay away from other agents more than extroverts, the net effect creates a world where the introverts and the extroverts tend to group together. Computer modeling is an incredible tool for experimentation. Over time, this model can and should be extended to show more detailed and meaningful results while comparing introverts and extroverts.

4 End Matter

4.1 Sources

Credit goes to Conway for The Game of Life, Epstein and Axtell for Sugarscape, the MASON team for developing MASON, the Myers-Briggs Type Indicator, NetLogo, Swarm, and Dr. John A. Johnson's IPIP-NEO.

4.2 Code

```
//Stephen Hilber
//Period 2 Latimer
//Dec. 2, 2004

package sim.app.project;
import sim.engine.*;
import sim.field.grid.*;
import sim.util.*;
import ec.util.*;

public class Project extends SimState
    {
    public DoubleGrid2D trails;
    public SparseGrid2D agents;

    public int gridWidth = 100;
    public int gridHeight = 100;
    //public int numAgents = 500;
    public int numAgentE = 250;
    public int numAgentI = 250;

        //start information
    public Project(long seed)
        {
        super(new MersenneTwisterFast(seed), new Schedule(3));
        }

    public void start()
        {
        super.start();
        trails = new DoubleGrid2D(gridWidth, gridHeight);
```

```
agents = new SparseGrid2D(gridWidth, gridHeight);
/*
Agent a;

        //create random drection, location, and agent
for(int i=0 ; i<numAgents ; i++)
    {
    a = new Agent(random.nextInt(5) + 1);
    schedule.scheduleRepeating(a);
    agents.setObjectLocation(a,
                            new Int2D(random.nextInt(gridWidth),random.
    }
*/
AgentE ae;
        //create random drection, location, and agent
for(int i=0 ; i<numAgentE ; i++)
    {
    ae = new AgentE(random.nextInt(5) + 1);
    schedule.scheduleRepeating(ae);
    agents.setObjectLocation(ae,
                            new Int2D(random.nextInt(gridWidth),random.
    }

AgentI ai;
        //create random drection, location, and agent
for(int i=0 ; i<numAgentI ; i++)
    {
    ai = new AgentI(random.nextInt(5) + 1);
    schedule.scheduleRepeating(ai);
    agents.setObjectLocation(ai,
                            new Int2D(random.nextInt(gridWidth),random.
    }

// Create & decrease trails
Steppable decreaser = new Steppable()
    {
    public void step(SimState state)
        {
        trails.multiply(0.9);
        System.out.println();
        }
```

```
            static final long serialVersionUID = 6330208160095250478L;
            };

        schedule.scheduleRepeating(Schedule.EPOCH,2,decreaser,1);
        }

    public static void main(String[] args)
        {
        Project project = null;

        // should we load from checkpoint?
        for(int x=0;x<args.length-1;x++)
            if (args[x].equals("-checkpoint"))
                {
                SimState state = SimState.readFromCheckpoint(new java.io.File(args[
                if (state == null) //error?
                    System.exit(1);
                else if (!(state instanceof Project))  //error
                    {
                    System.out.println("Checkpoint contains some other simulation:
                    System.exit(1);
                    }
                else
                    project = (Project)state;
                }

        // if nothing works, recreate the simulation ourselves
        if (project==null)
            {
            project = new Project(System.currentTimeMillis());
            project.start();
            }

        long time;
        while((time = project.schedule.time()) < 5000)
            {
            if (time % 100 == 0) System.out.println(time);
            if (!project.schedule.step(project))
                break;

            // checkpoint
            if (time%500==0 && time!=0)
                {
                String s = "project." + time + ".checkpoint";
                System.out.println("Checkpointing to file: " + s);
                project.writeToCheckpoint(new java.io.File(s));
                }
            }

        project.finish();
        }

    static final long serialVersionUID = 9115981605874680023L;
    }

//Stephen Hilber
//Period 2 Latimer
//Dec. 2, 2004

package sim.app.project;
import sim.engine.*;
import sim.display.*;
import sim.portrayal.grid.*;
import java.awt.*;
import javax.swing.*;

public class ProjectWithUI extends GUIState
    {
    public Display2D display;
    public JFrame displayFrame;

    SparseGridPortrayal2D agentsPortrayal = new SparseGridPortrayal2D();
    FastValueGridPortrayal2D trailsPortrayal = new FastValueGridPortrayal2D("Trail"

    public static void main(String[] args)
        {
        ProjectWithUI p = new ProjectWithUI();
        Console c = new Console(p);
        c.setVisible(true);
        }

    public ProjectWithUI()
```

```
                    {
                                    super(new Project(System.currentTimeMillis(
                    }

public ProjectWithUI(SimState state)
                    {
                                    super(state);
                    }

public String getName()
                    {
                                    return "Project: Erinth Simulator";
                    }

public String getInfo()
    {
                    return "<H2>Erinth Simulator</H2><p>Extraversion Relations
    }

public void quit()
    {
    super.quit();

    if (displayFrame!=null) displayFrame.dispose();
    displayFrame = null;
    display = null;
    }

public void start()
    {
    super.start();
    // set up our portrayals
    setupPortrayals();
    }

public void load(SimState state)
    {
    super.load(state);
    // we now have new grids.  Set up the portrayals to reflect that
    setupPortrayals();
    }
```

```
// This is called by start() and by load() because they both had this code
// so I didn't have to type it twice :-)
public void setupPortrayals()
    {
    //create the "template agents" for graphing purposes
    Agent one = new Agent(1);
    Agent two = new Agent(2);
    Agent three = new Agent(3);
    Agent four = new Agent(4);
    Agent five = new Agent(5);

    // tell the portrayals what to
    // portray and how to portray them
    trailsPortrayal.setField(((Project)state).trails);
    trailsPortrayal.setMap( new sim.util.gui.SimpleColorMap(0.0,1.0,Color.white
    agentsPortrayal.setField(((Project)state).agents);
    //agentsPortrayal.setPortrayalForAll( new sim.portrayal.simple.OvalPortraya
    agentsPortrayal.setPortrayalForClass(AgentE.class, new sim.portrayal.simple
    agentsPortrayal.setPortrayalForClass(AgentI.class, new sim.portrayal.simple
    /*
    agentsPortrayal.setPortrayalForObject(one, new sim.portrayal.simple.OvalPor
    agentsPortrayal.setPortrayalForObject(two, new sim.portrayal.simple.OvalPor
    agentsPortrayal.setPortrayalForObject(three, new sim.portrayal.simple.OvalP
    agentsPortrayal.setPortrayalForObject(four, new sim.portrayal.simple.OvalPo
    agentsPortrayal.setPortrayalForObject(five, new sim.portrayal.simple.OvalPo
    agentsPortrayal.setPortrayalForRemainder( new sim.portrayal.simple.OvalPort
    */
    // reschedule the displayer
    display.reset();

    // redraw the display
    display.repaint();
    }

public void init(Controller c)
    {
    super.init(c);

    // Make the Display2D.  We'll have it display stuff later.
    display = new Display2D(400,400,this,1); // at 400x400, we've got 4x4 per a
```

```
        displayFrame = display.createFrame();
        c.registerFrame(displayFrame);   // register the frame so it appears in the
        displayFrame.setVisible(true);

        // specify the backdrop color  -- what gets painted behind the displays
        display.setBackdrop(Color.black);

        // attach the portrayals
        display.attach(trailsPortrayal,"Trails");
        display.attach(agentsPortrayal,"Agents");
        }
    }
```

```
//Stephen Hilber
//Period 2 Latimer
//Dec. 2, 2004

package sim.app.project;
import sim.engine.*;
import sim.util.*;
import java.lang.Math.*;

/** The basic agent */

public class Agent implements Steppable
{
        /**CHARACTERISTICS.
         * All characteristics are measured on a scale of 1 to 5.
         * Characteristics change from generation to generation,
         * and this change is determined by a breeding method
         * which matches both of the parents for each trait.
         * 5 is a high score for a trait; 1 is low. Characteristics
         * are currently random at the start - let evolution work! */

        /**Extraversion.
         * Extraversion measures social interaction. On average,
 * agents prefer social interaction to intraversion. High
 * scores deliberately try to run into other agents, and
 * low scores do everything possible to avoid them. */
public int xExtra;  // 1, 2, 3, 4, or 5

public Agent(int extra)
       {
        this.xExtra = extra;
}

public int statNum(int num)
{
        switch (num)
        {
                case 1: return 1;
                case 2: return 2;
                case 3: return 2;
                case 4: return 3;
                case 5: return 3;
                case 6: return 3;
                case 7: return 4;
                case 8: return 4;
                case 9: return 5;
                default: return 3;
        }
}

public boolean equals(Object obj)
{
        Agent temp = ((Agent)obj);
        if(this.xExtra == temp.xExtra)
                return true;
        return false;
}

public boolean isValidLoc(Int2D loc, Project pro)
{
        if(loc.x < 0 || loc.x >= pro.trails.getWidth() || loc.y < 0 || loc.
                return false;
        return true;
}
```

```
public boolean isOccupied(Int2D loc, Project pro)
{
        if(pro.agents.numObjectsAtLocation(loc) > 0) //get cells with agent
                return true;
        return false;
}

public boolean isNotCell(Int2D loc1, Int2D loc2)
{
        if(loc1.x != loc2.x || loc1.y != loc2.y)
                return true;
        return false;
}

    public void step(SimState state)
{
        //get information, leave trail
        Project pro = (Project)state;
        Int2D location = pro.agents.getObjectLocation(this);
        pro.trails.field[location.x][location.y] = 1.0;

        //move
        this.move(location, pro);
        location = pro.agents.getObjectLocation(this);
}

public void move(Int2D location, Project pro)
{
        //temploc is the temporary location; temploc2 is likewise
        //nextloc is the next location that the agent will occupy
        //bestchoice is the cell number of the "best" cell; default to 4
        //choices stores the values for each cell (higher = more favorable
        Int2D temploc;
        Int2D temploc2;
        Int2D nextloc;
        //int bestchoice = 4
        //int bestvalue = 0;
        int[] choices = new int[9];
        for(int k = 0; k < 9; k++)
                choices[k] = 0;
```

```
//CHOICES: Determining what value each cell should have (higher = m
for(int k = 0; k < 9; k++)
{
        //Eliminating this cell if it’s already occupied - or if it
        temploc = new Int2D(location.x + (k % 3) - 1, location.y +
        if(isValidLoc(temploc, pro)) //temploc’s just an Int; it’s
        {
                temploc = new Int2D(location.x + (k % 3) - 1, locat
                if(pro.agents.numObjectsAtLocation(temploc) > 0) //
                        choices[k] = -1;
        }
        else
                choices[k] = -1;
        //the current cell will be treated as irrelevant - so agent

        if(choices[k] >= 0) //stop calculating values for this cell
        {
                //Calculate number of neighbors around the candidat
                int neighbors = 0;
                for(int j = 0; j < 9; j++)
                {
                        temploc2 = new Int2D(location.x + (k % 3) +
                        if(isNotCell(location, temploc2))
                        {
                                if(isValidLoc(temploc2, pro)) //don
                                {
                                        if(temploc.x != temploc2.x
                                        {
                                                if(pro.agents.numOb
                                                        neighbors++
                                        }
                                }
                        }
                }

                //EXTRAVERSION: Using neighbor values, figure out w
                int value = 10;
                switch (xExtra)
                {
                        case 1: value -= neighbors;
```

```
                                break;
                        case 2: value -= neighbors / 2;
                                break;
                        case 3: value = 10;
                                break;
                        case 4: value += neighbors / 2;
                                break;
                        case 5: value += neighbors;
                                break;
                        default: value = 10; //this is the average
                                break;
                }
                choices[k] = value;
        }
}

//SELECTION: Choosing a cell to move to - agents prefer RANDOM MOTI
int[] narray = new int[9]; // Values of 1 mean it's currently the b
for(int k = 0; k < 9; k++)
        narray[k] = 0;
int max = -5; //anything will be better!... hopefully
int bests = 0;
for(int k = 0; k < 9; k++)
{
        if(choices[k] > max)
        {
                max = choices[k];
                for(int j = 0; j < k; j++) //reset the array if you
                        narray[k] = 0;
                bests = 1;
        }
}
for(int k = 0; k < 9; k++)//redundant, but effective
{
        if(choices[k] >= max) //always keep a keeper
        {
                narray[k] = 1;
                bests++;
        }
}
```

```
                //DECISION: Putting the destination in the map!
                int n = 4; //default
                //System.out.print("" + bests);
                //System.out.print("" + rand);
                int movetemp = (int)(Math.random() * 10);
                boolean stophere = false;
                int check = 0;
                while(!stophere)
                {
                        for(int k = 0; k < 9; k++)
                        {
                                if(narray[k] > 0)
                                        movetemp--;
                                if(movetemp == 0)
                                {
                                        n = k;
                                        stophere = true;
                                }
                        }
                        check++;
                        if(check > 50)
                        {
                                n = 4;
                                stophere = true;
                        }
                }

                int newx = location.x + (n % 3) - 1; //x-coordinate in map
                   int newy = location.y + (n / 3) - 1; //y-coordinate in map
                       Int2D newloc = new Int2D(newx, newy);
                if(isOccupied(newloc, pro) && isNotCell(newloc, location)) //make s
                        pro.agents.setObjectLocation(this, location);
                else
                {
                        if(isValidLoc(newloc, pro)) //check one last time to make s
                                pro.agents.setObjectLocation(this, newloc); //you h
                        else
                                pro.agents.setObjectLocation(this, location);
                }
        }
}
```

```
//Stephen Hilber
//Period 2 Latimer
//Dec. 2, 2004

package sim.app.project;
import sim.engine.*;
import sim.util.*;
import java.lang.Math.*;

/** The basic agent */

public class AgentE extends Agent
{
        public int xExtra;  // 1, 2, 3, 4, or 5

        public AgentE(int extra)
               {
                super(extra);
                this.xExtra = 5;
        }

            public void step(SimState state)
        {
                //get information, leave trail
                Project pro = (Project)state;
                Int2D location = pro.agents.getObjectLocation(this);
                pro.trails.field[location.x][location.y] = 1.0;

                //move
                this.move(location, pro);
                location = pro.agents.getObjectLocation(this);
        }
}

//Stephen Hilber
//Period 2 Latimer
//Dec. 2, 2004

package sim.app.project;
import sim.engine.*;
import sim.util.*;
import java.lang.Math.*;

/** The basic agent */

public class AgentI extends Agent
{
        public int xExtra;  // 1, 2, 3, 4, or 5

        public AgentI(int extra)
               {
                super(extra);
                this.xExtra = 1;
        }

            public void step(SimState state)
        {
                //get information, leave trail
                Project pro = (Project)state;
                Int2D location = pro.agents.getObjectLocation(this);
                pro.trails.field[location.x][location.y] = 1.0;

                //move
                this.move(location, pro);
                location = pro.agents.getObjectLocation(this);
        }
}
```