Object Oriented Programming Final Project

Christian Burke

December 16, 2016

Introduction

For the final propject of Object Oriented Programming, I decided to write an ant simulator. This simulator would simulate a collection of agents searching for resources, and placing pheromones in their current location to signal other agents. I was interested in the emergent behavior such a system would generate.

Previous work

Previously, this project was sketched in Javascript, a weakly-typed language without support for classes. Functionally, the idea was succesful; the simulator worked almost as intended. Organizationally, however, this project was not successful. Without support for classes, the simulator was very difficult to structure. Without knowledge and implementation of OOP principles and practices, reading, writing and extending such a rough draft would be very difficult.
Java enforces such a structure on a project, and facillitates programming when the conventions are followed. In addition, many programming tools are available to speed up the often tedious process of writing Java code, which has a tendency to be very verbose. Finally, availability, support, and ease of use for the AWT and Swing libraries is, in my opinion, generally better than that for the newly minted HTML5 Canvas. This means graphical representations of the simulator are much easier.

Ant Colony Optimization Algorithm

The ideas for this project are based on the behavior of ants in nature. Ants have apparently random and simple behavior, but will find and extract sources of food in aggregate over time. Ants are thought to place pheromones on the ground to signal other ants to the locations of items in the environment that may be of interest to the colony.
Ant colony optimization algorithms are based on this principle. Typically these are a graph algorithm, with probabistic movements for the agent. The probability of the agent to move from node i to node j is usually given as

$$p_{ij} = \frac{t_{ij}a_{ij}}{\sum_k t_{ik}a_{ik}}$$

Where a_{ij} is a weight assigned to that movement by the agents, t_{ik} is an a priori desirability factor assigned at the start of the simulation. Such a desirability

factor may reflect the presence of resources, or conversely, a low desirability factor may reflect something dangerous.
The design of the project did not initially include an a priori desirability factor, but eventually one was added.

Implementation

A rectangular graph of Cell objects with implicit connections to each of the eight Cells around it formed the basis for the implementation. The probability of an agent in one cell transitioning to another is based on the summmation of the pheromones around it in these 8 directions, within a given radius around the agent. Weights proportional to each direction were assigned based on the pheromone levels and then a direction was randomly selected according to these weights. To prevent agents from responding to pheromones they themselves had just placed, and to mitigate rounding errors within the implementation, a smaller radius is given, within which pheromones are to be ignored. After implementing this, the behavior of the agents appeared to become more random.

Important Classes / Glossary

Basic classes

Cell(interface) - The basic building block for the simulator.
CellGrid2D - A grid of Cells.
Request - The base class for all types of requests – most requests do not alter the base functionality.
NumberGenerator - A base class for different types of random number generators (eg UniformGenerator, which is implemented) and a convenient wrapper for the Java Random class.
Pheromone - Represents a pheromone, implements decay() function and general lifecycle of the pheromone

Special instances of Cell

Ant - Represents an ant (the agent)
Anthill - Generates Ants and accepts ants with food.
FoodCell - Food
PheromoneCell - Wraps the Pheromone class to allow placement in a CellGrid2D

Special instances of CellGrid2D

Board - The main logic controller of the simulator.
PheromoneAreaMap - Auxillary grid to hold Pheromones

Algorithm for random movement

The movement of the agents is random, with weights assigned based on the number of pheromones in each direction. As a test, here is the algorithm with weights set to be equal.

```
weights = new int[]{1,1,1,1,1,1,1,1};
int[] cumSum = new int[weights.length];
int sum = 0;//weights[0];
for(int ii = 0; ii < weights.length; ii++){
        cumSum[ii] = sum;
        sum += weights[ii];
}
int ra = new NumberGenerator.UniformGenerator(0, sum).generate();
int winningIndex = 0;
while(winningIndex < weights.length-1 && ra > cumSum[winningIndex]){
        winningIndex++;
}
System.out.println(winningIndex + ",");
return winningIndex;
```

The frequency count of a specific run of the above algorithm is

0	1	2	3	4	5	6	7
3785	3844	3839	3754	3804	3697	3768	3776

which appears to be approximately uniform.

Algorithm for processing requests

A class called Request and a number of its subclasses were implemented. These were relatively simple classes, meant simply to act as tokens to the request handler.

The request handler is a simple queue in the Board class. At runtime, the Board is initialized to null and Anthills and Food are placed. Each cycle, every cell is checked, and if it is non-null, step() is invoked on it. In addition, the Ant class has LoadNeighborhood() called on it. The purpose of LoadNeighborhood() is to take small section of the Board and provide it to the Ant for use during its step function.

In the step function, each Cell is provided the main RequestQueue and allowed to add arbitrary requests to it. For instance, an Anthill may request to place an Ant on the Board. An Ant may request to place a Pheromone on the board or to be removed from the board.

A similar process occurs in the PheromoneAreaMap, but there is much less sophistication involved.

The following listing is the RequestQueue handler from the Board class and supporting functions.

```
public void run(){
        //System.out.println("executing run function");
        stepAllCells();
        //System.out.println("executing request processor");
        processRequests();
        //System.out.println("completed board functions");
        pheromoneMap.decayAll();
}

public void stepAllCells(){
        //System.out.println("stepping all cells in the board");
        int[] location = new int[2];
        Cell x;
        for(int ii = 0; ii < numRows; ii++){
        for(int kk = 0; kk < numCols; kk++){
                x = grid[ii][kk];
                if(x != null && x instanceof Ant){
                        //System.out.println(ii + "," + kk+" non-null");
                        location[0] = ii;
                        location[1] = kk;
                        ((Ant) x).loadNeighboorhood(this, location, this.pheromoneMap);
                        x.step(this.requestQueue);
                        //System.out.println("completed step");
                }else if(x != null && x instanceof Anthill){
                        ((Anthill)x).step(this.requestQueue);
                }
        }}
}

void processRequests(){
        //System.out.println("processing request queue");
        Request temp;
        while(!requestQueue.isEmpty()){
                temp = requestQueue.remove();
                if(temp instanceof MoveRequest){
                        safeMove(temp);
                }else if(temp instanceof PlacementRequest){
                        //System.out.println("placing... ");
                        safePlace(temp);
                        //System.out.println("done placing... ");
                }else if(temp instanceof PickupRequest){
                        safePickup(temp);
                }else if(temp instanceof RemovalRequest){
                        safeRemove(temp);
                }
        }
}
```

Determining the weights for each transition

Each cycle, the Ant checks its neighborhood and determines a move to make probabalistically. Different weights can be assigned in the Ant class for Food and Pheromones. Once an Ant has obtained Food, the weight of the transition in the direction of the Anthill which spawned it can be adjusted in a similar manner. These weights, together with the window size, determine the behavior of each Ant.
The implementation of this process is buggy. In particular I had a lot of difficulty with the atan2 function.
I noticed during testing that the Ants had a "swooping" behavior, where they were not attracted to the Pheromones but instead would move in a circular pattern around Pheromones. Upon fixing this, I noticed the ants began to clump up around high concentrations of Pheromones and stop collecting resources until the Pheromones had run out. I replaced the "bug" and found that the Ants seemed to be more efficient.
In the future I'd like to implement some sort of gradient Pheromone, which would indicate to the ants which direction they should go in order to reach the resource.

The function for loading the neighborhood

The following listing is the implementation of the weight assignment for the ant. The ant has 8 possible transitions, and each one is weighted according to the objects of interest in that direction.

```
public int[] getEmpiricalDist(){
        //System.out.println("generating empd");
        int[] vals = new int[]{4,4,4,4, 4,4,4,4};
        int l = phNeighborhood.length;
        //int[][] test = new int[l][l];
        if(!this.hasItems){
                for(int ii = 0; ii < l; ii++){
                for(int kk = 0; kk < l; kk++){
                        if(phNeighborhood[ii][kk] != null){
                                double angle = Math.atan2(ii - l/2, kk - l/2);
                                int wheel = (int) ((2*Math.PI + angle)/(Math.PI/4))%8;
                                vals[wheel] += 1;
                        }
                        if(this.cellNeighborhood[ii][kk] instanceof FoodCell){
                                double angle = Math.atan2(ii - l/2, kk - l/2);
                                int wheel = (int) ((Math.PI+angle + Math.PI/2)/(Math.PI/4))%8;
                                vals[wheel] += 100;
                        }
                }}
        }else{
                double angle = Math.atan2( location[0] - origin[0], location[1] - origin[1]);
                int wheel = (int) ((2*Math.PI+angle + Math.PI/2)/(Math.PI/4))%8;
                vals[wheel] += 100;
        }
        return vals;
}
```

UI considerations

The UI was very simple. Two classes were implemented, one to maintain the window state, and one to generate images based on the contents of the Board class. Ultimately, two images were generated and displayed in the window. In the future, I hope to add buttons and functionality to facillitate user interaction with the Ants (eg placing food, pheromones, attractors and repellors).

Conclusions/Further interests

While the Ants did acheive the goal (collecting food), they didn't do so especially efficiently. Implementation details, especially regarding the behavior of ants in response to Pheromones, still have a few kinks to work out.
In the future, I would like to work on UI, including displaying images and allowing more user interaction.
I would also be interested in adding new features, like predators, walls, gradient-inducing pheromones, attractors and repellors, etc.
Ultimately I felt the project was successful given the amount of time I spent on it and the outcomes I acheived. In addition, I gained valuable experience from implementing a large program in the Object-Oriented paradigm. The value of doing fairly extensive class documentation before writing any code is now very clear to me.