

*UNIVERSITY OF ATHENS*  
*M.Sc. in BIOINFORMATICS*  
*ATHENS 2004*

*Assignment in*  
***COMPLEX ADAPTIVE SYSTEMS***

**Topic**  
Clustering in social insects

*Elpida Tzafestas*

# 1 Objective of the project

The objective is the implementation of a model of object clustering starting from the research work by Deneubourg and associates (cf. references). According to this model, which initially addresses ant or more generally insect societies, each ant or insect (agent) follows a very simple behavioral model :

**(Step 1)**

Random move to a neighboring position

**(Step 2)**

- (a) If the agent does not carry food:  
if there is already food in the position of the agent,  
then with a probability  $p_c = (k_1 / (k_1 + f))^2$   
the agent charges a piece of food
- (b) Otherwise (if the agent carries a piece of food)  
then with a probability  $p_d = (f / (k_2 + f))^2$   
the agent discharges the piece

$k_1$  and  $k_2$  are constant  $\in (0,1)$

$f$  the quantity (fraction) of food around the agent

**(Step 3)**

Update of variables used for action selection, here possibly the  $f$  variable.

The above general case may be instantiated or further specialized in many ways :

- Defining differently the neighbourhood of an agent ; it is generally a 4- or 8- point neighbourhood, or with a radius of above 1.
- With different values for  $k_1$ ,  $k_2$ .
- With different computation methods for  $f$ . Some obvious methods are  $f$  computation in the neighborhood of the object (e.g. the same neighborhood that is used for motion) or  $f$  computation as the fraction of loading actions of the agent during the last  $N$  simulation cycles or as the fraction of occupied places the agent encountered during the last  $N$  simulation cycles.
- With “continuous load”, i.e. load taking not just the binary values 0 or 1, but taking values in an interval  $[0..L]$ , either discrete (e.g. 0,1,2,...L) or continuous (any real value in the same interval).
- With “continuous perception” in the same spirit as continuous load, i.e. with the possibility to recognize the amount of food present at its current position.
- With more complex formulae for computing the decision probabilities  $p_c$  and  $p_d$  or with different decision rules for loading/unloading (and especially in problems involving continuous load and/or continuous perception).
- With more object types (e.g. food of type A, B or C).

- Finally with specialization of the “goal” of the overall system. The typical case described above (with  $p_c$ ,  $p_d$ ,  $f$ ) can be characterized as object grouping in clusters (clustering). For more than one food types, we generally speak about collective sorting, i.e. clustering of objects according to type. Finally, building and assembly are problems involving some variant of continuous perception each.

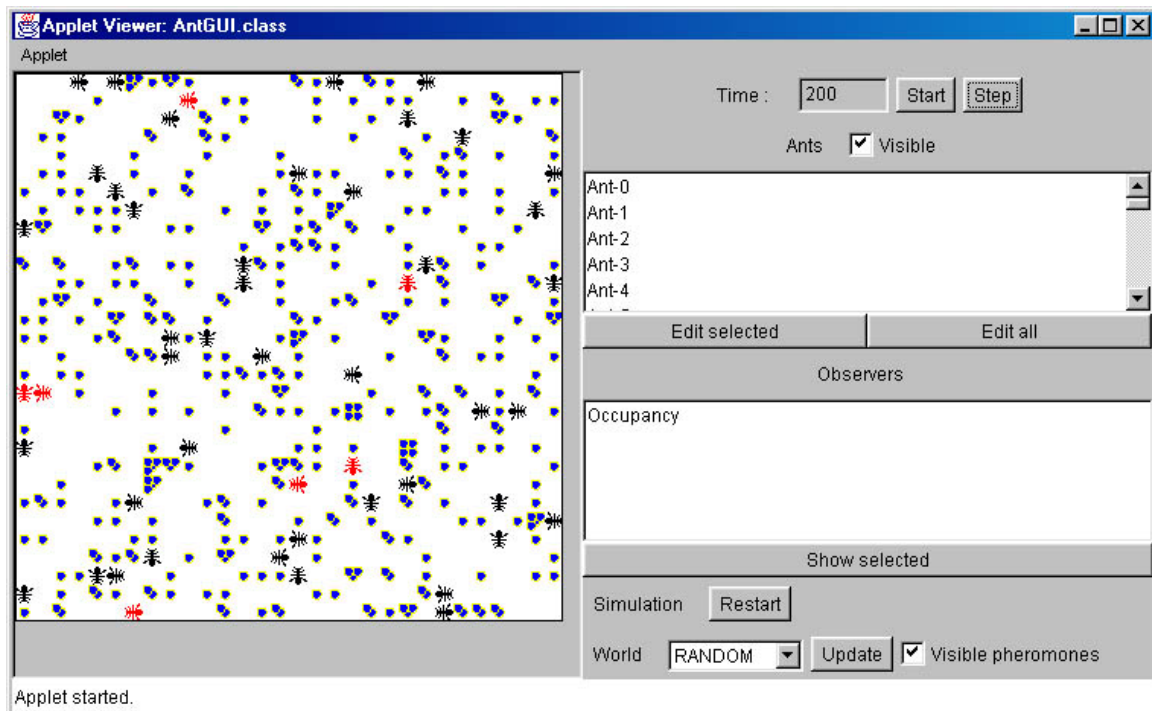
We present below a modeling and simulation testbed where a number of model-less agents are defined (i.e. agents without behavior). The testbed allows the experimentation with one or more agent behavioral models.

The performance of an agent depends not only on its behavioral model but also on the environment in which it is situated. Thus, the more independent a model is from environmental specificities, the better it is. The experimenter has to design and study different environments in which his/her agent model(s) will be tested.

### 1.1 Environment description

The environment is the 2D space in which a number of (passive) objects exists and a number of (active) agents live and move and can load/unload objects. Agents also have a (limited) perception capacity, for example in the simplest case they can see whether there is food in their position (and possibly how much this is). The food grains are visualized in blue, as the following picture shows.

## 2 Implementation of the testbed



## 2.1 User's manual

The applet's screen shows a white area in the left where lie the agents (simulated ants) and the objects to be clustered (simulated grains of food). By double-clicking on a position, a popup dialog appears that gives the number of objects and the names of agents in this position. Agents are visualized in black whenever they are free of charge, and in red whenever they are loaded with a grain. The right side of the applet includes a number of simulation control components.

- **The simulation clock.**

- **Three simulation control buttons.**

The **Start/Stop** button starts/stops the simulation and in every cycle executes all the agents. The **Step** button advances the simulation one cycle further. Finally, the **Restart** button restarts the simulation, i.e. reinitializes the clock to 0 and resets all agents and statistics observers.

- **A simulation start/stop button.**

In every simulation cycle all agents execute at random order. If we define dynamic objects, they will execute as well (e.g. we can imagine food sources that decay with time).

- **A simulation step button.**

It advances the simulation one cycle further.

- **A list of the simulated agents and a show/hide checkbox.**

- **Two buttons for agent editing.**

The "Edit" button is used to edit the selected agent from the above list, whereas the "Edit all" button is used for the complete list of agents. By default, the agent editor allows the user to modify the agent's behavioral model.

- **A list of the various "observers" of the simulation, that "watch" or "observe" the progress of the simulation and gather statistical data.**

By default, a single statistical variable is defined, the *occupancy* degree or environmental density, i.e. the space percentage that is covered by at least one object. The system automatically updates the statistical data for occupancy, that is a series of numbers (measurements) and that may be presented graphically in the form of a curve.

- **A button for statistics presentation.**

It opens up a window that presents graphically (as a curve) the measurements for the selected observer.

Finally, the testbed allows the following operations from its lower right end components.

- **Restart** – Reinitialization of the simulation. It reinitializes the clock to 0, resets all agents, but leaves their environment intact.

- **Update world** – Redefines the simulation environment, but leaves the simulation clock and the agents intact. The user may select one of the given environmental configurations (random, custom1, custom2 or custom3) and then confirm with the “Update” button.
- **A pheromone show/hide checkbox.**

The above features of the testbed allow the user to experiment systematically with several ant clustering models, in the predefined environmental configurations or with new constraints of his own choice.

### 3 Instructions

The testbed includes four types of classes :

- **User interface classes.** Those are the AntGUI, SimulationView, GenericEditor and GenericInterface classes, that implement respectively the applet, the canvas on which the simulation view is drawn, the editor (general form that is used by the agents) and an auxiliary interface.
- **Simulation class.** This is the Clustering class, that implements the simulation itself (for example, how the agents move around in the environment without falling on one another, how they load/unload objects and how the simulation progresses).
- **Agent class.** This is the Ant class that implements all the primary functions and possibilities of the agent (motion, perception, loading/unloading) and that may be appropriately customized by the user to fit his own study needs.
- **Auxiliary classes.** Those are the classes Observer, History, Curve, CurveViewer, ListModifierDialog, ListModifierListener and TextAreaDialog, that implement the statistical data recording as a time series that may be translated into a curve and visualized as a graph. The user does not need to know the internals of these classes, but rather it uses them as such. Only the classfiles are given for these classes. Other auxiliary classes are the classes MessageDialog and Place that implement respectively a popup dialog to present text messages and a discrete “place” of the simulated world (because a place may “contain” many different objects as well as agents, it is better to give it material quality, that of a container). Again, the user does not need to know the internals of these classes, but rather it uses them as such.

#### *What to do*

1. Implement an ant clustering model that conforms to the original Deneubourg model above and that achieves the best (maximum) clustering speed in different environments. This model has to be implemented in the Ant class for experimentation.

**ATTENTION!! It is forbidden to use variables that explicitly or implicitly record the absolute or relative position of the agent in the environment, for example it is forbidden for agents to “know” the positions of objects or to use odometric knowledge (such as how many forward movements and turns the agent has done). Without this restriction the system would not be biologically realistic.**

ATTENTION also to the implementation of random motion. Pure Brownian motion may lead to totally different results than other more realistic random motions of the type “*Motion toward a random direction and change of direction with a small probability*”.

2. Implement a second model of your choice that is either a specialization of the general case, such as an assembly model, or a model of your pure inspiration and of a form similar to the original one. This second model has to differ **qualitatively** from the previous one, for example resulting clusters could be linear. An indicative qualitative comparison between the Deneubourg model and three models called A, B, C is given below. Try to reproduce behaviors such as these.

**Deliverables.** The Ant and Clustering classes and a description text for both models. If necessary for completeness or presentation simplicity, you can give many versions of the two models that have emerged and have been evaluated gradually during the study. Also include systematic results of the application of the models, comparative or not, in any form you judge suitable (statistical data tables, charts, snapshots etc.). To answer the above question (2), it would be nice for the results to demonstrate the “quantitative” difference between the two models. If applicable, you can define and justify the type of “experiments” necessary for the evaluation of the models, for example to try them only in “sparse” environments (with a low environmental density) or in environments with specific object configurations (such as triangles or “forests” of objects etc.).

### 3.1 *Description of the Ant class*

The Ant class defines the following data :

- **boolean loaded**  
Denotes whether the agent is loaded with objects or not. It is also used to draw the agent with a different color on the simulation view (red or black, respectively).
- **int behavior**  
The behavioral model of the agent. It takes one of the (constant) values *MODEL\_0* or *MODEL\_1* or any other you will define yourself.
- **double x**  
“Useless” variable, that is only included to be used as a template for editing.
- **Point position**
- **Point heading**  
The position of the simulated ant in the environment and its heading.

- **static int T**
- **boolean f[]**
- **int fIndex**  
Data used for ‘f’ computation, as defined by the original Deneubourg model.
- **int id**
- **static int counterID**  
Data used for automatic name generation for ants (Ant-1, Ant-2,...).
- **Choice BehaviorChoice**
- **TextField XField**  
User interface controls for the variables “behavior” and “x”, respectively.
- **Button TestButton**  
Button to be used as a template for editing.  
For the purposes of your models, define any more variables of any type you judge necessary.

Moreover, the Ant class defines the following methods (on which you will not need to intervene) :

- **private void setID()**  
It is used for initialization of the agent’s name.
- **public void initAt(int,int)**  
It is used for initialization of the agent in a predefined position of the simulated world.
- **public void setPosition(int,int)**
- **public Point getPosition()**  
They set/get the agent’s position.
- **public String name()**  
It fetches the name of the agent.
- **public void drawOn(Graphics,Point,Point,AntGUI)**  
It draws the agent on the simulation canvas (that is, it draws the agent in its simulated environment). The first Point argument is the agent position, the second is the translation scale (it is given by the testbed).
- **private void setF()**
- **private double computeF()**  
Data used to update and retrieve the value of ‘f’.
- **public void editIn(AntGUI)**

It opens the agent editor.

- **public static double truncate(double,int)**

Utility for decimal digits truncation (it may be used during data gathering).

The Ant class that you will deliver should define/redefine the following methods :

- **public void reset()**

The agent reinitialization method. It is used to reinitialize the agent's variables.

- **public void execute()**

Primary "execution" method of the agent. Depending on the agent's behavior, it should call other methods. It is also responsible for the implementation of the agent motion model, that is of the selection of motion direction or goal-setting, without colliding with other agents.

- **public Panel editorPanel(GenericInterface)**

- **public void editorActionPerformed(ActionEvent)**

- **public void editorItemStateChanged(ItemEvent)**

- **public Vector getEditedValues()**

- **public void updateWith(Vector)**

Methods used by the agent editor: editor initialization, editor ActionEvent processing, editor ItemEvent processing, creation of a data list (vector) that are created during editing and agent variable update according to the above vector values. Study how editing of the variables 'behavior' and 'x' is done and implement accordingly editing of as many variables of yours as you judge necessary.

### 3.2 *Description of the Clustering class*

The Clustering class defines the following data :

- **AntGUI caller**

The simulation interface.

- **final static int WIDTH**

- **final static int HEIGHT**

- **final static int NUMBER\_OF\_ANTs**

- **final static int NUMBER\_OF\_SAMPLES**

Constants about the size of the simulated environment and the (initial) number of agents and objects.

- **Place world[][]**

The simulated environment (2D table). The auxiliary class Place holds all the necessary data and methods for object management, agent motion etc.

- **Vector ants**  
The agents.
- **int time**  
The time counter.
- **Vector observers**  
The simulation “observers” that gather statistical data.
- **static String OCCUPANCY**  
Constant used by the occupancy observer.

For the purposes of your own study, define any additional variables of any type you judge necessary. More specifically, you may define your own observers in the same way as the one used here by default to define the occupancy observer.

Moreover, the Clustering class defines the following methods (on which you will not need to intervene) :

- **public void initWorld()**  
It is used to initialize the environmental variables.
- **public void initRandomly()**
- **public void initObjectsRandomly()**  
It is used to initialize the simulated objects randomly (“random” configuration in the simulator selector).
- **public void reinitWorld(String)**  
It is used to reinitialize the object configuration within the environment.
- **public void initAnts()**  
It is used to initialize the agents.
- **public void restart()**  
It is used to reinitialize (restart) the simulation.
- **public int getTime()**
- **public void incrementTime()**  
It is used for simulation clock management (it advances the simulation by one step).
- **public void addAgentAt(int,int,Ant)**  
It is used during agent move from position to position within the environment.
- **public boolean isValid(int,int)**  
It returns true if and only if the position (x,y) is outside the environmental borders and is generally used by an agent to find and manage its neighbouring positions.

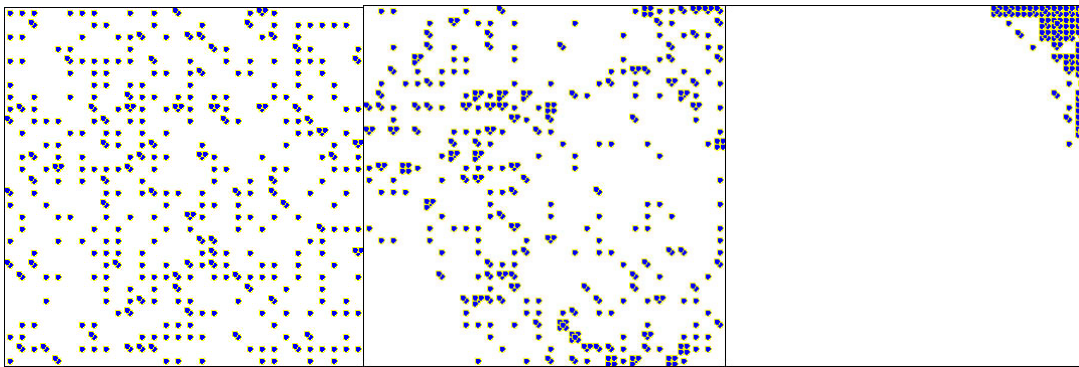
- **public boolean isEmpty(int,int)**  
It returns true if and only if the position (x,y) does not contain any agent and is generally used by an agent to probe its neighbouring positions.
- **public int samplesAt(int,int)**
- **public boolean hasSampleAt(int,int)**
- **public void addSampleAt(int,int)**
- **public void removeSampleAt(int,int)**  
They are used for object management (retrieval, testing, addition and removal of objects in a position).
- **public double pheromoneAt(int,int)**
- **public boolean hasPheromoneAt(int,int)**
- **public void addPheromoneAt(int,int,double)**
- **public void removePheromoneAt(int,int,double)**  
They are used for pheromone management (retrieval, testing, addition and removal of pheromones in a position).
- **public void moveAgentTo(Ant,int,int)**  
It moves an agent to a new position (removal from the old and addition to the new position).
- **public void execute()**  
It executes one simulation step. First, all agents execute once each and then their environment executes (if you have defined dynamic processes, such as time-decaying pheromones, look into the method *execute()* in class Place).
- **protected Vector antsAt(int,int)**  
It returns all the agents that exist in a specific place in the simulated world (for example you can define in the corresponding motion model that at most N agents can coexist in one position, otherwise the position is considered an obstacle to other agents).
- **public void updateOccupancy()**  
It updates the “Occupancy” observer. You can define accordingly your own observers with similar operations.
- **public void addObserver(String)**
- **public void removeObserver(String)**
- **public void updateObservers(String,double)**
- **public void showStatisticsFor(String)**  
They are used for initialization, retrieval and management of the simulation observers.

The Clustering class that you will deliver should define/redefine the following methods :

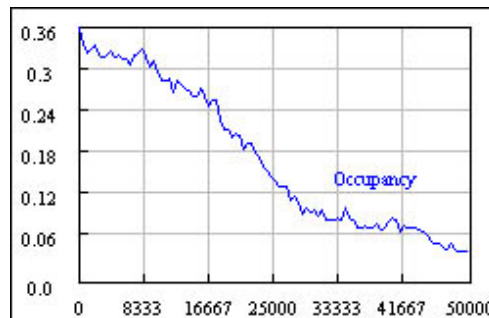
- **public void initCustom1()**
- **public void initCustom2()**
- **public void initCustom3()**  
Object initialization in “custom” configurations of your choice (configurations “custom 1”, “custom 2” and “custom 3” in the simulator selector).
- **public void initObservers()**  
Observers initialization. By default, only “Occupancy” is defined, but you may add your own observers.
- **public void executeAgents()**  
Its is used for the execution of the agents. The agents may execute in a predefined or (better) in a random order.

### 3.3 Example of a model study

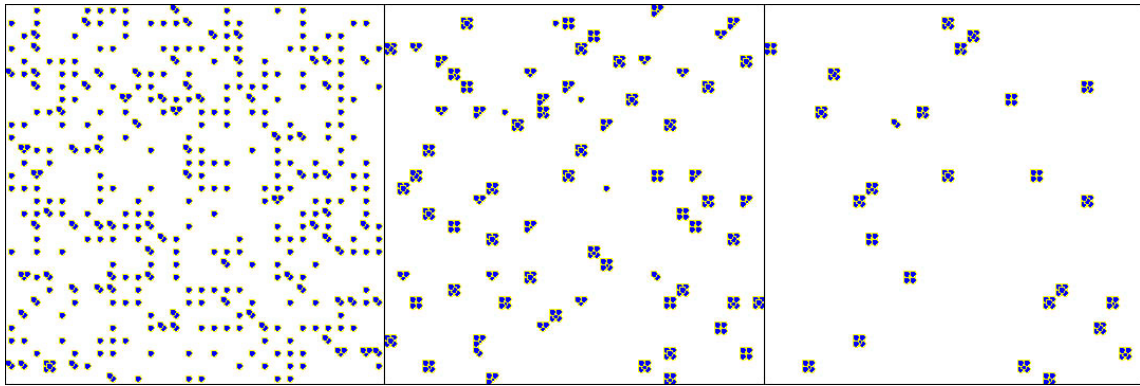
Below are given three states of a 50-agent system in a 30x30 environment with 450 objects -initially randomly distributed in space- for  $t=0$ ,  $t=10000$ ,  $t=50000$  respectively, with  $k_1=1.5$ ,  $k_2=0.01$ . The value of  $k_1$  has been defined to be more than 1, so that  $p_c$  be quite close to 1 and loading be frequent. Accordingly the value of  $k_2$  has been defined very close to 0, so that  $p_d$  be quite close to 1 and unloading be frequent. Finally, this value combination speeds up the clustering process :



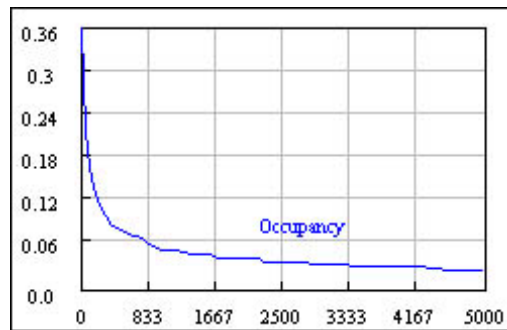
The following graph depicts the dynamics of the occupancy metric :



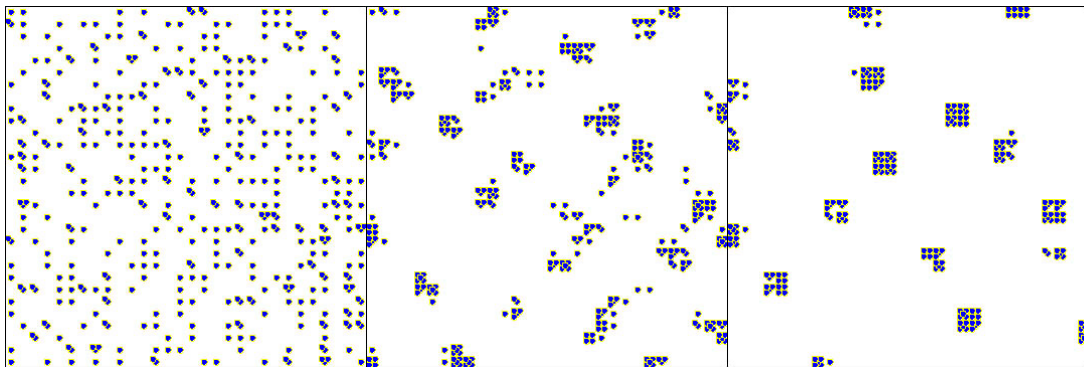
The corresponding system states for an A model are (for  $t=0$ ,  $t=500$ ,  $t=5000$  respectively) :



The following graph depicts the dynamics of the occupancy metric (note that, compared with the Deneubourg model, clustering is much faster, but leads to high disconnected heaps of objects) :



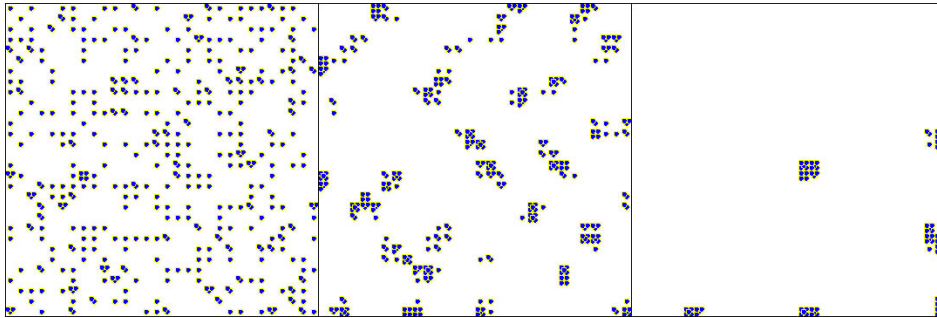
The corresponding system states for a B model are (for  $t=0$ ,  $t=500$ ,  $t=5000$  respectively) :



The following graph depicts the dynamics of the occupancy metric (note that, compared with the A model, clusters are not thin and high, but thick and low –and fewer!- so that a greater proportion of the environment is occupied by objects) :



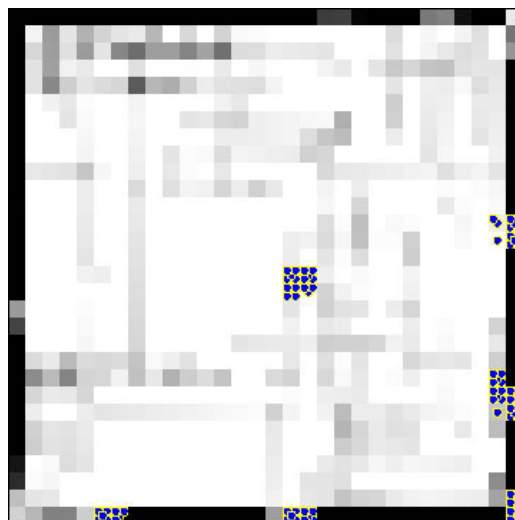
The corresponding system states for a C model, which is the B model with additional pheromone usage for cluster/heap marking, are (for  $t=0$ ,  $t=500$ ,  $t=5000$  respectively) :



The following graph depicts the dynamics of the occupancy metric (note that, compared with the B model, clustering is extremely faster and leads to fewer clusters that are thick and high) :



The corresponding pheromone occupancy is depicted below on the simulation view, where the darkness of a position is proportional to the amount of pheromone present in this position. Note that there is a higher pheromone density in the borders of the environment –this is due to the fact that the motion model is not Brownian, but the agents travel in long directions before changing direction randomly or if they encounter an obstacle or the borders of the environment :



## References

- E.Bonabeau, M.Dorigo, G.Theraulaz (1999). *Swarm Intelligence, From Natural to Artificial Systems*, Oxford University Press, 1999.
- S.Camazine, J.-L.Deneubourg, N.R.Franks, J.Sneyd, G.Theraulaz, E.Bonabeau (2001). *Self-Organization in Biological Systems*, Princeton University Press, 2001.
- J.-L.Deneubourg, S.Goss, N.R.Franks, A.Sendova-Franks, C.Detrain, L.Chretien. The Dynamics of Collective Sorting: Robot-Like Ants and Ant-Like Robots, *Proceedings First Conference on the Simulation of Adaptive Behavior, From Animals to Animats*, by J.-A.Meyer & S.W.Wilson (Eds.), MIT Press, 1990.
- E.Lumer, B.Faieta (1994). Diversity and Adaptation in Populations of Clustering Ants, *Proceedings Third Conference on the Simulation of Adaptive Behavior, From Animals to Animats 3*, by D.Cliff, P.Husbands, J.-A.Meyer & S.W.Wilson (Eds.), MIT Press, 1994.
- R.Beckers, O.E.Holland, J.-L.Deneubourg (1994). From local actions to global tasks: Stigmergy and collective robotics, *Artificial Life IV, Proceedings of the 4th Interdisciplinary Workshop on the Synthesis and Simulation of Living Systems*, by R.Brooks & P.Maes (Eds.), MIT Press, 1994.
- A.Martinoli, A.J.Ijspeert, F.Mondada (1999). Understanding collective aggregation mechanisms: from probabilistic modelling to experiments with real robots, *Robotics and Autonomous Systems* **29**(1999):51-63.

## See also ...

The Anthill Project @ Univ.of Bologna

<http://www.cs.unibo.it/projects/anthill/>

Marco Dorigo @ IRIDIA, Brussels

<http://iridia.ulb.ac.be/~mdorigo/HomePageDorigo/>

Swarm Intelligence Group @ Caltech

<http://www.coro.caltech.edu/Courses/EE141/>

Ant Robotics @ Georgia Tech

<http://www.cc.gatech.edu/fac/Sven.Koenig/ants.html>

ANTS Workshop Series

<http://iridia.ulb.ac.be/~ants/>