Intelligent Systems Simulation Project (236754)

Circle Mark Ant Walk Algorithm

Submitted by:
Kiril Lisovtsev
Anna Gavrilenko

Under the guidance of: Yotam Elor
Under the supervision of: Prof. Alfred M. Bruckstein

September, 2010

Table of Contents

Problem definition

In our project we present a Simulator for a new pheromone based domain covering algorithm. Our goal is to perform a continuous area patrolling using a robot. The said robot is capable of marking pheromone trails of varying intensity and is equipped with sensors, allowing him to identify previously marked pheromone circles upon crossing them. Also, the robot's memory size is independent of the domain size, meaning it has no capability of learning the said domain.

The simulator provides a Solution based on the paper "Robot Cover" by Yotam Elor and Alfred M. Bruckstein.

The CMAW algorithm

Originally, the algorithm describes a team of robots as described above, whilst each robot can sense the markings of the other robots. For simplicity means, we refer only to the one robot.

Let us define some notations:
R(r,2r,p) – A ring centered in p, between radius r and 2r
D(r,p) – A disc centered in p, with radius r.
ph(p) – the pheromone level at point p.

The CMAW ("Circle Mark Ant Walk") algorithm is an extension to the MAW ("Mark Ant Walk") algorithm, where it is assumed that the robot, located in a point p, can sense the amount of pheromone in every point in R(r,2r,p), and set the disc D(r,p) to any uniform value.
The CMAW algorithm proposes a solution to the case where the pheromone marking and sensing ranges are much smaller than the cover range, and in fact are constant and independent of the cover range.
The robot is moving in circles of radius 3r/2, and marks them by dispersing pheromone along its path. A pheromone circle induces virtual pheromone in D(r,p), where p is the center of the said circle.
The pheromone level of a point is the maximum pheromone of all the induced circles it belongs to.

Firstly, the robot performs a “reading” circle, where he senses the previous marks on the floor. Using the information about the intersections he can deduce the circles that he has met, and thus know the level of virtual pheromone in each point in the area R(r,2r,p). The robot then performs a “mark” circle, where he marks the circle with new pheromone and continues to the point with minimum pheromone in the area described above.

Circle Mark Ant Walk (CMAW) – pseudo code

```
The algorithm is executed for point p
   1. Travel a circle of radius 3r/2 around p while identifying
      intersecting pheromone circles.
   2. Compute the virtual pheromone map of R(r,2r,p)
   3. Let x be the point with the minimal virtual pheromone value in
      R(r,2r,p)
   4. If ph(p) <= ph(x)
            Mark the circle of radius 3r/2 around p with ph = ph(x) + 1
   5. Execute the algorithm for point x (In case of such several points,
      choose one randomly)
```

Project Goals

The projects goal is to implement the CMAW covering algorithm in Java, as well as to provide a user-friendly simulator, so the user will be able to see the robot patrolling the domain as well as the pheromone markings made by the robot, and other relevant visual aids.

Implementation

Algorithms

The main algorithm in this project is the identification of intersecting pheromone circles; it is mainly composed out of two parts:

1. Identification of intersections with intersecting circles.
2. Deduction of the circles from the list of identified intersections.

The first part is performed during the "reading" circle the robot performs upon arriving at a new point (equivalent to the first step of the CMAW algorithm described above).
In the program we keep a database of all the points the robot has "visited", and so when the robot performs the said reading, whenever he gets in a range of 3r/2 of a known visited circle, measurements of the angle between the circles (see figure 2), the pheromone level of the circle, as well as the coordinates are being made and sent to the class 'Robot', where they are stored in a temporary database.

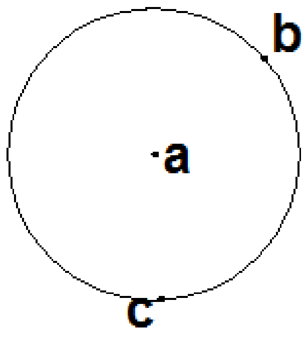


Figure 1 – After finishing step 1 around the point a, points b and c have been identified as intersections with previous circles.

Later on, when the robot has finished step 1, we proceed to the second part of the algorithm. Using the list we built in phase one, we now can recognize the exact circles the robot has met; this can be achieved by the fact that two intersections can belong to the same circle if and only if:

- They share the same meeting angle (see figure 2).
- Relative to that angle they are within a specific distance one of another.
- They have the same pheromone level.

Combining all those factors together, we are able to determine exactly where the centers of the intersecting circles are.

This is achieved in a few steps:
At first, for each two intersections we calculate the theoretical circle that crosses the origin circle in those two points (we are able to do that because of the fact that for any two points and a given radius, there are only two circles with that radius that passes throw those points, and one of them is our original circle).
Then we calculate the meeting angle of those two circles, and compare it to the angle of the intersections, if they are equal, we deduce that those two intersections belongs to the same circle, and add the said circle to the list of identified circles and continue to the next two intersections.

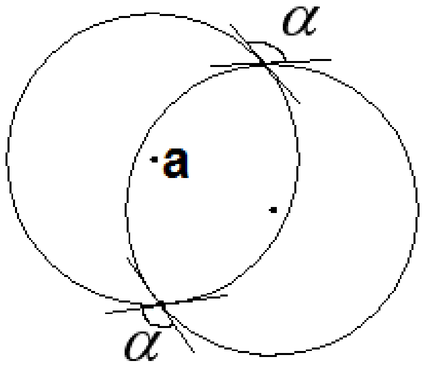


Figure 2 – α - the meeting angle between the two circles.

<u>Identification of intersecting pheromone circles – pseudo code</u>

```
1. Let L1 be the list containing all the intersections found while
   circling a circle C.
2. L2 ← empty list of circles.
3. For each two points: i, j in L1 do:
      3.1   Let C2 be the theoretical circle that intersect the circle
            C at points i and j.
      3.2   If the meeting angle between circle C and C2 equals to
            the measured angle in both intersections i and j
               Add C to L2
4. L2 Contains now all identified intersecting circles.
```

Another important algorithm is the algorithm of avoidance of obstacles along the robot's way.
Each step the robot takes, we check whether it hast stepped inside an obstacle, if it is the case (see points **A**, **C** and **F** in Figure 3), the robot

starts moving alongside the obstacle. An important point is that in this case, the robot has only four ways to go: left, right, up and down. Whilst moving alongside an obstacle we distinguish between two cases:

1. Originally, the robot was moving in a circle (see figure 3.1) – in that case the robot will continue moving near the obstacle, as long as it is still "inside" the original circle (hasn't reached point B). When the robot reaches point B, it stops following the obstacle and returns to the circular motion.
2. Before meeting the circle, the robot went in a straight line (usually it is the case when the robot moves from one circle to another, see figure 3.2) – in that case the robot will move alongside the obstacle until it will be on the same line again (point **E**), and then will continue walking in the original direction.

In both the cases above one of the following may occur:

1. The obstacle ends (see point **D** in Figure 3.2) – in that case the robot will start rounding the obstacle.
2. The robot meets a new obstacle in the way (see point **G** in Figure 3.3) – in that case it will simply continue moving alongside the new obstacle.

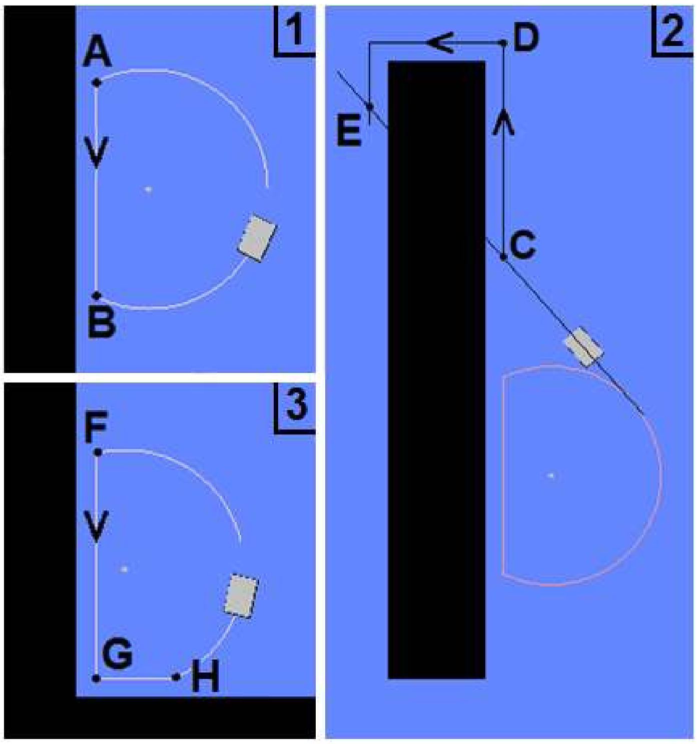


Figure 3

Main Data Structures and Classes

Data Structures

For all of our databases we've used the Java built In LinkedList data structure. Depending on the usage of the List, it contained different data such as: Obstacles, Coordinates or Intersections.

Main Classes

- **Robot** – Holds information about the robot's position and state (step in the algorithm) as well as the list of identified intersections. Includes functionality of drawing itself, and uses the Class 'Move' to calculate the next move.

- **Move** – Responsible for the robot's movement. It holds the robot's current position and state, and is responsible for retrieving the next step according to the robot's state and the obstacles it meets in its way.

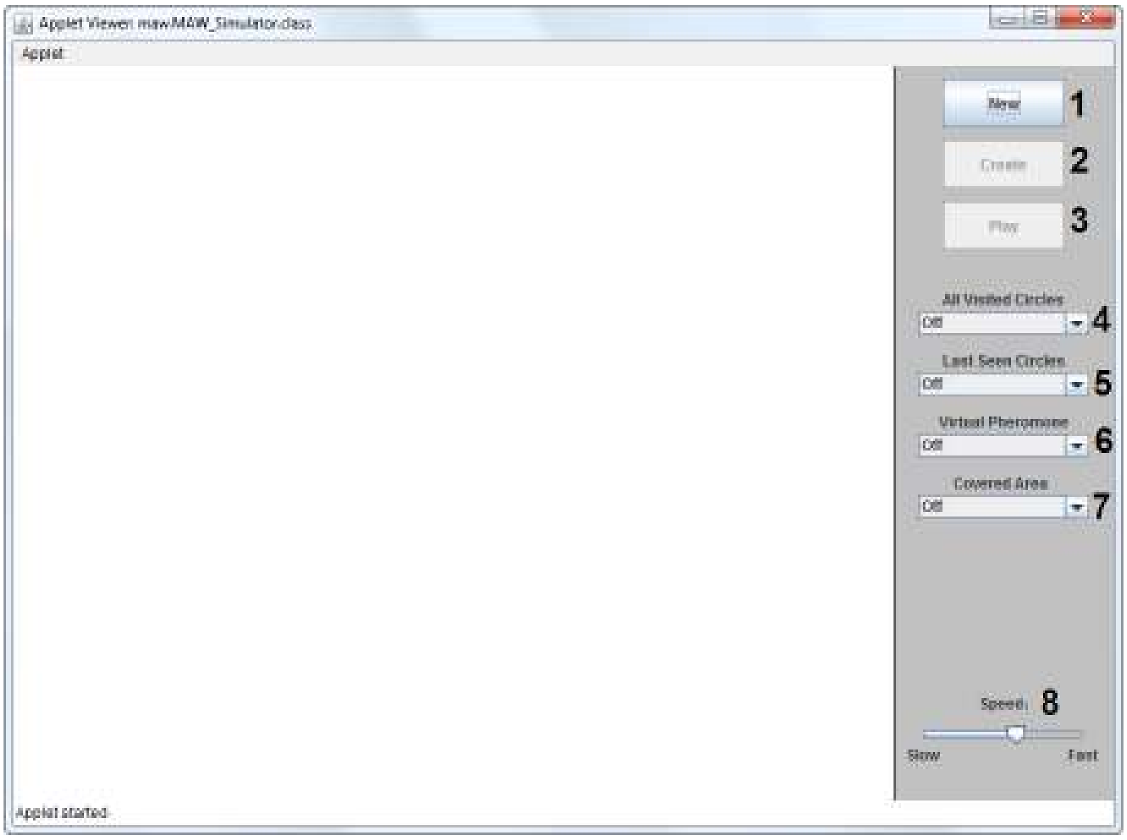
- **Obstacles** – Holds a list of all the Obstacles in the domain. Initially it includes the 4 walls, the domain is bordered by. As the user adds obstacles, they are added to this list. Includes functionality of drawing all the obstacles.

- **Markings** – Responsible for all visual output regarding the robot markings: the real and induced pheromone circles as well as the covered area. The class receives queries from the class 'Field' containing the database to be drawn (can be either the all visited circles DB or only the DB of the identified ones) and indicators as of what to draw: virtual pheromone, covered area etc.

- **Field** – Represents the environment of the robot. Is responsible for the maintenance of the database of all visited circles, as well as for identification of the intersections with previous circles.

Using the simulator

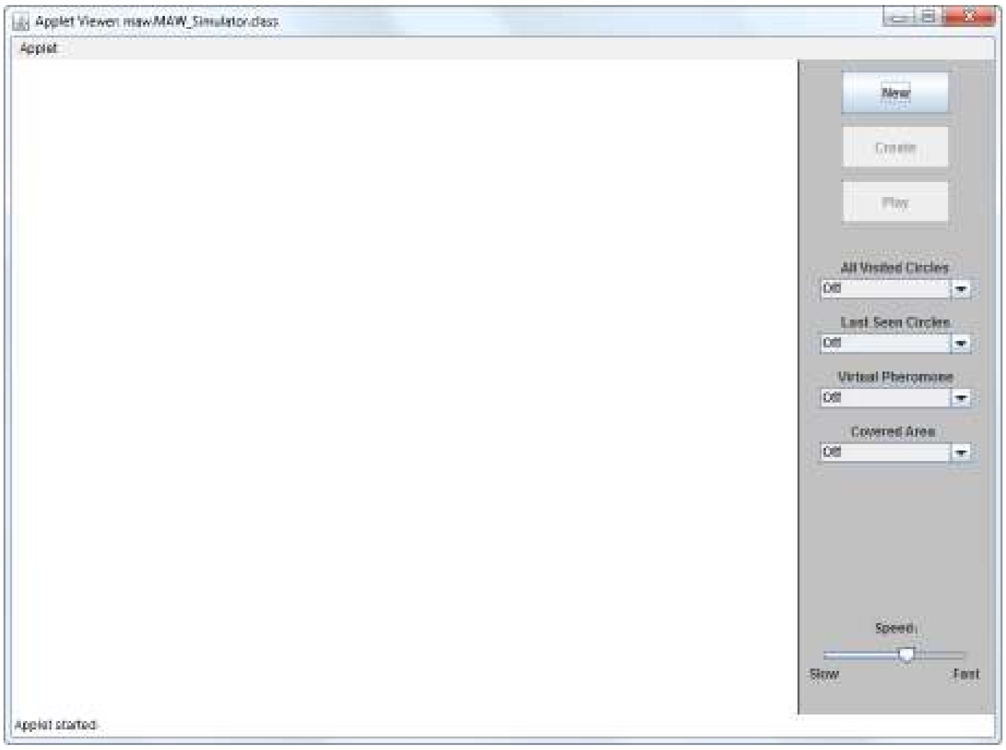
Interface:



1. New – Initializes a new domain, without obstacles.
2. Create – Creates the domain, with the obstacles that have been added. The robot's starting point is chosen.
3. Play – Starts the robot's movement.
   Pause – Pauses the robot's movement.
4. All Visited Circles – Toggles the display of all the circles which the robot marked.
5. Last Seen Circles – Toggles the display of the Circles the robot recognized in the last phase.
6. Virtual Pheromone – Toggles the display of the virtual pheromone induced by the circles the robot marked. (has no effect if option 4 is off)
7. Covered Area – toggles the display of the area covered by the robot.
8. Speed – Controls the robot's speed.

Operating guide

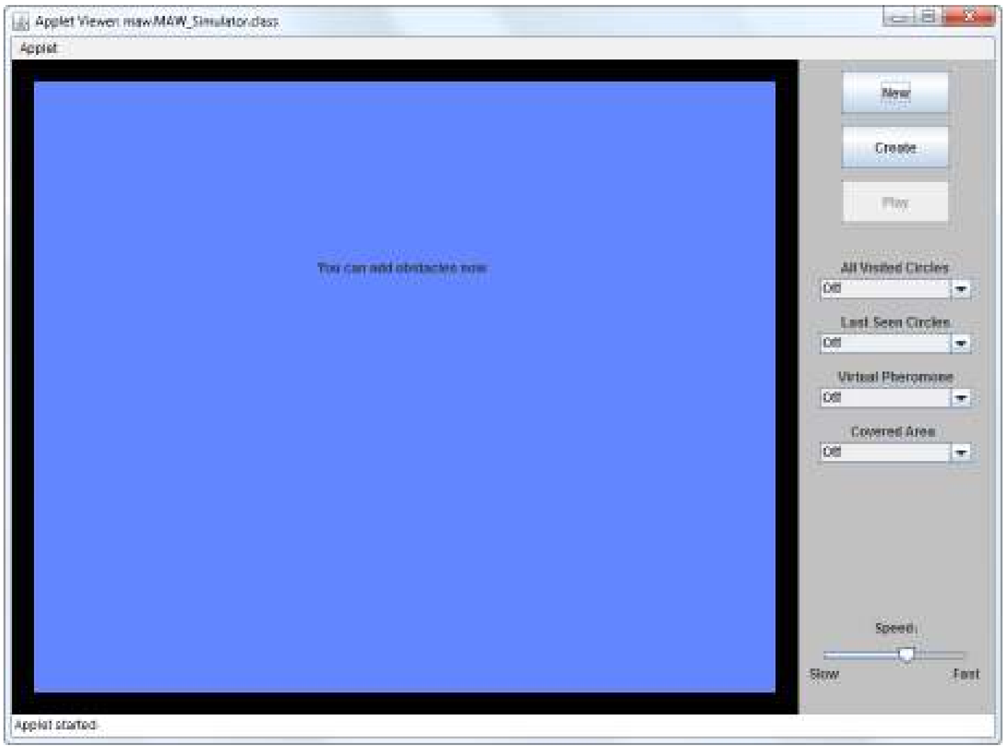
When starting the simulator, the following window opens:



All Visited Circles: Off
Last Seen Circles: Off
Virtual Pheromone: Off
Covered Area: Off

Applet started.

After clicking the ‘New’ button, a new domain is initialized:

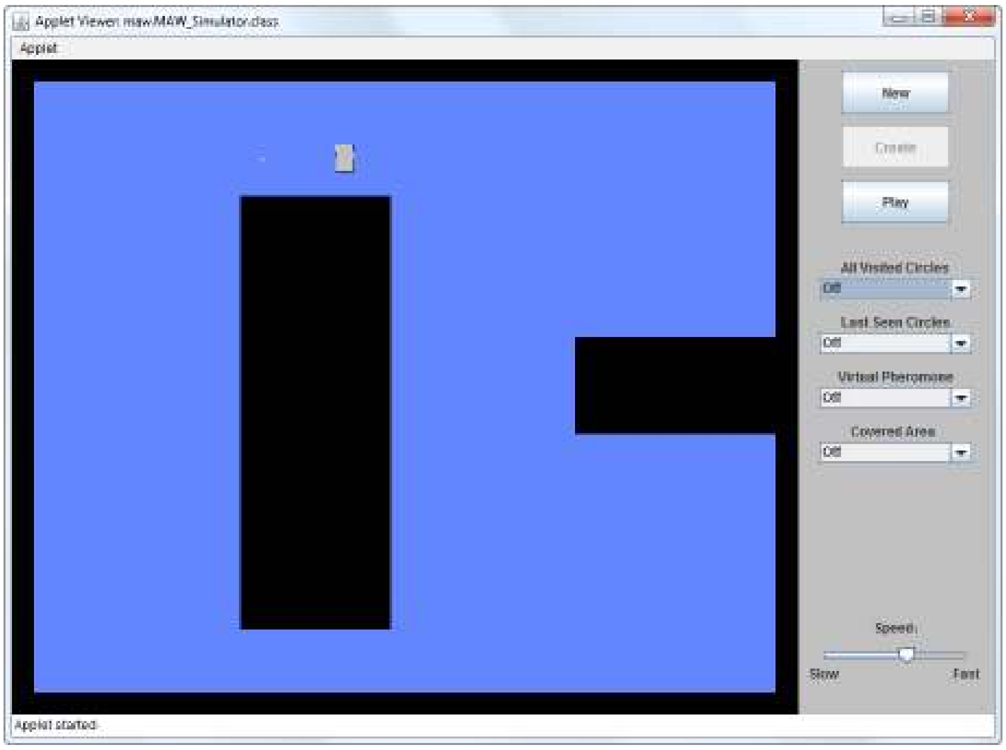


You can add obstacles now

All Visited Circles: Off
Last Seen Circles: Off
Virtual Pheromone: Off
Covered Area: Off

Applet started.

Now, obstacles can be added by dragging the mouse over the blue area. The last obstacle that was added can be removed by a click of the mouse.
All the obstacles can also be deleted by a click on the ‘New’ button.
After the user has finished adding obstacles, a click on ‘Create’ will choose the initial coordinates for the robot:

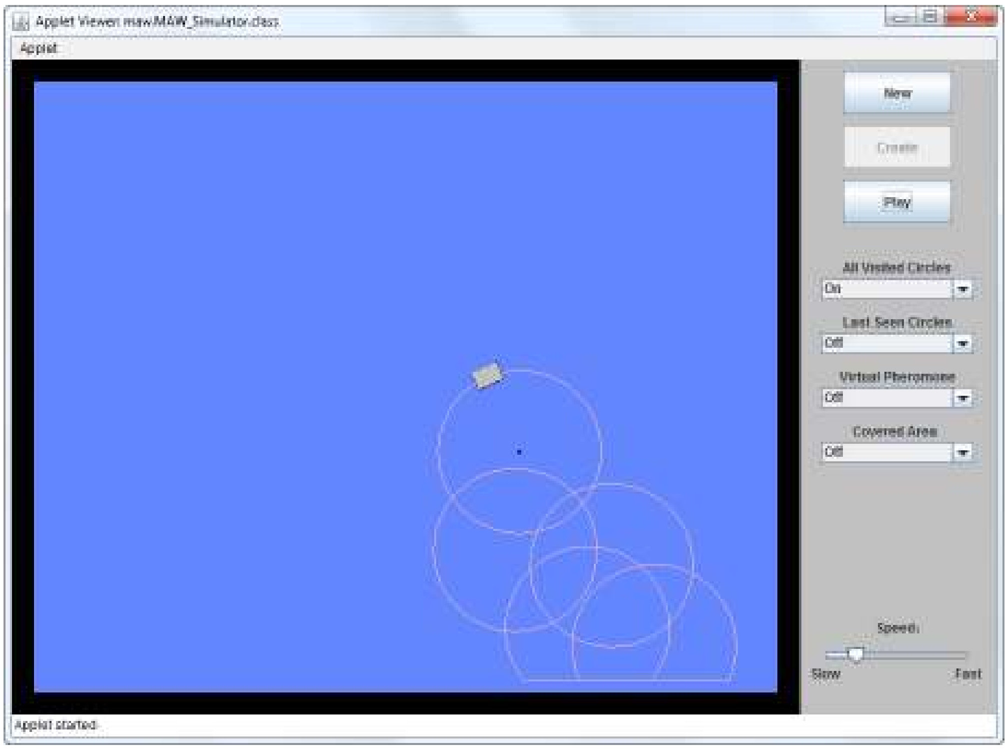


All Visited Circles: Off
Last Seen Circles: Off
Virtual Pheromone: Off
Covered Area: Off

Now the user can set the speed, and the data they wish to be displayed, and start the simulation by pressing ‘Play’. The above can also be changed during the simulation.

Displays

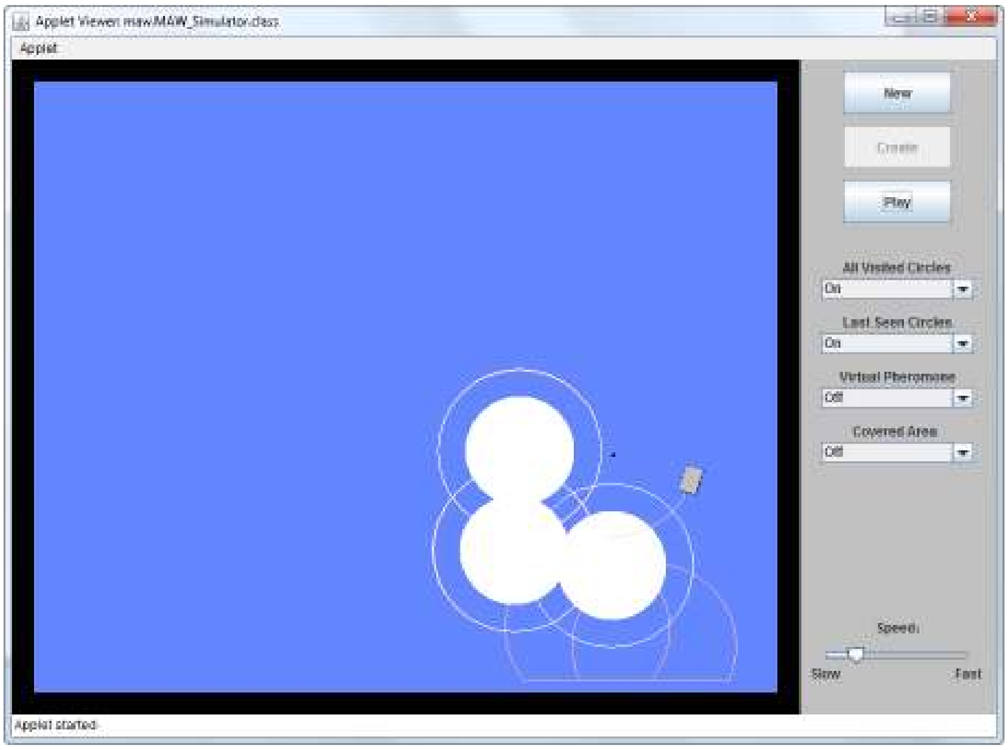
All visited circles:



All Visited Circles: On
Last Seen Circles: Off
Virtual Pheromone: Off
Covered Area: Off

Applet started.

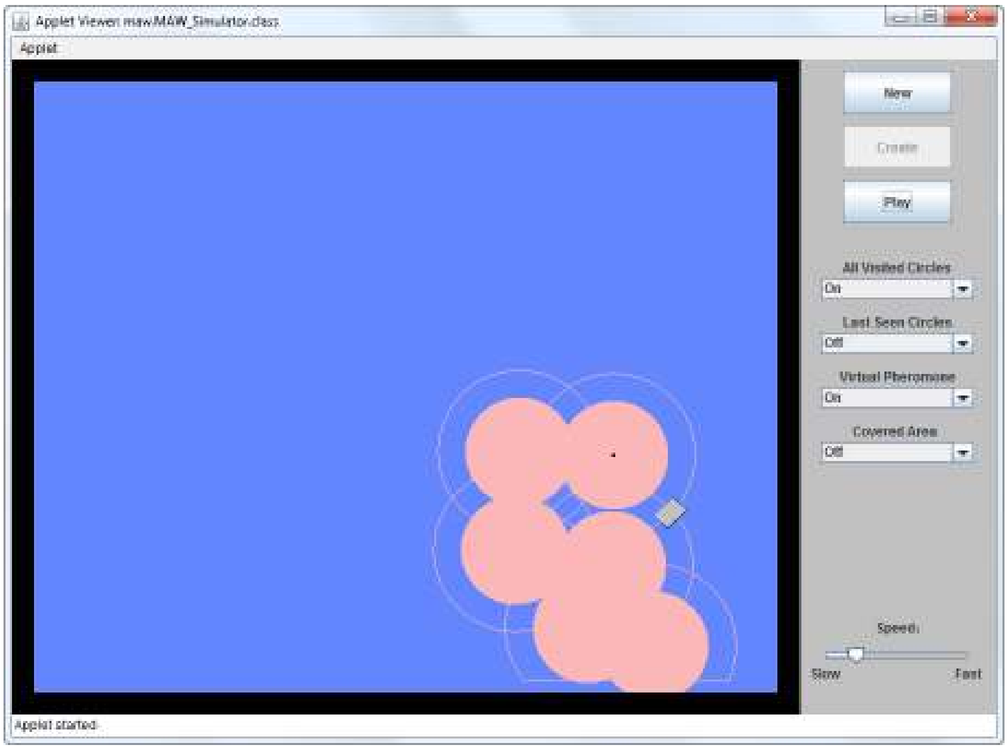
Last seen Circles (highlighted in white):



All Visited Circles: On
Last Seen Circles: On
Virtual Pheromone: Off
Covered Area: Off

Applet started.

Virtual Pheromone:


Applet Viewer: maw.MAW_Simulator.class

All Visited Circles: On

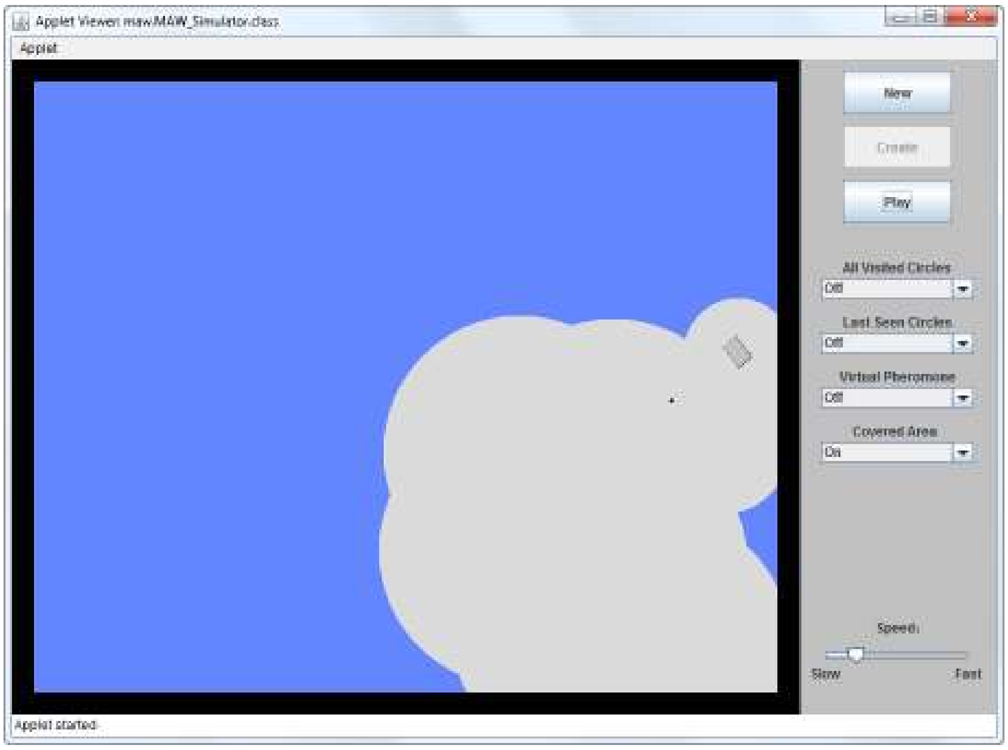
Last Seen Circles: Off

Virtual Pheromone: On

Covered Area: Off

Applet started.

Covered Area:


Applet Viewer: maw.MAW_Simulator.class

All Visited Circles: Off

Last Seen Circles: Off

Virtual Pheromone: Off

Covered Area: On

Applet started.