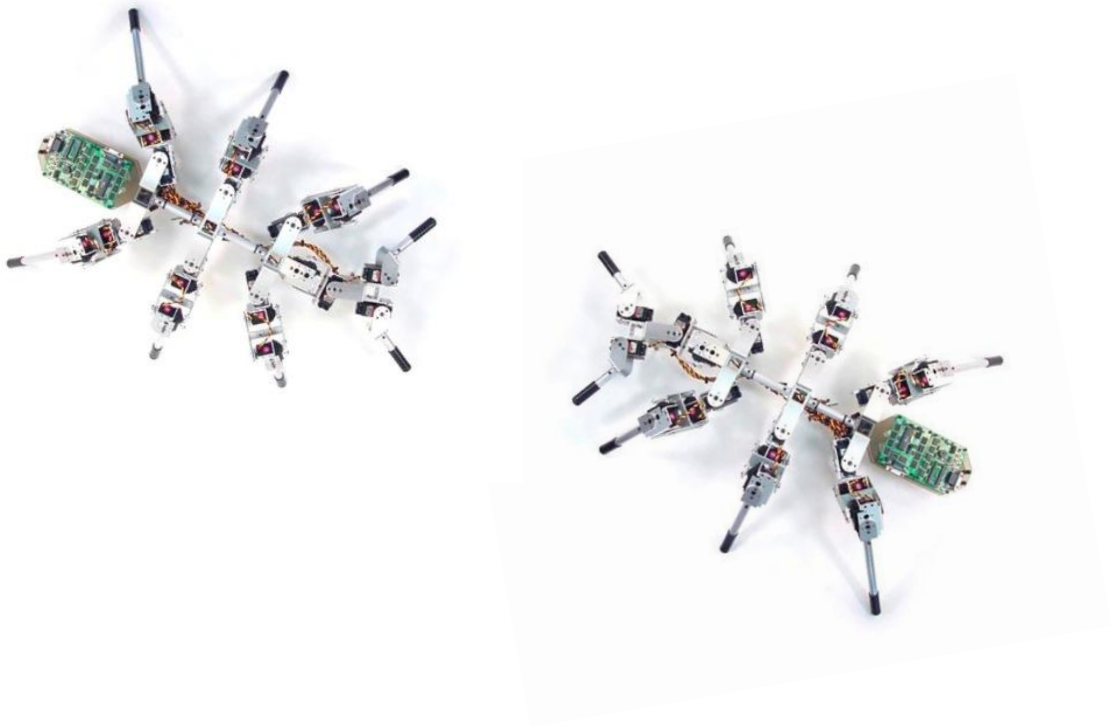
Intelligent Systems Simulation Project (236754)

Multi-A(ge)nt Graph Patrolling and Partitioning



Submitted by	**Oded Perez**
	Istratova Alexandra
Under the guidance of	**Yotam Elor**
Under the supervision of	**Prof. Alfred M. Bruckstein**

25.08.2010

Contents

Introduction

Problem Definition

In this project we present a simulator for one of the algorithms that provides a solution for graph patrolling. We are given an area (modeled as undirected graph) to patrol, and a number of low capability agents. The agents have limited memory and are able to perform local interactions between them. The goal of patrolling is to visit each point (vertex) as often as possible.

The simulator provides a solution based on the paper "Multi-A(ge)nt Graph Patrolling and Partitioning" by Yotam Elor and Alfred M. Bruckstein. The agents perform dynamic partitioning of the graph into sub-graphs, while each agent patrols its own sub-graph. Balanced graph partition is achieved over time due to "gas filled balloon" behavior – each agent applies pressure on the border vertices of its sub-graph. The agent that is applying the higher pressure (the agent that is controlling a smaller area) can conquer the vertex on the other side, and will do so with a certain probability. It has been shown empirically, that given enough time the agents will partition the graph to sub graphs almost equal in size, and reach a balanced partition, if it exists in the given graph (in a graph G with k agents the most balanced partition is a partition in which some agents control sub graphs of size $\lfloor |G|/k \rfloor$, and some of size $\lceil |G|/k \rceil$).

In this project we have implemented the Balloon DFS algorithm in Java programming language, which allowed us to run, check and research the algorithm.
In addition we have built a visual online simulator, providing a user-friendly tool for running the algorithm that is accessible online. The simulator allows control of the key algorithm/simulation parameters, as well as the control of graphical representation of the simulation.

In this report we will present detailed explanations of the main algorithms and data structures in our implementation, also providing a complete user guide for the simulator. We hope the end user can enjoy working with the MultiAgentsApplet as much as we've enjoyed building it.

Project Goals

The project aim is to implement the Balloon DFS patrolling algorithm in Java and provide an easy-to-use online simulator that allows user to witness the agents patrolling the graph as well as providing visual aids for following the algorithm's progress.

Simulator Description

The simulator enables the user to construct an undirected connected graph with no edges intersection, on which the Balloon DFS simulation is ran. It allows user to choose different parameters such as number of agents, number of vertices and others. The simulator uses the Voronoi diagram in order to visualize the area patrolled by every agent. The simulator also provides a chart of progress - how many vertices are controlled by each agent in every point of time.

Simulator Parameters

- Number of Vertices - number of vertices in the graph.
- Number of Agents - number of agents that are placed randomly on the graph vertices. Two agents cannot be placed on the same vertex, and the number of agents must be smaller or equal of the number of vertices.
- R - represents the maximal distance between two vertices that can have an edge between them.
- r - the probability that the agent will conquer the vertex when it is able to do so.

Simulator Capabilities

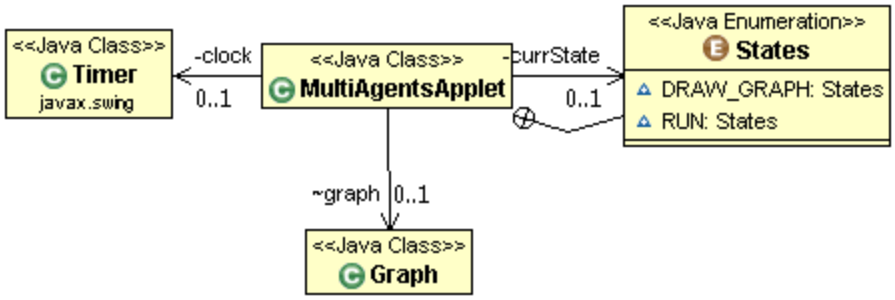
- Running the algorithm
- Constructing a new graph
- Animation speed control
- Showing or hiding agents
- Showing standard or connected Voronoi
- Displaying agents progress charts
- Pause
- Restart

Implementation

General Applet Overview

In our implementation the MultiAgentApplet holds an instance of a Graph along with an instance of javax.swing.Timer. The graph is the entity that manages all the other objects in the project.

The graph is the main class of the applet, all entities in the simulator are properties of the graph. There are two states the applet can be in: Draw_Graph or Run. At each timer event, the graph operates according to the state the applet is in.

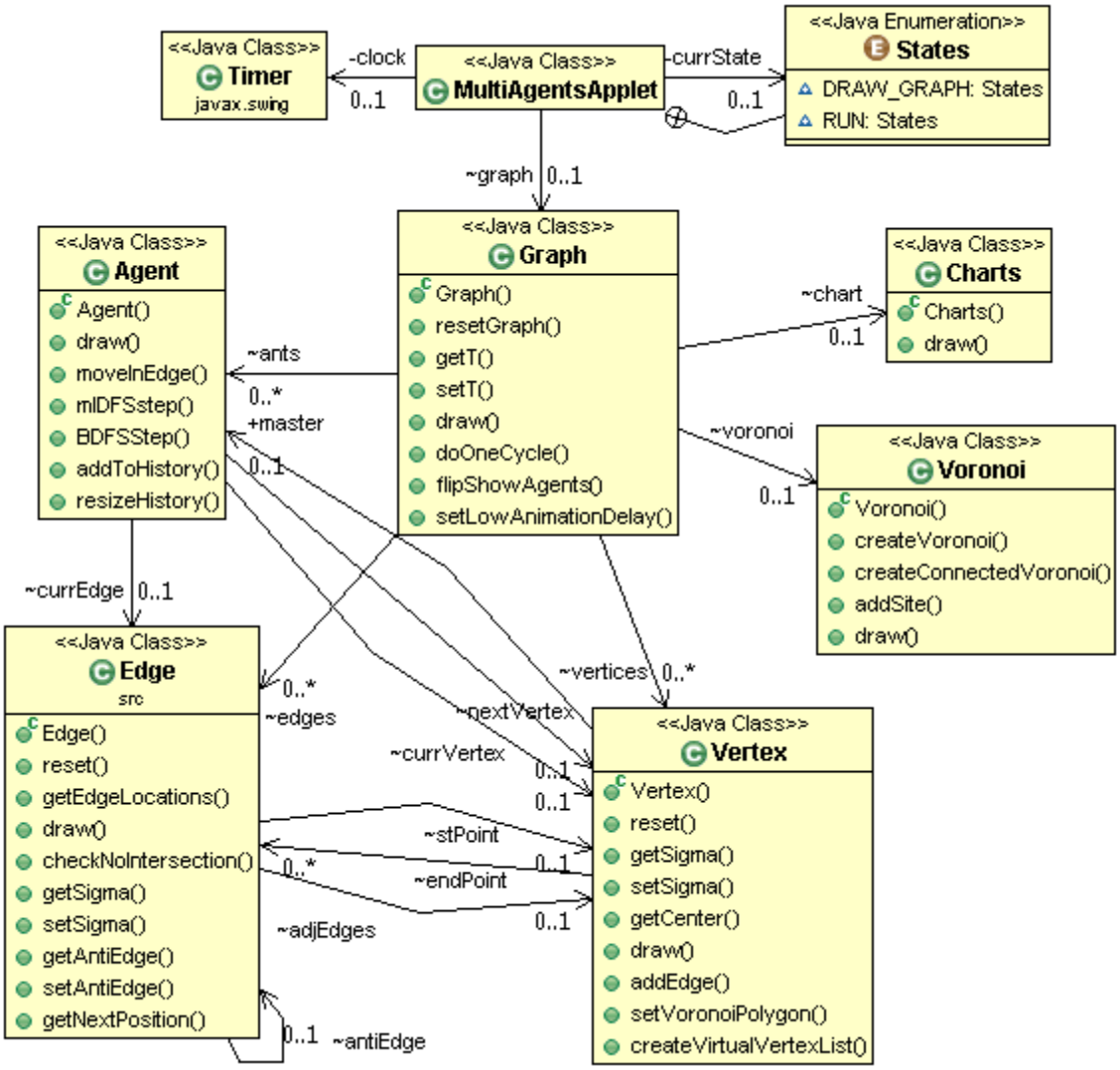


- **Draw Graph State**
  - The graph is drawn on the screen. The drawing is done by calling the draw functions of all graph sub elements.
  - If the graph is still empty, then calling the draw function will result in an empty image.
  - Drawing the graph repeatedly insures that it will be shown correctly in all times, even if the window the applet is located in will be hidden or moved.

- **Run State**
  - The applet calls the doOneCycle function of the graph, which in turn calls the BDFSStep on each agent. After all relevant changes were performed, the updated graph is drawn on the screen.
  - The call of the graph draw() function in the Run state occurs after either a BDFSStep was performed or the agents were moved on the edges for animation purposes.

Since the animation speed is set by the user, and we need to show the agents walking on the edges when the animation speed is low, we created a number of locations on each edge. At each timer tick the agent advances to the next location on the edge, therefore it takes several timer ticks before the agent reaches the next vertex, and only then t (the timer that counts BDFS steps) is updated and the next algorithm step is performed. This serves solely for the GUI look and feel. In case the agents are not shown or the speed is set to high, this part is skipped and each timer tick results in one algorithm step.

Detailed Class Overview



As can be seen from the class diagram, the Graph class contains references to:

Array of Agents

- The agent is an entity performing the Balloon DFS algorithm (the agents in our implementation can also perform mlDFS algorithm).
- The agent contains references to its current and next vertices. These references are needed to perform the algorithm step. The agent also holds a reference to its current edge used for rotating the ant image during the agent's advance on the edge.
- The agents are managed in an array data structure. An array was selected as the data structure for storing the agents information since we know their number at the creation of the graph, and we can access each agent in O(1) time complexity.

Array of Vertices

- The vertex has a LinkedList of adjacent edges, to make it easier to find a path from the vertex while running the algorithm. Vertex can be in two states: unconquered or controlled by an agent. While being controlled by an agent, the vertex will be surrounded by a Voronoi polygon colored with the agents color.
- The vertices are managed in an array data structure. An array was selected as the data structure for storing the vertices information since we know their number at the creation of the graph, and we can access each vertex in O(1) time complexity

LinkedList of Edges

- The edge has a reference to both of its vertices, and a reference to an antiEdge - the edge between the same vertices in the opposite direction.
- The edges are stored in a LinkedList because in graph initiation, each edge that is added to the graph is first checked and only then added. The edges are added and need to be accessed serially.

Voronoi Diagram

- The Voronoi Diagram is used to color the territory of each agent on the graph to make it easily distinguishable from other agents' territories.
- Two types of Voronoi are available:
    - Standard Voronoi diagram.
    - Connected Voronoi diagram.
- The connected Voronoi ensures that the territory controlled by a certain agent will look continuous. This is achieved by creating a number of virtual vertices along each edge, which are "controlled" by the same agent that controls the closest real vertex.

Charts

- The charts display how many vertices has each agent conquered over time. The progress of each agent is displayed by a single line in the agent's color.
- The chart is scaled on both axes over time.

Implementation Platform

The project was implemented on Java Applet platform, using Eclipse Java EE IDE for Web Developers.

Graphic User Interface

Jigloo Eclipse plug-in was used for GUI development.

Double buffering strategy was used to ensure the animation works smoothly without flickering.

We set the GUI “Look and Feel” to be “Windows Look and Feel” in order not to be dependent on OS or browser definitions for GUI components.

Voronoi Tessellation

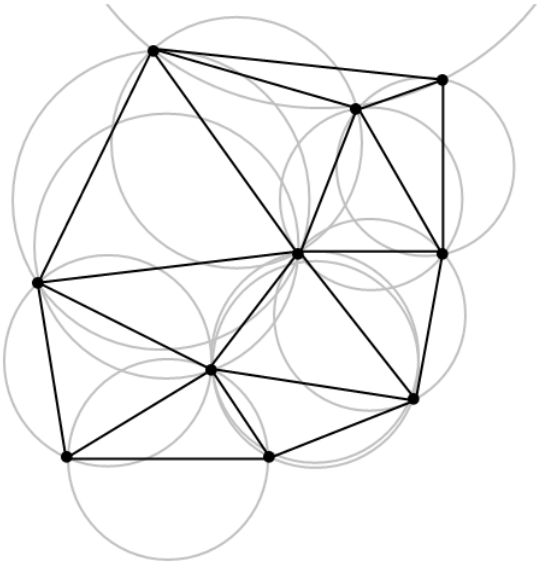
Delaunay Triangulation

A Delaunay triangulation for a set P of points in the plane is a triangulation DT(P) such that no point in P is inside the circumcircle of any triangle in DT(P). Delaunay triangulations maximize the minimum angle of all the angles of the triangles in the triangulation and tend to avoid skinny triangles.
The Delaunay triangulation of a discrete point set P in general position corresponds to the dual graph of the Voronoi tessellation for P.
By considering circumscribed spheres, the notion of Delaunay triangulation extends to three and higher dimensions, generalizations are possible to metrics other than Euclidean.
Our implementation, however, requires Delaunay Triangulation in 2D plane only.

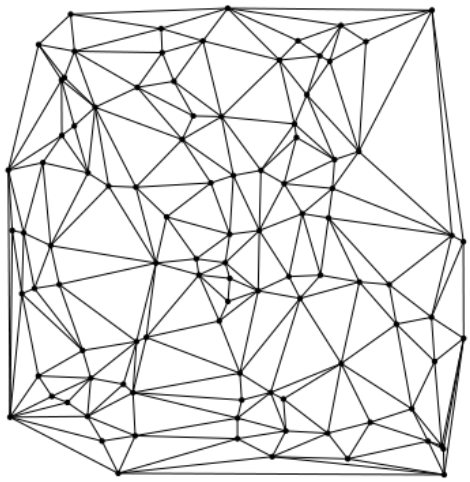
In this implementation of the Delaunay Triangulation, the triangulation is built within a large triangle whose vertices are well off-screen. This technique makes the code simpler as otherwise additional code would be needed to handle corner cases: new sites that are outside the convex hull of the previous sites, a set of points on the same line (notion of triangulation is undefined) and others.

The algorithm

To insert a new site, the algorithm walks across the triangulation, starting from the most recently created triangle, until it finds the triangle that contains the new site. This triangle and any adjacent triangles that contain this new site in their circumcircle are eliminated and the resulting empty spot is re-triangulated. The site-insertion part of this technique is commonly called the Bowyer-Watson Algorithm. The expected time to insert a new site is roughly $O(n^{1/2})$ where n is the current number of sites.



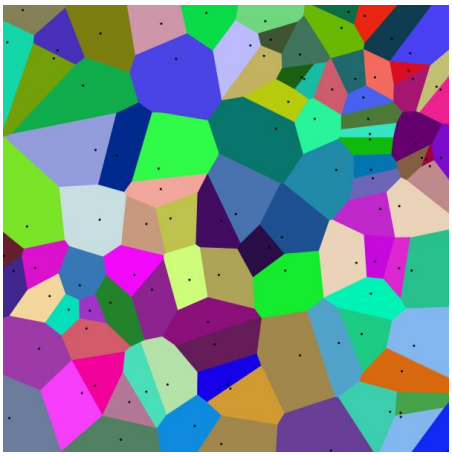
Delaunay Triangulation and Circumcircles



Delaunay Triangulation of 100 points

Voronoi tessellation

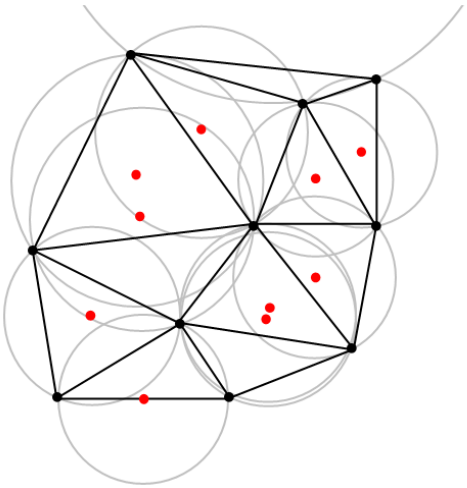
Voronoi diagram is a special kind of decomposition of a metric space determined by distances to a specified discrete set of objects in the space, e.g., by a discrete set of points.
In the simplest case, we are given a set of points S in the plane, which are the Voronoi sites. Each site s has a Voronoi cell, V(s) consisting of all points closer to s than to any other site.
The segments of the Voronoi diagram are all the points in the plane that are equidistant to the two nearest sites. The Voronoi nodes are the points equidistant to three (or more) sites.
Let *S* be a set of points in Euclidean space with all limit points contained in *S*. For almost any point *x* in the Euclidean space, there is one point of *S* closest to *x*. The word "almost" is used to indicate exceptions where a point *x* may be equally close to two or more points of *S*.



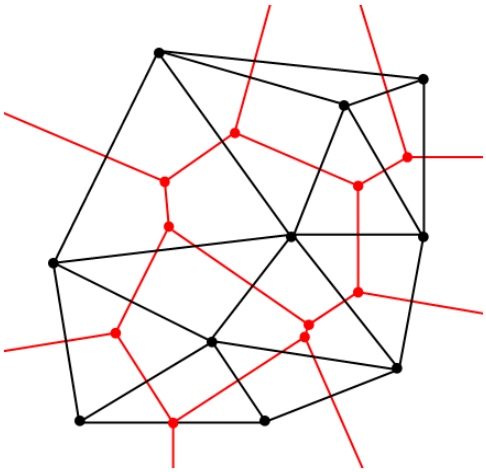
Voronoi tessellation

The dual graph for a Voronoi diagram corresponds to the Delaunay Triangulation for the same set of points *S*. The closest pair of points corresponds to two adjacent cells in the Voronoi diagram. Two points are adjacent on the convex hull if and only if their Voronoi cells share an infinitely long side.

In our implementation the Voronoi tessellation is built on-the-fly from the Delaunay Triangulation.
For each point we find all of its surrounding triangles.
For each triangle we find its circumcenter and connect all of the surrounding triangles circumcenters to create a Voronoi polygon.
All of the Voronoi polygons created define the Voronoi tessellation for the given graph.



Delaunay Triangulation and Circumcircles



Corresponding Voronoi tessellation

Our use of the Voronoi tessellation

We use the Voronoi tessellation in order to visualize the area that is patrolled by every agent in the given graph. Each Voronoi polygon is colored with the color of the agent that is patrolling the polygon's area.

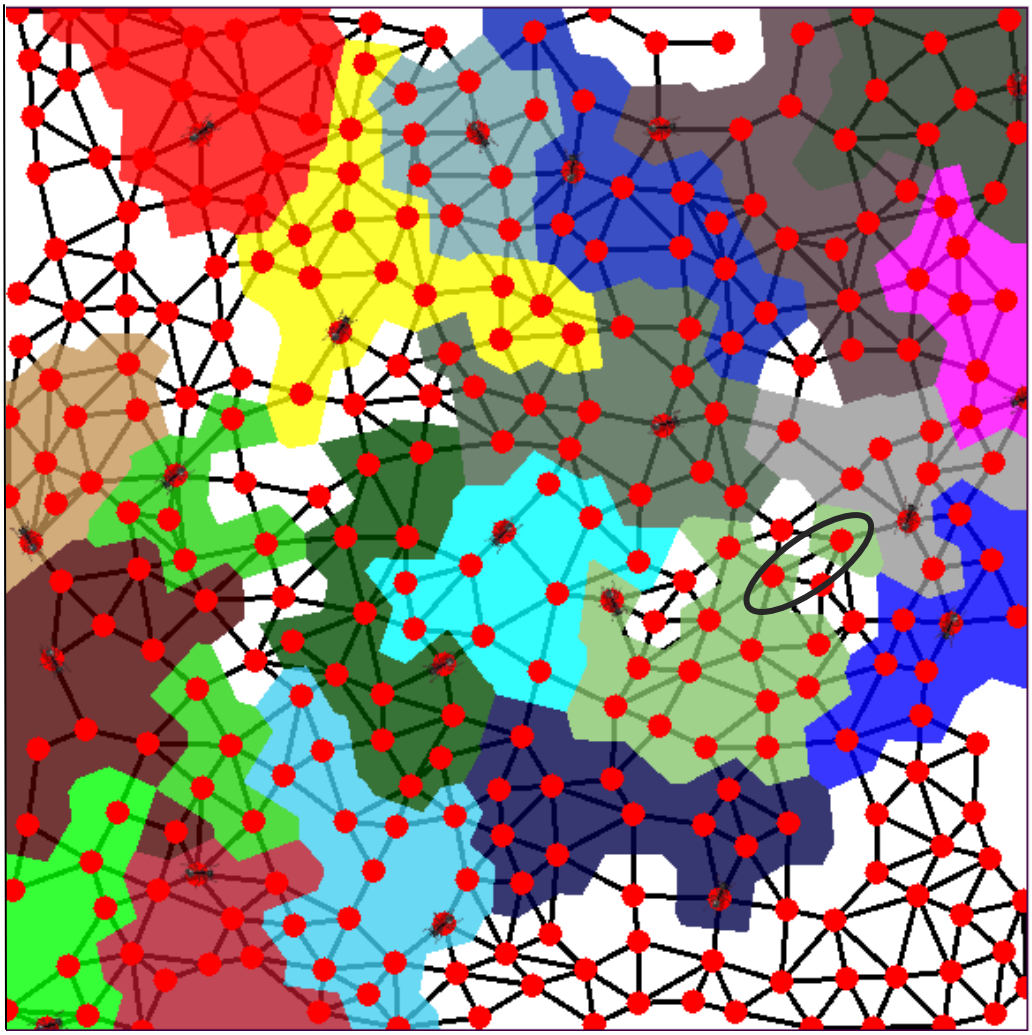
We have implemented two uses of the Voronoi tessellation:

1. Standard Voronoi - each point of the Voronoi is a vertex in the graph, which creates a surrounding polygon for every vertex. Every time an agent conquers a vertex, the polygon that is surrounding the vertex is colored with the agent's color.
2. Connected Voronoi - each point of the Voronoi is either an actual vertex or a virtual vertex that is marked on an edge. Every time an agent conquers a vertex, the polygon that is surrounding the vertex, as well as all the polygons surrounding the points on the edges that come out of the vertex and are the closest to the vertex, are colored with the agent's color.

Here are examples of the differences between the standard and the connected Voronoi:

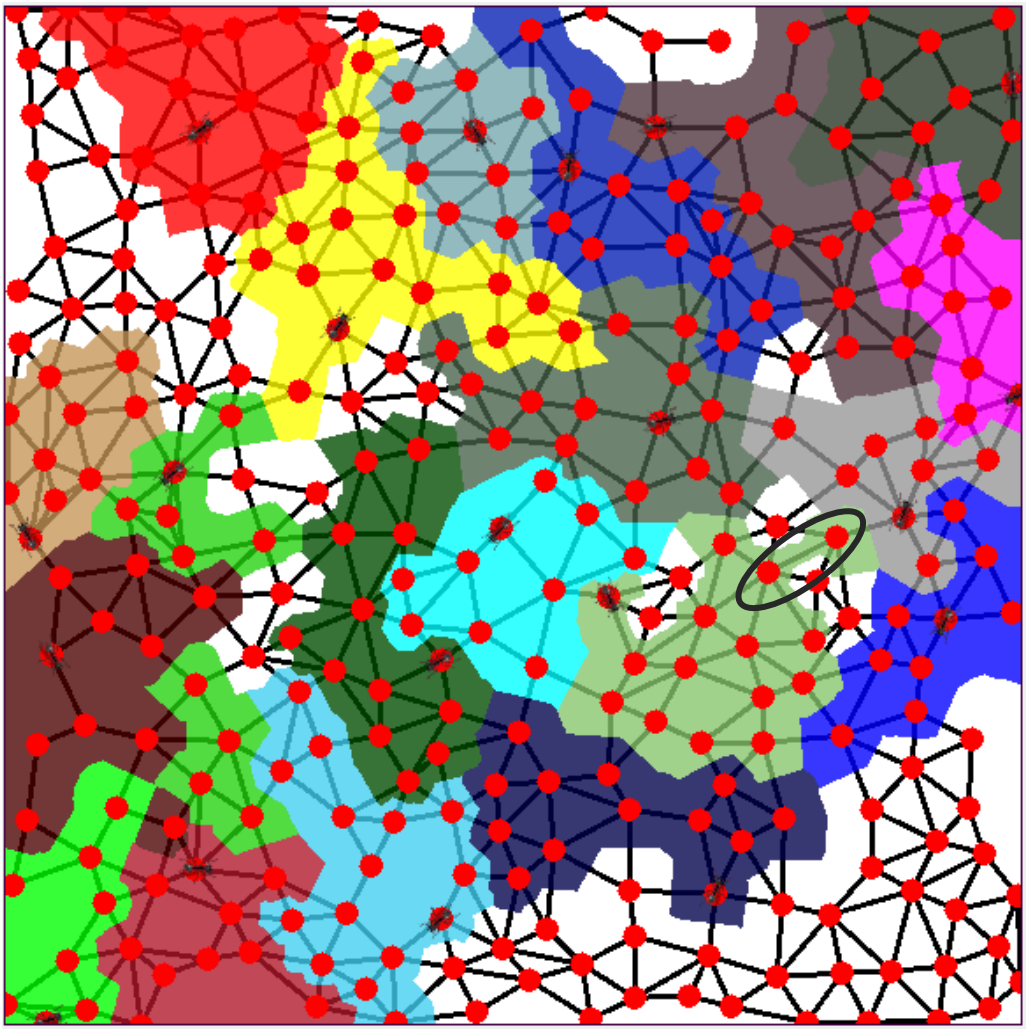
Standard Voronoi

The agent conquered two vertices, but the edge between them remained unpainted:



Connected Voronoi

The agent conquered two vertices, and the edge between the vertices is painted as well



Algorithms

Balloon DFS

The intuition behind the Balloon DFS algorithm is the behavior of gas filled balloons in a closed space. If we assume that the balloons are infinitely elastic and filled with an equal amount of gas, then over time the system will achieve its balanced state, and the balloons will fill the entire space.

In the same manner, the Balloon DFS agent controls a certain area on the graph, slowly spreading its control. The agent applies non-negative pressure on the border edges of its sub graph, and senses the pressure applied from the other side of the edge with an opposite sign.

The agent can conquer the vertex v with probability r in case $P(u,v) > 0$ (the pressure on the edge uv which the agent can see has to be strictly positive), which happens when the agent visits the edge uv twice, and no other agent visits this edge in the between (this, however, is insufficient for conquering the vertex). Thus, the agent that visits the border edges of its sub graph more often will be able (with a certain probability) to conquer their end vertices. From the definition of the DFS algorithm we can see that the agent who controls a smaller area will visit its border edges more often, therefore the agent with a smaller area size is likely to expand its control, and an agent with a bigger area size is more likely to "retreat".

As a consequence, given enough time, the agents will dynamically partition the graph into sub graphs of roughly the same size. It was shown empirically that in case the given graph can be partitioned in a balanced manner, the agents will be able to do so.

It was proven that the achieved patrol quality is at least half the optimal, since the maximum time lag between two successive visits to any vertex using the proposed strategy is at most twice the optimal.

The advantage of this approach is that the partitioning is dynamic, therefore:

- No preprocessing of the graph is needed.
- If the graph itself or the number of agents available is changed during the run, the graph will be repartitioned without external intervention.

The disadvantage of this approach is that reaching a good partition takes relatively long time.

BalloonDFS

1. *if u is not marked with AgentId then*
   1.1. go to random neighbor vertex marked by AgentId. (return)
2. *while* $\exists v \in N(u), \mid \sigma(u,v) < SearchLevel$ *and* $C(v) \neq AgentId$ *do*
   2.1. $\sigma(u,v) \leftarrow t$
      2.1.1. *if* $P(u,v) > 0$ *then*
         2.1.1.1. *if* $x < r$ *then*
            2.1.1.1.1. $\forall w \in N(v),\ \sigma(v,w) \leftarrow t+1,\ P(v,w) \leftarrow 0$
            2.1.1.1.2. $\sigma(v,u) \leftarrow t$
            2.1.1.1.3. $\sigma(v) \leftarrow t$
            2.1.1.1.4. $C(v) \leftarrow AgentId$
            2.1.1.1.5. *go to v (return)*
      2.1.2. *else*
         2.1.2.1. *if* $P(u,v) + AreaSize = -1$ *then*
            2.1.2.1.1. $P(u,v) \leftarrow 0$
         2.1.2.2. *else*
            2.1.2.2.1. $P(u,v) \leftarrow AreaSize$
3. *while* $\exists v \in N(u), \mid \sigma(u,v) < SearchLevel$ *and* $C(v) = AgentId$ *do*
   3.1. $\sigma(u,v) \leftarrow t$
   3.2. if $\sigma(v) < SearchLevel$ *then*
      3.2.1. $AreaSize \leftarrow (t - \sigma(v) + 1)/2$
      3.2.2. $\sigma(v,u) \leftarrow t$
      3.2.3. $\sigma(v) \leftarrow t$
      3.2.4. go to v (return)
4. *if* $\exists v \in N(u), \mid \sigma(u,v) = \sigma(u)$ *and* $C(v) = Agent$ *and* $\sigma(u) > SearchLevel$ *then*
   4.1. go to v (return)
5. *else*
   5.1. $SearchLevel \leftarrow t$
   5.2. $\sigma(u) \leftarrow t$
   5.3. $\forall w \in N(u),\ \sigma(u,w) \leftarrow \sigma(u,w) - 1$
   5.4. return

Algorithms for Graph Construction

Avoiding Edge Intersection

Before adding an edge to a graph we check whether it intersects with one of the existing edges. We use a simple geometric algorithm to ensure there are no intersections.

Say the first edge is between two points, (x_1, y_1) and (x_2, y_2), and has the gradient m_{12}, the second edge is between (x_3, y_3) and (x_4, y_4), and has the gradient m_{34}.

The intersection point will be:

$$x = \frac{m_{12} \cdot x_1 - y_1 - m_{34} \cdot x_3 + y_3}{m_{12} - m_{34}}$$

$$y = m_{12} \cdot x - m_{12} \cdot x_1 + y_1$$

And in case it belongs to both edges we should not add the new edge to the graph, as it causes an intersection with an existing edge.

The complete algorithm for no intersection is this:

noIntersection(Edge e, EdgeList edges)

1. *for each edge e' in edges*
   - *1.1. equation = linear equation that is defined by two points in e.*
   - *1.2. equation' = linear equation that is defined by two points in e'.*
   - *1.3. if (equation.gradient == equation'. gradient)*
     - *1.3.1. continue*
   - *1.4. find intersection point of equation and equation'*
   - *1.5. if (intersection point is on both edges)*
     - *1.5.1. return false*
2. *return true*

The algorithm complexity for each edge is $O(E')$, where E' is the number of existing edges in the current iteration. Thus the total algorithm complexity is $O(E^2)$, where E is the number of edges in the graph.

Ensuring the Graph Connectivity

When the graph is created, two vertices are connected if the distance between them is smaller than R and the new edge does not intersect with any of the existing edges (checked with *noIntersection* algorithm).

After that procedure, the following algorithm is run to ensure the graph is connected:

connectGraph(Graph G)

1. *run DFS on G and find maximal connected components.*
2. *while (Number of maximal connected components in G > 1) do*
   - *2.1. min_distance = INF*
   - *2.2. source_vertex = NULL*
   - *2.3. destination_vertex = NULL*
   - *2.4. choose one of the maximal connected component - mark it as C*
   - *2.5. for each connected component C' s.t. (C' != C) do*
     - *2.5.1. for each vertex v_1 in C do*
       - *2.5.1.1. for each vertex v_2 in C' do*
         - *2.5.1.1.1. if (distance between v_1 and v_2 < min_distance) && (no Intersection)*
           - *2.5.1.1.1.1. min_distance = distance between v_1 and v_2*
           - *2.5.1.1.1.2. source_vertex = v_1*
           - *2.5.1.1.1.3. destination_vertex = v_2*
   - *2.6. connect source_vertex and destination_vertex*

The algorithm's correctness is guaranteed by the fact that every connected component has at least one vertex. Thus, we are able to select two closest vertices on the perimeter of two connected components and connect them on every iteration.

The algorithm enforces the no edges intersection requirement as well.

The total algorithm's complexity is $O(N \cdot V^2 \cdot E)$

N – the number of connected components.

V – the number of vertices.

E – the number of edges.

Vertex Positioning

In our implementation, we wanted vertex position to be chosen randomly, but on the other hand, the graph needed to be distributed relatively uniformly under the given area.

In order to achieve that, we created the “Clear Radius”. The “Clear Radius” is a radius which ensures that for a given vertex, no other vertices will be located in the distance of the radius from its center. When we randomly create a vertex, we check no other vertex is inside its “clear radius”. In the case there is another vertex inside the “clear radius”, this vertex is dismissed and a new vertex is created at different random position.

The “clear radius” formula is:

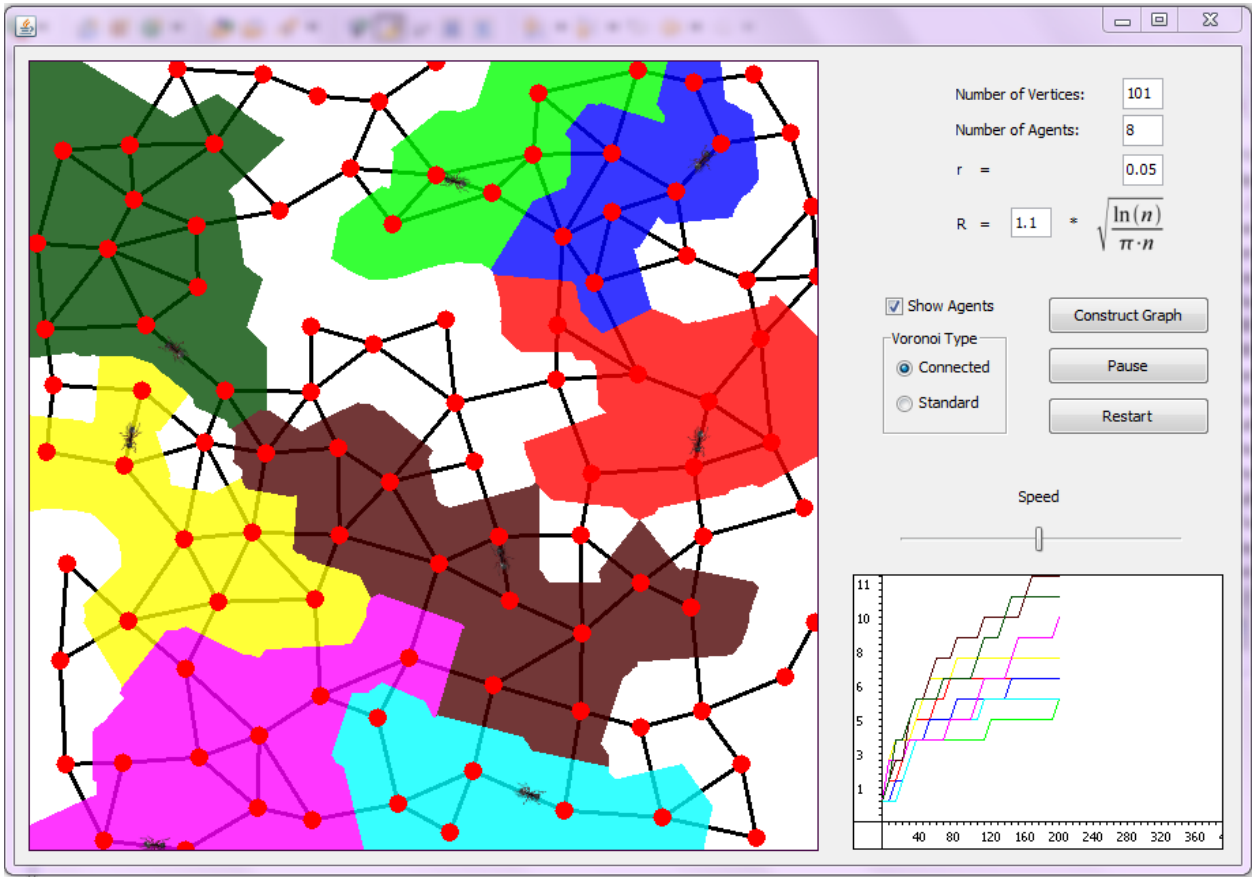
$$Clear\ Radius = \sqrt{1.5 * \frac{area\ size}{|V| * \pi}}$$

The formula was achieved by comparing the canvas area to the sum of all vertices “clear radius” areas. Since “clear radiuses” can overlap, as long as the vertices are not inside the overlapping area a $\sqrt{1.5}$ factor was added.

Simulation User Guide

The graphic user interface of the project is comprehensive and user friendly. In the following user guide we are going to explain the functionality of each GUI element and the influence of each input parameter, to enable the end user to get the exact output he/she wants.

The applet starts with a default set of parameters, which allows the user to start the simulation immediately. Running the simulation with these parameters will result in the graph below:



Applet Controls

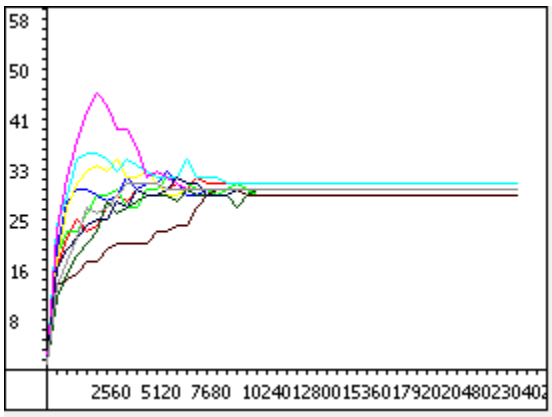
Construct Graph Button Creates a new graph for the simulation.
Constructing the graph can be done at any point of the simulation.
The result is always a new random graph.

Start/Pause Button	Starts the simulation, or pauses it if it's presently running.
Restart Button	Restarts the simulation (with the current graph).
Speed Slider	Controls the animation speed. When the speed is set to maximum the movement of the agents becomes jumping between the vertices.
Show Agents Checkbox	Controls the visibility of the agents - if they will appear on the graph or not.
Voronoi Type Control	Sets the type of the Voronoi diagram displayed on the graph: Connected Voronoi - the variation of Voronoi in which the intersection between two neighbor polygons contains more than one point. Thus all the area conquered by a certain agent (colored with its color) will look continuous. Standard Voronoi – regular Voronoi diagram.

Applet Input Parameters

Number of Vertices	The number of vertices in the graph should be chosen by the user prior to the graph construction.
Number of Agents	The number of agents in the simulation should be chosen by the user prior to the graph construction. Those agents are placed randomly on the graph vertices. Two agents cannot be placed on the same vertex, and the number of agents must be smaller or equal of the number of vertices
R	Represents the maximal distance between two vertices that can have an edge between them.
r	The probability that the agent will conquer the vertex when it is able to do so.

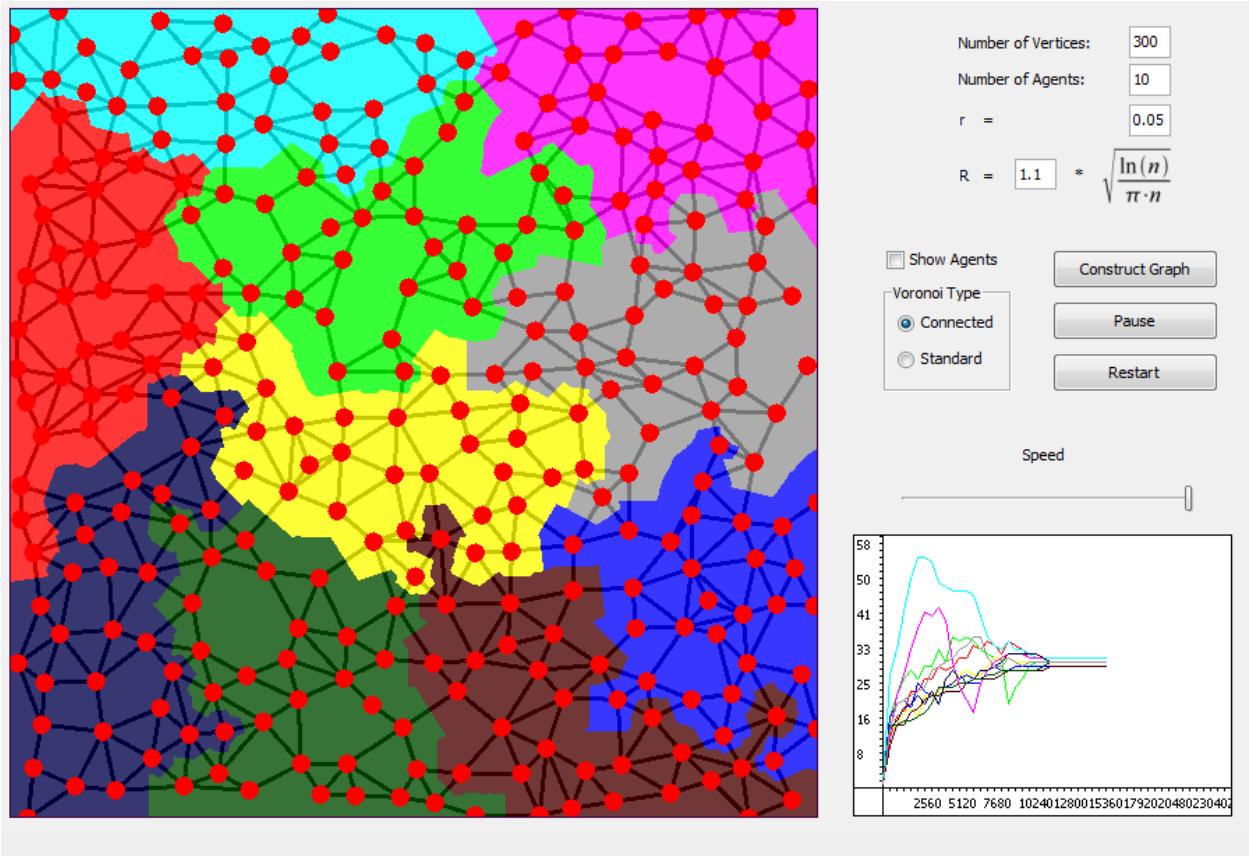
Chart



The chart shows the advance of each agent over time. Each agent is represented by a single line of its color. The Y axis stands for the number of vertices the agent controls. The X axis stands for the number of cycles passed since the start of the simulation.

It can be seen that over time the algorithm converges - each agent controls an area of roughly the same size as the others, and there are no more conquests. The lines representing the agents' number of vertices in the chart converge.

The algorithm convergence in the simulation on a graph with 300 vertices and 10 agents:



Summary

In this project we have implemented, under the dedicated supervision of Yotam Elor and Prof. Alfred M. Bruckstein, a multi-agent patrolling and partitioning algorithm, which is being run "simultaneously" by number of agents and results in a dynamic partitioning of the graph into agent controlled areas.

We have implemented an algorithm for creating the graph. The algorithm ensures graph connectivity while preventing edges intersection, as well as ensuring a "clear radius" around each vertex (in which no other vertices can be added). In order to visualize partitioning of the graph a Voronoi diagram is used. All of these operations are performed in a short matter of time for relatively large number of vertices.

In addition we've been exposed to a whole new world – Java Applet.

Using a simple timer, we have built a synchronized animation of A(ge)nts moving along the edges in the graph. We have also enabled the user with an option of changing the animation speed on the fly, causing the A(ge)nts to move slower or faster.

The Voronoi diagram is being colored dynamically upon vertex conquests, in transparent colors corresponding to the conquering agent color. We have also implemented an alternative type of Voronoi, a connected Voronoi that allows continuous coloring of the agent's controlled area. The agents advance is shown in a dynamically rescaling flow chart.

The applets biggest advantage is that it can be placed on a web server, so that the simulations can be run by multiple users all around the world, with no need for installation of the environment or even the application itself.

We enjoyed working on the project. We learned a lot and acquired a valuable experience in the field of multi agent robotics.

With gratitude,

Oded & Sasha.

References

[1] Multi-A(ge)nt Graph Patrolling and Partitioning by Yotam Elor and Alfred M. Bruckstein.

[2] http://en.wikipedia.org/wiki/Voronoi

[3] http://en.wikipedia.org/wiki/Delaunay_triangulation

[4] http://www.cs.cornell.edu/home/chew/Delaunay.html

[5] http://en.wikipedia.org/wiki/Bowyer%E2%80%93Watson_algorithm