

Intelligent Systems Simulation Project (236754)

Autonomous Multi-Agent Cycle Based Patrolling



Submitted by *Maxim Derkach, 318103140*

Under the guidance of *Yotam Elor*

Under the supervision of *Prof. Alfred M. Bruckstein*

Summer 2009, CS, Technion

Table of Contents

1. Introduction

1.1 Problem Definition

1.2 Solution Description

1.3 Project Goals

2. Implementation Design

2.1 Programing Concept

2.2 Data Structures and Algorithms

2.3 Implementation Platform

3. Running the Simulation

4. Summary

5. References

1. Introduction

1.1 Problem Definition.

We are dealing with the multi-agent patrolling task. Let us refer to wikipedia for the definitions of patrol and multi-agent system:

A **patrol** is commonly a group of individuals or units that are assigned to monitor a specific area.

A **multi-agent system (MAS)** is a system composed of multiple interacting intelligent agents.

Multi-agent systems can be used to solve problems which are difficult or impossible for an individual agent or monolithic system to solve. Examples of problems which are appropriate to multi-agent systems research include online trading, disaster response, and modeling of social structures.

Multi-agent systems can manifest *self-organization* and complex behaviors even when the individual strategies of the agents are simple.

The agents in a **multi-agent system** have three important characteristics:

- **autonomy**: agents are autonomous, i.e. there is no direct communication between the agents or between the agents and any control units.
- **local views**: no agent has a full global view of the system.
- **decentralization**: there is no controlling agent or unit.

Thus, our goal is to perform an area patrolling using ant-like agents. The area we deal with is a group of stations with paths between them. The stations are have to be visited as frequent as possible. The time intervals between two subsequent visits to the same station are called idleness. In other words, we can say, that our goal is to achieve the minimal idleness possible.

For the formal analysis we think of the area as of graph $G(V,E)$, where V is the group of vertices (each vertex models station) and E is the group of edges (each edge models a path between stations).

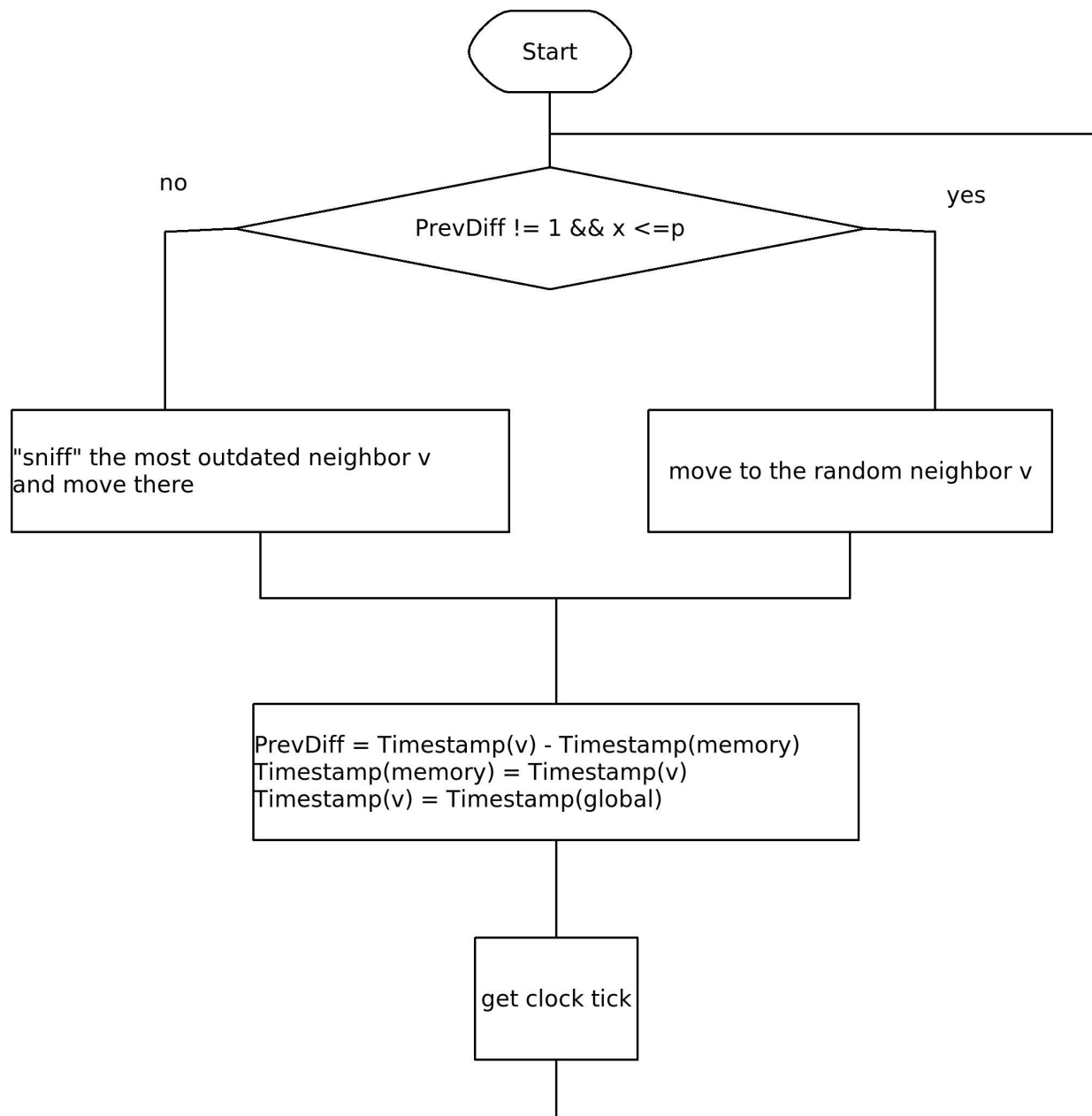
1.2 Solution Description

The approach we use is as follows. The agents are divided into a leader and a herd. Each type has a predefined behavior. A leader's function is to find the path on the graph and all the rest has to follow it. Every agent can mark the vertex with special markings based on time stamp. The leader's mark differs from herd-agent's mark. It is used by leader himself to find the path and by herd-agents to recognize the path to follow. The herd-agents use 2 markings and they are indistinguishable between different herd-agents. These marking are used to spread agents uniformly over the path.

Thus, the problem is divided into two parts: *circle finding* and *swarm deployment*.

Circle finding is the algorithm run by leader, the *swarm deployment* is algorithm run by herd-agents. Let us take a closer look at these algorithms.

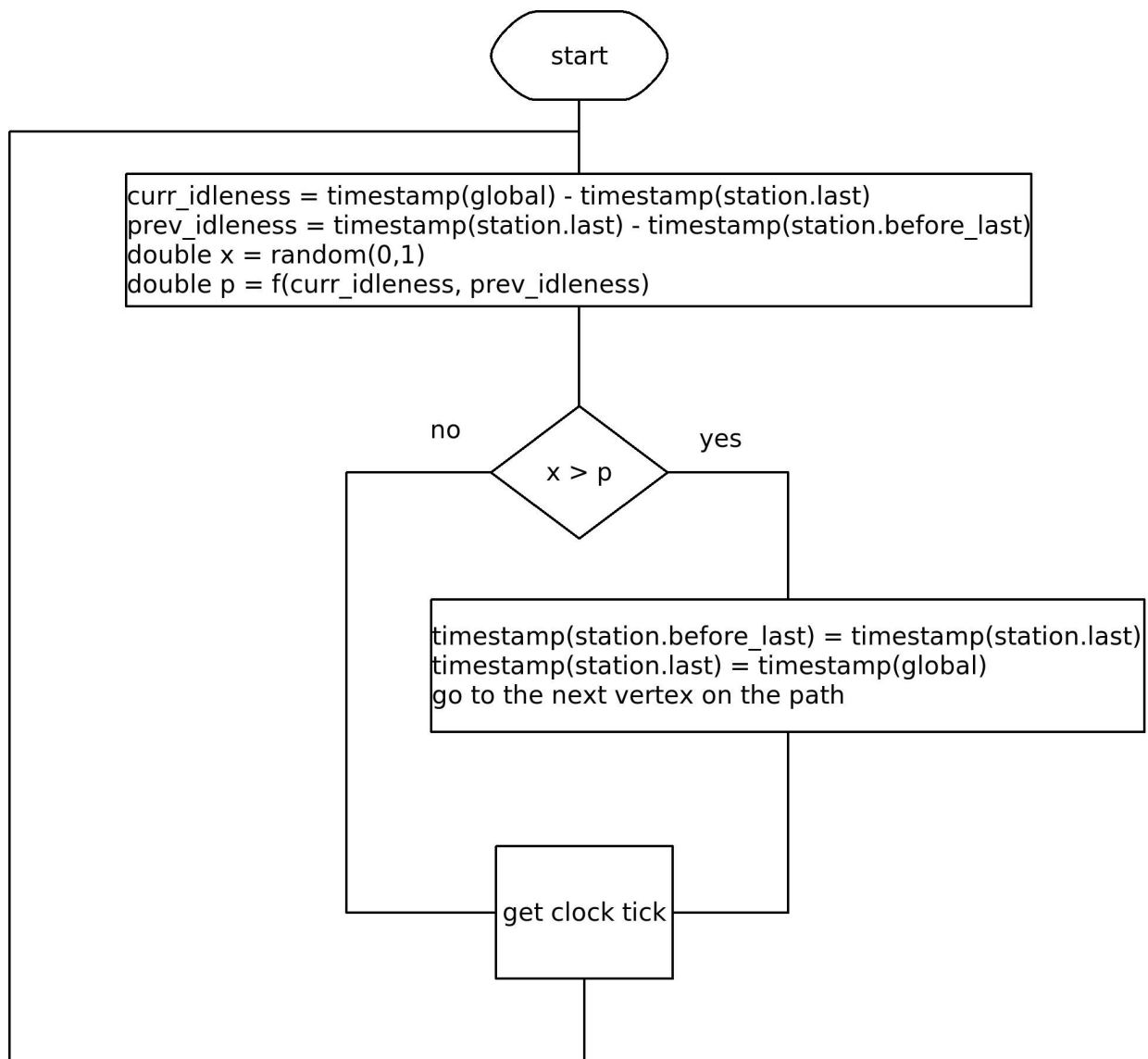
Cycle Finding. Note that, what we (the leader) are actually looking for is Hamiltonian cycle(which is optimal) on the provided graph. The leader makes a move every time cycle. The decision where to go is made based on the following scheme:



The schema above is general PVAW(probabilistic vertex-ant-walk) schema. The addition here with comparison to regular VAW is the possibility to make a random move in case that some vertex is visited not at the same order it was visited in the previous time. Which is sufficient to indicate that the path is not a Hamiltonian cycle.^[1] This addition allows to

prevent convergence to a non-Hamiltonian cycles, while always following a Hamiltonian cycle when found.^[1] Note also, that the neighborhood definition depends on how far the agent can “sniff”, in other words, the meaning of “move to the random neighbor” and “move to the most outdated neighbor” are subjects to change depending on how a neighborhood is defined. In strict terms, when we decide which algorithm PVAW, PVAW2 or PVAW3 to run, we actually define the neighborhood to be N_1 , N_2 or N_4 appropriately. The formal definition of $N_d(u)$ can be found at [1].

Swarm Deployment. In order to achieve the best perimeter patrol agents should spread uniformly over the cycle. Each herd agent has a simple decision routine. Upon arriving to a vertex he checks whether he needs to stay and increase the distance between himself and his predecessor or to rush forward because he is on time or being late. There is also a chance of probabilistic move when the previous idle period is greater than the current strictly by 1 time cycle. This allows to prevent agents grouping for low n/k ratios, where n – the number of vertices in the graph, k – the number of agents. Herd-agents decisions are made based on Algorithm 6^[1]. Its scheme is presented below:



Where the function $f()$ is defined by:

```

double f(int l1, int l2)
{
    if (l1 < (l2 - 1)) return 1;
    if (l1 == (l2 - 1)) return  $\frac{1}{n}$  ;
    return 0;
}
  
```

Let us note, that the cycle finding process has a probabilistic convergence time. This leads

us to the state, where the herd-agents markings are very outdated at some vertices, which in its turn causes convergence time to be very high. Consider a vertex v_i that has $\delta_1 \ll \delta_2$ at the moment the cycle is found. Agent a_x entering v_i will stay there for at least $T = \delta_2 - \delta_1 - 1$ which is big. Moreover, all the rest of agents will stuck at v_i for the same reason. After waiting enough for δ_1 to reach $\delta_2 - 1$ one of the agents a_y will have to leave v_i and update δ_2 and δ_1 . By tracing the definitions of deltas and following the deployment flow chart, one can see that after update made by a_y we have $\delta_1 = 0; \delta_2 = T$ and thus, $T^{new} = \delta_2 - \delta_1 - 1 = T - 1$. Which means that all agents currently at v_i won't move for the next T^{new} time cycles. And even if there are no jams at other stations, the deployment convergence time tends to be $O(T^2)$.

In this circumstances, it was decided that the leader will be responsible for one more function: resetting the herd-agent visits markings after the cycle was found. This requires the leader to know $|V|$, but not G's topology. And thus, its use in most cases may be considered as non-breaking decentralization concept. Such a knowledge allows the leader to notice when the cycle is found. After the cycle is found, the leader moves from vertex to vertex on the cycle and updates τ_1, τ_2 until it completes the whole loop. After resetting the taus of all the vertices, leader continues his usual routine.

While running simulations we found two good manners to do this:

- $\tau_1 = \tau_2 = currentTimestamp$
- $\tau_1 = currentTimestamp; \tau_2 = currentTimestamp - (\frac{n}{k-1})$

Both decrease the deployment convergence time to be $O(n^2)$, because after the taus resetting process is over, T is at most $n + \lceil \frac{n}{k} \rceil$, and the rest of analysis persists.

The second approach (though demanding from leader to know the herd size k) seems (experimental conclusion, no analysis was done here) to provide better results. We use the second approach in our simulation implementation.

1.3 Project Goals

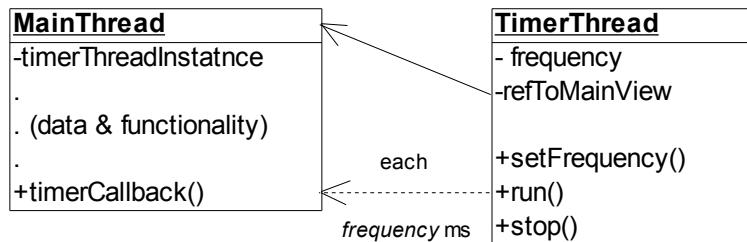
The project goals are to provide a visualization of the above algorithm. Simulation has to provide:

- Generate and draw $G_{n,r}$ graphs with given n and r .
- Place and show k agents, for a given k .
- Selecting neighborhood size by choosing algorithm (PVAW, PVAW2, PVAW3)
- Highlight the path (cycle, when found).
- Change the simulation speed.
- Restart the simulation for the same graph.
- Switch between initial and circle like views.

2. Implementation Design

2.1 Programing Concept

As long as we deal with animation, the need of periodical redrawing arises. It was found that very natural solution is to split the program into two threads: the main thread (responsible of interaction with a user and containing all the necessary data and functionality) and the timer thread (has only one purpose – generate callbacks to the main thread with a given frequency). Schematically it can be shown this way:

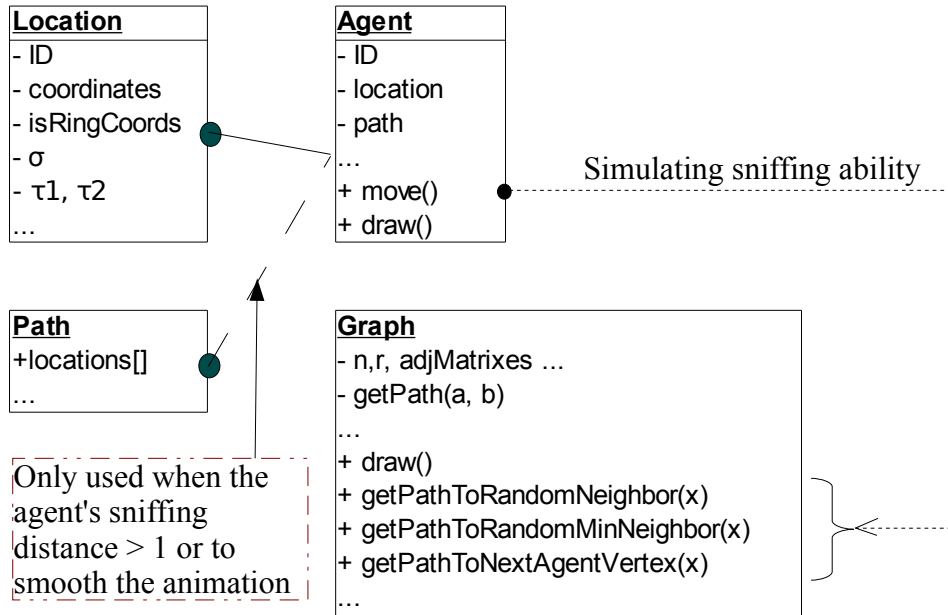


Thus, as we can see, the main thread holds an instance of the timer thread, which can be started or stopped upon user's action. Main thread is also responsible of gathering a user input. The input is divided into two groups: initial and runtime.

Initial input represents graph parameters (as we deal with $G_{n,r}$ graphs these are n and r), number of agents and which algorithm to use.

Runtime parameters are animation speed and view (initial vertices arrangement or circle-like).

For now, we have general understanding of how the animation runs. Let us take a closer look into the simulation core class diagram:



Each time cycle the main thread calls for each agent to move and after that for the draw methods both of the agents and of the graph.

Let us make a distinction: analysis and flowchart at 1.2 refer to *station*, while here we refer *location* as agent placement. The following terms will be used in future explanation:

- **real location** – the location that is equal to station for analysis and has the same coordinates as appropriate station;

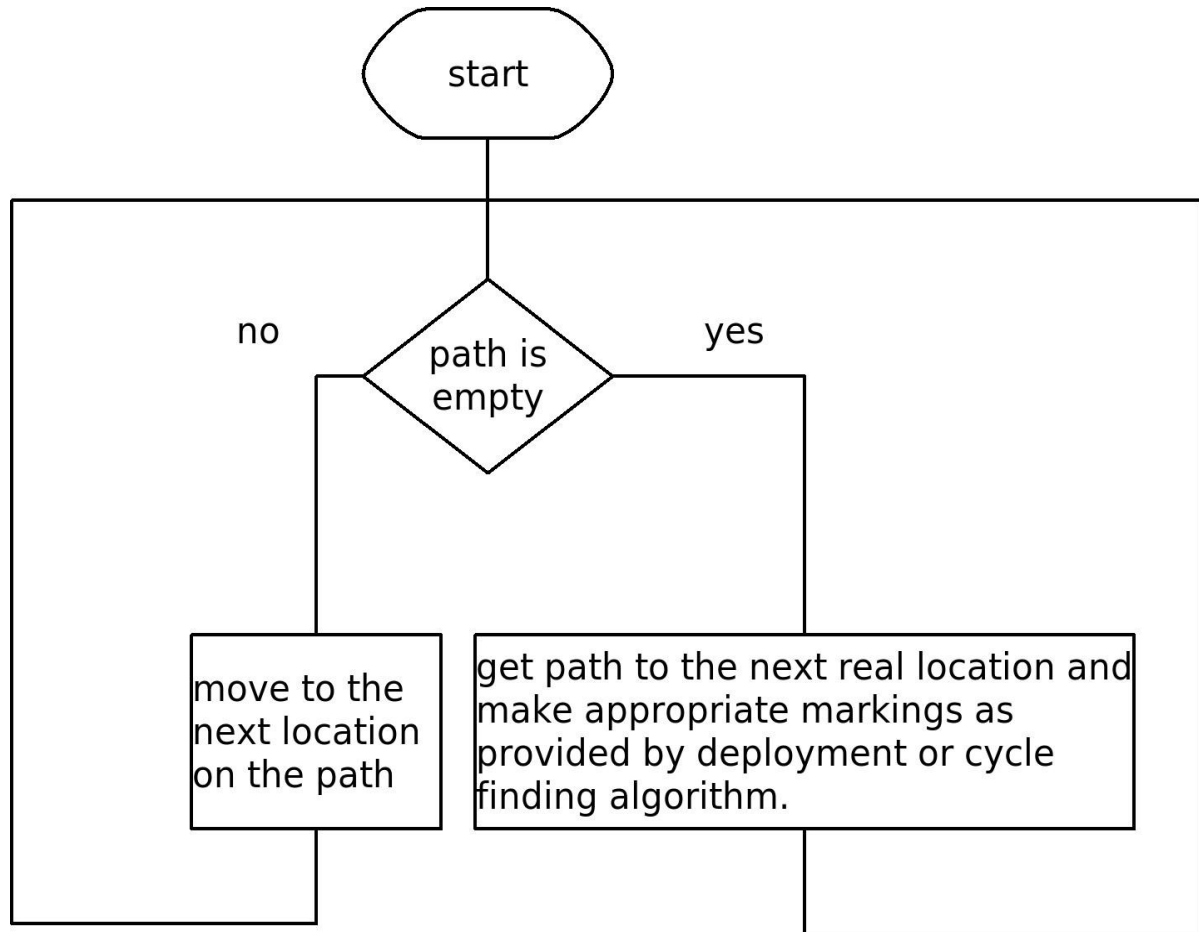
- **semi-real location** – the location that has the same coordinates as the appropriate station, but is irrelevant for the analysis. These locations are used for the animation purpose for PVAW2 and PVAW3 algorithms only, where the agent can has destination station not in his direct neighborhood. Thus, semi-real location is a matter of circumstances, in other words if the agent intent to bypass the station while moving to another one, its appropriate location called semi-real.

- **virtual location** – the location that has no common coordinates with any station nor participates in analysis. This kind of locations has bare animation purpose. They are actually have coordinates on edges and are aimed to smooth the animation.

Depending on the agent's role (leader or herd agent) and current circumstances (markings state) the agent check that he has a path to follow. If not, he retrieves it from the graph (sniffing the area and makes a decision). When the agent has a path, he removes the first location from it and goes there. Note, that the paths we refer to here are sets of locations

(real, semi-real, virtual). As a matter of convention, valid path has only one real location – the final one. Thus, following the path in this content has equal meaning as moving to appropriate neighbor(random, next on the cycle, with lowest σ) at 1.2.

Schematically it can be represented by:



Thus, upon arriving to the real location (which is always final) on the path, the agents act as defined by the appropriate algorithm, while being on the path serves simulation and animation needs only.

2.2 Data Structures and Algorithms

The core algorithms which are the cycle finding and the deployment have already been explained. Their formal definition and analysis can be found in [1]. Let us take a look on some additional algorithms used in the simulation.

- **graph generation.** Simple algorithm for $G_{n,r}$ generation, based on random number generator. Each vertex position is defined randomly in $((0,0) , (1,1))$ square and then

mapped to the necessary square size. Edges are defined based on the next condition:

$$\forall v_i, v_j \in V(G_{n,r}), i \neq j: (v_i, v_j) \in E(G_{n,r}) \Leftrightarrow |(v_i, v_j)| < r$$

- **graph road map**. Finding a path on a graph is $O(n^l)$, where l is the path's length. Although, we are interested only in paths with $l \leq 4$, running path finding algorithm each time agent “sniffs” the area is highly inefficient. Therefore, it was decided that all available paths will be calculated during initialization stage and stored for future use. That is, depending on the used algorithm (PVAW1, PVAW2 or PVAW3) the road map is filled. The road map we refer to is actually a bi dimensional matrix $RM[n][n]$, where $RM[i][j] = calculatePath(v_i, v_j)$, if there is no such path, $calculatePath()$ will return *null*. In its turn $calculatePath()$ runs DFS to find the path, while its depth is limited by the “sniffing” range.

Data structures.

Most of data structures used are represented by uni- and bi- dimensional arrays. Thus, agents herd, adjacent matrices, road map are all managed with arrays.

The most complex data structure used is built-in java template type **ArrayList**. It is a linked list with random index-based access. It was chosen as one which best suits for handling paths and neighborhoods of vertices.

This project does not require more complex data structures.

2.3 Implementation platform

The project should it should be easily accessible through the web. Thus, facing the request for web-based application, there were several options available:

- Java applet
- Flash applet
- JavaScript

All the above candidates has pretty equal programming accessibilities for simulating task. As we saw there are no complex data structures or algorithms for data manipulation needed here. So, all the three would perfectly match under this criterion.

One more criterion is animation. Animation in JavaScript, though possible to achieve, especially with modern JavaScript extensions as JQuery, Prototype, DOJO etc., still is a very resource consuming and some-what tricky. The most valuable advantage of this platform

would be independence on user's software. The only program user actually needs here is an internet browser (otherwise it would confront the requirement of web-based application) with JavaScript support (which is available in more than 99% of cases), while Java and Flash require additional installations to work on the user's machine.

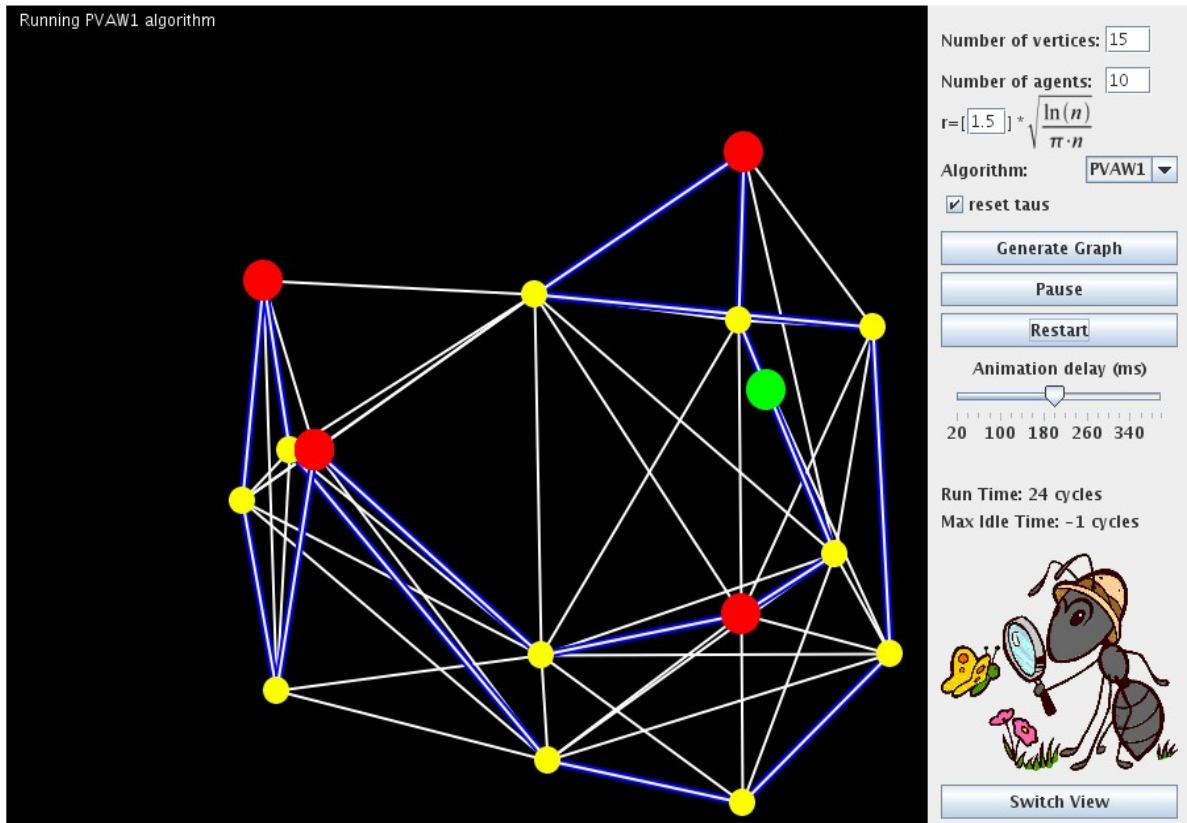
Flash has the most powerful abilities in graphics processing, with no known limitation for simulation implementation we needed. The only counterargument is our lack of experience with Flash ActionScript.

Java Applet platform also has more than sufficient tools for animation complemented with almost full featured abilities of Java programming language. And, as long as we had programming experience with Java, it was decided to use this platform for the implementation.

3. Running the simulation

Let us take a look at the project's main view:

✧ Intelligent Systems Simulation Project ✧



The screenshot displays the main interface of the Intelligent Systems Simulation Project. On the left, a graph is shown with 15 vertices and 10 agents. The vertices are represented by colored circles (red, yellow, green, and blue), and the edges are represented by white lines. The graph is titled "Running PVAW1 algorithm". On the right, a control panel allows users to set simulation parameters. The parameters are: Number of vertices: 15, Number of agents: 10, $r = \lceil 1.5 \rceil \cdot \sqrt{\frac{\ln(n)}{\pi \cdot n}}$, Algorithm: PVAW1, and a checked box for "reset taus". The control panel includes buttons for "Generate Graph", "Pause", and "Restart", an "Animation delay (ms)" slider (ranging from 20 to 340), and a "Switch View" button. The simulation status is shown as "Run Time: 24 cycles" and "Max Idle Time: -1 cycles". A cartoon illustration of an ant with a magnifying glass is also present.

The right control panel allows us to set the simulation parameters (number of vertices, number of agents, the value of r , which algorithm to use, whether or not the leader updates τ_1, τ_2 upon successful cycle finding) along with the simulation state control buttons (graph generation, simulation start/stop/restart, view switching) and the animation speed control slider.

After n, k, r are set, "Generate Graph" button can be pressed. Each following pressing will bring newly generated graph with the provided parameters. After the user is satisfied with the graph he/she got, "Start/Pause" can be pressed to launch the animation. If the animation is already started, the only options you have are: changing the animation speed, pausing/continuing the simulation and switching the view to circle-like after a cycle was found and back.

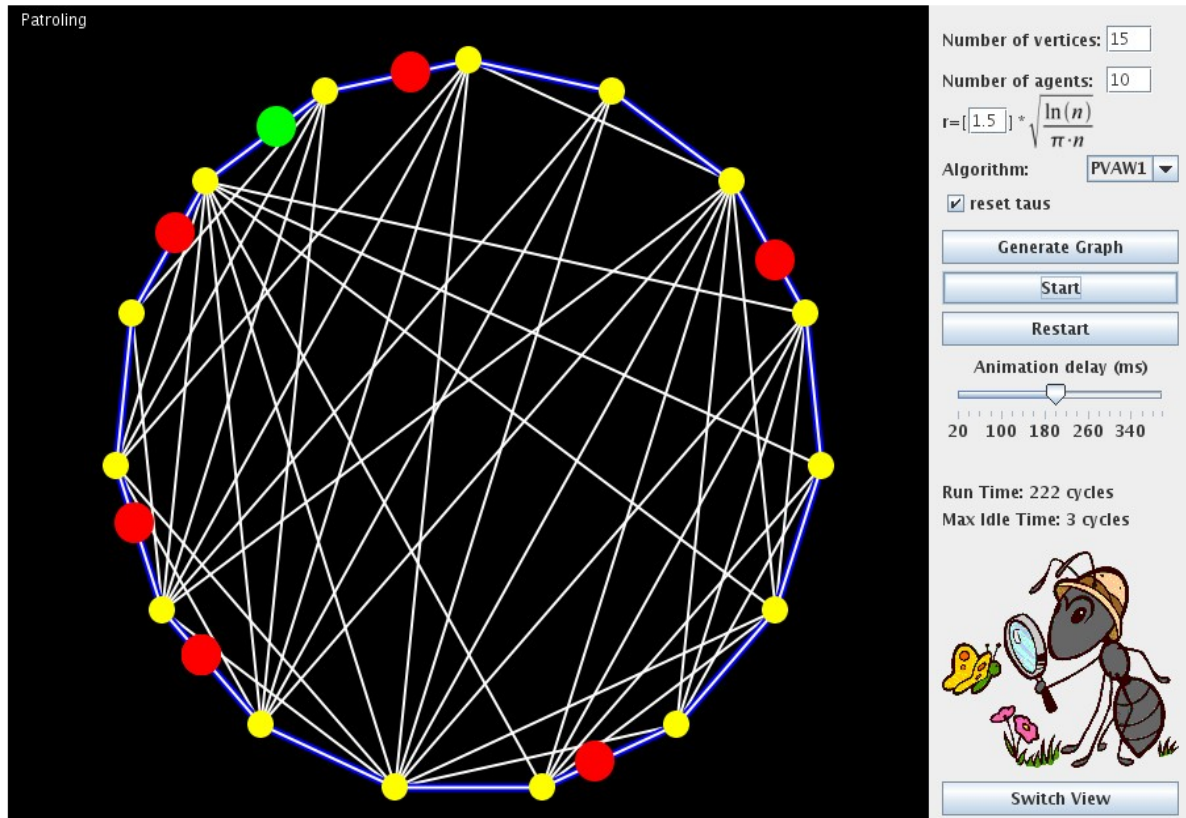
If the user wishes to switch algorithm or/and the leader's behavior regarding taus resetting,

he/she can change these parameters and press restart. This will restart the simulation on the same graph with new parameters.

Note, that *n.k.r* parameters change will need that “Generate Graph” be pressed.

Circle-like view example is presented below:

✧ Intelligent Systems Simulation Project ✧



This view is meant to provide tracking after the deployment process. After a cycle was found, it can be thought of as a circle perimeter, that allows to have a clear look of how the spreading over the cycle is going on.

Output. Besides simulation's animation itself, total run time and max idle time (idleness) are displayed on the control panel. This, along with the top left state message, helps to see and estimate what is current cycle finding/convergence state and compare it to the theoretical prediction.

4. Summary

Multiple simulation runs showed results within the theoretical limits. Both the cycle finding and the deployment processes seem to agree with theory. Thus, the application showed itself to be capable of modeling a multi-agent patrolling system.

Future work:

- Improvement of tau resetting process. This issue solution would bring valuable investigation in speeding up the deployment process. Currently, after the tau updating is done by existing scheme, herd-agents tend to group at one point, and, only after this to spread over the cycle. Seemingly, though in theory that was the analyzed situation, it is the worst possible case. Changing this may drastically improve the average convergence time. Taking into account the fact, that agents move on the cycle only in one direction, the leader could assign taus with future (or combining actual and future) timestamps until he encountered a herd-agent. Decision of actual values to use is left for future investigation along with simulations to check the approach.
- Eliminating global timing. Global timing elimination would avoid the main SPF(single point of failure) of the system. If some agent's clock is wrong, this may lead to complete system malfunctioning. While synchronization between the moving agents is very tricky, if possible. It was proposed to use station based clocks. Such an approach gives an opportunity to use one of the distributed clock synchronization algorithms presented in [4].
- Logging. Logging can provide the ability to trace simulation backwards. Which in its turn would strongly increase debugging abilities, if such a need arises. Also, it seems like running the simulation backward could be a nice feature even for user needs.

5. References

- [1] Autonomous Multi-Agent Cycle Based Patrolling, Yotam Elor and Alfred M. Bruckstein, CS, Technion, Haifa 32000, Israel
- [2] <http://en.wikipedia.org/wiki/Patrol>
- [3] http://en.wikipedia.org/wiki/Multi-agent_system
- [4] Distributed Operating Systems, Andrew S. Tanenbaum, Prentice Hall Inc 1995.