PROJECT

DARWINISM, LAMARCKISM, AND KNOWLEDGE EXCHANGE AMONG ANIMATS

By Ankit Arora and Luis Ángel Larios Cárdenas

University of California, Los Angeles *Animats-Based Modeling* *March, 2011*

PROJECT

DARWINISM, LAMARCKISM, AND KNOWLEDGE EXCHANGE AMONG ANIMATS

By Ankit Arora and Luis Ángel Larios Cárdenas

INDEX

Content *Page*

Appendix

PROJECT

DARWINISM, LAMARCKISM, AND KNOWLEDGE EXCHANGE AMONG ANIMATS

By Ankit Arora and Luis Ángel Larios Cárdenas

1. HYPOTHESES

1. Lamarckian-evolution-based animats outperform their Darwinian analogues in the task of reproduction when the environment is ***static***, while in a ***dynamic world***, Darwinian agents outnumber Lamarckian ones, considering that both gain experience just by ***directly interacting*** with their habitat.

2. Darwinian and Lamarckian populations are larger when individuals obtain knowledge both by ***directly interacting*** with their static/dynamic world and by ***acquiring information from peers***.

2. GOALS

- Simulate an artificial world where ***animats*** learn to avoid traps, recognize where food is, and look for opportunities to reproduce if they have enough energy to give birth.
- Generate ***static*** and ***dynamic*** worlds in order to see how well the agents can adapt to environmental conditions when they:
    a. Can ***exchange information*** about the world they know with peers.
    b. Come from a Lamarckian evolutionary process and are capable of passing knowledge to the future generations.
    c. Come from a Darwinian evolutionary process and are born without knowledge about their world.
- Implement ***genetic algorithms*** to assist the evolution of agents and improve their ability to fit the world they live in.
    a. Make use of the ***crossover*** technique to produce enhanced genotypes for newborns.
- Implement ***egocentric maps*** as artificial means of learning, which are constantly updated as the animats move around and discover new things in the simulated world.
    a. Make use of ***shortest path algorithms***, such as A*, and integrate them to a deterministic decision module.
- Observe the consequences of ***information flow*** among individuals and how knowledge can guide the entire population either to succeed or to fail in their task of consuming resources and survive as long as the environment and their adaptation allow.
- Tie algorithmic specifications to a ***graphical interface*** that allows evaluating the agents' performance.

3. ISSUES

- ***Early extinction***. Two issues that avoided to measure the emergence of collective behavior were:
    a. Having the first-generation of animats scattered throughout the world preempted a necessary initial increase in the population in the case of a dynamic environment.
    b. A high rate of swapping food into traps, and vice versa, made agents disperse continually and reduced their opportunities to mate, particularly in a dynamic environment.
- ***Population explosion***. The artificial world experienced overpopulation in two ways:
    a. In the case of static environment, the number of individuals grew exponentially when the amount of resources did not deplete over time. The immediate consequence was slow runs, which later required to constraint the length of simulations.
    b. Lamarckian agents tended to agglomerate in groups of over 100 individuals in a dynamic environment with high level of resources when they had the ability to exchange information. As a result, animats used to procreate tens of newborns after eating up the resources at their current cell, causing simulations to go slow because of the massive information exchange among them.
- Definition of the ***Experience*** function and the diffusion of knowledge among animats at the same cell:
    a. Animats tended to enter a 'vicious circle' of information exchange after all of them reached the same level of experience; as a consequence, they used to get experience values over 1000 (clearly incorrect) and remained at their common cell forever exchanging egocentric maps.

4. Methodology

- Simulations consist on comparing the performance of two classes of animats: those procreated under the Darwinian principle of evolution, and those from the Lamarckian approach, both inhabiting dynamic and static environments and with/without knowledge interchange allowed.
- The agents learn through sensing their current cell and storing data in an ***egocentric map***, which is an array of cells that one can consider as a miniature version of the environment.
- Animats can learn from three different processes:
    a. As they move around and discover cell's content that does not match what they know about that particular location.
    b. When they acquire information from another peer.
    c. If their parents pass on an egocentric map to them (Lamarckian individuals only).
- Agents' ultimate goal is breeding; they must eat as much food as they can, given their ***energy capacity***, and achieve a high *energy level*/*energy capacity* ratio if they want to pass the most significant bits of their gene's share when mating with another individual at the same cell.
    a. The system implements genetic algorithms in a haploid fashion.
    b. Reproduction disregards parents' egocentric maps and concentrate only on phenotype attributes in case of Darwinian evolution.
    c. Newborns inherit their fittest parent's egocentric map together with the resulting characters from the genetic crossover in case of Lamarckian evolution.
- Animats can perform knowledge exchange with peers at the same cell if they are not eating nor falling into a trap; however, it is an individual's choice to take as true what the teller is informing by observing its level of experience and age, considering their own ***trust*** gene.
- Agents die as their ***energy level*** reaches zero or if their *age* equals their ***life span***.
- Regarding abstraction and accuracy, it is not the intention in the project to model exact phenotypes or the physics of mass, volume, and forces with respect to the animats ability to move and perform actions.

5. Environment and Physics

- The artificial world is a non-toroid, squared terrain that encloses a set of cells in four walls.
- Cells store a numeric value that determines if there is:
    a. Food when the value is *positive (0, 750]*.
    b. A trap when the value is *negative [-750, 0)*.
    c. Nothing when the value is *zero*.
- The system keeps track of where agents are through a list attached to each cell.
- Information such as food amount, trap, and animats list at a particular cell is available for individuals to sense once they reach that location and need to take further actions.
- There are two classes of environment:
    a. ***Static***, if food springs and traps keep their condition (numeric value) constant during the whole simulation. (Developers included a variation in the static environments, having food depleting as agents ate from them).
    b. ***Dynamic***, if food springs and traps swap their condition periodically (every 20 seconds). Traps turn into fully replenished food sources in order to keep the population high, for food resource decreases as animats feed from it.
- Food springs and traps do not produce any kind of signal. Animats sense cell's content as soon as they happen to be at that location.
    a. Animats eat from food sources at rates equivalent to their ***feeding speed***.
    b. Agents lose energy points when they accidentally fall into a trap.
- Agents which *age* equals ***life span*** or *energy level* reaches zero disappear.
- The first-generation's individuals appear in the world at random locations according to the following:
    a. In a static environment the system scatters individuals one by one, taking care of avoiding locations that overlap food or trap cells.
    b. In a dynamic environment the system chooses four cells, divides the initial population accordingly, and assigns the groups to those locations.
- Agents do not block each other when they share a cell. There is not limit in the number of individuals that can feed from or just be at the same location.

6. Animat Specifics

There is only one type of animat inhabiting the simulated environment. The artificial agent's characteristics are:

Brain Controller

- It consists of an ***egocentric map*** that stores the animat's current knowledge with respect to the world.
- The agent's egocentric map is basically an array of cells that mimic those from the real environment.
- The agent keeps its knowledge by setting the state of its "mental cells" as follows:
    a. *Food* if it thinks there is a food spring at that location.
    b. *Trap* if it thinks there is a trap.
    c. *Empty* if it thinks there is nothing at that cell.
    d. *Unexplored* if it has not been at that location yet and, hence, does not know what could be there.
    e. *Unreachable* if it tried to compute a path to that cell but could not get to it.
- The agent can modify its mental cells' content via:
    a. *Direct learning* if it had the chance to check by itself the respective real-world cell.
    b. *Knowledge acquisition* if it received information from a peer.
    c. *Inheritance* if the animat is product of a Lamarckian evolutionary process.

Sensors and Effectors

- The animat's only one sensor consists of its whole body.
    a. It can sense just one cell every time step since the agent must learn by ***direct*** experience.
- The animat's effectors are quite abstract and relate directly to its phenotype as follows:
    a. *Mouth*: tied to its ***feeding speed***.
    b. *Limbs*: tied to its linear ***speed***.

Body – Environment Interaction

- The agent must sense its current cell at every time step in order to make a decision. Consequently, the world retrieves a numeric value that equals food, trap, or nothing (>0, <0, or 0 respectively), and a list of the animats at the same cell. Based on that, the individual chooses an action from:
    a. ***Exploring*** randomly the environment as its energy level is above 70% of its energy capacity.
    b. ***Following*** a path to the closest presumably food spot that it previously computed when it realized its energy level was below 70% of its energy capacity.
    c. ***Mating*** with another partner when both satisfy the requirements:
        - *Energy level* greater than 70% the energy capacity.
        - *Age* between 12 and 52 seconds.
        - It has a zero *mating waiting time*.
        - Current cell is neither a food container nor a trap.
    d. ***Feeding*** from a food source in order to increment its energy level until it replenishes its energy capacity, or there is no more resource available at the spot.
    e. ***Exchanging knowledge*** with the *most reliable partner* if the next equation holds:

$$\frac{Experience_{recipient}}{Age_{recipient}} < \frac{Experience_{teller}}{Age_{teller}} + \frac{Age_{teller} - Age_{recipient}}{Age_{teller} + Age_{recipient}} \quad Trust_{recipient}$$

- The animat enters a "*doomed*" state when it thinks there is no more food available. Then, it just stops and waits for dying, unless another agent modifies its mental state through knowledge exchange.

Costs

- All animats have a maximum time to live, their ***life span***.
- The act of ***mating*** implies time and energy costs:
    a. Takes 50% of ***energy capacity*** away from each parent's *energy level*.
    b. Establishes a *waiting time* of 5 seconds during which none of the parents can mate again.
- ***Falling into a trap*** costs 10% of energy capacity to agent's energy level.
- ***Exploring*** takes away 1 energy level unit per simulated second.

7. System Architecture

Figure 1 shows the overall system architecture. The general process is:

1. The agent perceives its current cell.
2. It updates its egocentric map according to the sensorial information.
3. The animat makes a decision based on what it is sensing, its internal state, and what it knows.
4. It updates its orientation to compute its position for next time step.
5. System updates agent's internal state variables and go back to step 1.

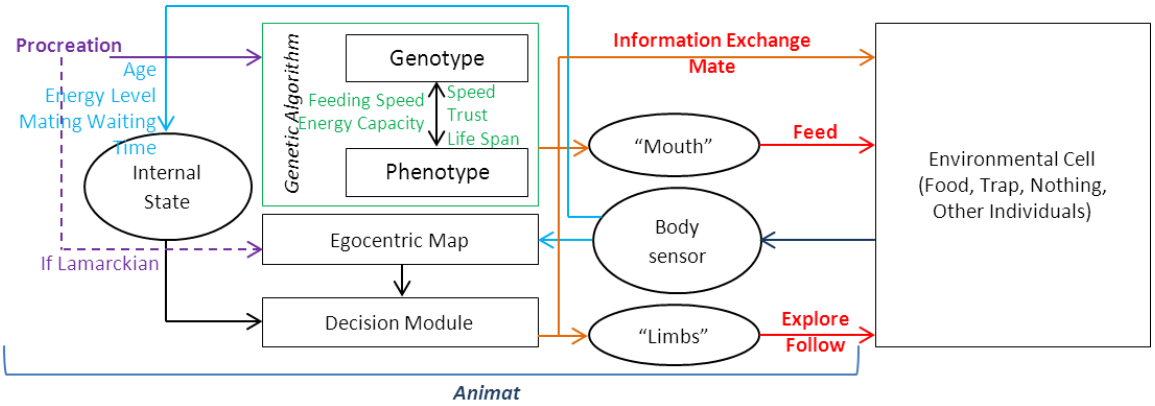


Figure 1. System Architecture.

There are three fundamental pieces in the system: ***Learning***, ***Evolution***, and ***Decision Making***.

Learning

- Agents learn via *direct experience*, *information exchange*, and *inheritance* (in case of Lamarckism).
- System rewards individuals with points of ***experience*** when they modify their mental cells' content:
  a. 1 point for discovering an empty cell.
  b. 3 points for discovering food or a trap.
- ***Experience*** points are necessary to increase agent's reliability and improve its chances to pass on its egocentric map to a peer during information exchange.

Evolution

- The model makes use of genetic algorithms for constructing a newborn's genotype.
- Genotypes are encoded in 8-bit-long strings that describe the following phenotype features:
  a. ***Speed***: how fast the animat moves, in units per second, ranging in [1, 4].
  b. ***Feeding speed***: how fast the agent eats, in units of food per second, ranging in [1, 4].
  c. ***Life span***: how many seconds it is going to live, ranging in [10, 70].
  d. ***Energy capacity***: maximum amount of energy it can get, in units of food, ranging in [10, 50].
  e. ***Trust***: a value in [0, 1] that weights the teller's reliability during knowledge exchange.
- The fittest parent (that one with higher *energy level*/***energy capacity*** ratio) passes its genotype bit-strings' most significant bits during ***crossover*** to create a composed gene for a newborn.
- Lamarckian fittest parent provides its egocentric map to the child, unlike Darwinian individuals that are born with blank mental maps.

Decision Making

An animat chooses an action every time step as stated on algorithm 1.

1. If state is FOLLOWING a path, then
   a. If there is food, then start eating and change state to FEEDING.
   b. Else
      i. If there is trap, then step back and change state to EXPLORING.

ii. Else, continue following the path towards goal cell.
c. Get information from other agents if system allows knowledge exchange.
2. Else
a. If state is EXPLORING, then
i. If there is food, then
1. If energy level is below 70% of energy capacity, then start eating and set state to FEEDING.
2. Else, get a random direction.
ii. Else
1. If there is a trap, then step back to previous cell.
2. Else
a. If energy level is below 70% of energy capacity, then
i. Compute a path to closest food source, unexplored cell, unreachable cell, or trap (if world is dynamic) in that order as known resources start depleting.
ii. If there is a path, then change state to FOLLOWING and set the direction.
iii. Else
1. If there are no more potential food cells, then change state to DOOMED.
2. Else, mark mental cell as unreachable.
b. Else
i. If energy level is above 70% of energy capacity and age is between 12 and 52 seconds, then find a mate that meets same requirements and breed.
ii. Else, get a random direction.
3. Exchange information if system allows.
b. Else
i. If state is FEEDING
1. If there is food, then continue feeding until replenishing energy capacity.
2. Else
a. If there is a trap, then step back to previous cell and change state to EXPLORING.
b. Else, set state to EXPLORING, set random direction, and exchange information if allowed.

Algorithm 1. The decision-making module.

8. Project Phases

1. Implementation of a ***static*** environment ***without*** knowledge exchange, with Darwinian and Lamarckian animats inhabiting it at separate times.
2. Implementation of a ***dynamic*** environment ***without*** knowledge exchange, with Darwinian and Lamarckian animats inhabiting it at separate times.
3. Implementation of a ***static*** environment ***with*** knowledge exchange, with Darwinian and Lamarckian animats inhabiting it at separate times.
4. Implementation of a ***dynamic*** environment ***with*** knowledge exchange, with Darwinian and Lamarckian animats inhabiting it at separate times.

9. Current Status of Work

Currently, the three main modules of the system (learning, evolution and decision making) work as expected and established in the Goals section, together with other auxiliary constructs such as:

- A* algorithm.
- Fundamental classes for priority queues, linked lists, and simulator.
- Graphic environment.

10. Instrumentation

Developers gathered the following data every time step in the experiments in order determine how individual phenotypes adjusted to the environment characteristics in a collective fashion:

- Population.
- Average experience, fitness (energy level/energy capacity), speed, feeding speed, life span, energy capacity, and trust.

A separate window showed animats' states and actions, such as:

- The animat started feeding.
- The agent finished feeding because it was full or food vanished from its spot.
- The animat exchanged information with another partner.
- The animat became doomed.
- The agent died.
- A new animat was born.

Visually, developers could tell the physical state of animats through color indicators attached to agents:

- A bright red body indicated animat's energy level was high; it turned black as the agent lost energy.
- A yellow dot indicated animat's age; the brighter the dot, the younger the agent.

11. EXPERIMENTS

Developers divided experimentation into two main categories:

- Agents in a ***dynamic*** environment with Lamarckian and Darwinian agents separately.
- Agents in a ***static*** environment with Lamarckian and Darwinian agents separately.

Developers varied for each category the following parameters:

- Distribution of food sources and traps (high, medium, low).
- Information exchange being allowed or not.
- Time rate for food to turn into traps (and vice versa) in a dynamic environment only.
- Food resources amount being constant or updatable in a static environment only.
- Size of the grid (number of cells per side)

Results correspond to data collected from simulations where:

- The *initial population* was 100 individuals, either Lamarckian or Darwinian.
- Agents had/had not the ability to exchange knowledge.
- Food distribution was *high* or *low*, where a cell had probabilities of 10% or 20%, respectively, to become a resource spot. If so, then the system assigned a negative value (trap) to 65% of chosen cells.
- The number of cells per side was *15*.
- Developers run simulations twice for static environment: with depleting/non-depleting food resources.
- The maximum simulation time was *175* (simulated) seconds for static environment and *500* (simulated) seconds for dynamic one, with time steps of *0.01* sec.

EXPERIMENTS IN STATIC ENVIRONMENTS

Results in static-environment simulations were more uniform than those from dynamic worlds. Figure 2 shows the population tendency and average experience in 4 different system configurations: having Lamarckian/Darwinian agents with/without knowledge exchange. The results were:

- *Lamarckian agents' population was higher than Darwinian one's*. It was a consequence of two factors:
    a. Lamarckian animats, having a preset egocentric map from one of the parents, knew exactly where the food resources were available and tended to conglomerate around those locations, thus creating more opportunities for mating when they diverged after feeding.
    b. Lamarckian agents did not lose required energy wandering around, exploring the world and thus had greater chances of mating.
- *Darwinian agents' average experience was higher than Lamarckians*. The reasons were:
    a. Lamarckian individuals' pre-programmed egocentric map kept them from falling into traps. They used to move within their safe regions and, later, unexplored sites when they needed food, which did not make any significant contribution to increasing their level of experience.
    b. Darwinian agents were born with empty brains that they needed to fill out with the most recently updated information. The necessity for food pushed them to explore new cells, discovering resources or traps (which hold higher experience value than unexplored sites) that Lamarckian individuals never bothered to stop by because of their out-of-date assumptions.
- *The above mentioned trends repeated in both the cases*: when food resources were depleting and when they were immutable (figure 3).
    a. Information exchange did not help towards a population boom when food remained constant since animats headed to the nearest resource they had already learnt from an experienced parent.

b. Information exchange did help agents survive longer in a food-depleting environment provided they knew about other food locations once a spring had exhausted its resources.

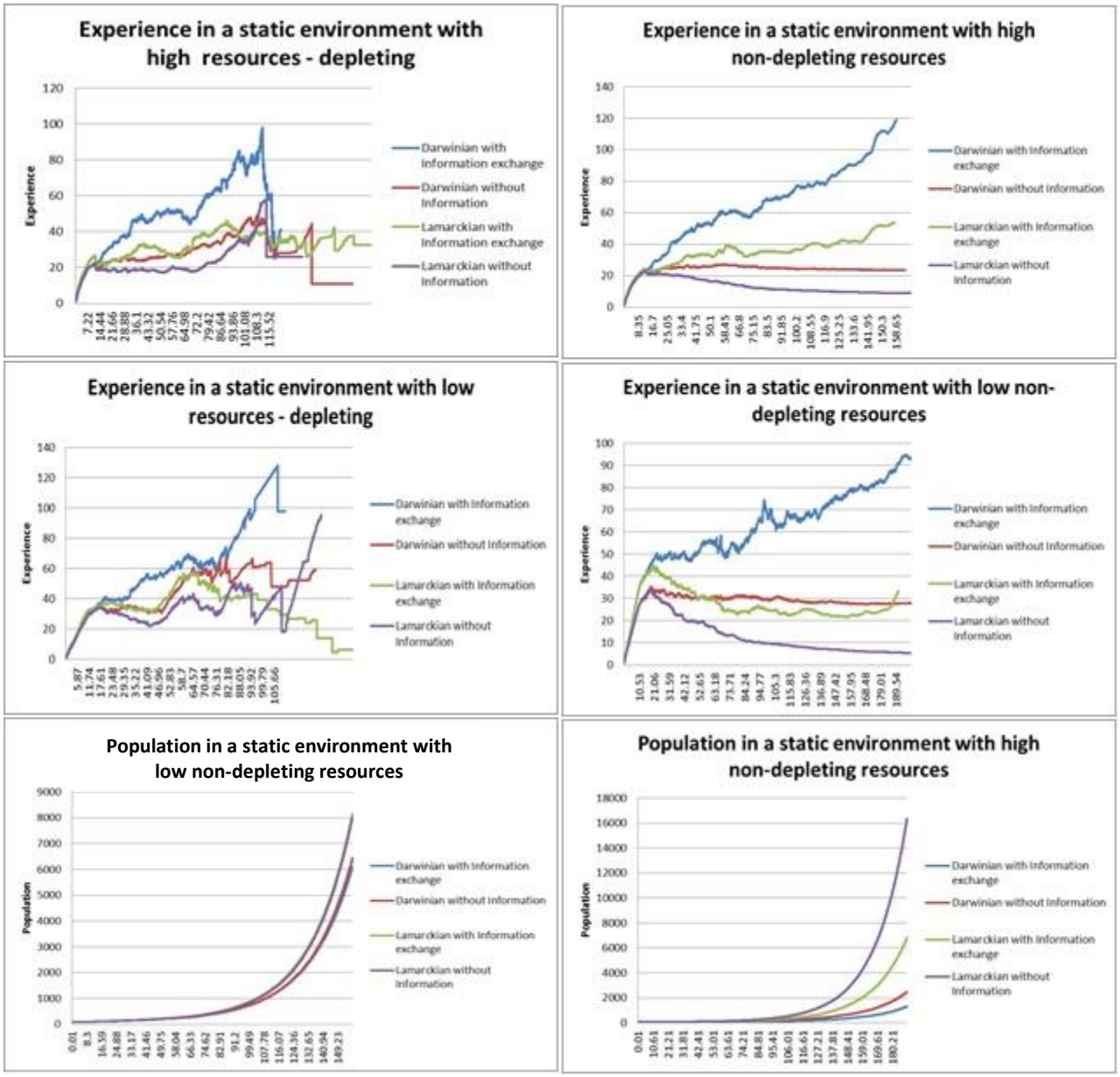


Figure 2. Population tendency and experience in a static environment when food is non-depleting.

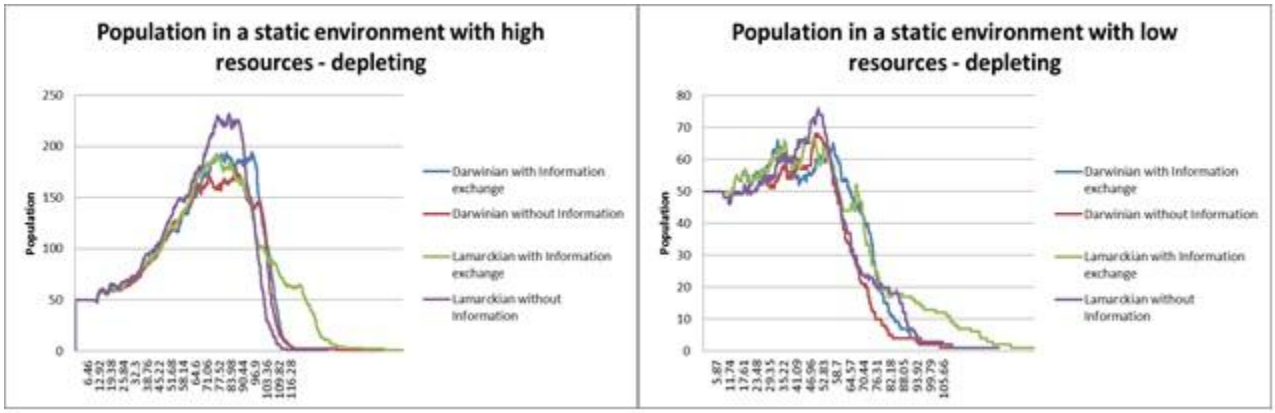


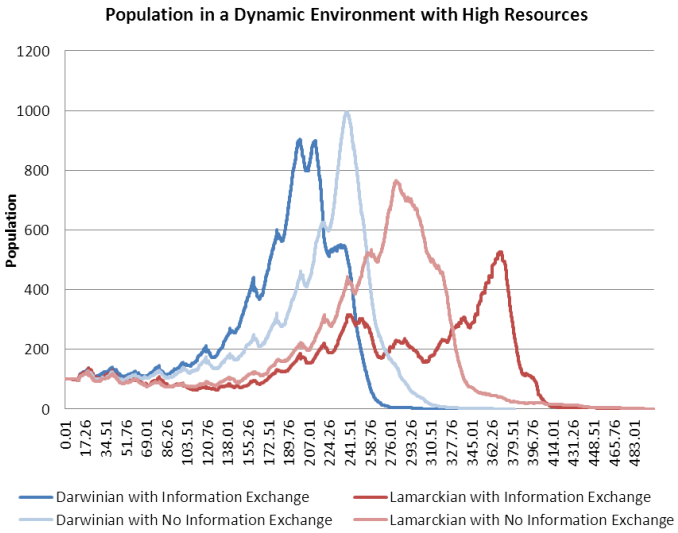
Figure 3. Information exchange favoring longer group survival when food is low and no constant.

In conclusion, for a static environment:

- Hypothesis 1 proved to be ***correct*** since Lamarckians outnumbered Darwinian agents.
- Hypothesis 2 was ***partially correct*** because information exchange did aid in longer survival but did not directly help in population explosion.

EXPERIMENTS IN DYNAMIC ENVIRONMENTS

High Amount of Resources Available



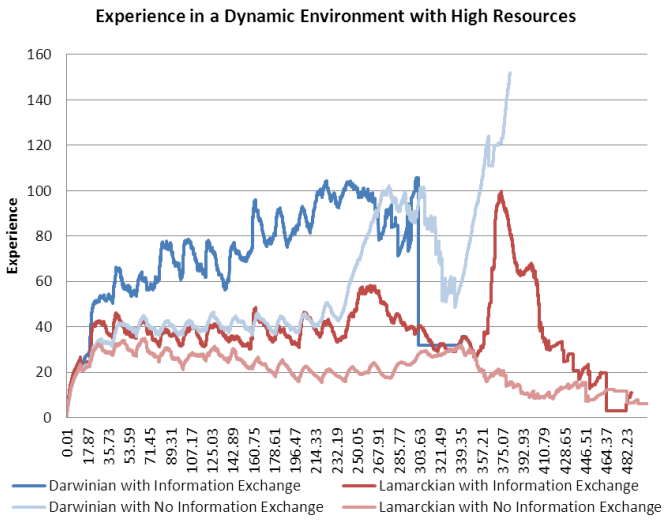
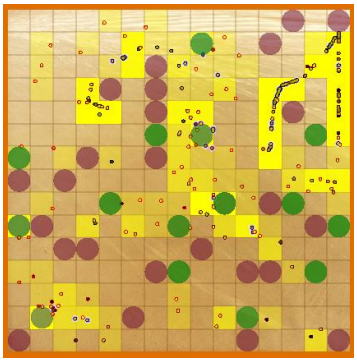


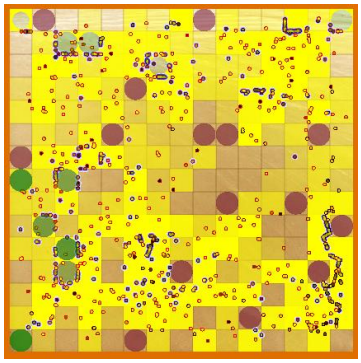
Figure 4. Population tendency and experience in a dynamic environment when the level of food was high.

Figure 4 shows the population tendency and average experience in 4 different system configurations: having Lamarckian/Darwinian agents with/without knowledge exchange. The results were:

- *Darwinian population was higher than Lamarckian one*. It was a consequence of two factors:
  a. Darwinian animats scattered more throughout the world since they had no preset influence from their parents, facilitating distribution of mating in the presence of plenty of food (figure 5b).
  b. Lamarckian agents tended to create groups of influence that converged towards common goal cells they thought to have food, especially when they could exchange information; hence, recurrent parades of agents used to migrate long distances just to find out that their idealized goal had turned into a trap, losing tens of members that starved to dead during the journey (figure 5a).
- *Darwinian population died off before simulation ended*. The successful large population distributed along the grid eased the consumption of resources, including the traps at the time they swapped to food.
- *Darwinian agents' average experience was higher than Lamarckian ones'* because they lacked a preset brain that kept them from adventuring to unknown sites, especially when in need for food.
- *Both Lamarckian and Darwinian animats had higher populations when they did not exchange information*. Sharing information is not helpful if an idealized food source is actually a trap, and especially if traps are very close to each other when the level of resources is high; so those groups of animats that made a collective wrong assumption caused the loss of individuals because of the energy cost of moving with a group towards a non-beneficial place.
- *Both Lamarckian and Darwinian animats had higher experience average when they exchanged information*. Animats equaled their experience average (*experience*/*age*) to their teller's *reliability* shortly after exchanging knowledge,



a) Lamarckian agents form long groups that lead to a trap at t = 259 sec



b) Darwinian agents are more scattered throughout the grid at t = 196 sec

Figure 5. Behavior in a dynamic environment with high level of resources and information exchange allowed.

raising, consequently, the experience of the group implied in the exchange and that of the whole community at the same time.

In conclusion, for a dynamic environment with high level of resources:

- Hypothesis 1 proved to be ***correct*** since Darwinian population outnumbered Lamarckian one.
- Hypothesis 2 ***failed*** because neither Darwinian agents nor Lamarckian ones increased their population with respect to the population they achieved having no knowledge exchange.

Low Amount of Resources Available

Figure 6 shows a comparison between Lamarckian and Darwinian populations when the level of resources was low and with/without information exchange allowed. The results were:

- *Darwinian population was higher when they could exchange information than when they could not*. Having resources so scarce made knowledge valuable since it helped individuals find food faster than exploring cell by cell (figure 7b).
- *Lamarckian population was higher when they could exchange information than when they could not, during the first half of simulation* (figure 7a).
    a. Information was helpful to find food at the beginning; however, inheriting out-of-date knowledge from parents trimmed the chances to find food that had not turned into a trap for one individual and for others that unfortunately got information from the former. As a consequence, Lamarckian agents could not keep up with environmental changes and, then, did not raise their population.
    b. Having no information exchange reduced the "error" spreading among individuals. What someone inherited from their parents did not affect others' assumptions about the world, improving slightly their odds to avoid traps and reproduce more.
- *Lamarckian population edged Darwinian one after t = 166 when they could exchange information*. Knowledge from parents was helpful considering that there was a population deficit to eat up resources and that Lamarckian agents were born at times that matched the swapping rate; during 30 seconds, newborns ate from sources that their parents knew as food places and avoided them when turned into traps, which coincided with their periods of being above 70% of their energetic capacity; eventually, however, the synchronization broke and the agents started visiting traps successively, which caused their earlier extinction in comparison with the Darwinians case.
- *Darwinian individuals' average experience was higher than Lamarckian ones'* because:
    a. Darwinian agents pushed themselves to explore and discover more up-to-date cell contents than Lamarckian ones, who kept moving in their safety zone, reducing their chances to learn.
    b. Agents could not find someone to exchange information with as population reduced when they could communicate their knowledge; therefore, everyone relied only on what they found out by themselves rather than from what others informed, making experience acquisition independent from information exchange roughly after population dropped below 40.

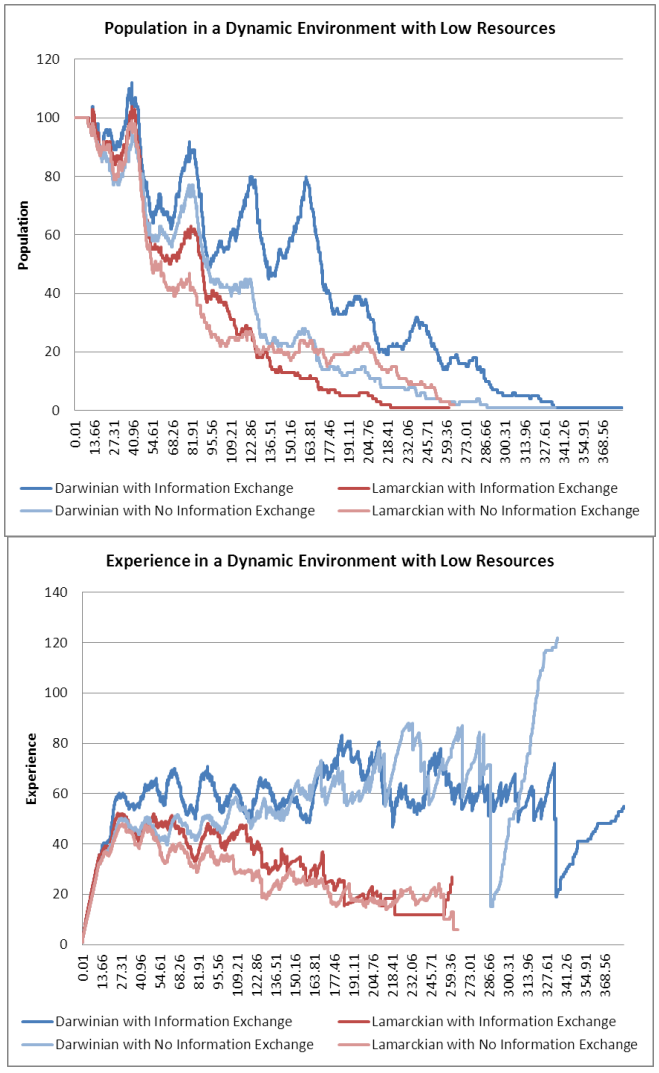
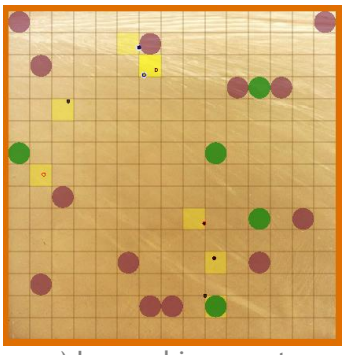
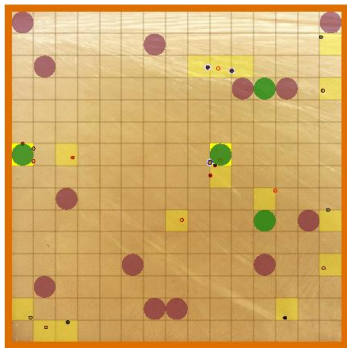


Figure 6. Population tendency and experience in a dynamic environment when the level of food was low.

- *Both Lamarckian and Darwinian agents died off before reaching the maximum simulation time* because of the lack of resources to give an initial impulse to their populations.
- *The long distance between food sources contributed to eliminating a large portion of both classes of populations*. Slow agents could not get to food and died soon after simulation started, producing an increase in the population average speed (figure 8).



a) Lamarckian agents survived longer *with no* information exchange



b) Darwinian agents survived longer *with* information exchange

Figure 7. Collective behavior of animats in a dynamic environment with low level of resources at t = 250 sec.

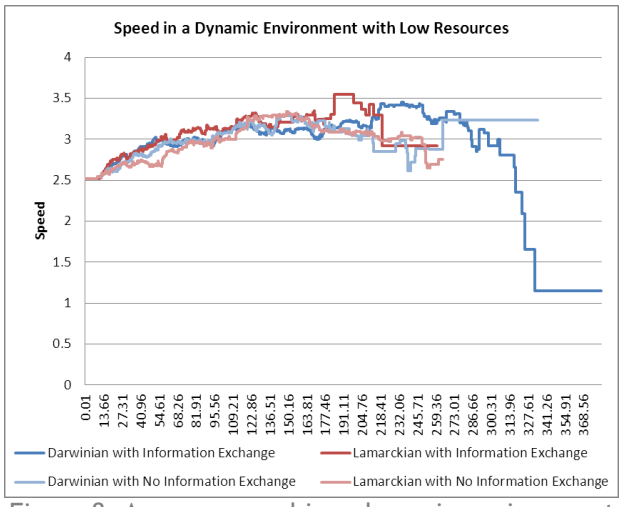


Figure 8. Average speed in a dynamic environment where the level of resources was low.

In conclusion, for a dynamic environment with low level of resources:

- Hypothesis 1 was ***partially correct*** since Darwinians outnumbered Lamarckian agents only during the first half of simulation. Scarce resources disturbed the overall performance of the population because they prevented the community from generating enough individuals to survive longer.
- Hypothesis 2 proved to be ***correct*** for Darwinian agents but ***failed*** for Lamarckians because:
    a. Darwinian population with information exchange allowed was always higher than the population with knowledge communication disabled.
    b. Lamarckian population with no information exchange allowed was higher during the second half of the simulation than the population of individuals with the feature enabled.

12. Individual Contributions to the Project

Ankit Arora contributed with:

- Genetic algorithm implementation.
- Experimentation over static environments.
- Base classes for linked lists.
- Simulator implantation.

Luis Ángel contributed with:

- Learning implantation via egocentric maps.
- Decision module development.
- Graphical environment.
- Priority queues and A* algorithm.
- Experimentation over dynamic environments.

13. Platform

- ***Environment***: Microsoft Visual Studio 2010 (http://www.microsoft.com/visualstudio/en-us/).
- ***Language***: Visual C++.
- ***Graphics library***: OpenGL (http://www.opengl.org/)

14. References

Sasaki, T., and Tokoro, M. (1997). *Adaptation toward Changing Environments: Why Darwinian in Nature?* In P. Husbands and I. Harvey (eds.) Fourth European Conference on Artificial Life. Pages 145-153.

APPENDIX

DARWINISM, LAMARCKISM, AND KNOWLEDGE EXCHANGE AMONG ANIMATS

A1. SOURCE CODE

Note: In most of the code you will find instructions referring to "ant" objects since developers started working with ant-like animats at the beginning of the project. Consider, now, that programmers actually refer to "ANimaTs" in all those cases. Documentation and system outputs have the correct terminology though.

LEARNING AND DECISION MODULE

Learning and the Decision Module are embedded in the Simulator's main step function.

```
/*************** Executes every simulation step **************/
int WorldSimulator::step(double time)
{
        AStar pathFinder;
        float food, gap, request, minFoodDistance, minTimeStamp, minUnreachableTimeStamp, minUnexploredDistance;
        Ant *ant, *mate;
        IntNode *c, *antID;
        IntList *antIDList;
        Vector direction, possiblePosition, distance;
        int goal, goalUnexplored, goalFood, goalTrap, goalUnreachable, knownWorld;
        bool found;

        ///////////////// Counters //////////////
        int population = 0;
        double avgEnergyLevel = 0.0;
        double avgExperience = 0.0;
        double avgEnergyCapacity = 0.0;
        double avgLifeSpan = 0.0;
        double avgSpeed = 0.0;
        double avgFeedingSpeed = 0.0;
        double avgTrust = 0.0;

        if(time <= worldSystem->getMaxTime())
        {
                ////////////////// Change world every x seconds if it is dynamic ////////////////////////////////
                if(fmod((float)time, 20.0f)==0.0 && worldSystem->isDynamic())
                {
                        for(int I=0; I<pow(worldSystem->getSideLength(), 2.0); I++)
                                worldSystem->getCell(I)->swapContent();  //Transform food into traps.
                }

                ant = worldSystem->getAntList()->gotoStart();    //Check for all living animats.
                while(ant != NULL)
                {
                        ////////////////////////////////// Sense the world ////////////////////////////////////
                        food = worldSystem->senseFoodAt(ant->getPosition());   //Senses food, trap, or nothing.
                        antIDList = worldSystem->senseAntsIDAt(ant->getPosition()); //Senses other animats.
                        ///////////////////////////////////////////////////////////////////////////////////////

                        ////////////////////////////// Learning something new ////////////////////////////////
                        if(food > 0.0)  //Found food?
                        {
                                if(ant->getMentalMap()->getCell(
                                        worldSystem->worldToGridCoordinates(
                                        ant->getPosition()))->getContent()!=FOOD)
                                {       //It did not know there was food.
                                        ant->getMentalMap()->getCell(worldSystem->worldToGridCoordinates(
                                                ant->getPosition()))->setContent(FOOD);  //Change thought.
                                        ant->getMentalMap()->getCell(worldSystem->worldToGridCoordinates(
                                                ant->getPosition()))->setTimeStamp(time); //Stores timestamp.
                                        ant->setExperience(ant->getExperience() + 3.0);  //Gain three points of
                                                                                         //experience.
                                }
                        }
                        else
```

```
{
        if(food < 0.0)          //Found a trap!
        {
                if(ant->getMentalMap()->getCell(worldSystem->worldToGridCoordinates(
                        ant->getPosition()))->getContent()!=TRAP)
                {       //It did not know there was a trap.
                        ant->getMentalMap()->getCell(
                                worldSystem->worldToGridCoordinates(
                                ant->getPosition()))->setContent(TRAP);  //Change mental
                                                                         //cell content.
                        ant->getMentalMap()->getCell(
                                worldSystem->worldToGridCoordinates(
                                ant->getPosition()))->setTimeStamp(time) //Stores
                                                                         //time stamp.
                        ant->setExperience(ant->getExperience() + 3.0);  //Gain three
                                                                 //points of experience.
                }
        }
        else
        {       //There is nothing at that cell.
                if(ant->getMentalMap()->getCell(worldSystem->worldToGridCoordinates(
                        ant->getPosition()))->getContent()!=EMPTY)
                {       //It did not know there was nothing.
                        ant->getMentalMap()->getCell(
                                worldSystem->worldToGridCoordinates(
                                ant->getPosition()))->setContent(EMPTY); //Change mental
                                                                         //cell content.
                        ant->getMentalMap()->getCell(
                                worldSystem->worldToGridCoordinates(
                                ant->getPosition()))->setTimeStamp(time);  //Stores
                                                                           //time stamp.
                        ant->setExperience(ant->getExperience() + 1.0);  //Gain just one
                                                                 //point of experience.
                }
        }
}
////////////////////////////////////////////////////////////////////////////////

////////////////////////////////// Make a decision ////////////////////////////////////
if(ant->getState() == FOLLOWING) //Was the animat following a goal?
{
        if(food > 0.0)  //Found food (the ultimate goal)?
        {
                ant->getPath()->deleteList();    //Destroy remainders of list.
                zeroVector(ant->getVelocity());  //Remains in same cell.
                ant->setState(FEEDING);          //Changes its status to feeding.
                animTcl::OutputMessage("Animat %d started feeding\n", ant->getId());
        }
        else
        {
                if(food < 0.0)  //Found a trap while following a path (recoil from it!)?
                {
                        ant->getPath()->deleteList();    //Destroy path because it has
                                                         //been interrupted.
                        ant->setEnergyLevel(ant->getEnergyLevel() -
                                ant->getEnergyCapacity()*0.1);   //Discounts 10% to
                                                                 //energy capacity.
                        VecScale(ant->getVelocity(), -1.0);      //Plans to move to
                                                         //where it was previously.
                        ant->setState(EXPLORING);
                }
                else    //Current cell is neither a goal nor a trap to start over again.
                {       //Has it reached a cell considered in its path? Then, delete it
                        //and set a next cell to achieve.
                        if(worldSystem->worldToGridCoordinates(ant->getPosition()) ==
                                ant->getPath()->gotoStart()->getValue())
                        {
                                c = ant->getPath()->gotoStart(); //Locate the cell index
                                                                 //in the path.
                                ant->getPath()->deleteInt(c);    //Erase cell from path
                                                //(now there are less cells to traverse).
                                c = ant->getPath()->gotoStart(); //Get next goal cell.
                                if(c == NULL)    //It reached goal but realized there is
```

```
                                        //no food there.
                                {
                                        ant->setState(EXPLORING); //Explore again;
                                                //since it is hungry, it will
                                                //choose another path next time.
                                        zeroVector(ant->getVelocity());  //Stay where
                                                                         //it is.
                                }
                                else    //It has not reached its goal, continue moving
                                        //to next cell.
                                {       //Computes new direction.
                                        VecSubtract(direction,
                                                worldSystem->getCell(
                                                c->getValue())->getCellCenter(),
                                                ant->getPosition());
                                        VecNormalize(direction);
                                        ant->setVelocity(direction);     //Sets new
                                                                         //velocity.
                                        ant->setState(FOLLOWING);        //Continues to
                                                                        //follow the path.
                                        ant->setEnergyLevel(ant->getEnergyLevel() -
                                                deltaT);         //The act of moving
                                        //represents 1 unit of energy per second.
                                }
                        }
                        else    //It is on its way to a goal cell.
                        {
                                c = ant->getPath()->gotoStart(); //Get next.
                                VecSubtract(direction, worldSystem->getCell(
                                        c->getValue())->getCellCenter(),
                                        ant->getPosition());
                                VecNormalize(direction);
                                ant->setVelocity(direction);     //Sets new velocity.
                                ant->setState(FOLLOWING);//Continues to follow the path.
                                ant->setEnergyLevel(ant->getEnergyLevel() - deltaT);
                                //The act of moving represents 1 unit of energy per
                                //second.
                        }

                        // Exchange information with other animats with no cost ////////
                        if(worldSystem->isExchangeInformationAllowed())
                                worldSystem->getAdvice(ant, antIDList);
                }
        }
}
else    //The animat is on its normal living state: EXPLORING.
{
        if(ant->getState() == EXPLORING)
        {
                if(food > 0.0)  //Is there food?
                {
                        if(ant->getEnergyLevel() < ant->getEnergyCapacity()*0.7) //Is
                                                                        //it hungry?
                        {
                                zeroVector(ant->getVelocity());  //Remain in the same
                                                        //cell to avoid wasting energy.
                                ant->setState(FEEDING);  //Next time step will eat.
                                animTcl::OutputMessage("Animat %d started feeding\n",
                                        ant->getId());
                        }
                        else    //Animat has enough energy.
                        { //Even though it did not stop for eating, it has memorized
                                //where food is.
                                generateRandomValidVelocity(ant, &direction[0]);
                                        //Compute new velocity randomly.

                                while(direction[0]==0.0 && direction[1]==0.0 &&
                                        direction[2]==0.0) //It does not want to stay
                                                           //at the same place.
                                        generateRandomValidVelocity(ant,
                                                &direction[0]); //Compute new velocity
                                                                //randomly.
```

```
                        ant->setVelocity(direction); //Set velocity for
                                        //displacing position next time step.
                        ant->setState(EXPLORING);
                        ant->setEnergyLevel(ant->getEnergyLevel() - deltaT);
                  //The act of moving represents 1 unit of energy per second.
                }
        }
        else
        {
                if(food < 0.0)   //Found a trap?
                {
                        ant->setEnergyLevel(ant->getEnergyLevel() -
                                ant->getEnergyCapacity()*0.1);   //Discounts 10%
                                                        //to energy capacity.
                        VecScale(ant->getVelocity(), -1.0);      //Plans to move
                                                //to where it was previously.
                        ant->setState(EXPLORING);
                }
                else    //There is neither food nor trap, the animat can do
                        //other activities.
                {
                        if(ant->getEnergyLevel() < ant->getEnergyCapacity()*0.7)
                        //Did it reach threshold for hunger?
                        {
                                //It will try to find a place to go to find
                                //food depending on what it already knows.
                                minFoodDistance = 10000; //Initialize variables
                                        //in order to find the smaller one.
                                minUnexploredDistance = 10000;
                                minTimeStamp = 3600;
                                minUnreachableTimeStamp = 3600;
                                goalUnexplored = -1;
                                goalFood = -1;
                                goalTrap = -1;
                                goalUnreachable = -1;
                                knownWorld = 0;
                                for(int I=0; I<pow(
                                        worldSystem->getSideLength(), 2.0);
                                        I++)    //Visit every cell in its
                                                //mental map.
                                {
                                        if(ant->getMentalMap()->getCell(I)
                                                ->getContent() != UNEXPLORED)
                                                //Does it know this location?
                                        {
                                                knownWorld ++;  //Even if the
                                                //cell is UNREACHABLE and it
                                                //was UNEXPLORED before, the
                                                //animat must consider it
                                                //"known".
                                                if(ant->getMentalMap()->
                                                        getCell(I)
                                                        ->getContent() == FOOD)
                                                        //Does it think there
                                                        //is food?
                                                {
                                                        VecSubtract(distance,
                                                                worldSystem->
                                                                getCell(I)->
                                                                getCellCenter()
                                                                , ant->
                                                                getPosition());
                                                        if(VecLength(distance)
                                                                <
                                                                minFoodDistance)
                                                                //This position
                                                                //is closer?
                                                        {
                                                                minFoodDistance
                                                                = VecLength(
                                                                distance);
                                                                goalFood = I;
                                                        }
```

```
                                }
                                else
                                {
                                        if(ant->getMentalMap()
                                                ->getCell(I)->
                                                getContent() ==
                                                TRAP)   //It
                                        //thinks there is a
                                        //trap at I.
                                        {
        if(ant->getMentalMap()->getCell(I)->getTimeStamp() <
                minTimeStamp)
        {

                goalTrap = I;   //Chooses the trap with the
                                //oldest timestamp.

        minTimeStamp = ant->getMentalMap()->
                        getCell(I)->getTimeStamp();
        }
}
else    //Consider also the oldest unreachable spots.
{
        if(ant->getMentalMap()->getCell(I)->getContent() ==
                UNREACHABLE)
        {
                if(ant->getMentalMap()->
                        getCell(I)->getTimeStamp() <
                        minUnreachableTimeStamp)
                {

                        goalUnreachable = I;
                        minUnreachableTimeStamp = ant->
                                getMentalMap()->
                                getCell(I)->getTimeStamp();
                }
        }
}
                                }
                        }
                        else    //Unknown places: before
        //resorting a trap, it must try with unexplored cells.
                        {
                                VecSubtract(distance,
                                        worldSystem->
                                        getCell(I)->
                                        getCellCenter(), ant->
                                        getPosition());
                                if(VecLength(distance) <
                                        minUnexploredDistance)
                                        //This position is
                                        //closer?
                                {
                                        minUnexploredDistance =
                                        VecLength(distance);
                                        goalUnexplored = I;
                                }
                        }
                }
                //Now must decide where to go.
                if(goalFood != -1) //It "knows" about a place
                                        //with food?
                        goal = goalFood;
                else
                {
                        if(knownWorld < pow(worldSystem->
                                getSideLength(), 2.0))   //It //has
                        not visited all places? (Or are //some
                        reachable and unexplored?)
                                goal = goalUnexplored;
                        else
                        {
                                if(goalTrap!=-1 &&
```

```
                                        worldSystem->
                                        isDynamic())//If this //is
                                a dynamic world, some //traps may
                                have turned into //food.
                                {
                                        goal = goalTrap;

                                        //Change its thougts
                                        //(that there is food).
                                        ant->getMentalMap()->
                                        getCell(goalTrap)->
                                        setContent(FOOD);
                                }
                                else    //Choose as goal a
                        //previously-thought unreachable place.
                                {
                                        if(goalUnreachable!=-1)
                                                goal =
                                                goalUnreachable;
                                        else
                                                goal = -1;
                                        //The animat thinks
                                        //there are no more
                                        //resources.
                                }
                        }
                }
                if(goal != -1)  //Are there chances to find
                                //food?
                {
                        if(!pathFinder.findPath(
                                worldSystem->
                                worldToGridCoordinates(
                                ant->getPosition()),
                                goal,
                                worldSystem->getSideLength(),
                                ant->getPath(),
                                ant->getMentalMap()))    //If
                                        //there is not path?
                        {
                                ant->getMentalMap()->
                                        getCell(goal)->
                                        setContent(UNREACHABLE);
                                //Mark cell unreachable in //order
                                to avoid considering //multiple
                                times because of its //closeness.

                                ant->getMentalMap()->
                                        getCell(goal)->
                                        setTimeStamp(time);
                                ant->setState(EXPLORING);
                                //There are no more resources,
                                //just wait to die.
                                animTcl::OutputMessage("<<<<
                                        Animat %d thinks the
                                        goal node %d is
                                        unreachable >>>>\n",
                                        ant->getId(), goal);
                        }
                        else
                        {
                                ant->setState(FOLLOWING);
                                //It found a path to any
                                //possible place with food.
                        }
                }
                else
                {
                        ant->setState(DOOMED);
                        //Animat did not find any goal node to
                        //go, it is lost until someone tells it
                        //different information.
                        animTcl::OutputMessage("Animat %d is
```

```
                        doomed!\n", ant->getId());
                }
                zeroVector(ant->getVelocity());  //Remain at
                                                 //same cell.
        }
        else    //Animat is healthy, exploring, and on an empty
                //cell, perhaps with other animats.
        {
                if((ant->getAge()>=12.0 && ant->getAge()<=52.0)
                        && ant->getMatingWaitingTime() == 0.0)
                        //Reproductive age between 20 and 50
                        //seconds.
                {
                        antID = antIDList->gotoStart();
                        found = false;
                        while(antID!=NULL && !found)
                        //Traverse the list of ids of animats //at
                        the same cell.
                        {
                                mate = worldSystem->
                                        getAntList()->
                                        findAnt(antID->
                                        getValue());    //Who //is
                                the animat with this ID?
                                if(mate!=ant && (mate->
                                        getAge()>=12.0 &&
                                        mate->getAge()<=52.0)
                                        && mate->
                                        getEnergyLevel()>=
                                        0.7*mate->
                                        getEnergyCapacity()
                                        && mate-
                                        >getMatingWaitingTime()
                                        ==0.0)
                                {
                                        found = true;
                                        worldSystem->
                                        procreate(ant, mate);
                                        //Executes routine to
                                        //create another
                                        //individual.
                                        ant->
                                        setMatingWaitingTime(
                                        5.0);   //5 seconds to
                                        //wait for next mating.
                                        ant->
                                        setEnergyLevel(
                                        ant->getEnergyLevel() -
                                        ant->
                                        getEnergyCapacity()
                                        *0.5);  //Discounts 50%
                                        //of energy.
                                        mate->
                                        setMatingWaitingTime(
                                        5.0);   //5 seconds to
                                        //wait for next
                                        //maiting.
                                        mate->
                                        setEnergyLevel(mate->
                                        getEnergyLevel() –
                                        mate->
                                        getEnergyCapacity()
                                        *0.5);  //Discounts 50%
                                        //of energy.
                                }
                                else
                                        antID =
                                        antID->getNext();
                                        //Check with next
                                        //animat.
                        }
                }
                else    //It cannot reproduce, then just wander
```

```
                                        //around at the cost of moving.
                                        ant->setEnergyLevel(ant->
                                                getEnergyLevel() - deltaT);
                                                //The act of moving represents
                                                //1 unit of energy per second.

                                        generateRandomValidVelocity(ant,
                                                &direction[0]);  //Compute new velocity
                                                                 //randomly.

                                        while(direction[0]==0.0 && direction[1]==0.0 &&
                                                direction[2]==0.0)//It does not want to
                                                                  //stay at the same
                                                                  //place.
                                                generateRandomValidVelocity(ant,
                                                        &direction[0]);  //Compute new
                                                                         //velocity
                                                                         //randomly.

                                        ant->setVelocity(direction);     //Set velocity
                                              //for displacing position next time step.
                                        ant->setState(EXPLORING);
                                }
                        }

                        //// Exchange information with other animats with no cost ////
                        if(worldSystem->isExchangeInformationAllowed())
                                worldSystem->getAdvice(ant, antIDList);
                }
        }
        else
        {       ////////////////// The animat is feeding //////////////
                if(ant->getState() == FEEDING)
                {
                        if(food > 0.0)  //Is still there food?
                        {
                                if(ant->getEnergyLevel() < ant->getEnergyCapacity())
                                        //It will try to fill up its energy capacity.
                                {
                                        gap = ant->getEnergyCapacity() -
                                                ant->getEnergyLevel();   //Is there room
                                                                         //in its energy
                                                                         //capacity.
                                        request = (gap > ant->
                                                getFeedingSpeed()*deltaT)? ant->
                                                getFeedingSpeed()*deltaT: gap;   //How
                                                //much food can it request?
                                        ant->setEnergyLevel(ant->getEnergyLevel() +
                                                worldSystem->provideFood(ant->
                                                getPosition(), request));        //Updates
                                        animat and environment.
                                        ant->setState(FEEDING);
                                }
                                else    //Animat is full.
                                {
                                        ant->setState(EXPLORING);//It will go explore
                                                                 //randomly.
                                        animTcl::OutputMessage("Animat %d has stopped
                                                feeding because it is full\n", ant->
                                                getId());
                                }
                                zeroVector(ant->getVelocity());  //Remains in the same
                                                //cell (EXPLORING will change that).
                        }
                        else
                        {
                                if(food < 0.0)  //Food became a trap while it was
                                                //eating.
                                        ant->setEnergyLevel(ant->getEnergyLevel() -
                                                ant->getEnergyCapacity()*0.1);
                                                //Discounts 10% to energy capacity.
                                else
                                {//Exchange information with other animats with no cost/
```

```
                                                if(worldSystem->isExchangeInformationAllowed())
                                                        worldSystem->getAdvice(ant, antIDList);
                                                }

                                                //Here the animat must start moving because there are no
                                                //more food and it does not want to wait until next time
                                                //step to get a nonzero velocity.
                                                generateRandomValidVelocity(ant, &direction[0]);
                                                //Compute new velocity randomly.

                                                while(direction[0]==0.0 && direction[1]==0.0 &&
                                                        direction[2]==0.0)//It does not want to stay at
                                                                                //the same place.
                                                        generateRandomValidVelocity(ant,
                                                        &direction[0]);  //Compute new velocity
                                                                        //randomly.

                                                ant->setVelocity(direction);//Set velocity for
                                                                        //displacing position next time
                                                                        //step.
                                                ant->setState(EXPLORING);//Go to normal state and decide
                                                                        //what to do there.
                                                animTcl::OutputMessage("Animat %d has stopped feeding
                                                        because there is no more food at its spot\n",
                                                        ant->getId());
                                        }
                                }
                        }
                }
                ////////////////////////////////////////////////////////////////////////////////

                ////////////////////////////////////////////////////////////////////////////////
                // Increase counters.
                population++;
                avgEnergyLevel += ant->getEnergyLevel()/ant->getEnergyCapacity();
                avgExperience += ant->getExperience();
                avgEnergyCapacity += ant->getEnergyCapacity();
                avgLifeSpan += ant->getLifeSpan();
                avgSpeed += ant->getSpeed();
                avgFeedingSpeed += ant->getFeedingSpeed();
                avgTrust += ant->getTrust();
                ////////////////////////////////////////////////////////////////////////////////

                ant = ant->getNext();    //Next animat in the list.
        }

        ////////////////////////////// Draw current state of animats ////////////////////////////
        glutPostRedisplay();

        ////////////////////////// Computing next position of animats ////////////////////////////
        Vector v, p, q;
        for(ant = worldSystem->getAntList()->gotoStart(); ant!=NULL; ant=ant->getNext())
        {
                VecCopy(v, ant->getVelocity());          //Current velocity vector.
                VecScale(v, ant->getSpeed()*deltaT*10.0/worldSystem->getSideLength());//Determines how
                        //far the animat will advance considering deltaT, speed, and grid scaling.
                VecCopy(p, ant->getPosition());          //Current position.
                VecAdd(q, ant->getPosition(), v);        //Displacement.
                ant->setPosition(q);                             //q is new position.
                worldSystem->updateAntLocation(ant, p, q);       //Informs world of motion.
        }

        ////////////////////// Increase age and remove animats that must die ////////////////
        aging();

        ////////////////////////////////////////////////////////////////////////////////
        // Update files.
        if(population > 0)       //Write data if there are living animats.
        {
                fprintf(filePopulation, "%f, %d\n", time, population);
                fprintf(fileFitness, "%f, %f\n", time, (double)avgEnergyLevel/population);
                fprintf(fileExperience, "%f, %f\n", time, (double)avgExperience/population);
                fprintf(filePhenotype, "%f, %f, %f, %f, %f, %f\n", time, (double)avgSpeed/population,
```

```
                                (double)avgFeedingSpeed/population, (double)avgLifeSpan/population,
                                (double)avgEnergyCapacity/population, (double)avgTrust/population);
                }
                ////////////////////////////////////////////////////////////////////////////
        }
        return 0;
}
```

The Procreation Function

```
/************** Function to simulate procreation (evolution) **************/
void WorldSystem::procreate(Ant *a, Ant *b)
{
        Vector position;
        Ant *bestParent;
        int x;
        VecCopy(position, cells[worldToGridCoordinates(a->getPosition())].getCellCenter());

        Ant *newBorn = antList.addAnt(position, sideLength);     //New animats are born at parents' cell center.

        //Choose the fittest parent to give out more significant genes and pass on its mental map to the next
        //generation (if lamarckian).
        if(a->getEnergyLevel()/a->getEnergyCapacity() > b->getEnergyLevel()/b->getEnergyCapacity())       //Choose
the fittest parent.
        {
                bestParent = a;
                x = 0;
        }
        else
        {
                bestParent = b;
                x = 1;
        }

        newBorn->getGenotype()->setec(newBorn->getGenotype()->recombine(a->getGenotype()->getbi_e(),
                b->getGenotype()->getbi_e(),5,x));
        newBorn->getGenotype()->settrust(newBorn->getGenotype()->recombine(a->getGenotype()->getbi_t(),
                b->getGenotype()->getbi_t(),4,x));
        newBorn->getGenotype()->setlifespan(newBorn->getGenotype()->recombine(a->getGenotype()->getbi_l(),
                b->getGenotype()->getbi_l(),3,x));
        newBorn->getGenotype()->setfs(newBorn->getGenotype()->recombine(a->getGenotype()->getbi_f(),
                b->getGenotype()->getbi_f(),2,x));
        newBorn->getGenotype()->setspeed(newBorn->getGenotype()->recombine(a->getGenotype()->getbi_s(),
                b->getGenotype()->getbi_s(),1,x));

        ////////////////////// Set phenotype ////////////////////
        newBorn->setSpeed(newBorn->getGenotype()->convertintofloat(newBorn->getGenotype()->getspeedb(),
                1.0, 4.0));
        newBorn->setFeedingSpeed(newBorn->getGenotype()->convertintofloat(newBorn->getGenotype()->getfsb(),
                1.0, 4.0));
        newBorn->setEnergyCapacity(newBorn->getGenotype()->convertintofloat(newBorn->getGenotype()->getecb(),
                10.0, 50.0));
        newBorn->setTrust(newBorn->getGenotype()->convertintofloat(newBorn->getGenotype()->gettrustb(), 0.25,
                1.0));
        newBorn->setLifeSpan(newBorn->getGenotype()->convertintofloat(newBorn->getGenotype()->getlifespanb(),
                10.0, 70.0));
        newBorn->setEnergyLevel(newBorn->getEnergyCapacity());

        //Print new animat's phenotype data.
        animTcl::OutputMessage("_____________________________________________________________________\n");
        animTcl::OutputMessage("A new animat [%d] has been born from %d and %d\n", newBorn->getId(),
                a->getId(), b->getId());
        animTcl::OutputMessage(">>Speed:           %f\n", newBorn->getSpeed());
        animTcl::OutputMessage(">>Feeding speed:   %f\n", newBorn->getFeedingSpeed());
        animTcl::OutputMessage(">>Energy capacity: %f\n", newBorn->getEnergyCapacity());
        animTcl::OutputMessage(">>Trust:           %f\n", newBorn->getTrust());
        animTcl::OutputMessage(">>Lifespan:        %f\n", newBorn->getLifeSpan());

        //Fill attributes according to genotype and process of recombination, etc.
        if(lamarckian)          //Are they lamarckian?
        {               //They will pass on what they have learned.
                newBorn->getMentalMap()->copyMentalMap(bestParent->getMentalMap());//Copy fittest parent's
```

```
                                                                    //mental map into newborn's.
                newBorn->setExperience(bestParent->getExperience()/bestParent->getAge() * 0.01); //Adjust
                                                                    //experience for newborn.
                animTcl::OutputMessage(">>Mental map from: %d\n", bestParent->getId());
        }

        animTcl::OutputMessage("_____________________________________________________________\n");

        cells[worldToGridCoordinates(a->getPosition())].getAntIDList()->insertAtEnd(newBorn->getId());    //Says
                                                                    //to cell where new born has appeared.
}
```

The Exchange-Information Function

```
/********* Function to exchange information with another animat ***********/
void WorldSystem::getAdvice(Ant *ant, IntList *othersList)
{
        Ant *partner;   //Pointer to every animat the current one is going to evaluate.
        Ant *teller;    //Partner animat which will exchange information with current recipient animat.
        float reliability = -1.0;
        IntNode *partnerID;
        float antAvg, partnerReliability;

        if(othersList->getElementsCount() > 1 && ant->getAge()>=1.0)     //Animat is not alone and its age is
                                                                    //more than one second?
        {
                antAvg = ant->getExperience()/ant->getAge();     //Computes recipient animat's average
                                                                    //experience.
                partnerID = othersList->gotoStart();     //Get the id of the first animat id.
                while(partnerID!=NULL)                   //Traverse the whole list.
                {
                        if(partnerID->getValue() != ant->getId() && ant->getAdvisors()->
                                findInt(partnerID->getValue())==NULL)    //Other than the animat itself? Animat
                                                                    //can exchange with any of the others just once.
                        {
                                partner = antList.findAnt(partnerID->getValue());//Finds partner in world animat
                                                                    //list.
                                if(partner->getAge()>=1.0 && partner->getState()!=DOOMED)//Evaluate only a
                                        //partner with enough age (also to avoid division by 0).
                                {       //Function to compute reliability based on average experience and biased
                                        //towards older animats than the recipient.
                                        partnerReliability = (partner->getExperience()/partner->getAge() +
                                                (partner->getAge()-ant->getAge())/(partner->getAge()+ant->
                                                getAge()))*ant->getTrust();
                                        if(partnerReliability > antAvg && partnerReliability > reliability)
                                        {
                                                reliability = partnerReliability;//Keep track of possible
                                                                    //candidates to exchange information.
                                                teller = partner;
                                        }
                                }
                        }
                        partnerID = partnerID->getNext();        //Next id of the partners in the same cell.
                }

                ////////////////Exchange mental maps if a teller passed the reliability test ////////////////////
                if(reliability > -1.0)
                {
                        ant->getAdvisors()->insertAtEnd(teller->getId());//Insert a new advisor for this cell.
                        ant->getMentalMap()->copyMentalMap(teller->getMentalMap());//Copy teller's mental map
                                                                    //into current animat's.
                        ant->setExperience(reliability * ant->getAge()); //Adjust experience to keep the same
                                                                    //ratio as the teller's reliability.
                        animTcl::OutputMessage("Animat %d has received information from animat %d\n", ant->
                                getId(), teller->getId());

                        ////////////// Check that, maybe the animat was FOLLOWING a path with previous thoughts;
                        //////////////let it update its path because there could be a closer spot with food
                        //////////////according to the new information.
                        if(ant->getState() == FOLLOWING)
                        {
                                ant->setState(EXPLORING);
```

```
                                ant->getPath()->deleteList();
                                animTcl::OutputMessage("Animat %d will try to find a path to a closer food
                                        source, according to its new information\n", ant->getId());
                        }
                }
        }
}
```

The A*-Based Path Finder Function

```
/************* Function to compute the shortest path to a cell goal ************/
bool AStar::findPath(int start, int goal, int cellsPerSide, IntList *path, MentalMap *mm)
{
        AStarCell *v;
        int x, y;
        bool tentativeIsBetter;
        int tentativeGScore;

        Heap openSet(cellsPerSide*cellsPerSide); //Define the heap for the open set.
        AStarCell::aStarCellsCount = 0;  //Initializes static variable for allowing cells to determine their
                                         //indices.
        AStarCell *map = new AStarCell[cellsPerSide*cellsPerSide];       //A map of cells that correspond to
                                                                         //animat's mental cells.

        map[start].setG(0);                     //Initializes variables for start cell.
        map[start].setH(manhattanDistance(start, goal, cellsPerSide));
        map[start].setF(map[start].getH());             //Estimated total distance from start.
        openSet.insertElement(&map[start]);             //Insert address of initial cell in open set.

        while((v=openSet.extractMinimumElement())!=NULL) //While open set is not empty.
        {
                x = v->getIndex();      //Smallest-f-value element.
                if(x == goal)           //Reached goal?
                {
                        buildPath(map, start, goal, path);       //Constructs a path from start to goal cells.
                        delete [] map;                           //Delete dynamic allocated memory.
                        return true;                             //Finishes routine.
                }
                else
                {
                        map[x].setClosedSet(true);               //Now x cell is in the set of evaluated nodes.
                        int neighbors[8];                        //Up to 8 neighbors for x.
                        int numberOfNeighbors = getNeighbors(x, neighbors, mm, cellsPerSide);     //Get
                                                                                                  //neighboring cells.
                        y = 0;
                        while(y < numberOfNeighbors)                     //Do for every neighbor of x.
                        {
                                if(!map[neighbors[y]].getClosedSet())    //Is neighbor not at the closed set?
                                {
                                        tentativeGScore = map[x].getG() +
                                                distanceBetweenNeighboringCells(x, neighbors[y], cellsPerSide);
                                        if(map[neighbors[y]].getHeapPosition()==-1)      //Current neighbor is
                                                                                        //not at open set?
                                                tentativeIsBetter = true;
                                        else
                                        {
                                                if(tentativeGScore < map[neighbors[y]].getG())   //Better path?
                                                        tentativeIsBetter = true;
                                                else
                                                        tentativeIsBetter = false;
                                        }

                                        if(tentativeIsBetter)    //Modify values of current neighbor?
                                        {
                                                map[neighbors[y]].setFrom(x);            //Set parent.
                                                map[neighbors[y]].setG(tentativeGScore); //Set g value.
                                                map[neighbors[y]].setH(manhattanDistance(neighbors[y],
                                                        goal, cellsPerSide));    //New h.
                                                map[neighbors[y]].setF(map[neighbors[y]].getG() +
                                                        map[neighbors[y]].getH());       //New f.
                                                if(map[neighbors[y]].getHeapPosition()!=-1)      //Is it already
                                                                                                 //at open set?
```

```
                                                openSet.updateElement(&map[neighbors[y]]); //Update
                                                                //element at heap.
                                        else
                                                openSet.insertElement(&map[neighbors[y]]);      //New
                                                                //element at heap.
                                }
                        }
                        y++;    //Next neighbor.
                }
        }
        path->deleteList();     //Clear integer list (path).
        delete [] map;                                          //Delete dynamic allocated memory.
        return false;           //There is not path from start cell to goal.
}
```

GENETIC ALGORITHM

```
/*Function to recombine allele values of parents to obtain the genotype of the child animat*/
unsigned long Genotype::recombine(int *arr1, int *arr2, int y, int select_dominant_parent)
{
        unsigned long val=0;
        int x=rand()%8;
        int arr[8];
        if(select_dominant_parent==0)
        {

        for(int i=0;i<x;i++)
        {
                arr[i]=*(arr2+i);               //Take the MSB from appropriate parent
        }
        for(int j=x;j<8;j++)
        {
                arr[j]=*(arr1+j);
        }

        }

        else
        {

                for(int m=0;m<x;m++)
        {
                arr[m]=*(arr1+m);
        }
        for(int l=x;l<8;l++)
        {
                arr[l]=*(arr2+l);
        }

        }

        for(int k=0;k<8;k++)
        {
                val+=pow(10.0,k)*arr[k];
                if(y==1)
                        binaryspeed[k]=arr[k];
                if(y==2)
                        binaryfs[k]=arr[k];
                if(y==3)
                        binarylifespan[k]=arr[k];
                if(y==4)
                        binarytrust[k]=arr[k];
                if(y==5)
```

```
                    binaryec[k]=arr[k];
        }

        return val;
}
```

A2. Simulation Outputs

Figures A1 and A2 display the simulation windows when the system starts up. Figure A1 corresponds to a dynamic environment while figure A2 corresponds to a static one. Both simulations are initialized with high level of resources and 15 cells per side.

Figure A1. Initialization of a dynamic-environment simulation.

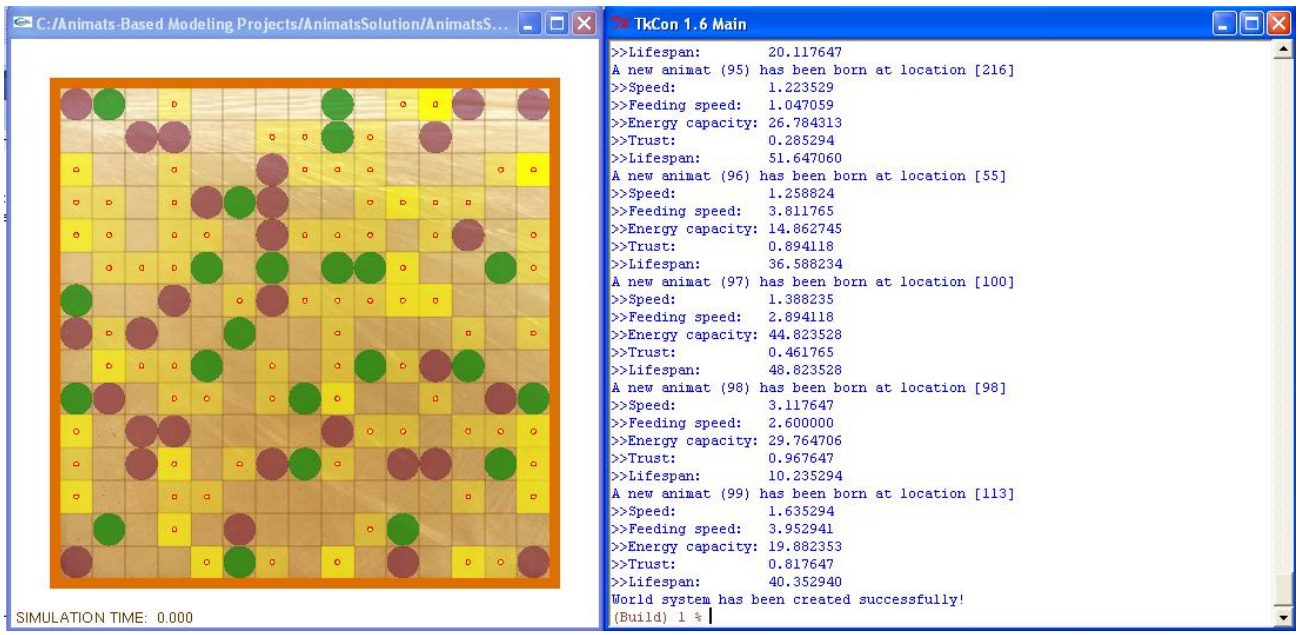


Figure A2. Initialization of a static-environment simulation.

Figure A3 shows sample generated messages in the output window when the simulation in a dynamic environment with information exchange allowed among Lamarckian individuals reached 31 seconds.

Figure A3. Sample output messages during a simulation in a dynamic environment.

Figure A4 shows information generated at the output window at t = 31 seconds, regarding a simulation of a static environment with high level of resources and knowledge exchange allowed among Darwinian agents.

Figure A4. Sample output messages during a simulation in a static environment.

Table A1 shows information regarding average population experience, collected from a simulation in a dynamic environment with information exchange allowed and high level of resources during 99 time steps.

Time	*Darwinian*	*Lamarckian*
100.01	66.523837	33.790883
100.02	66.530206	33.836338
100.03	66.599312	33.836338
100.04	66.605682	33.870428
100.05	66.62479	33.904519
100.06	66.622355	33.904519
100.07	67.172986	33.942828
100.08	65.990483	34.517019
100.09	67.607415	34.524006
100.1	69.058313	33.145656
100.11	69.093035	33.185129
100.12	69.172775	33.24166
100.13	69.193108	33.24166
100.14	69.220886	33.24166
100.15	69.220886	33.294291

100.16	69.227831	33.333765
100.17	69.307917	33.350557
100.18	69.422404	33.390031
100.19	68.991905	33.429504
100.2	69.094232	33.429504
100.21	69.121818	33.429504
100.22	69.149404	33.458976
100.23	68.953829	33.472134
100.24	68.960678	33.497623
100.25	69.00376	33.497623
100.26	69.031157	33.510781
100.27	69.079103	33.563412
100.28	69.099651	33.727325
100.29	69.099651	33.727325
100.3	69.20423	33.727325
100.31	69.271989	33.766799
100.32	69.656018	33.766799
100.33	69.710812	33.845747
100.34	69.745059	33.858904
100.35	69.788896	33.872995
100.36	70.406737	33.899311
100.37	70.510101	33.899311
100.38	70.564895	33.965101
100.39	70.574962	33.965101
100.4	70.602359	34.03089
100.41	70.609209	34.044048
100.42	70.658449	34.073049
100.43	70.687916	34.073049
100.44	70.722163	34.099365
100.45	70.797505	34.099365
100.46	70.838601	34.112523
100.47	71.108659	34.112523
100.48	71.21082	34.112523
100.49	71.217669	34.204628
100.5	71.238217	34.204628
100.51	71.39633	34.204628
100.52	71.413045	34.204628
100.53	71.600385	34.217786
100.54	71.627152	34.244102
100.55	71.640851	34.25726
100.56	71.640851	34.349365
100.57	71.748924	34.388839
100.58	71.776321	34.520259
100.59	71.905594	34.559733
100.6	71.980936	34.559733
100.61	72.157038	34.599206
100.62	72.191285	34.63868
100.63	72.313408	34.691312
100.64	72.705482	34.691312
100.65	72.760277	34.704469
100.66	72.790845	34.704571
100.67	72.877304	34.929968
100.68	72.884154	34.969441
100.69	72.408751	34.982599
100.7	72.435962	35.129173
100.71	72.510792	35.168646
100.72	72.589878	35.051541
100.73	72.612806	35.090502
100.74	72.680833	35.098559
100.75	72.721649	35.111546
100.76	72.975743	35.111546
100.77	73.150347	35.13752
100.78	72.747758	35.202455
100.79	72.810119	35.321781
100.8	72.810119	35.613053
100.81	72.816876	35.652014
100.82	72.843918	35.652014
100.83	73.064936	35.23357
100.84	73.118991	35.710634
100.85	73.166288	35.736275
100.86	73.179801	35.798855
100.87	73.220342	35.850137
100.88	73.339175	35.914239
100.89	73.473824	35.952701
100.9	73.487337	35.965522
100.91	73.603743	36.055265
100.92	73.603743	36.093727
100.93	73.6105	36.132188
100.94	73.709591	36.156813
100.95	73.716348	36.156813
100.96	73.864996	36.167641
100.97	73.871753	36.206102
100.98	73.940116	36.531316
100.99	74.128882	36.531316

Table A1. Experience data sample from a simulation in a dynamic world with high level of resources.

Table A2 contains information from average population trust, collected from a simulation in a dynamic environment with no information exchange allowed and low level of resources during 99 time steps.

Time	Darwinian	Lamarckian
20.01	0.61629	0.609719
20.02	0.61629	0.609719
20.03	0.61629	0.609719
20.04	0.61629	0.609719
20.05	0.61629	0.609719
20.06	0.61629	0.609719
20.07	0.61629	0.609719
20.08	0.61768	0.609719
20.09	0.62039	0.609719
20.1	0.621974	0.608693
20.11	0.625992	0.612426
20.12	0.625992	0.612426
20.13	0.625992	0.612426
20.14	0.625992	0.612426
20.15	0.625992	0.612426
20.16	0.625992	0.612426
20.17	0.625992	0.612426
20.18	0.625992	0.612426
20.19	0.625992	0.612426
20.2	0.625992	0.612426
20.21	0.624577	0.612426
20.22	0.624577	0.612426
20.23	0.624577	0.609967
20.24	0.624577	0.609967
20.25	0.624577	0.609967
20.26	0.628476	0.609967
20.27	0.628476	0.609967
20.28	0.628476	0.609967
20.29	0.628476	0.609967
20.3	0.628476	0.609967
20.31	0.628476	0.609967
20.32	0.628476	0.609967
20.33	0.628476	0.609967
20.34	0.628476	0.609967
20.35	0.628476	0.609967
20.36	0.628476	0.609967
20.37	0.628476	0.609967
20.38	0.628476	0.609967
20.39	0.628476	0.609967
20.4	0.628476	0.609967
20.41	0.628476	0.609967
20.42	0.628476	0.609967
20.43	0.628476	0.609967
20.44	0.628476	0.609967
20.45	0.628476	0.609967
20.46	0.628476	0.609967
20.47	0.628476	0.609967
20.48	0.628476	0.609967
20.49	0.628476	0.609967
20.5	0.628476	0.609967
20.51	0.628476	0.609967
20.52	0.628476	0.609967
20.53	0.628476	0.609967
20.54	0.628476	0.609967
20.55	0.628476	0.609967
20.56	0.628476	0.609967
20.57	0.628476	0.609967
20.58	0.628476	0.609967
20.59	0.628476	0.609967
20.6	0.628476	0.609967
20.61	0.628476	0.609967
20.62	0.628476	0.609967
20.63	0.628476	0.609967
20.64	0.628476	0.609967
20.65	0.628476	0.609967
20.66	0.628476	0.609967
20.67	0.628476	0.609967
20.68	0.628476	0.609967
20.69	0.628476	0.609967
20.7	0.628476	0.609967
20.71	0.628476	0.609967
20.72	0.632218	0.613417
20.73	0.632218	0.613417
20.74	0.632218	0.613417
20.75	0.632218	0.613417
20.76	0.632218	0.613417
20.77	0.632218	0.613417
20.78	0.632218	0.613417
20.79	0.632218	0.613417
20.8	0.632218	0.613417
20.81	0.632218	0.613417
20.82	0.632218	0.613417

20.83	0.632218	0.613417
20.84	0.632218	0.613417
20.85	0.632218	0.613417
20.86	0.632218	0.613417
20.87	0.632218	0.613417
20.88	0.632218	0.613417
20.89	0.632218	0.613417
20.9	0.632218	0.613417
20.91	0.632218	0.613417
20.92	0.632218	0.613417
20.93	0.632218	0.613417
20.94	0.63379	0.613417
20.95	0.63379	0.613417
20.96	0.63379	0.613417
20.97	0.63379	0.613417
20.98	0.63379	0.613417
20.99	0.63379	0.613417

Table A2. Trust data sample from a simulation in a dynamic world with low level of resources.