TAZBOTS:

an agent-based model inspired by Tasmanian devils

Arianna Pace, Giulia Paci and Manami Zanzi

a.a. 2013-2014

Contents

Introduction

The main goal of this project is to simulate a system in which agents (bots), representing animals, can interact with each other and with the environment in order to perform different actions. Each bot is equipped with a neural network which determines its actions, and the population is evolved through a genetic algorithm as they learn tasks such as searching for food or biting.

Biological background The inspiration for this project came from an article on the Tasmanian devil's transmissible cancer, published on the June 2011 issue of Scientific American [3]. The Tasmanian devil facial tumour disease (DFTD) was first reported in northeastern Tasmania in 1996 and has spread quickly, putting the population survival at risk. In 2009 it was reported that the population decline had surpassed the 60%, with peaks of the 90% in the areas where the disease had been present for the longest time[4]. This unusual infection process is thought to be possible due to the extremely low genetic diversity of this population, and the primary means of tumor transmission is believed to occur through biting. This is a common behaviour for these animals, which bite each other when competing for food or during sexual encounters. Among the most interesting studies, Hamede and others reported in [2] that the tumor transmission most likely occurs from an infected animal which is bitten to the biter: the authors suggest that this disease may thus favour less aggressive phenotypes in evolution.

Main features of the simulation The simulation was implemented in C++ and it can be run in two different "modes": evolution or graphics, which are selected by the user through a keyboard input. In the first case the population of bots is evolved through a genetic algorithm and the graphics is completely switched off; in this mode the user can monitor the evolution through different files that are continuously update and contain information such as the number of survived bots and the fitness function. In the other case it is possible to inspect graphically the behaviour of the bots previously evolved and, on the other hand, the genetic algorithm is switched off. The simulated environment is 2D with a toroidal geometry in which the opposite sides are connected, and the bots positions are given by two continuous-valued coordinates, whereas food is distributed in blocks of a fixed dimension. Each bot has a neural network, which structure depends on the phase being simulated (i.e. which task the bots are learning). The main actions that these agents perform are looking for food, biting each other and trying to escape when being bitten. Our interest in the simulation of the biting process naturally comes from the biological inspiration

of this project.

Structure of the report This report is organised as follows: in the first chapter we outline the structure of the program, describing the main classes and functions implemented for this simulation. The second chapter is focused on a detailed description of the three different simulation phases and the respective fitness functions employed in the genetic algorithm. In the third chapter we present and comment the results obtained for the three different phases and discuss the conclusions and future directions of this work. Finally in the appendix we outline the major changes that the program has undergone in order to solve specific issues or add extra features.

Chapter 1

Structure of the Program

The program is structured in seven blocks: each one includes a .cpp file as well as a header file. The inclusion hierarchy is shown in Fig.1.1 and the details of each block are listed below:

- Brain: in this module the neural networks of the brain are implemented.
- Diavoli: the class Diavolo is defined here, as well as functions for the creation of random bots and for computing inputs.
- Environment: this module contains functions for the management of the bots' environment, such as functions for the distribution of food, for the performing of actions as consequences of the brain outputs. The update function deals with the different actions that are carried out recurrently at each time step.
- Utilities: in this module functions for the saving and loading of data are implemented.
- Gen_ alg: functions for the creation of new generations through a genetic algorithm are implemented in this block.
- Grafica: deals with the graphic aspects of the simulation.
- Main: this module allows the user to select different simulation modes (e.g graphic or evolution) and to set the values of different parameters. It is the block which calls all functions necessary for the functioning of the program.

1.1 Brain

This module contains the implementation of the neural network that controls the behaviour of the bots.

The inputs are:

- Food view: the bot inspects the food boxes around its position (up to a distance of DciboV and included in the bot's angle of view) and each eye selects the closest box containing some food; the amount of food - divided by its distance - is then set as the input.

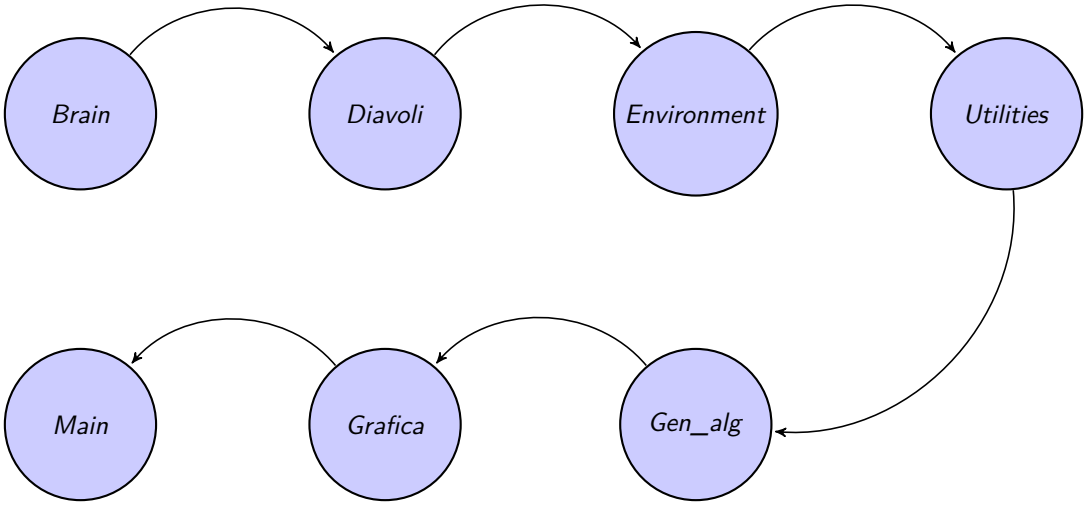


Figure 1.1: Inclusion hierarchy

- Food smell: this input is the difference between the cumulative sum of the food around the bot's position up to Dcibo, weighted by the food's distance, in the current time and in the previous time step.
- Bots view: the bot inspects the bots around its position (up to a distance of DdiavV and included in the bot's angle of view) and each eye selects the nearest bot. The input is proportional to the closeness of this bot to the selected one.
- Bots smell: this input is the cumulative sum of the bots around the bot's position up to Ddiav, weighted by the bots distance in the current time.
- Bitten / Biter: the value is 1 if the bot was bitten or bit in the previous time step.
- Health / health change: health of the bot and difference in health between the current and previous time steps.

The outputs are:

- Right wheel, left wheel: the speed of the bots' wheels.
- Bite: if this output (weighted by the bot's aggressiveness) is over a fixed threshold the bot bites.
- Boost: this output enables the bot to speed up for a time step.
- Right boost wheel / left boost wheel: extra wheels that substitute the normal ones when the boost is activated.

The inputs are processed by the neural network. The simulation requires three phases of evolution, so the bots' brain is divided into three parallel components (see Fig.1.2). Each part is a fully connected feedforward neural network, with a number of hidden neurons equal to double the number of inputs. The neuron zero of each layer is employed as the bias, therefore it does not receive any inputs and it has a constant value equal to one. The transfer function employed is tanh. We decided not to employ any learning algorithm, so the neural network evolves only through the genetic algorithm.

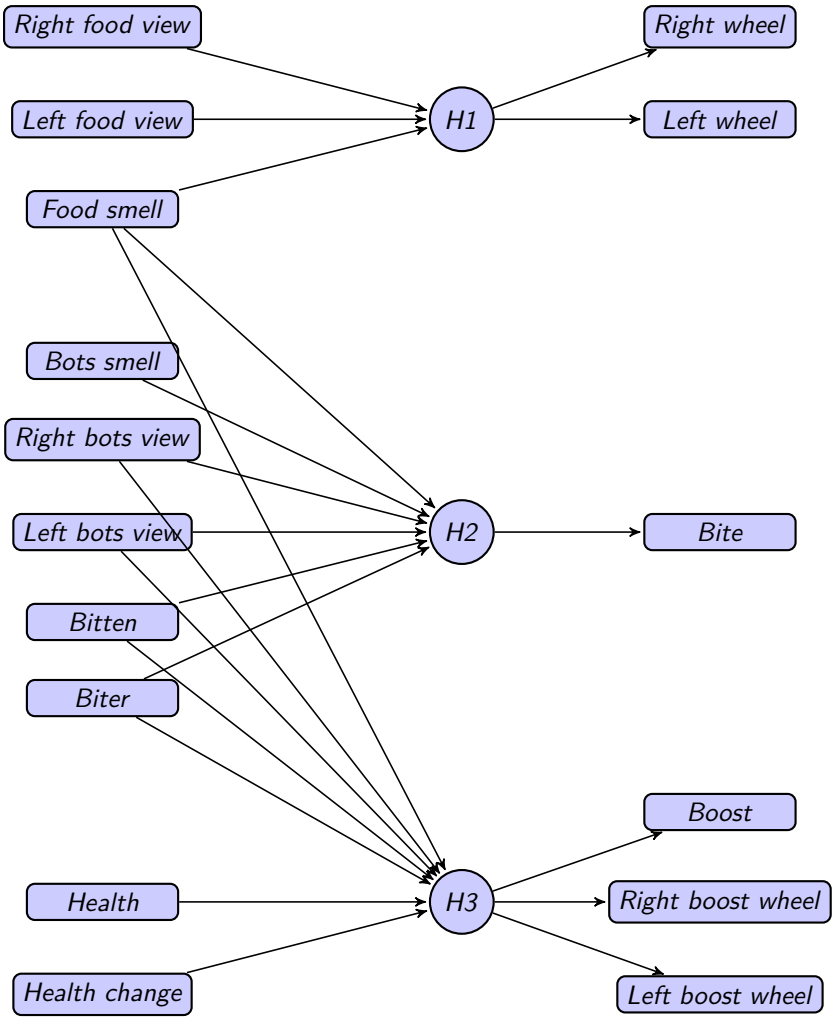


Figure 1.2: Structure of the bots' brain

The first component is designed to making bots capable of searching for food. Therefore the inputs in this phase are only the ones related to food and the outputs are the speed of the two wheels, that allow the steering of bots towards the food.

The second one deals with the action of biting. The objective here is to evolve the bots in order to discourage them from biting when no other bots are around and encourage them to bite when food is near. The inputs are related to the bot's sensing of other bots and food (only smell) and the output enables the bot to bite.

In the third component the goal is to link the two behaviours previously evolved, for example by allowing the bots to escape when they are bitten or to search food while also avoiding other bots. Therefore a new output is added: boost, which enables the bot to move at a double or triple speed. The final output relative to the movement and bite of the bot is obtained by linearly combining the corresponding outputs from the two previous components.

The structure of the network is defined through a hierarchy of classes:

- Cervello: this class defines the complete NN, implemented as a vector of 3 Network.
- Network: a fully connected feedforward NN, with a vector of 3 Layer (input, hidden and output);
- Layer: one layer of the NN,with its vector of Neurone;
- Neurone: the neurons, each holding a vector of Dendrite;
- Dendrite: the links outgoing a neuron, each one pointing at a different neuron and holding its own weight;

1.2 Diavoli

The bots are defined as elements of the Diavolo class, which is defined in this module, and are contained in a vector called Diavoli.

Each bot has various attributes, such as:

- id: ID of the bot;
- salute, sal_old: health of the bot in the current and in the previous time step, values are in [0,SALMAX];
- sesso: true for female, false for male;
- pos: a two dimensional vector representing the x and y coordinates of the bot's position;
- angolo: it indicates the direction of the bot, values between $(-\pi,\pi]$;
- w1, w2: speeds of the bot's wheels;
- boost: specifies when the bot uses the boost function;
- brain: an object of the *Cervello* class representing the bot's neural network;
- in, out: inputs and outputs for the neural network;
- bravura: plays the role of the cost function for our genetic algorithm;
- odore_cibo_old: memory for the food smell sensed in the previous time step;
- biter, bitten: bite status of the bot at the current time step;
- genitoreA, genitoreB: IDs of the bot's parents.

The following attributes are inherited and mutated at every new generation:

- aggress: aggressiveness of the bot;
- Dcibo: distance at which food can be smelled;
- DciboV: distance at which food can be seen;
- Ddiav: distance at which other bots can be smelled;
- DdiavV: distance at which other bots can be seen;
- MUTRATE1: rate of mutation of the neural network;
- MUTRATE2: maximal intensity of mutation of the neural network;
- MUTRATE_DNA1: rate of mutation of the inherited attributes;
- MUTRATE_DNA2: maximal intensity of mutation of the inherited attributes;
- MUT_DNA_SOGLIA: threshold for food range mutation.

The module also contains some functions for managing the bots.

addRandomBots: a simple function that, as its name says, creates a desired number of random bots. This is used at every new generation to create the bots.

setInputs: a function that deals with the computation and assignation of the inputs. This includes the computation of what the bots see and smell.

1.3 Environment

The Environment module is the core of the simulation, as it contains the functions that determine what happens every time step. In this module some important parameters are also defined, such as:

- NUMTAZ: the number of bots at the beginning of the simulation, the default value is 250;
- TAZRAD: the bots' dimension, used both for drawing purposes and to regulate the distance at which bots can bite each other;
- BOTSPEED: the bots' speed when they are not using the boost option;
- INITFOODin and INITFOODfin: the percentages of the simulation environment covered by food for the first and last generations of the evolution, the default values are both 10%.

The most important functions defined in this module are:

processOutputs This is a complex function in which the bots' neural network outputs are used to control their actions: namely, their movement and the possibility to bite and boost. In addition to this, the fitness of each bot is updated in this function; the details of this process are given in Chapter 2.
The outputs are processed as follows:

- **bit**: if the sum of the bit output and morso_soglia, i.e. the bite threshold, is higher than a fixed threshold the bot attempts to bite. When the bit output is high enough the bite is stronger, causing a double damage. The bite, however, is only successful if there is a bot near enough to be bitten.

The effects of the bite, besides those on the fitness functions, are basically two: the bitten bot modifies its bitten parameter from the previous value to 1 and its health decreases by BITELOSS/10.

- **boost**: in the *boost* case the output also has to exceed a certain threshold in order for the action to happen. The bot can move twice or three time faster than the normal speed depending on by how much the threshold is surpassed. If a bot uses the boost, the dedicated wheels for the boost will be employed to move it. If the boost output is negative, the bot will go backwards.
- **wheels**: the two wheels are rotated around the bot's center by an angle equal to BOTSPEED * w (where w is either w1 or w2), first the right wheel and then the left one. If the movement causes the bot to exit the perimeter of the simulation, its position is calculated again as if the bot has entered the environment from the opposite side.

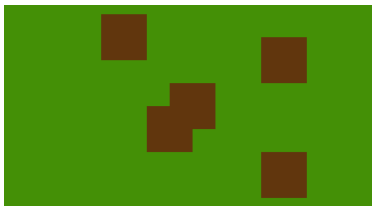


Figure 1.3: Example of the environment with the food (brown).

set_cibo and ripopola_cibo These two function deal with the food in the map. set_cibo receives as an input the number of slots in the environment that are to be filled with food. This value is calculated in the main module and the function is called there at the beginning of each new generation. This function spreads the food randomly in blocks of four adjacent slots, as shown in Figure 1.3. The ripopola_cibo function, on the other hand, is called every time step and it restores the food to the same percentage as at the starting point, distributing the food slots four at a time.

update This is the function that makes the simulation advance at every time step: it is called in the main module for NUMGIRI times.

Each time the update function is called, the modcounter increases its value by one unit. This allows us to keep trace of the *time passing by*. Then a series of actions follows, as listed below:

1. the inputs of the bots are recalculated and set as input nodes in the brain using the setInput function;
2. the cervellosTick function is called and the output are calculated for each bots' brain;
3. the processOutput function takes the outputs of the first and, possibly, of the third component of the brain and calculates the movement of the bots;
4. each bot undergoes a health loss, proportional to the energy it has spent for moving;
5. the inputs are then calculate again, so that the bots are *aware* of the new position of the bots around them;

Dcibo:	indicates how far a bot can *smell* the food
DciboV:	indicates how far a bot can *see* the food
AngoloV:	is the angle of view of each eye
AngoloVmin:	is the angle of overlap or separation between the two eyes
Ddiav:	indicates how far a bot can *smell* another bot
DdiavV:	indicates how far a bot can *see* another bot
morso_soglia:	is the threshold for the bite action
boost_soglia:	is the threshold for the boost action

Table 1.1: The parameters listed in the table are saved in the Cervelli_gen_i_dna.txt file.

6. the bite outputs are processed with the processOutputs function and the bots bite each other;
7. the health of the bots is checked and if it is lower that 0 the bots are considered dead and removed from the simulation;
8. the environment is updated: if there is less food than it should be, it is added up to the established percentage with the ripopola_cibo function;
9. the costo is calculated as the sum of the bravura of every bot; this step can be skipped when costo is not needed.

salva_costi This function is called in the main only when an evaluation the appropriate number of NUMGIRI for the simulation is wanted . This function writes costo/modcounter and the number of bots still alive in the simulation for each time step in a file. A new file is created for each new generation and this is called Costo_gen_i.txt, where i is the number of the generation.

1.4 Utilities

The utilities module contains three functions that save and load useful bots' parameters and values as listed below.

SalvaCervelli The SalvaCervelli function creates four files containing all the information about the bots currently present in the simulation: these files can be loaded in a new simulation to obtain a perfect copy of the saved bots.

The first three files contain all the information about the brains: specifically, each file stores one component of the brain. They are called Cervelli_gen_i_f_n.txt, where i is the number of the generation of the simulation and n indicates the component of the brain saved in the file. The first three rows of these files store the number of input, hidden and output nodes of the network, followed by the list of the weights of the dendrites.

The last file, called Cervelli_gen_i_dna.txt, saves the "DNA" of the bots, which includes all the genetic parameters listed in table 1.1. In all files the relevant information is preceded by the ID number of the bot that is being saved, this allows to ascertain that the features of one bot are loaded in the same bot in the LoadCervelli function.

LoadCervelli This function loads the parameters from the files saved with the SalvaCervelli function. Some information is needed in order to load them correctly. The user has to provide it through a keyboard input, when the program is running, as follows:

1. it is asked whether to use or not any saved file (answer with 0 for *no* and 1 for *yes*);
2. the user then has to insert the name of the files without the ending _f_n.txt or _dna.txt;
3. when the program is running in the evolution mode, it is asked if the last component of the brain has to be develop from a random or from a previous one (answer 0 for the *random* or 1 for the *saved* one).

A number of errors can occur during the loading process:

- the file does not exist, that is the input file name does not correspond to any file;
- the number of nodes of the loading network does not correspond to the number of nodes fixed in the simulation, probably some changes in the brain network has occurred between the saving and the loading;
- the ID of the bot in the file differs form the ID of the one in the simulation, this means that the file is probably corrupted, causing a wrong reading.

In the function LoadCervelli all these possible sources of errors are checked and the user is warned accordingly by different messages in case one of the errors should occur.

SalvaMutrate This function is used to keep track of the changes in the *mutrates* (see Table 1.2) during the evolution of the bots. It produces a file called Mutrate.txt, containing the average values of the various *mutrates* and the values for each bot.

1.5 Genetic algorithm

The Gen_alg module contains functions devoted to the different steps of the genetic algorithm. The most important ones are:

compara_bravura: a simple function which returns the result of the fitness comparison between two bots.

roulette: in order to stochastically choose bots that are to be combined to form a new population we employ fitness proportionate selection (also called roulette-wheel selection). The main idea is that fitter individuals have a better probability of survival and will form the mating pool for the next generation, however also "weaker" individuals have a chance to reproduce. In practice, this translates into the following steps:

- compute the fitness for each individual;

- sort the population by fitness;
- compute a set of decreasing values (we employ $i^{-}3/4$ with i from 1 to the number of bots that are fit enough for reproduction);
- get a random number in the interval of the values previously computed;
- translate the random number in the corresponding individual to be chosen from the sorted array.

newgen: this function deals with the creation of a new population for the genetic algorithm, by cloning and crossover of the previous one (plus additional mutations). These operations apply both to the neural network and to the inheritable features of bots. We decided to limit the number of individuals participating into the mating pool to only the best 10% of the population, and 25% of them are cloned. Details are illustrated below:

- Cloning: the fittest individuals are cloned. Three clones are created for the two fittest bots, and two clones for every other bot included in the toptoptaz (top 25% of best 10%). At the end of the process, the positions and angles of all clones are randomised.

- Crossover: bots belonging to the toptaz group (10% best individuals) are eligible for crossover and couple of bots to be combined are extracted by the roulette method previously introduced. The crossover is done as follows: the inheritable features (range of view/smell...) are chosen randomly from one of the two "parents" and copied in the new bot. As for the neural network, a random number in the range [0, number of neurons] is obtained for each layer and the new bot inherits a portion of neurons from one parent and the rest from the other parent according the number extracted (i.e if we have 10 neurons in a certain layer and we extract a number equal to 5, half neurons will be copied from the first parent and half from the second). This operation is repeated until the desired number of individuals for the new generation have been generated.

- Mutation: according to the different mutation rates, the neural network weights are changed with a certain probability (we have two different parameters that control how likely a mutation is and also its intensity). Inheritable features are also mutated according to a specific rate ("DNA" mutation rate): according to their nature some are changed in a continuous fashion (i.e. angle of view) or discretely (i.e. the range of view that is measured in food blocks and can go up or down by one). The mutation rates can also undergo small changes, according to a metamutrate parameter. See Table 1.2 for the different values and descriptions.

1.6 Graphics

As it is evident from its name, this module is dedicated to the management of the graphics in the simulation. The implementation was done mainly using OpenGL and GLUT (OpenGL Utility Toolkit), which provide many functions useful for the graphics and for the interaction with the Operating System (i.e.

Mutation rate	Description	Value
MUTRATE 1	rate of mutation neural net	0.005
MUTRATE 2	amount of mutation neural net	0.4
MUTRATE_DNA1	rate of mutation DNA	0.1
MUTRATE_DNA2	amount of mutation DNA	0.05
MUT_DNA_SOGLIA	threshold for food range mutation	0.2
METAMUTRATE 1	mutation in MUTRATE1	0.02
METAMUTRATE 2	mutation in MUTRATE2	0.5
METAMUTRATE_DNA1	mutation in MUTRATE_DNA_1	0.1
METAMUTRATE_DNA2	mutation in MUTRATE_DNA_2	0.5
METAMUTRATE_DNA_SOGLIA	mutation in MUT_DNA_SOGLIA	0.05

Table 1.2: Description of the mutation rates employed in the genetic algorithm.

creating a window, handling key and mouse inputs...).

The main function is renderScene, which controls the following actions:

- Draw food: according to the amount of food present in a certain area, the corresponding food box is drawn in a different color, from brown (box full of food) to green (no food). We also introduced an additional feature (mainly for debugging purposes), in which when a bot is selected the food inside its range of view is coloured differently.
- Draw bots: the bots are drawn as simple circles, with a small icon depicting a tasmanian devil superimposed on them. We also display lines indicating the angle of view of the bot: the left eye angle is represented by a red arc and the right one by a blue one. The angle of view in which the two overlap is drawn in pink and the length of the lines is proportional to the range of view. A number indicating the unique bot ID is written on the bottom left of the bot icon and a small bar on the right indicates the bot's health (the colour is red when the it is maximal). When a bot is selected or to represent different interactions (i.e. bite) we display a corresponding indicator. For example we employ a white indicator for a selected bot (see Figure 1.4) and yellow, red and black indicators to visualise the biting process. Another graphics feature we added (also shown in Figure 1.4) is the possibility of seeing the inputs and outputs of the bot's neural network when it is selected: these are displayed as boxes with a colour ranging from blue (negative values) to red (positive values). Finally the use of the boost is displayed as two concentric circles, which enlarge and then disappear as soon as the bot stops using the boost.

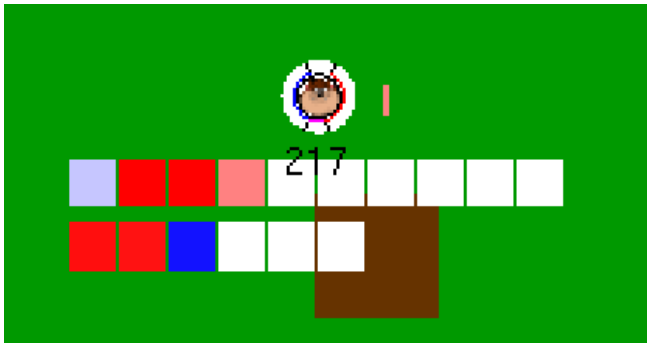


Figure 1.4: A bot which has been selected (white indicator): notice the inputs/outputs active in this phase displayed below.

Chapter 2

Simulations

In this chapter we describe the three different simulation phases, illustrating their main objective as well as which parameters are mutated in each one. We also explain in detail how the fitness function is computed in the three different cases.

2.1 First phase: food

2.1.1 Objective

The first phase of simulation tries to create a population of bots capable of moving towards food, in order to eat and survive. In this phase we only use the first portion of the neural network: namely we only have the food smell and sight as inputs, and the the speeds of the wheels as outputs (see Figure 1.2).

At each time step the bots move according to the output of their neural network. If they are standing on food and if their health is not maximum, they eat. Eating food restores the bots' health, and the food intake is inversely proportional to the speed of the bot. We keep track of the amount of food consumed during each time step, so that it can be replenished and kept at a fixed density.

2.1.2 Parameters' evolution

All the genetic parameters are inherited from one of the two parents. In this first phase we chose to mutate only those connected with the search for food: Dcibo, DciboV, AngoloV and AngoloVmin.

2.1.3 Fitness function

In this phase, the fitness function of a bot will increase:

- when it is getting closer to food;
- when it succeeds in actually eating food.

On the other hand we decrease its fitness when the output is very similar to that of the previous step: this should encourage the bots to change their output,

rather than proceed in one fixed direction (this is an issue we encountered in the first developing phases). Finally, we gradually decrease the fitness at each time step to stimulate the bots to keep eating.

2.2 Second phase: bite

2.2.1 Objective

In the second phase, our goal is to evolve a population of bots capable of biting each other. In particular we wish the bites to be "meaningful": the bots should bite only when other bots are around, otherwise it would be a fruitless waste of energy. For this part of the simulations the neural network added in parallel to the first one has the food smell, the bots' smell and sight and a bitten/biter status as inputs, and the bite as output: the motion of bots is still controlled by the part of the neural network developed in the first phase.
If a bot is bitten (as indicated by the bitten attribute) it loses health and in addition it will not be able to eat food in the next round: this introduces an advantage for more "aggressive" bots. We consider two possible bite intensities, depending on the output of the neural network, which is the reason why the bitten and biter attributes can have values bigger than one. In addition to this we also introduced a random aggressiveness parameter, which controls how likely the bot is going to bite. The biting process is visualised graphically by "auras" of different colours around the bot indicators, as explained in more detail in the Graphics section.

2.2.2 Parameters' evolution

As far as the inheritable genetic parameters are concerned, in this phase only the following ones are mutated: Ddiav, DdiavV and morso_soglia. The latter is a value which is summed to the bite output of the neural network and thus determines how easy it is for the bot to bite.

2.2.3 Fitness function

The fitness function of a bot in this phase will increase:

- when it bites if another bot is near;
- additionally if it bites near food (assessed by smell);
- additionally if the bite is stronger;
- if it doesn't bite.

We decided to introduce an increase in the fitness function for the bots which do not bite in order to allow evolution of less "aggressive" bots which could rather avoid contact with other bots as a survival strategy (if the fitness function were to be increased only when bots bite, the genetic algorithm would only select bots which bite often).

On the other hand, the fitness function of a bot will decrease:

- when it bites far from other bots;
- when it is bitten;
- additionally if the bite is stronger.

2.3 Third phase: boost

2.3.1 Objective

The aim of this third evolutionary step was to create bots capable of escaping from other aggressive ones. This is needed because otherwise when they encounter other bots remain in a state of bite/bitten until they run out of energy, and this is not what we look for. A secondary scope for this acceleration could also be to reach the food faster than the others around.

These are the reasons why the third component of the brain was designed in order to give the bots the ability to go more rapidly than usual and change their direction, maybe even going away from where the food is, when this is necessary. In fact, the first output of the network defines when the decision of using the boost is made and how much faster (two or three times) the bot is going to be, i.e. how many time to repeat the single move action in processOutput function, depending on the comparison with a personal threshold which can vary from bot to bot. If the bot uses this option it will have to spend more energy and so its health will be decreased twice or three times more as it would be otherwise with the baseloss. This is because we don't want the bots to use the boost continuously, but only when it's needed.

2.3.2 Parameters' evolution

All the genetic parameters are inherited from one of the two parents, but in this third phase only one of them is mutated: the threshold for the boost usage. This is the only parameter that can influence the result of the current fitness function, defined as explained in the next section.

2.3.3 Fitness function

The fitness function of this phase has to take into account two factors:

- whether the bot has wisely decided to escape (we could call this *decision fitness*),
- whether the direction where he moved was a good one (*direction fitness*).

About the decision fitness we decided to reward or punish the bots only on the base of them having been bitten or not, because if they boost towards food they are rewarded in the direction fitness. So, the bot is more fit if it uses the boost when it has just been bitten (bravura+=BRAV*10), but it is less fit when it was neither bitten nor getting closer to the food (bravura-=BRAV*60). The penalty is much higher than the bonus because we want to suppress the misuse of this tool as soon as possible.

With respect to what we called direction fitness, we increase the fitness if the bot is no more under attack thanks to the boost usage or if he is still getting

nearer to a food source. In the first case we add BRAV*20*a->boostpassato, where boostpassato is a counter that is set at 10 when the bot uses the boost and decreases its value of one unit every time step, to its bravura. In the second case we increase it of one BRAV. The difference between the two amounts of reward is due to the difference of the probabilities of the two events. What is more, since we want the bots to learn how to avoid repeated injuries, when the bot is bitten after the use of the boost bravura decreases of -BRAV*20*a->boostpassato.

Even if not closely related to the boost action, there is an additional prize of BRAV*10 for the bots which have eaten in this time step. This is because we still want the bots to eat as much as they can.

Chapter 3

Results and conclusion

In this chapter we present the results of simulations of the three different phases of evolution and discuss the conclusions and future directions of this project. For each phase we first performed an assessment of the cost function in order to choose the optimal number of time steps for the generations in the genetic algorithm. Once this evaluation had been made it was possible to proceed and perform different rounds of simulations with the genetic algorithm: these were evaluated by inspecting the fitness function behaviour for the different generations. Naturally, many simulations were performed for each phase, but for brevity we report here only on specific ones (usually the best ones).

3.1 First phase

In Figure 3.1 the evaluation for the cost at the different time steps for 6 different generations is shown. We perform the analysis on a maximum of 10000 time steps and 25 generations: for this first phase we estimate that 3000 time steps for each generation of the genetic algorithm will suffice, in fact the cost curves have already stabilised well at this step number.

In Figures 3.2 and 3.3 we report the plots of the fitness function and of the surviving number of bots for a phase 1 simulation up to the 500th generation. We can see that on average the fitness increases and then reaches a "plateau", moreover the plot for the worst bot presents wide oscillations when compared to the best ones. As for the number of surviving bots, the results are excellent in this phase: after the first few generations a 100% survival rate is quickly reached.

In Figures 3.4 and 3.5 the plots relative to the bots' DNA are shown for the two best bots. We specifically report the inheritable characteristics important in this phase: the range of food smell and sight, the angle of view and the overlap angle. We can see that there is a big variability among different generations, and that the parameters value often stabilises around values different from the default ones. For example we can notice how the range of view decreases markedly from a default of 10 to around 4: this could be interpreted as an advantage for an increased "specificity" of view.

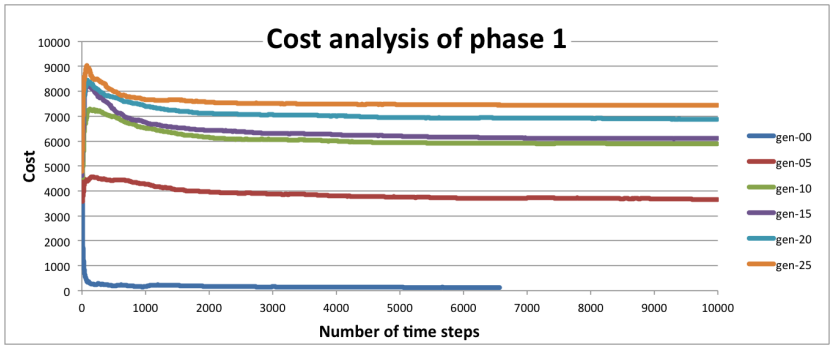


Figure 3.1: Cost analysis for phase 1.

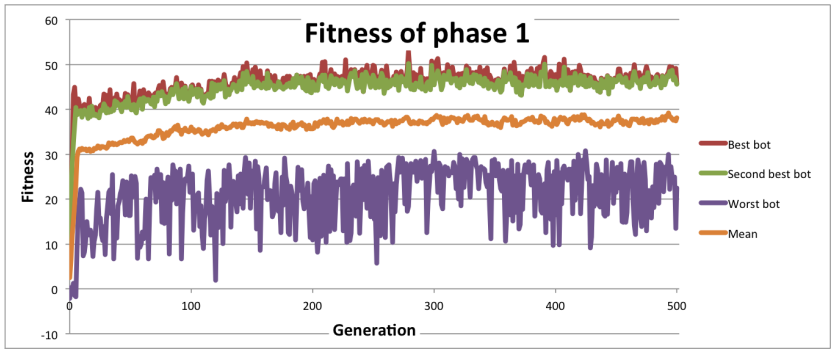


Figure 3.2: Example of fitness evaluation for phase 1.

3.2 Second phase

In Figure 3.6 the evaluation for the cost at the different time steps for 6 different generations is shown as done previously. Notice how in this case the curves stabilise more slowly compared with the first phase: we therefore decided to set 5000 as the number of time steps in phase 2.

In Figures 3.7 and 3.8 we report the plots of the fitness function and of the surviving number of bots for a phase 2 simulation up to the 1000th generation. The curve of the fitness undergoes a "jump" after a first few generations and then stays approximately constant until another increase occurs between generation 300 and 400. The value then stabilises, and as earlier we always observe wide oscillations in the plot relative to the worst bot. In this case we decided to continue the simulation up until 1000 generations (instead of 500) because at the 500th generation it wasn't clear whether the fitness function had really stabilised or not. In this phase the bots' survival doesn't reach 100% as earlier, but this is expected as the biting process was introduced.

In Figure 3.9 the plots relative to the bots' DNA are shown for the two best bots. We specifically report the inheritable characteristics important in this

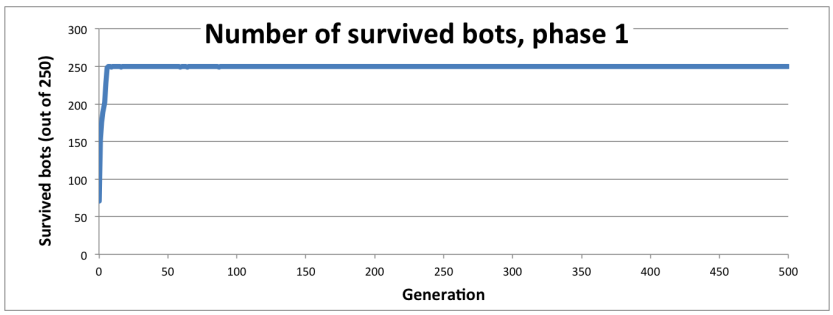
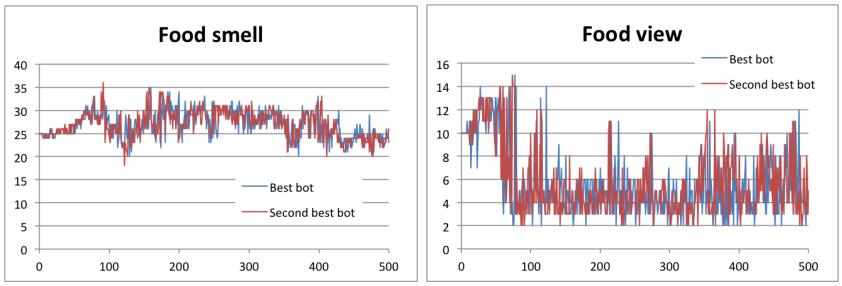


Figure 3.3: Example of plot of the surviving number of bots in phase 1.



(a) (b)

Figure 3.4: Example of DNA evolution in phase 1.

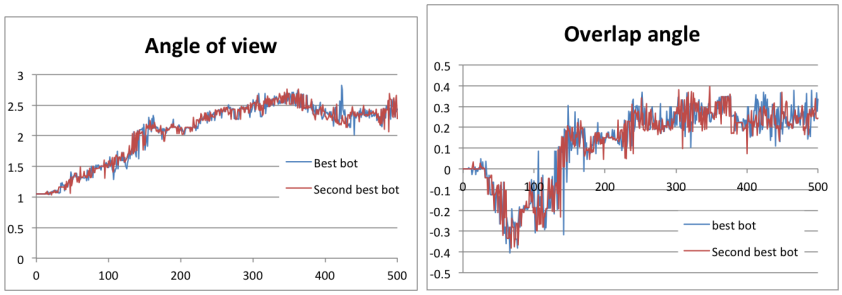
phase: the range of bots' smell and the range of bots' sight. These parameters stabilise around values different from the default ones and, as observed earlier, this is especially noticeable for the range of view.

3.3 Third phase

In Figure 3.10 the evaluation for the cost at the different time steps for 6 different generations is shown as done previously. Here we decided to set 4000 as the number of time steps for this phase, intermediate between the first and second phases.

In Figures 3.11 and 3.8 we report the plots of the fitness function and of the surviving number of bots for a phase 3 simulation up to the 500th generation. As observed earlier, the fitness increases slowly and then stabilises, even if the plot for the worst bot always presents wide oscillations. As for the number of surviving bots, this is low for the first few generations and then approximately reaches 100%: even if there are some oscillations, survival in this phase is very good.

We do not show any DNA evolution plots for this phase as there are no relevant inherited parameters of interest to be inspected.



(a) (b)

Figure 3.5: Example of DNA evolution in phase 1.

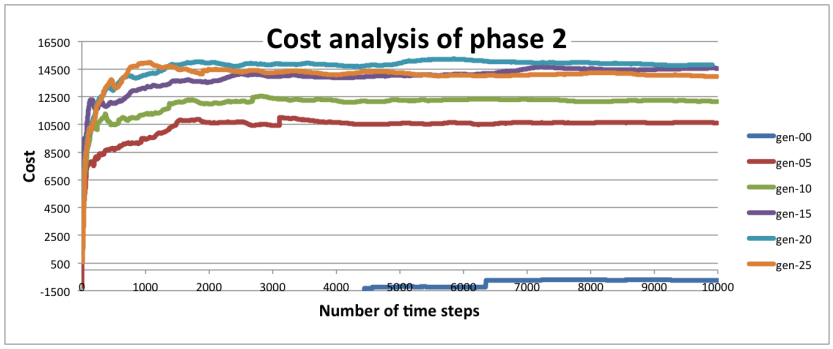


Figure 3.6: Cost analysis for phase 2.

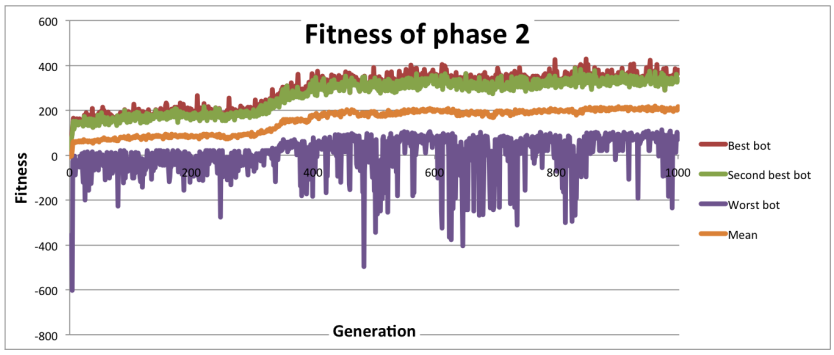


Figure 3.7: Example of fitness evaluation for phase 2.

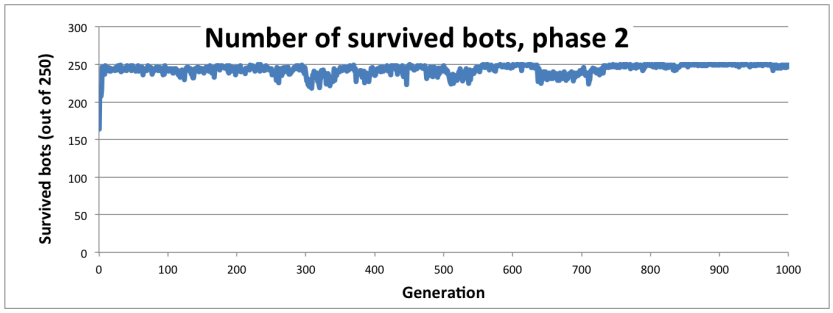
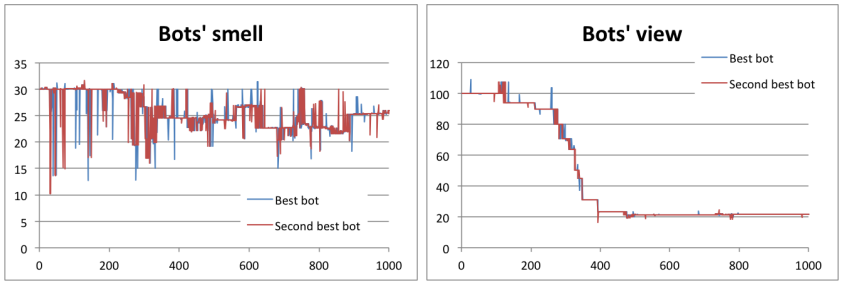


Figure 3.8: Example of plot of the surviving number of bots in phase 2.



(a) (b)

Figure 3.9: Example of DNA evolution in phase 2.

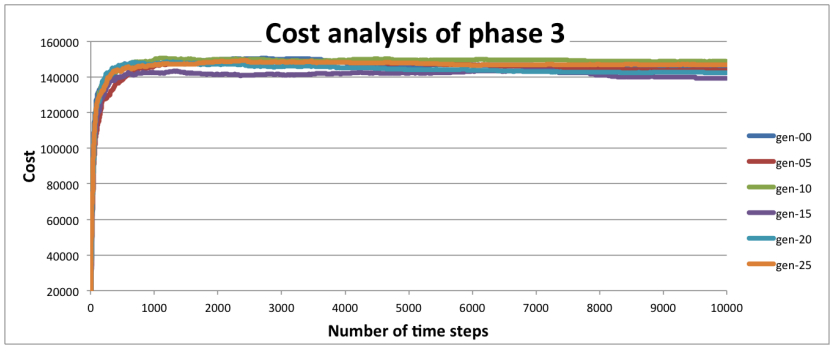


Figure 3.10: Cost analysis for phase 3.

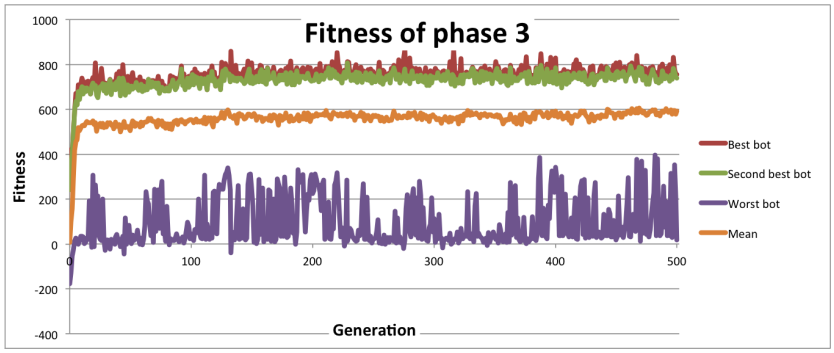


Figure 3.11: Example of fitness evaluation for phase 3.

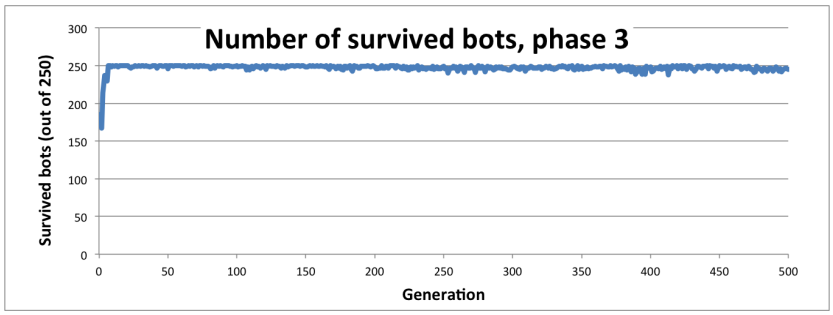


Figure 3.12: Example of plot of the surviving number of bots in phase 3.

3.4 Conclusion and future directions

The main goal of this project was to implement a simulation of an environment in which simple agents, representing animals, would be able to interact and perform different actions. Our approach was to equip each bot with a neural network that would control its behaviour through different sets of inputs and outputs, and to employ a genetic algorithm to evolve the population. The evolution was split into three different phases, devoted to development of different actions such as searching for food or biting.

We successfully managed to simulate different behaviours of the bots: some of these, however, proved to be more challenging that other ones. For example, the development of a search strategy for food in phase 1 turned out to be relatively simple, requiring only a few number of generations and time steps for each generation. When running many different phase 1 simulations, we always found the evolution to be very effective, reaching a 100% survival rate after a few generations (even if different DNA parameters, so different "evolutionary strategies", were obtained). The following phases, which were devoted to the evolution of more complex behaviours, proved to be more challenging. The second phase, focused on the action of biting, delivered results that differed greatly from one simulation to the other and required a greater number of generations (the double of the number in phase 1) and of time steps for each generation. We nevertheless find the results obtained satisfactory, as this task presented greater challenges: for example it was difficult to find the balance between encouraging the bots to bite (by rewarding them through the fitness function) and avoiding the selection of a hyper-aggressive population of bots that would immediately die off. The third phase required different attempts before finding a suitable fitness function for the evolution of the boost action, which is used primarily to escape from a bite but also to reach food faster than usual. The number of generations and time steps required was intermediate between the other two phases, and the final fitness function employed allows the evolution of a population with a very good survival rate. Overall we are very satisfied with the results obtained: the simulation is capable of evolving a population of bots which can perform different tasks as was desired, but at the same time the behaviour of the bots is not completely constrained and we can observe interesting "survival strategies" emerging from the evolution.

Among the possible future directions of this work, there is one analysis in particular which we believe will be very interesting and that we hope to carry out in the future. It is also inspired by the biological starting point of our project: we would like to introduce the mechanism of disease spread through the bites in our simulation and assess whether, in these new environmental conditions, less aggressive bots would be advantaged (as hypothesised in [4] and discussed in the Introduction).

Appendix A

Overview of major changes in the simulation

During the development of this project different issues were encountered and the simulation was modified accordingly in order to deal with them. Other changes were also introduced to improve the simulation and add extra features: we find it instructive to briefly illustrate here the most important ones.

Independent graphics and evolution modes

In the first version of our simulation, the evolution through the genetic algorithm and the graphics were processed at the same time. This, however, was quite inefficient so we decided to separate the two modalities, in order to allow the graphics to be "switched off" during the evolution. Following this change the computational time required for the evolution was significantly reduced.

Different neural networks for different phases

At first, our intent was to employ the same neural network for the different phases: the fully connected network was evolved for the first phase (focused on the search for food), with satisfying results. However, when we proceeded to the second phase (focused on the biting process), we found it very difficult to evolve the new behaviour while also keeping what had been learned in the previous phase. We therefore decided to make a quite drastic change to the structure of the simulation, dividing the neural network in three different modules that are evolved independently and act "in parallel", as illustrated in this report. This modification turned out to be very effective and allowed a much better evolution of the different behaviours.

Inheritable parameters

Initially we set most of the characteristic parameters of the bots, such as the range of view and angle of view, as constants. Once the genetic algorithm was correctly set up we decided to modify the parameters, making them inheritable and thus allowing them to change during the evolution. This increased flexibility allowed us to investigate whether the parameters for the different bots converged

during evolution to an optimal value, or whether they diverged to different values. This also allows interpretation of the parameters as different "survival strategies": for example some bots could evolve a long range of view and other could on the other hand evolve a shorter range for an increased "specificity".

Boost wheels

The boost action was originally dealt with by modifying the same wheel outputs used in the first phase. We lated decided to modify this by introducing a new "set of wheels" (i.e. new outputs) specifically for the boost and this allowed an easier management of this action.

Bibliography

[1] Budinich, M. (2011) *Introduzione alla Teoria delle Reti Neurali.* (Lecture notes for the "Introduzione alla teoria delle reti neurali" course, University of Trieste).

[2] Hamede, R. K., McCallum, H., & Jones, M. (2013). Biting injuries and transmission of Tasmanian devil facial tumour disease. *Journal of Animal Ecology*, 82(1), 182-190.

[3] McCallum, H., Jones (2011). The Tasmanian Devil's Cancer: Could Contagious Tumors Affect Humans? *Scientific American*, June issue.

[4] McCallum, H., Jones, M., Hawkins, C., Hamede, R., Lachish, S., Sinn, D. L., ... & Lazenby, B. (2009). Transmission dynamics of Tasmanian devil facial tumor disease may lead to disease-induced extinction. *Ecology*, 90(12), 3379-3392.

[5] Rao, V. B., & Rao, H. (1995). *C++, neural networks and fuzzy logic.* Mis: Press.

[6] Schrum, J. (2006). *Genetic Algorithms and Neural Networks: The Building Blocks of Artificial Life.* (Doctoral dissertation, Southwestern University).