Artificial Brains as Networks of Computational Building Blocks

Telmo Menezes and Ernesto Costa
Centre for Informatics and Systems
of the University of Coimbra (CISUC)
{telmo, ernesto}@dei.uc.pt

***Abstract*— The latest advances in the gridbrain agent brain model are presented. The gridbrain models brains as networks of computational components. The components are used as building blocks for computation, and provide base functionalities like: input/output, boolean logic, arithmetic, clocks and memory. The multi-grid architecture as a way to process variable sized information from different sensory channels is addressed. We show how an evolutionary multi-agent simulation may use the gridbrain model to emerge behaviors. The Simulation Embedded Genetic Algorithm (SEGA), aimed at continuous multi-agent simulations with no generations is described. An experimental scenario is presented where agents must use information from two different sensory channels and cooperate to destroy moving targets in a continuous physical simulation. Results are analysed and synchronization mechanism are shown to emerge.**

I. Introduction

The growing interest in multi-agent simulations, influenced by the advances in fields like complex adaptive systems and artificial life is related to a modern direction in computational intelligence research. Instead of building isolated artificial intelligence systems from the top-down, this new approach attempts to design systems where a population of agents and the environment interact and adaptation processes take place. As proposed by Holland [1], intelligence can be seen as the internal model of agents. In adaptive systems, this internal model can develop as the agent adapts to the environment. In nature, adaption encompasses varied processes, from *neo Darwinian* evolution to learning in the brain or in the immune system. We will focus on adaptation based on evolution, which may be considered the fundamental driving force of complexification in Nature.

We present for the first time the latest advances in the *gridbrain* [8], [10], a model that attempts to address several important limitations of current artificial brains used in evolutionary multi-agent systems. Two main classes of models are in use nowadays: symbolic approaches like production rule systems [2], [1], [6] or decision trees [3] and artificial neural networks [4], [5], [6]. Evolutionary systems based on *IF/THEN* rules tend to lead to simple, reactive agents. They can be very effective in developing models to abstract and test ideas about biological, social or other systems, but they are limiting when it comes to evolving more complex computational intelligence. Artificial neural networks are inspired in biological nervous systems. A multitude of algorithms exist for both learning and evolution of ANNs, with many successful implementations. Recurrent neural networks can be shown to be Turing-complete [7] and are theoretically capable of complex computations. It is important to note, however, that biological neural networks are analogic and highly parallel systems, while modern computers are digital and sequential devices. The implementation of artificial neural networks on digital computers demands for a significant simplification of the biological models. In neural networks, neurons are the building blocks. We believe that for the purpose of evolving artificial brains in multi-agent simulations, it is interesting to experiment with computational building blocks that are a more natural fit to von Neumann's modern digital computer model. We deconstruct the von Neumann machine [11] into a set of computational building blocks that fall into the categories of input/output, boolean logic, math operations, memory and clocks. This way we expect to facilitate the evolution of systems that take advantage of the processing capabilities of the computer, and more easily develop behaviors that require memory and synchronization.

Another limitation of agent models in current evolutionary multi-agent simulation is the sensory system. Many such simulations use simple 2D grid models where an agent is only capable of perceiving one other world entity per simulation cycle. As we move towards continuous simulations and more sophisticated sensors like vision or audition, we become confronted with the problem of dealing with multiple-object perceptions per simulation cycle. A common approach to this problem is to pre-design a layer of translation for the agent's brain. In rule systems this can be done by defining input variables like *number_of_visible_food_items* or actions like *go_to_nearest_food_item*. In artificial neural networks, it is common to define fixed sequences of input neurons, sometimes called radars, that fire according to the distribution of a certain type of entity in the agent's vision range. These radars are predefined for certain types or properties of world objects. Predefinition of sensory translations limit the range of behaviors that the agent may evolve. In the architecture we present, this problem is addressed by dividing the brain in sensory layers (alpha grids) and a decision layer (beta grid). In a way loosely inspired by animal brains, a layer exists for each sensory channel (ex: vision, audition, self state). In a brain cycle, alpha grids first evaluates each of the objects perceived by their sensory channel at the moment and extracts general information, which is then transmitted to the beta grid, that then fires actions. It is our goal to create systems where the perception layers can evolve with a great degree of freedom.

In this paper we present the latest version of the *gridbrain* agent brain model, including a set of computational

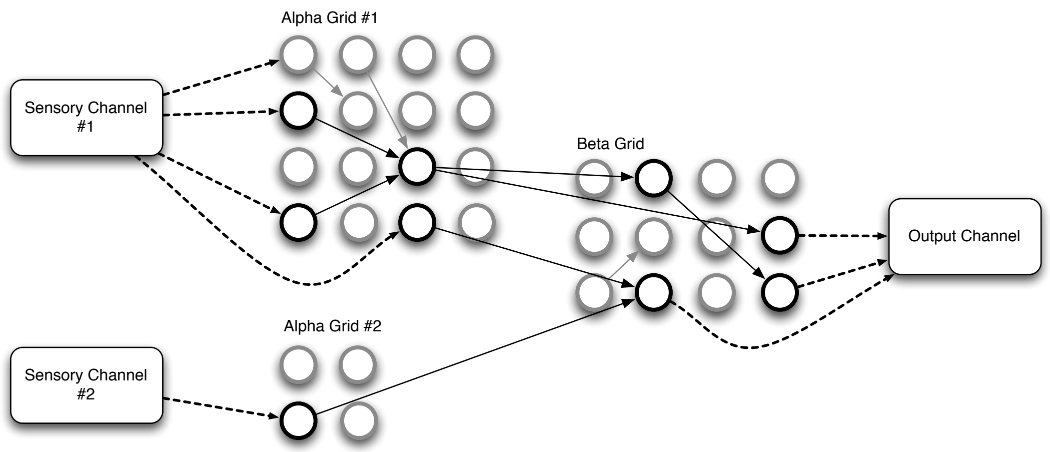


Fig. 1. Gridbrain computational model.

components and genetic operators to be used in evolutionary environments. Following that we describe the *Simulation Embedded Genetic Algorithm (SEGA)*, developed for continuous multi-agent simulations with no generations. We then present experimental results, where gridbrains are evolved in the context of a multi-agent simulations, in a scenario where they must use information from two different sensory channels and cooperate to shoot moving targets. We end with some final remarks.

II. The Gridbrain

The gridbrain is a virtual machine designed to serve as a brain for an autonomous agent. It belongs to the family of genetic programming, with some similarities to Parallel Distributed Genetic Programming [13], [14] and Cartesian Genetic Programming [15], [16].

As can be seen in figure 1, it consists of a network of computational components placed on rectangular grids. There are two types of grid: alpha and beta. Alpha grids are associated with sensory channels and are responsible for processing perceptual information. The beta grid receives inputs from the alpha grids and outputs decisions. A gridbrain can have any number of alpha grids (one of each sensory channel), but only one beta grid. This architecture is inspired on the organization of animal brains, which have areas for processing sensory information, and others for decision making, planning and muscle control. It is not intended to be an accurate model of such brains, but only an application of the concept of dedicated layers in a network of components. Furthermore, it is not intended to be a reductionist model with rigid hierarchies. The only limitation imposed is that sensory information is fed to the alpha grids and output is provided by the beta grid. As we will detail later, this allows the gridbrain to deal with variable sized perceptual information, while the evolutionary process remains free to test a diversity of structures.

Connections between components represent flows of information. A connection from component A to component B means that, during each computation cycle, the output value of component A is fed as an input to component B. The information propagated takes the form of unbounded floating point values. Two types of connections are possible: feed-forward connections inside a grid and connections from any component in an alpha grid to any component in a beta grid. In neural networks, recurrent connections allow the system to keep information about the past. Feed-forward neural networks model purely reactive agents. In the gridbrain, we provide the system with explicit memory mechanisms. Components may be able to conserve their state across computation cycles. We thus chose to only allow feed-forward connections in the models studied in this work for simplicity.

In each gridbrain cycle, the outputs of components are computed in order. We define a coordinate system where each component position is identified by a tuple (x, y, g), x being the column number, y the row number and g the grid number. Components are processed from the first to the last column, and inside each column from the first to the last row. The feed-forward restriction is imposed by only allowing connections inside the same grid to target components with an higher column number than the origin. The rectangular shape of the grids facilitates the definition of both parallel and sequential computational processes.

Both components and connection may either active or inactive. In figure 1, active components and connections are represented in black, inactive ones are grey. Active status is not explicitly encoded in the gridbrain, but derived from the network configuration. To explain how this works, we must first introduce the concept of *producer* and *consumer* components. *Producer components* are the ones that introduce information into the gridbrain, while *consumers components* are the ones that send information from the gridbrain to the outside. *Input components* are the alpha grid components associated with sensory information. They are updated with current sensory data in the beginning of each alpha grid computation cycle. Input components are producers. Other components that output values different than 0 without any input present are also considered producers. Examples of this are a random value component, that outputs a random value or a component that outputs 1 if the sum of its inputs is 0. *Output components* are the beta grid components from which decision information is extracted after a gridbrain compoutation cycle. In an agent environment, they are associated with triggering actions. Output components are consumers. An active path is a sequence of connections that links a producer component to a consumer component. A connection is considered active if it belongs to an active path. A component is considered active if it has at least one ingoing or outgoing active connection.

Only active components and connections influence the computation performed by the gridbrain. From a genetic perspective, we can say that active elements are the ones that have phenotypical expression, while inactive elements can be seen as analogous to junk DNA. As we will show later, inactive elements are valuable to the evolutionary process.

In a computation cycle, the sequences are executed for each grid. There are two evaluation stage, alpha and beta. In the first stage alpha grids are evaluated, once for each entity in the sensory channel they are associated with. In the second stage the beta grid is evaluated once. Alpha grid evaluation consists of two passes. This is done so that certain alpha grid components, which we call *aggregators*, have the chance of calculating their output based on information about the entire set of entities. An example of this is a maximizer component that outputs 1 if the current value inputed is the maximum for the set of entities present, 0 otherwise. For this to be possible, a first pass is performed on the grid where only intra-grid connections are active. In this phase, aggregators can compute their internal state so that they produce the correct outputs on the second pass. On the second pass, inter-grid connections are also active so that information can be propagated to the beta grid. In the example of the maximizer, the first pass is used to determine the maximum value, while the second is used to signal this value when it is found.

A. Component Model

Components are information processing units. They are the computational building blocks of gridbrains. In the generic gridbrain model does not define a specific set of components. Much like machine code instructions, components belong to classes of functionalities: input/output, arithmetic, boolean logic, information aggregation, synchronization and memory. Component sets may be conceived for different environments and problem domains. However, we will propose a set of components to be used in physically simulated worlds. These were applied in the experimentation presented in latter chapters.

Components have an arbitrary number of inputs and outputs, to allow for a high degree of freedom in network topologies. This contrasts with conventional genetic programming systems, where functions have fixed numbers of parameters that must be respected for the resulting programs to be valid. We strive for a connectionist model closer to natural

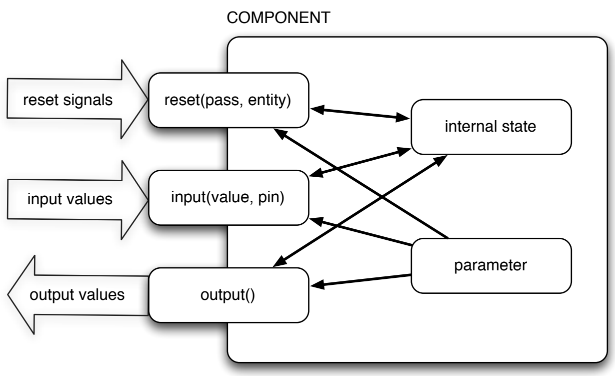


Fig. 2. Component model.

brains. To achieve this, we developed the component model presented in figure 2.

The gridbrain design follows an object oriented approach, where a base abstract component class defines a set of interfaces for interaction. Specific components are defined by inheriting from this base class and defining the internal mechanisms. This way we can have gridbrains made up of an heterogeneous mix of component, and yet allow the evolutionary process and computation cycle algorithm to treat these components as black boxes with a fixed set of external interfaces.

A component has three interfaces: *reset*, *input* and *output*. The reset interface is used to signal the component that a new grid evaluation has started. The input interface is used to feed a value from downstream components, and the output interface is used to produce a value to be fed to upstream components.

Components have internal states. The data structure that maintains the state is defined by each specific component. We classify components as *operators*, *aggregators* or *memories* according to the persistence of the state.

Operators are information processing components that perform, for example, arithmetic or boolean operations. Their output is determined only by information present in the current grid pass. Aggregators extract general information from a set of entities present in the sensory channel of an alpha grid, and their output is determined by all the values received during an alpha stage. They could, for example, provide minimum, maximum or average values for that stage. Memories conserve their state across computation cycles and provide the gridbrain with information about the past.

Components also have a parameter value, which is a floating point number in $[0, 1]$. Components may use this value to adjust their behavior. For example, an amplifier component can use the parameter to determine its amplification factor or a clock component may use the parameter to determine its ticking frequency.

A gridbrain computation cycle is performed each time we want to feed current sensory information and obtain a decision. Input components have a special $input\_type$ field that is used to identify the type of information from the sensory channel they are associated with. Likewise, output components have an *output_type* field that identifies the type of output information they are associated with. In a typical use of the gridbrain as an autonomous agent controller, each output type is associated with an action that the agent can perform.

B. A Component Set

In this section we present the component set we use in the experimentation presented in this paper. As stated before, the gridbrain model does not specify a component set, but only an abstract component model. Components may be tailored to specific applications and environments. The components presented resulted from experimentation and a trial and error process, where we attempted to evolve gridbrains in a environments. We did however try to create general purpose components, and expect the ones presented to have broad application.

Input and output of values in the gridbrains is done by way of the IN and OUT components. Previously we showed that alpha grids are associated with sensory channel processing and the beta grid with decision making. Alpha grids are where sensory information enters the system and the beta grid is where decision values are output. This way, IN components are only to be places on alpha grid sets and OUT components on beta grid sets. Both IN and OUT contain a floating point value state. In both components, the rest interface changes this state to 0 and the output interface returns the current state value. The input interface of IN writes a value received to the internal state, while the input interface of OUT writes a value received that is different from zero to the internal state.

The two boolean logic components are NOT and AND. Both are operators with a floating point value internal state. These components treat their input values as boolean values. They define a threshold value t to determine if the input is considered on or off. If $|input| \leq t$ it is considered off, otherwise on. The NOT component reset interface sets the internal value to 1. The input interface checks if the module of the input value is greater then the threshold, in which case it changes the internal value to 0. The output interface returns the internal state. This way, the NOT component returns 1 if none of its inputs is on. The AND component has a boolean flag in its internal state, additionally to the floating point value. This flag signals if the current input is the first one in the grid evaluation. The rest value initializes the flag to true and the floating point value to 0. The input interface checks if the module of the input value is greater that the threshold. If it is not, it multiplies the current state by 0, otherwise it multiplies it by the input value if the first input flag is false, or just stores the input value in the state if it is true. The first input flag is set to false after an input value is processed. The AND component can return -1, 1 or 0. We provided it with the ability to return the signal of the product of the input values, as this can be useful in sensory information with symmetry properties, as usually found in continuous physical environments. The output interface returns 1 if $state > 0$, -1 if $state < 0$ and 0 otherwise.

Arithmetic components can also be used as logic gates, albeit without the threshold. In fact, we decided to not include an OR component, as several of those components can be used as such. The AND component with a single input connection can be seen as a converter from continuous to boolean values. We believe the combination of the boolean components presented with the arithmetic ones is sufficient for complete boolean logic.

The ten arithmetic components are: SUM, MUL, INV, NEG, MOD, AMP, RAND, EQ, GTZ and ZERO. These operators have a floating point value internal state, except for RAND. RAND outputs an uniformly distributed floating point value in the $[0, 1]$ interval, regardless of its inputs, and is a source of randomness.

The MUL component computes the product of all its inputs. The EQ component outputs 1 if all of its inputs are equal, 0 otherwise.

All other components have the same input interface: the input value is summed to the internal state. Their state is always set to 0 by the reset interface, before a grid evaluation.

SUM outputs the value stored in the state, thus providing the summation of its inputs. NEG outputs the negative: $-state$. MOD outputs the module: $|state|$. GTZ returns 1 if $state > 0$, 0 otherwise. ZERO returns 1 if $state = 0$, 0 otherwise. INV truncates the internal state value to $[-1, 1]$ and then outputs $-1 - state$ if $state < 0$, $1 - state$ if $state > 0$. AMP operates as an amplifier. It outputs the product of its internal state by an amplification factor, which is determined by its parameter.

NOT, ZERO and RAND are considered producers, because they can output a value different from 0 when not receiving any input.

As defined, aggregator components conserve an internal state across the alpha grid evaluations of a gridbrain computation cycle. They serve the purpose of extracting information from the entire set of entities in a sensory channel. Aggregators are of no use in beta grids, so they should only be included in alpha grid component set. The aggregator components we describe here are MAX, MIN and AVG.

Aggregator computations have two phases, related to the two passes in alpha grid evaluations. In the first phase, an internal state is computed from the entire set of entities in the sensory channel of the containing grid. In the second phase, this internal state is used to produce an output.

In the first phase, the MAX component finds the maximum value different from 0 produced by its inputs, across the alpha grid evaluations in a gridbrain computation cycle. In the second phase, a value of 1 is output when it is again found. In a gridbrain computation cycle, the MAX component outputs 1 during the alpha grid evaluation linked to the entity that produced this maximum, and 0 in other evaluations. If more than one entity produce a maximum, the 1 value is output only for the first occurrence. The MAX component is thus used to signal a unique maximum in alpha grid evaluations.

The MIN component operates in a similar fashion to MAX, except that it signals the minimum value found. The AVG component computes the average of its inputs with values different from 0, across the alpha grid evaluations.

The components that are able to conserve a state across the lifespan of the gridbrain are: MEM, SEL, DMUL, CLK and TMEM. These components are used for persistent memory and synchronization.

The MEM component is a simple memory cell. Its persistent internal state is a floating point variable which we will call *memory state*. The memory state is initialized to zero when the component is created. Inputs are written to the memory state if the state is 0. The memory state is always output.

The CLK component is a clock device, producing a 1 signal at fixed intervals, otherwise outputting 0. The interval between ticks is a number of simulation cycles determines by the component parameter. It is given by the expression $I(p) = p.I_{max}$, where p is the parameter value and I_{max} is the maximum possible value for the period. I_{max} is a pre-configured value relative to the environment the gridbrain is operating on. In a multi agent simulation, it is a good choice to make I_{max} equal to the maximum lifespan of the agent. A clock may be synchronized by way of its input connections. A clock is forced to fire and restart its firing interval if its input state changes from 0 to another value.

The TMEM component is a temporary memory. It combines the functionalities of the MEM and the CLK components. It keeps a memory state like the MEM component, and has a periodic triggering mechanism like the CLK component. It always outputs the current value stored in the memory state. When the clock mechanism triggers, the memory state is set to 0. It is a memory cell with the capacity of *forgetting*.

The DMUL component is a delayed multiplier. It waits until all of its input connections have produced a value different from 0, and then outputs the product of the last value different from 0 received from each connection. Its internal state consists of an array of float values, with one value for each input connection the component has. When the component is created, the array is initialized to 0 values. The input interface checks if the input value is not 0. If so, this value is written to the corresponding position in the array. The output interface verifies the values in the array. If all of them are not 0, the output is the product of these values and the array is reset to 0 values. If at least one of the values in the array is 0, the output is 0.

The SEL component is an entity selector for alpha grids. It selects one entity present in the sensory channel of their containing grid, and keeps producing outputs only during the grid evaluation corresponding to this entity, while it is present. When it is no longer present, it selects a new one. Only entities for which the input state is not 0 are considered for selection. The selector component allows the gridbrain to keep track of a specific entity, using its inputs as a filtering condition. It only makes sense to use this component in alpha grids.

C. Genetic Operators

We must provide a set of genetic operators that can be used by an evolutionary algorithm to produce replication with change. In the following section we will propose an evolutionary algorithm suitable for continuous multi agent simulations that takes advantage of these operators. We provide two types of genetic operators that are usual in evolutionary computation systems: mutation and recombination. We also provide a formating operator that deals with adapting the shape of the grids as evolution progresses and is related to the complexification of gridbrains.

We define mutation operators at connection level and component level.

There are two pairs of connection level operators: *add/remove* and *split/join*. The operators in each pair are symmetrical, one performing the inverse operation of the other.

The add operator inserts a new valid connection and the remove operator deletes an existing connection from the gridbrain. These mutations occur with respective probabilities of p_a and p_r. These probabilities are relative to the number of connections in the gridbrain. For each existing connection there is a p_a probability that a new one is generated. Each existing connection has a p_r probability of removal. Multiple connections may be added or removed in the same mutation step. This defines a process that adapts to the size of the gridbrain. The number of mutations tends to increase as the gridbrain size increases, but the probability per connection remains the same. Also, if $p_a = p_r$, and disregarding evolutionary pressure, the number of connections will tend to remain stable. This is part of the measures we take to avoid bloat [12].

The split operator routes an existing connection through an intermediary component in the grid. If we have a connection from component A to component B, two new connections will be created, from A to the intermediary component C and from C to B. The original connection is removed. If the B or C connection already existed, their respective add operation is just ignored. If A and B are on the same grid, the component C must be in a column with a number higher than A and lower than B. If they are on different grids, C must be in either the origin grid with a column number higher than A or in the target with a column number lower than B.

Splits and joins occur with respective probabilities of p_s and p_j. Again, these probabilities are per existing connection. A join will only take place if the target component of the selected connection has an outgoing connection with an equal t_c. Due to the way the split/join operators work, only one connection at most will meet this condition. The join will be performed to the selected connection and the one that meets the condition. With $p_s = p_j$ and no evolutionary pressure, the number of connections will tend to remain stable. Notice that when a connection is split, only the first resulting connections will be eligible for the symmetrical join, so the overall probabilities remain balanced.

There are two component level operators: *change component* and *change parameter*. Change component replaces an existing component with a new one. The new component is randomly selected from the grid component set, and its parameter is initialized with a random value extracted from a uniform distribution in $[0, 1]$. These mutation occur with a probability of p_c per component in the gridbrain. A variation of this operator is *change inactive component*, which works the same way but only affects inactive components. This variation is less destructive as it only produces neutral mutations, which are mutations that do not affect the phenotype.

The *change parameter* operator alters the parameter of a component by adding a value x to it. The resulting value is truncated to the $[0, 1]$ interval. The value x is randomly generated from a normal distribution $x \sim N(\mu, \delta^2)$, where the average, μ is 0. The operator may be parameterized by the standard deviation, δ, as well as the probability to occur per component, p_p. An higher standard deviation will produce larger changes in parameters.

A viable recombination operator for the gridbrain has to deal with the recombination of differently shaped grids, containing networks with different topologies. Furthermore, there are different types of components, so the network nodes are heterogeneous.

The operator we present performs the following steps:

- Create the child gridbrain, with grids the same size and with the same rows and columns as parent A;
- Recombine equivalent connections from parents A and B into child;
- Recombine components from parents A and B into child.

When recombining connections, we start with parent A and iterate through its connection set. For each equivalent connection we find, we check if it is present in parent B. If it is, we import the group from one of the parents with equal probability. If it is not, the group may be imported or discarded, with equal probability. Then we iterate through the connection set of parent B. Again we check for each connection if it has an equivalent on the other parent. If it does, we ignore it, has it was already recombined. If it does not, the same random process of importing or discarding is used, as with the connection groups that only exist on parent B.

The last step is to recombine components. For each component in the child, we check to see if a component in the equivalent position (given by its column and row IDs) exists in each parent. If it exists in both parents, one of them is chosen with equal probability to have the component copied from. If it exists in only one parent, the component is copied from it. If it exists in neither parent, a new component is randomly selected from the grid connection set. A component in a child may have no equivalent component in any of the parents if it is situated in an intersection of a column that only exists in one of the parents with a row that only exists on the other.

The mechanism used to identify which connections are equivalent is *connection tags*. It consists of assigning three integer values to each connection. There is one value for the connection, one for the origin an one for the target. We use three values instead of just one because of the split/join mutation operator.

Connection tags are (t_c, t_o, t_t) tuples, where t_c is the connection value, t_o is the origin value and t_t is the target value. When a connection is split, both new connections inherit the t_c value from the original connection. The first new connection inherits the t_o value and the last new connection inherits t_t. A new tag is generated, and is assigned to t_t of the first connection and t_o of the second. This way, the origin and target components of the original connection remain equivalent for the purpose of generating new connection tags. Also, it can be determined if two connections originated from the same connection via split, by checking if they share the same t_c value. The reverse operation, join, is performed only to adjacent connections with equal t_c values. A split followed by a join of the resulting connections will result in the original connection, with the original connection tags.

Two connections are equivalent if their tag values are equal. During the evolutionary process, when a connection is passed from the parent to a child, its connection tags are also passed. When a new connection is created, tags have to be assigned to it. In this case there are two alternatives: either we find an equivalent connection in the population and copy its connection tags to the new one, or we generate new values. New values are generated by simply incrementing a global variable in the systems that holds the last value assigned.

This process requires that when a gridbrain with new connections is added to the population, these new connections are compared against all connections in all existing gridbrains to attempt to find an equivalent. A connection with unassigned tags is considered equivalent to another one if, from the connection network perspective, both its origins and targets are equivalent. Two origins are equivalent from a connection network perspective if the following conditions are met:

- Origin component are equal;
- Origin components both have no incoming connections or they share at least one equivalent incoming connection.

In the same vein, two targets are equivalent if:

- Target component are equal;
- Target components both have no outgoing connections or they share at least one equivalent outgoing connection.

These rules define a constructive mechanism that ties in with the progressive complexification of the gridbrains in the evolutionary process. They allow us to match connections that create the same functionality.

This recombination operator always produces valid gridbrains and is able to recombine functionalities in a meaningful way. In the absence of evolutionary pressure, it does not introduce any probabilistic bias towards the increase or decrease in the total number of connections.

Formating is an operator that adapts the shape of the grids according to the network contained in the gridbrain. It is of a non-stochastic nature and can be seen as an adaptation mechanism. The changes it performs do not affect the phenotypical expression of the individual. The purpose of formating is to regulate the search space of the evolutionary process. It is part of the complexification process. We attempt to create systems that are initialized with empty brains, and that undergo an increase in complexity as higher quality solutions are found. Solutions are constructed through iterated tweaking, in a similar fashion to what happens in nature. Our goal is to have the size and complexity of the gridbrains to be determines by the demands of the environment.

Formating operates at grid level and performs changes taking into account the current *active network*, which is the set of active connections. For each grid, this operator determines if rows and columns should be added, removed or kept.

Under the genetic operator framework we are describing, complexification is driven by connection level mutations. Add/remove and split/join are the operations that directly affect network complexity. Formating alter the shape of the grid in such a way that the active network is kept unaltered, but the following mutation step has freedom to complexify the network in all the manners available to connection level operators. This means that the following should be possible:

1) Add a new valid connection between any of the active components;
2) Add an intra-grid incoming connection to any active component;
3) Add an intra-grid outgoing connection to any active component;
4) Branch the output of any active component to an inactive component in any of the following columns;
5) Split any of the active connections.

The first condition is guaranteed by not altering the active network. The second condition is met by having no active components in the first column of the grid, and the third by having no active components in the last one. The fourth condition is met by having at least one inactive component in all rows and the last condition is met by having all active connections skip at least one column. Formating alters the grid by inserting and deleting rows and columns, making sure that all these conditions are met with the smallest possible grid, without changing the active network.

If new rows or columns are created in a formating step, the new components are selected randomly from the grid component set. Furthermore, if a grid is null, meaning it has no columns or rows, a row and column is always created so that the evolutionary process can move forward. In fact, in the experiments we will describe, we just initialize the gridbrains with null grids.

The format operator can work at grid level and still enforce the restrictions stated above for the entire gridbrain. Maintaining inactive columns at the beginning and ending of all grids guarantees that any inter-grid connection is splittable, and that it can be split by a component in the origin or target grids.

The format operator should be applied before the mutation step in the generation of a new gridbrain. This way the gridbrain is formated according to the above mentioned rules when connection level mutations are performed. The sequence of genetic operations when generating a gridbrain is $recombine(parent1, parent2) \rightarrow format \rightarrow mutate$ if recombination is being used, or $clone(parent) \rightarrow format \rightarrow mutate$ for a single parent reproduction. We are considering the $mutate$ operation to be the combined application of all the mutation operators described above, according to their respective probabilities.

III. Simulation Embedded Genetic Algorithm

The *Simulation Embedded Genetic Algorithm* (SEGA) is a steady-state genetic algorithm that we developed for the purpose of evolving agents in continuous simulations without generations. It allows the seamless integration of an evolutionary process with a multi-agent simulation. New individuals may be generated at any moment and on demand.

SEGA keeps a buffer of individuals for each species in the simulation. Each time an individual dies or is removed from the population, its fitness is compared to the fitness of a random individual in the buffer. If the fitness is equal or greater, the recently removed individual replaces the old one in the buffer, otherwise the fitness of the old individual is updated using the expression:

$$f_{new} = f_{old} \cdot (1 - a)$$

where a is the *fitness ageing factor*.

The purpose of fitness ageing is to maintain diversity in the buffer. Individuals that make it to the buffer have a chance of producing offspring. The higher their fitness, the more offspring they are likely to produce, but eventually they will be replaced, even if it is by a lower fitness individual. Fitness ageing takes place when an individual in the buffer is challenged by an individual removed from the population, so it adapts to the current rate of individual removal, which can be variable and unpredictable.

When the simulation requests a new individual, two elements in the buffer are selected at random and recombined if recombination is to be done, or one is selected at random and cloned for simple mutation. Mutation is applied, followed by formating, and the offspring is then placed in the simulation environment. A probability of recombination, p_{rec}, is defined. This probability is used to randomly decide in each individual request if recombination is to be applied.

In simulations with fixed populations, as the one presented in this paper, agent removal is always followed by the creation of a new agent.

IV. Experimental Setup

Experimentation is done with LabLOVE [9], a tool that we developed for our research and that is available to the

Vision Input	Description
Position	Position of the object relative to the agent's vision field. -1 is the farthest possible to the left and 1 is the farthest possible to the right.
Distance	Distance from the object, divided by its view range, resulting in a value in $[0, 1]$.
Target	1 is object is on target, 0 otherwise.
Line of Fire	1 is currently on the target of this object, 0 otherwise.
Color	Distance between agent's color and this object's color.
Sound Input	**Description**
Position	Position of sound origin relative to agent.
Distance	Distance of sound source, divided by sound range.
Action	**Description**
Go	Apply forward force to agent.
Rotate	Apply rotation torque to agent.
Fire	Fire laser.

TABLE I
SENSORY INPUTS AND ACTIONS USED IN THE EXPERIMENT.

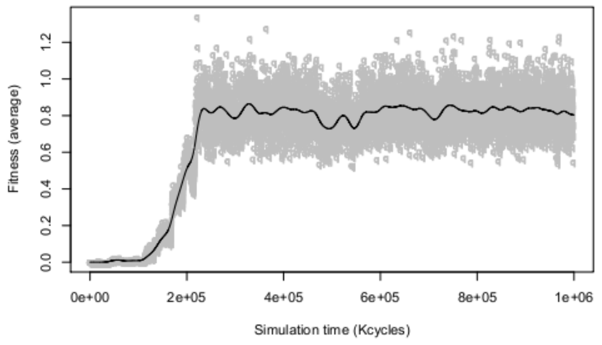
community as open source. LabLOVE includes an implementation of the Gridbrain, the SEGA algorithm and a physically simulated multi-agent environment, as well as real-time visualization and data recording modules.

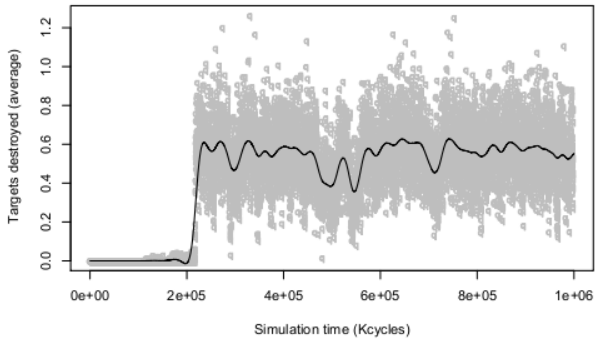
We define a videogame inspired scenario where agents are evolved to shoot moving targets. Agents have the ability to move in a 2D physically simulated world and shoot. They have two sensory channels: vision and audition. Vision provides them with information about objects in the environment, including other agents and the targets. Audition allows them to perceive sounds emitted by other agents shooting. Each agent's laser is not sufficiently strong to destroy a target, so they must cooperate. The effectiveness of a laser shot is related to the amount a time an agent spends aiming at a target. A simulation parameter named *laser_interval* (l_i) establishes the number of simulation cycles that an agent must spend aiming at a target for it to have full effect. The amount of damage that a target suffers from a shot is determined by the expression $d = l_i \cdot l_t \cdot d_{max}$, where d is the damage, l_t is the time the agent spent aiming at that object and d_{max} is the maximum damage that a shot can cause. We set d_{max} to 1 and the amount of damage needed to destroy a target to 1.1, so that only at least two simultaneous shots can destroy the target.

Targets are moving at a constant speed, bouncing when they hit a limit of the world. Agents must follow the target's movement to produce effective shots.

Gridbrains are configured to have three grids: one alpha grid to process vision sensory information, another alpha grid to process audition sensory information and the beta grid. Additionally to the components described in section II-B, input and output components of the types detailed in table I are included in the respective grids component sets.

The input value of the action component, i determines the intensity with which the action is triggered. The force applied in go and rotate actions, as well as the intensity of the laser shot is proportional to i. The energy costs of the actions are as follow: $0.01 \cdot i$ for go and rotate and $0.1 \cdot i$ for fire. Agents





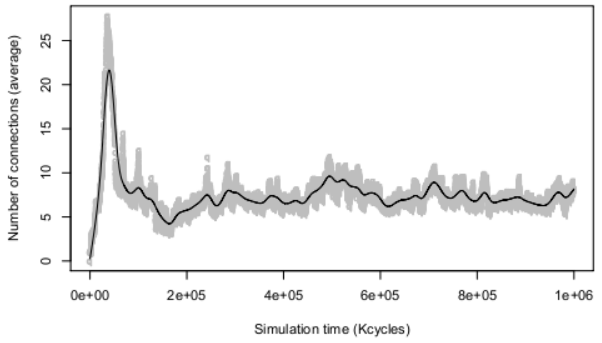


Fig. 3. Evolution of metrics in a simulation run.

are initialized to having an energy of 1.

Every time an agent shoots at a target, a score is calculated by multiplying the damage caused by this shot with the number of simultaneous successful shots from other agents. We define the fitness function as the best score obtained during the agent's lifetime. This encourages the agents to evolve mechanisms to produce more effective shots, but also to synchronize their shooting with the other agents.

Evolutionary parameters are configured as following: add/remove connection probability is set to $p_a = p_r = 0.01$; split/join connections probability is set to $p_s = p_j = 0.01$; the change inactive component operator is used, with $p_c = 0.2$; change parameter is set to $p_p = 0.01, \delta = 1.0$;

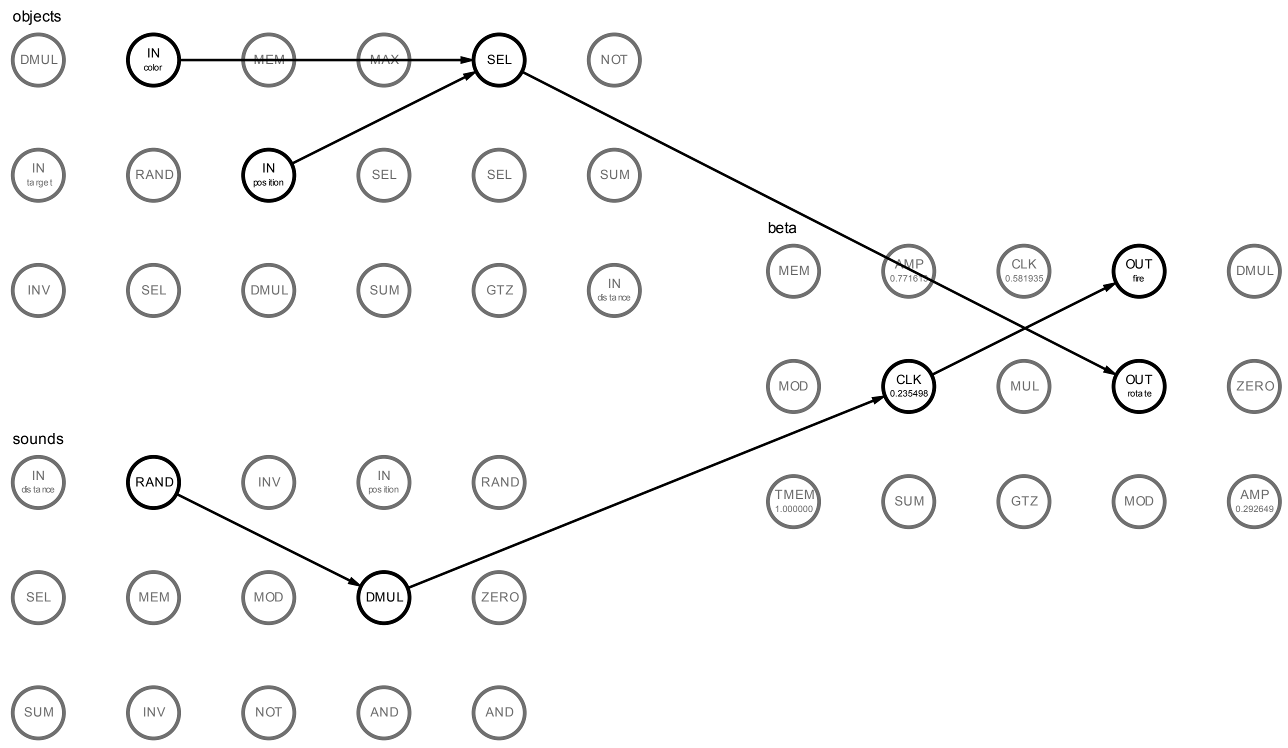


Fig. 4. An evolved gridbrain.

recombination probability is set to $p_{rec} = 0.25$; SEGA buffer size is set to $s_{buf} = 100$ and the fitness ageing factor is set to $a = 0.5$. These values where found to be good choices by previous benchmarking.

The simulation is defined to have 10 agents and 5 targets. Agents have a random maximum lifespan of 9.5 to 10.5 Kcycles. They can die because this lifespan is reached, or by expending to much energy or being shot. The complete experiment definition can be found in the *targets.lua* file, that is in the *experiments* directory of the LabLOVE release.

In figure 3 we can observe the evolution of several metrics during 10^6 Kcycles of simulation. These metrics are extracted by calculating the respective average value for dead agents during 100Kcycles. The first one is the final fitness value, the second is the number of targets that the agent collaborated in destroying and the third is the number of connections in the agent brain. As can be seen, the increase in fitness successfully drives the agents to develop the ability to destroy targets.

The number of connections in gridbrains is used as a simple complexity metric. As can be observed, there is a spike in this value in the begining of the run, followed by a sharp decrease in size. The mechanism propose appear to show the ability to adapt gridbrain sizes to the demands of the environment. As fitness stabilizes, so does the number of connections.

In figure 4, we show a sample gridbrain extracted from an agent dead in the last 100 Kcycle time interval of the simulation run. Two distinct mechanism have emerged: one to select and follow a target, and another to fire at regular intervals, synchronizing this action with the other agents. The first mechanism uses the SEL component to select an object amongst visible targets and use its relative position to rotate in its direction. The second feeds a sound perception to a clock component, this synchronizing this clock with shooting sounds. This mechanism evolved a clock parameter that produces ticking intervals very close to the laser interval defined in the simulation, which is clearly a good strategy for shooting efficiency.

V. Final Remarks

The experimental results obtained indicate the gridbrain to be a promising approach for emerging behaviors in evolutionary multi-agent simulation. The evolutionary process was able to take advantage of the computational building blocks to create mechanisms of synchronization, temporal-based behavior and processing of sensory information from multiple channels and a variable number of entities. We believe that the evolution of these mechanisms would be problematic for more traditional approaches, like rules systems and artificial neural networks.

We expect out work to have application in the development of more sophisticated scientific multi-agent simulations. Since the gridbrain is well suited for continuous, real-time environments, we also believe that interesting engineering applications exist - for example in the development of decentralized robot swarms, as well as agents for video games and virtual reality worlds.

Acknowledgments

The first author would like to acknowledge grant SFRH/BD/19863/2004 from *Fundação para a Ciência e Tecnologia (FCT)*, Portugal.

References

[1] Holland, J. H., *Hidden Order - How Adaptation Builds Complexity*, Addison-Wesley, 1995.

[2] Holland, J.H. and Holyoak and K.J. and Nisbett, R.E. and Thagard, P., *Induction: Processes of Inference, Learning, and Discovery*. Cambridge, MA: MIT Press, 1986.

[3] Eiben, A. E. and Griffioen, A. R. and Haasdijk, E., *Population-based Adaptive Systems: concepts, issues, and the platform NEW TIES*. In Proc. of the ECCS07 - European Conference on Complex Systems, Dresden, Germany, 2007.

[4] Stanley, K. O. and Bryant, B. D., and Miikkulainen, R., *Evolving neural network agents in the nero video game*. In Proceedings of the IEEE 2005 Symposium on Computational Intelligence and Games, 2005.

[5] Yaeger, L.S., & Sporns, O., *Evolution of Neural Structure and Complexity in a Computational Ecology*. In: Rocha, L. et al (ed), Artificial Life X, Cambridge, MA: MIT Press, 2006

[6] Adamatzky, A. and Komosinski, M., *Artificial Life Models in Software*. Springer, 2005

[7] Hyotyniemi, H., *Turing machines are recurrent neural networks*. In Proceedings of STeP'96, Finnish Artificial Intelligence Society, pp. 13–24, 1996.

[8] Menezes, T. and Costa, E., *The gridbrain: an heterogeneous network for open evolution in 3d environments*. In Proc. of the The First IEEE Symposium on Artificial Life, Honolulu, USA, 2007.

[9] Menezes, T. and Costa, E., *Modeling evolvable brains - an heterogeneous network approach*. International Journal of Information Technology and Inteligent Computing, 2(2), 2008.

[10] Menezes, T., *Lablove - laboratory of life on a virtual environment*. http://sourceforge.net/projects/lablove, 2007.

[11] von Neumann, J, *First Draft of a Report on the EDVAC*. technical report, Moore School of Electrical Engineering, University of Pennsylvania, 1945.

[12] Langdon, W. B., *The evolution of size in variable length representations*. In 1998 IEEE International Conference on Evolutionary Computation, pages 633–638, Anchorage, Alaska, USA, IEEE Press, 1998

[13] Poli, R. 1996. *Parallel Distributed Genetic Programming*. technical report CSRP-96-15. The University of Birmingham, UK.

[14] Poli, R. 1999. Parallel distributed genetic programming. *Chap. 27, pages 403–431 of:* D. Corne, et al. (ed), *Optimization, Advanced Topics in Computer Science*. Maidenhead, Berkshire, England: McGraw-Hill.

[15] Miller, J. F. 1999. An Empirical Study of the Efficiency of Learning Boolean Functions using a Cartesian Genetic Programming Approach. *Pages 1135–1142 of: GECCO 1999: Proceedings of the Genetic and Evolutionary Computation Conference*. Orlando, Florida: Morgan Kaufmann, San Francisco.

[16] Miller, J. F., & Thomson, P. 2000. Cartesian Genetic Programming. *Pages 121–132 of: Proceedings of the 3rd European Conference on Genetic Programming*. Edinburgh: Springer Verlag, Berlin.