Stack-Based Gene Expression

Samuel LANDAU, Sébastien PICAULT

Laboratoire d'Informatique de Paris 6 (LIP6)
Pôle IA – thème OASIS
8, rue du Capitaine Scott
75 015 Paris

{landau,picault}@poleia.lip6.fr

May 7, 2002

Abstract

We present a new model of gene expression, which relies on an interpretation with a stack. We first give an outline of the main morphogenic evolutionary computation approaches, and bring out the properties we wish to have from a gene expression model used to build structures. Then we present our model and an application of its principles. Finally, we give a discussion on the properties of the model and on experimental results.

Résumé

Nous présentons un nouveau modèle d'expréssion génétique, qui repose sur une interprétation à l'aide d'une pile. Nous dressons d'abord un tableau des principales approches morphogénétiques en informatique évolutionniste, et faisons ressortir les propriétés que nous recherchons dans un modèle d'expression génétique utilisé pour construire des structures. Ensuite nous présentons notre modèle et une application de ses principes. Enfin, nous discutons sur les propriétés du modèle et sur des résultats expérimentaux.

Contents

1 Introduction

A major trend in evolutionary computing is about automatically designing structures, let them be structures that are used to solve a problem (e.g. a circuit [1], a finite state machine [2]) or structures that are used to have agents behave (e.g. neural networks [3], program trees [4]). Most of the time, these are *executable* structures [5] the behavior of which is evaluated.
This evolutionary computing process is inspired by the representation transformations offered by the natural gene expression process, where information is taken from one structure (the genotype) to build another one (a protein) after various transformations, mainly the gathering of amino-acids into a peptides chain, and its folding to form an actual protein, helped by specific enzymes. Some evolutionary computing approaches respect this paradigm of successive transformations steps (*indirect* genetic encoding) while the others directly use the genotype as the structure (*direct* genetic encoding).
Choosing a genetic encoding is thus a main issue in gene expression computation. This choice has to be done on a multi-criteria basis: the encoding must allow to describe solutions to the given problem to solve (that is the expression part) and it must also allow to gradually find adapted solutions despite evolution is a blind process.
The existing evolutionary computing approaches provide different answers according to the features they focus on, but the first requisite of choosing between direct and indirect encoding already has strong consequences. On the one hand, we already can observe that over time, different approaches have converged on many points (e.g. the use of crossing over as a genetic operator, the use of a genetic mutation rate included in the genotype to obtain adaptivity) but on the other hand this convergence didn't happen yet for this first choice : it is a fact that direct and indirect encoding both have advantages.
We think that these advantages may not be contradictory, and we propose a new approach based on genetic *interpretation* to build structures, that could conciliate them.

We first summarize and discuss the different existing approaches of gene expression already used to evolve structures, in the light of the type of encoding they use (direct or indirect). Afterwards, we explain what we expect from a gene expression process that has to build structures, and how a gene expression process that uses a meaningless bitstring genome and an stack-based interpreter to build a structure from it might be an interesting alternative for the design of evolvable structures. We then detail an application of these principles to build some kind of automaton (an ATN[1]) and give some experimental results.

2 Existing Models of Gene Expression to Evolve Structures

We can classify the existing models of gene expression aimed at evolving structures – morphogenic evolutionary computation [6] – in two main groups: the ones that use direct encoding, and the ones that use indirect encoding. One the one hand, direct encoding is motivated by phenotypical considerations, i.e. producing (executable) structures in which there is a strong behavioral linkage between parent and offspring, and on the other hand indirect encoding allows faster or finer-grain search space exploration, and is also nearer to biological models of gene expression – the primary inspiration source for evolutionary computation.

2.1 Direct Encodings

There are roughly two approaches to produce structures with direct encoding. The first one sets parameters for a prior given structure; it allows fine-tuning of a given structure, but does not allow to generate it from scratch. The second one makes no separation between the genotype and the structure; it allows to generate any structure, but to do this the designer has to think of genetic operators that must face many constraints.

2.1.1 The Genotype as a Set of Parameters for a Prior Given Structure

With this approach, the genes do not encode directly a structure, but rather parameters to be used by a preexisting structure. The evolutionary process thus consists in optimizing a set of parameters [7] for a structure that has already been designed and mainly involves Evolution Strategies [8, 9, 10] or Genetic Algorithms [11, 12, 13]. This approach has been mainly used to

[1] ATN stands for Augmented Transition Network

evolve artificial neural networks (see [3], section 3.1), for example [14, 15, 16, ...].
The optimization is often reported to produce good results compared to other learning algorithms. In addition the approach allows to fine-tune the structure parameters for a given problem. But on the other hand, it does not allow plasticity, since the structure architecture is given, which sets strong assumptions and restrictions on the search space. We retain that direct parameter encoding lacks *expressive power*.

2.1.2 The Genotype as the Structure Itself

This approach makes no distinction between the genotype and the phenotype: the genes are components of the structure. Many structures are reported to be evolved this way: finite state machines [2, 17], graphs [18], neural networks [19], trees [4], ... The evolutionary algorithms involved are mainly Evolutionary Programming [2] and Genetic Programming (EP and GP) [4, 20] for trees.
Structures can be built from scratch, which is a strong advantage over parameters optimization because it allows a wide search in the space of structures. However, the design of the genetic operators has to face more constraints. The main ones address the following issues:

- The *syntactic* issue consist in ensuring the consistency of the structures that are produced. A random generation, mutations on a structure or crossing over of two structures must produce valid offspring.

- The *semantic* issue is brought by the phenotypical orientation of the approach: the behavior of the executable structures. Operators should produce offspring whose behavior is close to the behavior of the parents.

- Finally, the *completeness* issue means the ability to explore the whole search space without excluding portions of the search space that may eventually contain better solutions, or not biasing too much operators towards a part of the search space (for the same reason).

To sum up, this approach allows to *build* structures, but it suffers from difficult design of genetic operators. These issues have been widely discussed and offered solutions [4, 20, 21, 22, 23, ...]. This difficulty partly relies on the fact that the genotype is divided into parts that have differentiated properties (due to the structure). Thus the modifications induced by genetic operators have to take into account these different roles according to syntactic and semantic issues. A similar problem has been encountered in classical biological paradigms for gene expression [24]. As a consequence, the balance between trying to maintain behavioral proximity between parents and offspring on the one hand and exploring the search space on the other hand is made more difficult.

2.2 Indirect Encodings

Indirect encodings involve more complex gene expression processes to produce the evaluated structures (*morphogenic evolutionary computation*, [6]). The indirection features several advantages, mostly:

- *adaptibility* in the creation process: the evaluated structure might not be straight-forward interpretation of the genes. This may help to solve syntactic issues mentionned before (section 2.1.2).
- *compacity* of the genetic code: the phenotype structure might be much more larger than the genetic representation.

In a similar way to section 2.1, we may distinguish two main approaches to produce structures with indirect encoding. The first one sets describing parameters for a structure allowing fine-tuning of a given structure, but it has "bad" genetic properties. The second one uses a describing structure as a genotype to build the evaluated structure (phenotype); it allows to generate *any* structure and to compress gene representation, but again the designer has to face design difficulties as argued in section 2.1.2.

2.2.1 The Genotype as a Set of Describing Parameters

This approach relies on parameters arrays that are interpreted to form the structure. It has been used to evolve artificial neural networks (see [3], section 3.4), for example [25, 26, ...].
Like in direct encoding (section 2.1.1), the parameterization allows to fine-tune the structure, and adds the possibility to partially build it from scratch. But even if the structure is not a priori given, the gene expression process needs fixed assumptions about the structure which are made according to prior knowledge. The prior knowledge necessary to structure building makes

this approach not very flexible, because all the possibilities have to be foreseen to be parametered. From the completeness point of view, the portions of search space that can be explored are thus limited. And finally, since the genotype is tightly linked with the expressed structure, the designer has to choose very enforcing genetic operators to ensure syntactic correctness of the generated genotypes.
This approach allows to carefully fine-tune structures on which the designer already has some prior knowledge, but it is not convenient if we need to evolve structures in a more plastic way.

2.2.2 The Genotype as a Describing Structure

This approach, the most sophisticated one, evolves structures as genotypes to describe the evaluated, phenotypic structures. The latter are generated according to an interpretation of the first structure, which may be of a totally different nature. The gene expression process is referred to as *development*, by analogy with the biological development. This approach has been used to design networked structures, among which electrical circuits [1], and especially artificial neural networks (see [3], section 3.2.2). The gene expression process involved can operate by successive rewritings, of e.g. grammatical rewriting rules (L-systems [27]) in [28], matrix-based rewriting rules [29], or by executing a developmental program tree that can be syntactically constrained [30] or not [31, 32].

The size of the describing structure and that of the phenotype structure may not be correlated. This indirect gene expression scheme allows to generate phenotype structures the size of which is greater than the genotype one by orders of magnitude. Development not only permits to produce compressed genetic representation, but also structures that might be of any complexity, still keeping simple genotype representation. However, the underlying structure which is manipulated by the operators still has to indirectly address the issues detailed in section 2.1.2: its expression must produce a valid evaluated structure (syntactic constraint), the behavior of offspring should be as close as possible to the parents one (semantic constraint) and no portion of the search space that could contain interesting solutions should be excluded (completeness constraint).
The syntactic constraint is generally easy to deal with: through using grammatical genotype structures, generating valid phenotype structures is straight-

forward. But the semantic and completeness constraints are the hard part, because of the indirection level: it becomes even harder to foresee if most of the manipulations on the gene structure will induce small changes in the phenotypic structure, and will allow full exploration of the interesting portions of the search-space (that are most of the time unknown). This is due to the lack of knowledge on the effect of smaller manipulations of the genotype structure, that could be expressed as too coarse grain modifications at the phenotype structure level. Thus, again (like in section 2.1.2), this approach implies much work from the designer, and it is often necessary to introduce strong biases in the genetic operators to correct unwanted effects.

3 Stack-Based Gene Expression and Structure Design

In [33], we have stated principles that an evolutionary process that builds behaviors for agents [34] or animats [35] – autonomous entities behaving in an environment – should comply with. These principles, called *Ethogenetics*, are an attempt to cumulate the advantages of Genetic Algorithms – *cumulative adaptation* – and those of Genetic and Evolutionary Programming – *power of expression* – that are not contradictory. These principles can be imported without many changes in their formulation for the evolution of structures, which are the substratum for the behavioral expression in agents.
In the next sections, we first summarize the characteristics of the existing approaches of gene expression to build structures as detailed in section 2. Then we state the qualities we are seeking in an evolutionary process that evolves structures, and argue that these properties are not contradictory, by outlining and defining a new model of gene expression that fulfills them. This is a fine-grained, stack-based gene expression model. Finally we detail its functioning and how to evolve structures with it.

3.1 Toward Stack-Based Gene Expression

Direct encoding approaches allow to have a direct access to the structure, and thus facilitate engineering and human manipulations. That could be very useful to introduce prior knowledge and "bootstrap" the evolution process, or for refining or testing a solution made by hand. Then, the phenotypical considerations of strong behavioral linkage between parents and offspring seem

an interesting property to search for in an evolutionary approach, because it allows *cumulative adaptation*: evolution is a blind process in which the genetic operators act randomly and are not very liable to bring about adaptive improvements. So, if there is no *continuity* with the expressed behavior of the structure, the evolutionary process would not be able to *improve* the behaviors. Continuity is the property of having small variations of the genotype inducing most of the time, small variations in the phenotype – the behavior expressed by the structure. But direct encoding either lacks expressive power, or raises many genetic operators issues because of strong syntactic constraints.
On the other hand, indirect encoding approaches allow to have more flexibility and placticity (e.g. by building the structure in an adaptive way depending on its environment) or compact genetic representations. Besides, it is closer to biological gene expression process. But it is also harder to foresee the impact of the gene manipulations, thus much design work has to be done to try to obtain cumulative adaptation because of indirect semantic and completeness constraints.

Thus, the qualities we are seeking to build structures through an evolutionary computation process are:

- relative simplicity of gene expression process in order to facilitate human engineering and manipulations.

- easiness of genetic operators conception, with regards to the syntactic, semantic and completeness constraints expressed in section 2.1.2.

- continuity in the genotype-to-phenotype function, in order to allow cumulative adaptation. Continuity implies strong behavioral linkage between parents and offspring, since small differences in the genotype mostly induce small variations in the phenotype – anyway, evolutionary approaches work optimally with gene expression processes that respect this property.

- power of expression, in order to be able to generate any structure of any complexity, this without having to make changes to the gene expression process (e.g. adding biases to genetic operators to counterbalance unwanted effects).

Since the existing approaches do not fulfill all these requirements simultaneously, we propose a new one, that uses a stack-based gene expression process. Interpretation with a stack is a simple expression process, some common computer languages are stack-based (e.g. PostScript, Forth). The use of a stack also allows to have instruction locally interacting through the stack during the structure building process. This way most modifications on the genotype will only have local impact on the structure, hence ensuring the continuity property.
We then have to choose a langague that would be rather declarative, in order to avoid having a hierarchical structure (which carries many problems in the design of genetic operators). The language should support locality by avoiding atoms that deeply modify the structure, thus we propose to use fine-grain structure-building languages. Finally, this language should obviously be able to describe any structure of the kind we are evolving.

3.2 Related Works

[36] has first used a stack-based genetic programming model with success. He didn't use it to build structures, but only to execute a program. This approach resulted to be competitive compared to classical GP.
[37] have then also used this approach in a parallelized way (HiGP), and added reflection to the language in order to obtain adaptivity during the expression of a program [38, 39]. This resulted in a faster evolutionary process, but they didn't use this approach to produce structures either.

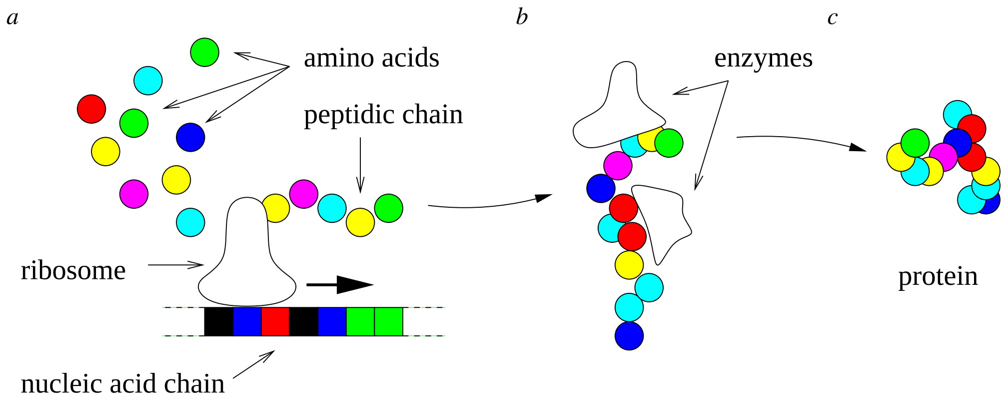
3.3 The Stack-Based Gene Expression Model

The model we propose is inspired by biological gene expression process (see figure 1), with two separate steps: a translation and an interpretation.

The stack-based gene expression model we use is an indirect encoding of the structure, with a plain bitstring as genotype.

In the biological process, the production of a protein mainly involves a traduction by the ribosome, that gathers amino-acids in a peptidic chain, and then the folding of this peptidic chain into the actual protein, with the help of enzymes. The "behavior" (i.e. function) of the protein is mainly influenced by its *shape*, which means that its "semantics" is not directly dependent on the "syntax" of its amino-acids components. If the same peptidic chain is folded into two different shapes, for example because different enzymes were

Figure 1: Ribosomic gene expression. a. the ribosome reads a nucleic acids chain and gathers amino-acids into a polypeptidic chain; b. the polypeptidic chain folds, helped by enzymes...; c. ...into the actual protein

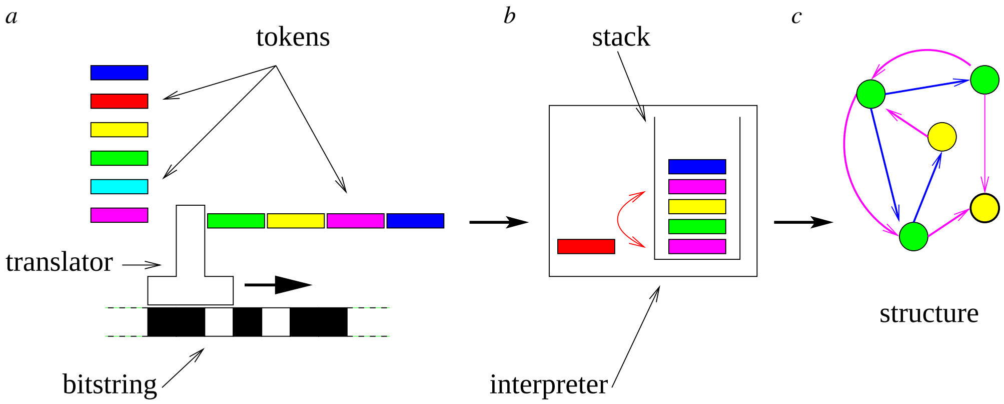


involved during the folding step, they are liable to behave differently. However this process always relies on the same physico-chemical laws, involved only in local interactions. In stack-based gene expression, we also use a two-step expression process (figure 2) using translation and interpretation. The interpretation process is always the same, whatever the kind of structure we evolve or the building language we use.
The first step (figure 2) is the translation of the bitstring into a tokens stream, that is sent to a stack-based interpreter (that produces the structure, as the second step).
The translator uses a genetic code, i.e. a function

$$\mathcal{G} : \{0,1\}^n \longrightarrow \mathcal{T} \quad (|\mathcal{T}| \leq 2^n)$$

where $\mathcal{T}$ is a set of tokens, and n is the size of our "codons". The tokens are divided into two categories: some of them are instructions of the stack-based language (*stack* tokens, e.g. the `swap` token the action of which is to swap the two top elements of the stack) and the others ones are specific to the kind of structure we want to evolve (*structure* tokens, e.g. the `connect` token which creates an edge between nodes for a graph structure). Notice that the genetic code may be redundant (many codons associated to the same token), and this contributes to the continuity between genotype and phenotype (a

Figure 2: Stack-based gene expression. a. the translator reads a bit string and outputs a token stream ; b. the tokens are sent to the interpreter, that may push the token on the stack and/or perform an action associated to the token ; c. the actual structure is popped from the stack



mutation is likely to transform a codon into another one that produces the same token).
The behavior of the structure used as the phenotype is also mainly dependent on its shape, so the analogy with the biological process is quite straightforward.

4 ATNoSFERES: Application to the Design of ATNs

The goal of the ATNoSFERES model [40] is to automatically design evolvable agents behaviors, in a way that fulfills the Ethogenetics principles [33].
ATNoSFERES uses the SFERES framework [41] as a tool for modeling the agents classes, integrating those classes to the system, designing an environmental simulator and providing classical evolutionary techniques. The agents behaviors are described by a labelled graph structure, an ATN[2] (see figure 3). An ATN[42] is a particular kind of graph that was primarily used in language

[2]ATN stands for Augmented Transition Network

processing theory and have also then been used by [43] for designing agent behaviors.

We will not discuss here the behavioral issues specific to agents, but rather

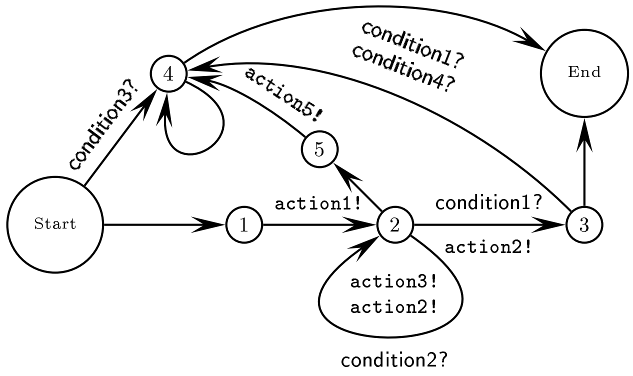


Figure 3: An example of ATN.

focus on the evolution features of our model. Then we will show an example of ATN building and give some experimental results.

4.1 The ATNoSFERES Model

In the ATNoSFERES model, we assume that the behavior of any agent may be described by a directed labelled graph (ATN). Thus we propose to produce it from a bitstring genotype, following the stack-based expression principle explained in section 3.3.

Each agent category is associated with two collections of tokens: *condition* ones and *action* ones. The actions are behavioral "primitives" that can be performed by the agent, the conditions are perceptions or stimuli that induce action selection. The edges of the graph are labelled with a set of conditions and a sequence of actions (see figure 3).

The ATN is built by creating nodes and edges and finalize the structure by adding two particular nodes: a "Start" node and an "End" node.

At each time step, the agent (initially in the "Start" state) randomly chooses an edge among those having either *no condition* in their label, or *all conditions simultaneously true.* It performs the actions associated with this edge and jumps to the destination node. It stops working when its state is "End". This is a state-based architecture: the agent goes through the states of the

ATN one at a time during its whole life-cycle.

We will now detail more precisely the translator and interpretor used to build ATN graphs from a bitstring. We will then give an illustrated example of stack-based expression to build the ATN.

4.1.1 The Translator

The translator is completely defined by the genetic code it uses. There are 2 types of tokens (see table 1):

- *stack* tokens, that are used to manipulate the structure under construction.
- *structure* tokens, including:
  - *ATN* tokens, such as those used to create nodes and to connect them.
  - *agent* tokens, such as conditions and actions ones, that are used to label edges between nodes in the ATN.

4.1.2 The Interpreter

The purpose of the interpreter is to build an ATN from the tokens. It successively reads them from the translator, and computes the eventual actions that are associated with them (see table 1):

- each stack token computes a specific action on the stack,
- condition and action tokens are just pushed on the stack,
- nodes and connection creation tokens push a node (it may be a node connected to “Start” or to “End”) on the stack, or connect the two last nodes on the stack, labelling the edge with the action and condition tokens that are between them.

If an instruction cannot execute successfully, it is simply ignored. Finally, when the interpreter does not receive tokens any more, it terminates the ATN: actions and conditions tokens still present between nodes are treated as *implicit connections* (so that new edges are created) and the consistency

token	(initial list state)	$\longrightarrow$	(resulting list)
dup	$(x\ y\ ...)$	$\longrightarrow$	$(x\ x\ y\ ...)$
del	$(x\ y\ ...)$	$\longrightarrow$	$(y\ ...)$
dupNode	$(x\ y\ N_i\ z\ ...)$	$\longrightarrow$	$(N_i\ x\ y\ N_i\ z\ ...)$
delNode	$(x\ N_i\ y\ N_i\ z\ ...)$	$\longrightarrow$	$(x\ y\ N_i\ z\ ...)$
popRoll	$(x\ y\ ...\ z)$	$\longrightarrow$	$(y\ ...\ z\ x)$
pushRoll	$(x\ ...\ y\ z)$	$\longrightarrow$	$(z\ x\ ...\ y)$
swap	$(x\ y\ ...)$	$\longrightarrow$	$(y\ x\ ...)$
node	$(x\ ...)$	$\longrightarrow$	$(N_i\ x\ ...)$[a]
connectStart	$(x\ ...)$	$\longrightarrow$	$(N_i\ x\ ...)$[b]
connectEnd	$(x\ ...)$	$\longrightarrow$	$(N_i\ x\ ...)$[c]
connect	(`c1?` `c2?` $x\ N_i\ y$ `c1?` z `a2!` `a1!` $t\ N_j\ u$ ...)	$\longrightarrow$	$(x\ N_i\ y\ z\ t\ N_j\ u\ ...)$[d]
condition?	$(x\ ...)$	$\longrightarrow$	(condition? x ...)
`action!`	$(x\ ...)$	$\longrightarrow$	(`action!` x ...)

Table 1: The ATN-building language.

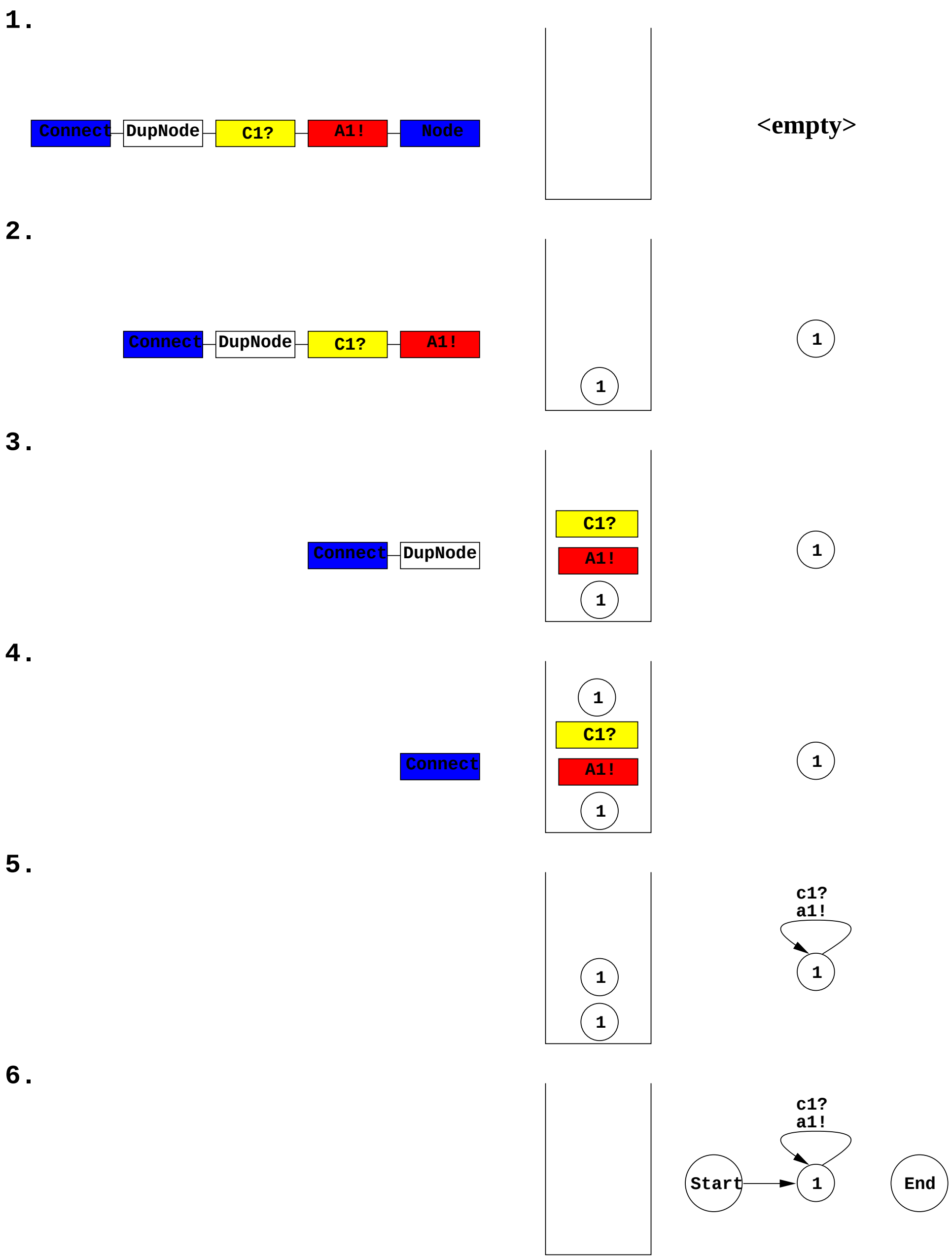
[a] creates a node N_i
[b] creates a node N_i and connects "Start" to it
[c] creates a node N_i and connects it to "End"
[d] creates an edge between N_j and N_i, with (`c1?`& `c2?`) as condition label and the list {`a1!`,`a2!`} as action label

of the ATN is checked ("Start" node is linked to nodes having no incoming edges, except from themselves; in the same way, nodes having no outgoing edges are linked to "End"). This may be considered as the actions associated with a virtual `finish` token that would always be at the end of any token stream: there is no contradiction with the genericity of the interpreter of the stack-based model.
We note that the programming language implemented by the interpreter is quite declarative (only locally imperative) and does not require an exhaustive explicitation of the structure.

4.2 Example of ATN building

(see figure 4, page 17)

Figure 4: Example of stack-based ATN building. 1. initial state: no structure and stack empty ; 2. a first node is created ; 3. conditions and actions are pushed on the stack ; 4. *dupNode* pushes a copy of the first node encountered ; 5. node 1 is connected to itself ; 6. final state, after an implicit connection to the `Start` node.



4.3 Experiments

The experiment we will describe here is just a simple illustration of how behavior based on ATN graphs may evolve through our model. It addresses a simple problem in order to emphasize that specific constraints on the structure may be taken into account and solved by genotype reorganization, due to the stack-based building process.
Additional experiments on coevolution of predators and preys in an ecosystem are currently ongoing and should provide soon results regarding the issue of the complexity of the behaviors.

4.3.1 Experimental setup

To illustrate the evolution of simple behaviors, we will consider an experiment with a discrete environment containing a color light bulb and a single agent with the action and perception abilities described in table 2. We want the agent to go to the right when the light is green and to the left when it is red. To make the agent behavior evolve, we apply the rules of darwinian selection over a population of 100 homogeneous agents.

Actions		**Conditions**	
`N!`	no action		
`R!`	move to the right	g?	true when the light is green
`L!`	move to the left	r?	true when the light is red
`U!`	move up	rand?	true with probability $p = 0.5$
`D!`	move down		

Table 2: Action and condition tokens of the agent.

The genetic code for these agents contains the 11 stack and ATN tokens and the 8 action/condition tokens; thus it needs at least $32{=}2^5$ codons. In the following experiments, a 32-codon genetic code has been automatically built from the list of all the tokens. The genotype of the agents is initially a *random* bitstring (*chromosome*) with a *random length*.

We evaluate the fitness of each agent by making it run during 100 time steps in its environment. The color of the light bulb randomly flips (with probability 0.05 at each time step) from green to red and vice-versa. The rewarding rules in the fitness function are: +1 point if the move is correct, -1 point if it is erroneous (e.g. left when green), 0 in the other cases (e.g. move up). Only the first move performed during the current time step is rewarded (`N!` is not considered as a move).

At each generation, the agents are evaluated through their average fitness (calculated over 10 runs in the above conditions) and selected to produce 30 new agents (by crossing over chromosomes), thus replacing 30 agents removed from the old population (depending on their fitness, too).

We have experimented with various mutation strategies, among them:

1. before their evaluation, *all agents* are subject to punctual random mutations of their chromosome with rate r (r % of the bits are randomly flipped);

2. same situation, but p % of the mutations are random insertions or deletions of codons in the chromosome, instead of punctual mutations.

4.3.2 Results

As agents are initialized in their "Start" state, the first time step is used to jump to one of the available nodes. Then, during the 99 other time steps, the behavior of the agents only depends on their ATN structure and its ability to respond to environmental changes. Thus, the maximum fitness in these experiments is 99 (correct answers at each time step after the first one).

Figure 5 shows the average evolution of the fitness with the first mutation strategy. It has been calculated from 10 experiments. Figure 6 shows the average evolution of the fitness with the second mutation strategy, and the evolution of the chromosome length (10 experiments too).

4.3.3 Results analysis

Though the problem to solve is very simple, these results validate the ATNoSFERES model for evolving agent behaviors from a fine-grain substrate and they already show interesting organizational genetic properties of the stack-based gene expression process. We would like to focus on the specificity of this model and give a qualitative analysis of the behaviors produced by natural selection.

The ATN described on figure 7 is the "optimal" solution to this problem (99 points with the simplest ATN structure): when the light is red, go to the

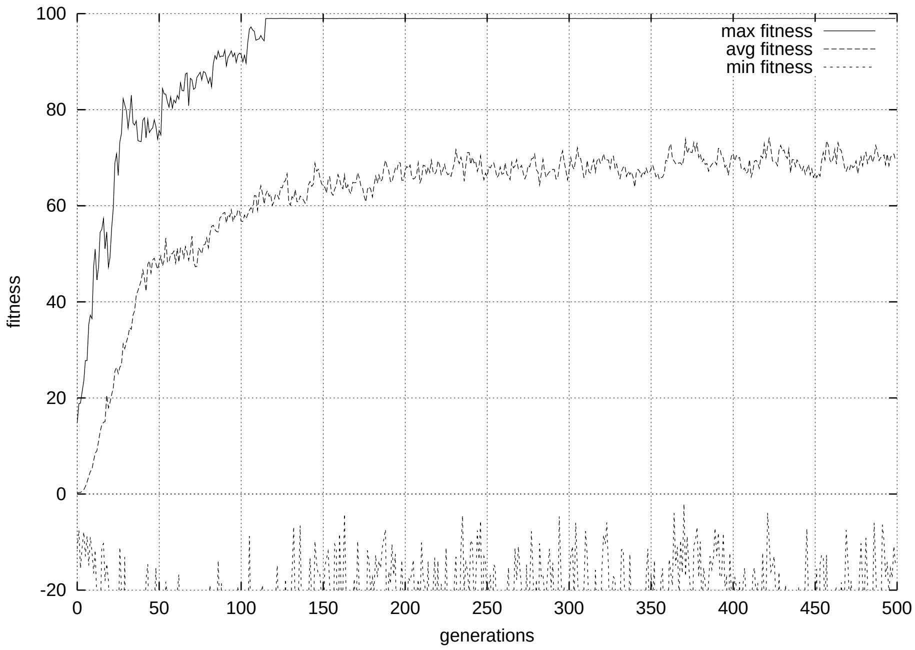


Figure 5: Evolution of the fitness (punctual mutations only, $r = 1$ %).

left; when it is green, go to the right. To produce this ATN, only 35 bits are theoretically required, e.g. to encode the following tokens:

$\boxed{\textit{node}, \texttt{g?}, \texttt{R!}, \textit{dupNode}, \texttt{L!}, \texttt{r?}, \textit{dupNode}}$

(with a maximal use of implicit connections). But in this solution the order and nature of tokens is crucial, thus it is highly vulnerable to mutations. In addition to this, the agents have much more bits in their chromosome than necessary – this can be a source of inadequate behavior.

The experimental results show that two strategies are used to produce an adequate behavior (99 points). The first one consists in building a simple ATN (very close or same than the "optimal" ATN on figure 7), by delaying the node creation and thus using tokens that have no effect. For instance, a 207-bit chromosome encoding the following tokens:

`L!`, *dup*, *swap*, *popRoll*, *popRoll*, *del*, *popRoll*, *swap*, *dup*, *del*, *popRoll*, `R!`, `L!`, *swap*, rand?, `L!`, *del*, *connectStart*, `R!`, *dup*, g?, *dupNode*, *popRoll*, *pushRoll*, `L!`, *del*, *pushRoll*, `L!`, r?, *connectStart*, *del*, *dup*, *dupNode*, r?, `R!`, *del*, *dup*, `R!`, *dup*, g?, *popRoll*

produces an ATN represented on figure 8, very close to figure 7, with labels (r?, `L!L!R!R!`) on one edge and (g?, `R!R!`) on the other. This is a good example of using the properties of the ATN-building language.

The agents using the second strategy build a complex ATN in which a small subset only is used, for instance like the ATN on figure 9. It leads to exactly the same behavior than the "optimal" solution, since a large part of it cannot be reached from the other nodes.

5 Discussion

We will discuss stack-based gene expression with regards to the three issues raised by morphogenic evolutionary computing as expressed in section 2.1.2, i.e. syntax,semantics and completeness.

5.1 Syntactical Issues

5.1.1 Dynamic String Representation

As most evolutions of executable structures, the stack-based gene expression uses a dynamic representation as a genotype [21, 44]. This is more convenient, as we do not know by advance what structure will be the fittest, and thus what bitstring length will be needed to build it.
But while in other works representation size is limited by the designer (and not only by the host system memory), e.g. trees depth in GP [4], or string length in Stack-Based GP [36] or in HiGP [37], we did not have to add such a constraint. We observed convergence of strings length to a (reasonable) finite limit, so we did not need to constrain genotype maximal representation size. We are currently investigating this property more precisely. We think that it is linked with the locality property, and that an equilibrium is found between the growing and the shortening tendencies. Our hypothesis is that:

- if the string length is too short, mutations impact is probabilistically stronger (due to a reduced redundancy in the tokens, see sections 4.3.3

and 5.1.3) and the genotype does provide few "junk" code to protect itself from mutations. By "junk" code we intend substrings that either are not expressed, or produce parts of the structure that are not used (e.g. in ATNoSFERES, subgraphs that are not connected with the "Start" node (see section 4.3.3 and figure 9)).

- if the string length grows too long, it becomes even harder for crossing over to make efficient exchanges of genetic materials. Each exchanged substrings are likely to carry a usable part of the final structure (thanks to the locality property), since they tend to be longer and longer. Thus the crossing over is more and more liable to juxtapose incompatible graph subparts, and thus becomes inefficient.

5.1.2 Separation between Syntax and Semantics

As the genotype syntax is not closely tight to the structure semantics, the stack-based gene expression is even less constrained than for example indirect grammar-based encodings, or GP. With stack-based gene expression there is no need to constrain the genotype syntax like in other approaches (e.g. closure principle [4] or on the contrary strong types [20] in Genetic Programming). This allows greater flexibility for the genetic operators. The bitstring can be manipulated through a "blind" process without any regard to its interpretation and we do not need operators that are specific to the structure that is being built.
Furthermore, the genotype that is manipulated is a meaningless bitstring structure, and as *any* genotype can be expressed as a valid structure, any genetic operator imported from the GA may be used without any restriction. We even can think of some additional genetic operators like insertion or deletion (see section 4.3), that could manage resizings of the genotype string in a smoother way than crossing over.

5.1.3 Redundancy

The description of the various parts of the structure is not position dependent, many combinations of genotype substrings may produce the same structure. What seems more important is the relative positions and distances of the tokens involved in the design of a given part, because the tokens interact locally through the stack. Furthermore, the interpretation of some token may not take into account order of the tokens they are acting on in the stack.

For example in the ATNoSFERES model (see section 4.1), the `connect` token does not set any constraint nor on the order of conditions used to produce a set labeling an edge (since all the conditions need to be true for an edge to be eligible), neither on the conditions-actions order (since all the conditions will be tested before any action occurs when crossing the elected edge). This redundancy in the interpretation increase the evolvability of the model, since the same structure may be obtained in different ways.

Another kind of redundancy is that of the genetic code (see for example section 4.1.1 for details about the genetic code used in ATNoSFERES): depending on the number of tokens available, the genetic code might be more or less redundant. If necessary, it can be designed in order to resist mutations, or to encourage structure-building operations (having probabilistically more codons for stack or structure manipulations).

5.2 Semantical Issues

5.2.1 Strong Parents-Offspring Behavioral Linkage

We assumed that we use a fine-grain building language (section 3.1). Locality of action of the language atoms, and their fine-grain scope ensures that with stack-based gene expression, most of the manipulations on the genotype have only a local impact, thus offspring structures are most of the time produced from weak variations of their parents.
This strong parents-offspring behavioral linkage is an important feature because it allows to have continuity, which smoothes the fitness landscape and facilitates convergence toward adapted structures.

5.2.2 Cumulative Adaptation

The combination of on the one hand continuous variations occurring in the genotype (through genetic operators operating on a fine-grain substratum), and on the other hand the continuity between parents and offspring phenotype (due to the continuity in the genotype-to-phenotype function), makes cumulative adaptation possible through a selection process, i.e. individuals are getting more and more adapted to the environmental constraints. This ensures that solutions space is not explored through pure random search but according to improving solutions.

5.3 Completeness Issues

5.3.1 Power of Expression

The stack-based ATN building language we propose in the ATNoSFERES model allows to build *any* ATN (we have implemented an algorithm to produce a bitstring from a specified graph). It is possible to explore the *whole* solution space.
Our approach also allows to fine-tune the structures: this seems a good compromise with regards to [6].

5.3.2 Limited Search Operators Bias

Since any genetic operator is structure independent, by using our simple and uniform model of gene expression, we ensure that we can put as few bias as possible in the search operators: the exploration of the solution space can be performed by generic, classical operators (mutations, crossing overs, mutations/deletions, ...
Still, it might be necessary to choose a convenient genetic code (which influences the probability of occurence of each token). It did not appear as a central issue in our works until now, but it is a point that deserves attention, and will be experimented soon.

6 Conclusions and Perspectives

We have presented a new approach of morphogenic evolutionary computation, that uses an indirect encoding and also features the advantages of direct encoding. This stack-based gene expression is inspired from biological gene expression process and has interesting properties that we have illustrated through simple experiments and discussed.
In current works we are more deeply investigating the model properties, e.g. the adaptation of bit string length.
In future works, we plan to formalize and generalize the properties that a structure building language should comply with in order to keep the desired properties (especially, locality) independently from the nature of the structure being built (graphs, Petri nets, trees, . . .).

References

[1] J. R. Koza, F. H. Bennett III, D. Andre, and M. A. Keane, "Automated design of both the topology and sizing of analog electrical circuits using genetic programming," in *Artificial Intelligence in Design '96* (J. S. Gero and F. Sudweeks, eds.), (Dordrecht), pp. 151–170, Kluwer Academic, 1996.

[2] L. J. Fogel, A. J. Owens, and M. J. Walsh, *Artificial Intelligence through simulated Evolution.* John Wiley & Sons, 1966.

[3] X. Yao, "Evolving artificial neural networks," *PIEEE: Proceedings of the IEEE*, vol. 87, 1999.

[4] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* 1992.

[5] P. J. Angeline, "A historical perspective on the evolution of executable structures," *Fundamenta Informaticae*, vol. 36, no. 1-4, pp. 179–195, 1998.

[6] P. J. Angeline, "Morphogenic evolutionary computations: Introduction, issues and example," in *Evolutionary Programming*, pp. 387–401, 1995.

[7] T. Bäck and H.-P. Schwefel, "An overview of evolutionary algorithms for parameter optimization," *Evolutionary Computation*, vol. 1, no. 1, pp. 1–23, 1993.

[8] I. Rechenberg, *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzi pien der biologischen Evolution.* Stuttgart: Frommann–Holzboog, 1973.

[9] H.-P. Schwefel, *Evolutionsstrategie und numerische Optimierung.* Dr.-Ing. Thesis, Technical University of Berlin, Department of Process Engineering, 1975.

[10] H.-P. Schwefel, *Evolution and Optimum Seeking.* John Wiley and Sons, Inc., 1995.

[11] J. H. Holland, *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence.* Ann Arbor: University of Michigan Press, 1975.

[12] K. A. De Jong, *An analysis of the behavior of a class of genetic adaptive systems.* PhD thesis, Dept. of Computer and Communication Sciences, University of Michigan, 1975.

[13] D. E. Goldberg, *Genetic algorithms in search, optimization, and machine learning.* Addison-Wesley Pub. Co., 1989.

[14] G. F. Miller, P. M. Todd, and S. U. Hegde, "Designing neural networks using genetic algorithms," in *Proceedings of the Third International Conference on Genetic Algorithms* (J. D. Schaffer, ed.), p. 6580, Morgan Kaufmann, 1989.

[15] D. Whitley, T. Starkweather, and C. Bogart, "Genetic algorithms and neural networks: optimizing connections and connectivity," in *Parallel Computing*, vol. 3, p. 347361, august 1990.

[16] G. W. Greenwood, "Training partially recurrent neural networks using evolutionary strategies," in *IEEE Trans. on Speech and Audio Processing*, vol. 5, pp. 192–104, 1997.

[17] L. J. Fogel, P. J. Angeline, and D. B. Fogel, "An evolutionary programming approach to self-adaptation on finite state machines," in *Evolutionary Programming*, pp. 355–365, 1995.

[18] K. Sims, "Evolving 3d morphology and behavior by competition," in *Artificial Life IV: Proceedings of the Fourth International Workshop on the synthesis and simulation of living systems* (R. Brooks and P. Maes, eds.), (Cambridge, MA), MIT Press, 1994.

[19] P. J. Angeline, G. M. Saunders, and J. P. Pollack, "An evolutionary algorithm that constructs recurrent neural networks," *IEEE Transactions on Neural Networks*, vol. 5, pp. 54–65, January 1994.

[20] D. J. Montana, "Strongly typed genetic programming," in *Evolutionnary Computation*, 1995.

[21] P. J. Angeline, "Genetic programming: A current snapshot," in *Proceedings of the Third Annual Conference on Evolutionary Programming* (D. B. Fogel and W. Atmar, eds.), Evolutionary Programming Society, 1994.

[22] D. B. Fogel and L. C. Stayton, "On the effectiveness of crossover in simulated evolutionary optimization," in *Biosystems*, no. 32, pp. 171–182, 1994.

[23] P. J. Angeline, "An investigation into the sensitivity of genetic programming to the frequency of leaf selection during subtree crossover," in *Genetic Programming 1996: Proceedings of the First Annual Conference* (J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, eds.), (Stanford University, CA, USA), pp. 21–29, MIT Press, 28–31 1996.

[24] J.-J. Kupiec and P. Sonigo, *Ni Dieu Ni Gène. Pour une autre théorie de l'hérédité.* Science ouverte, Paris: Seuil, 2000.

[25] S. A. Harp, T. Samad, and A. Guha, "Towards the genetic synthesis of neural networks," in *Proc. of the Third Int. Conf. on Genetic Algorithms and Their Applications* (J. D. Schaffer, ed.), pp. 360–369, Morgan Kaufmann, 1989.

[26] V. Maniezzo, "Genetic evolution of the topology and weight distribution of neural networks," in *IEEE Transactions on Neural Networks*, vol. 5, pp. 39–53, 1994.

[27] A. Lindenmayer, "Mathematical models for cellular interaction in development, parts i and ii," 1968.

[28] E. J. Boers and H. Kuiper, "Biological metaphors and the design of modular artificial neural networks," Master's thesis, Dep. CS and Exp. and theor. psych., Leiden University, 1992.

[29] H. Kitano, "Designing neural networks using genetic algorithms with graph generation system," in *Complex Systems*, pp. 461–476, 1990.

[30] J. Kodjabachian and J.-A. Meyer, "Evolution and development of neural controllers for locomotion, gradient-following, and obstacle-avoidance in artificial insects," 1998.

[31] F. Gruau, *Neural Network Synthesis Using Cellular Encoding and the Genetic Algorithm.* PhD thesis, ENS Lyon – Université Lyon I, january 1994.

[32] S. Luke and L. Spector, "Evolving graphs and networks with edge encoding: Preliminary report," in *Late Breaking Papers at the Genetic Programming 1996 Conference Stanford University July 28-31, 1996* (J. R. Koza, ed.), (Stanford University, CA, USA), pp. 117–124, Stanford Bookstore, 28–31 1996.

[33] S. Picault and S. Landau, "Ethogenetics and the Evolutionary Design of Agent Behaviors," in *Proc. of the 5th World Multi-Conference on Systemics, Cybernetics and Informatics (SCI 2001)* (N. Callaos, S. Esquivel, and J. Burge, eds.), vol. 3, pp. 528–533, 2001.

[34] J. Ferber, *Les systèmes multi-agents : Vers une intelligence collective.* 1995.

[35] J.-A. Meyer and S. W. Wilson, eds., *From Animals to Animats: Proceedings of the First International Conference on Simulation of Adaptive Behavior*, 1991.

[36] T. Perkis, "Stack-based genetic programming," in *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, vol. 1, (Orlando, Florida, USA), pp. 148–153, IEEE Press, 27-29 1994.

[37] K. Stoffel and L. Spector, "High-performance, parallel, stack-based genetic programming," in *Genetic Programming 1996: Proceedings of the First Annual Conference* (J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, eds.), (Stanford University, CA, USA), pp. 224–229, MIT Press, 28–31 1996.

[38] L. Spector and K. Stoffel, "Automatic generation of adaptive programs," in *From Animals to Animats 4: Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior (SAB96)* (P. Maes, M. Mataric, J. Meyer, J. Pollack, and S. W. Wilson, eds.), (Cambridge, MA), The MIT Press, 1996.

[39] L. Spector and K. Stoffel, "Ontogenetic programming," in *Genetic Programming 1996: Proceedings of the First Annual Conference* (J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, eds.), (Stanford University, CA, USA), pp. 394–399, MIT Press, 28–31 1996.

[40] S. Landau, S. Picault, and A. Drogoul, "ATNoSFERES: a Model for Evolutive Agent Behaviors," in *Proc. of AISB'01 symposium on Adaptive Agents and Multi-agent systems*, pp. 95–99, 2001.

[41] S. Landau, S. Doncieux, A. Drogoul, and J.-A. Meyer, "SFERES, a Framework for Designing Adaptive Multi-Agent Systems," technical report, LIP6, Paris, 2001.

[42] W. A. Woods, "Transition networks grammars for natural language analysis," *Communications of the Association for the Computational Machinery*, vol. 13, no. 10, pp. 591–606, 1970.

[43] Z. Guessoum, *Un environnement opérationnel de conception et de réalisation de systèmes multi-agents.* PhD thesis, LIP6 – Université Paris VI, may 1996.

[44] P. J. Angeline, "Genetic programming and emergent intelligence," in *Advances in Genetic Programming* (K. E. Kinnear, Jr., ed.), pp. 75–98, MIT Press, 1994.

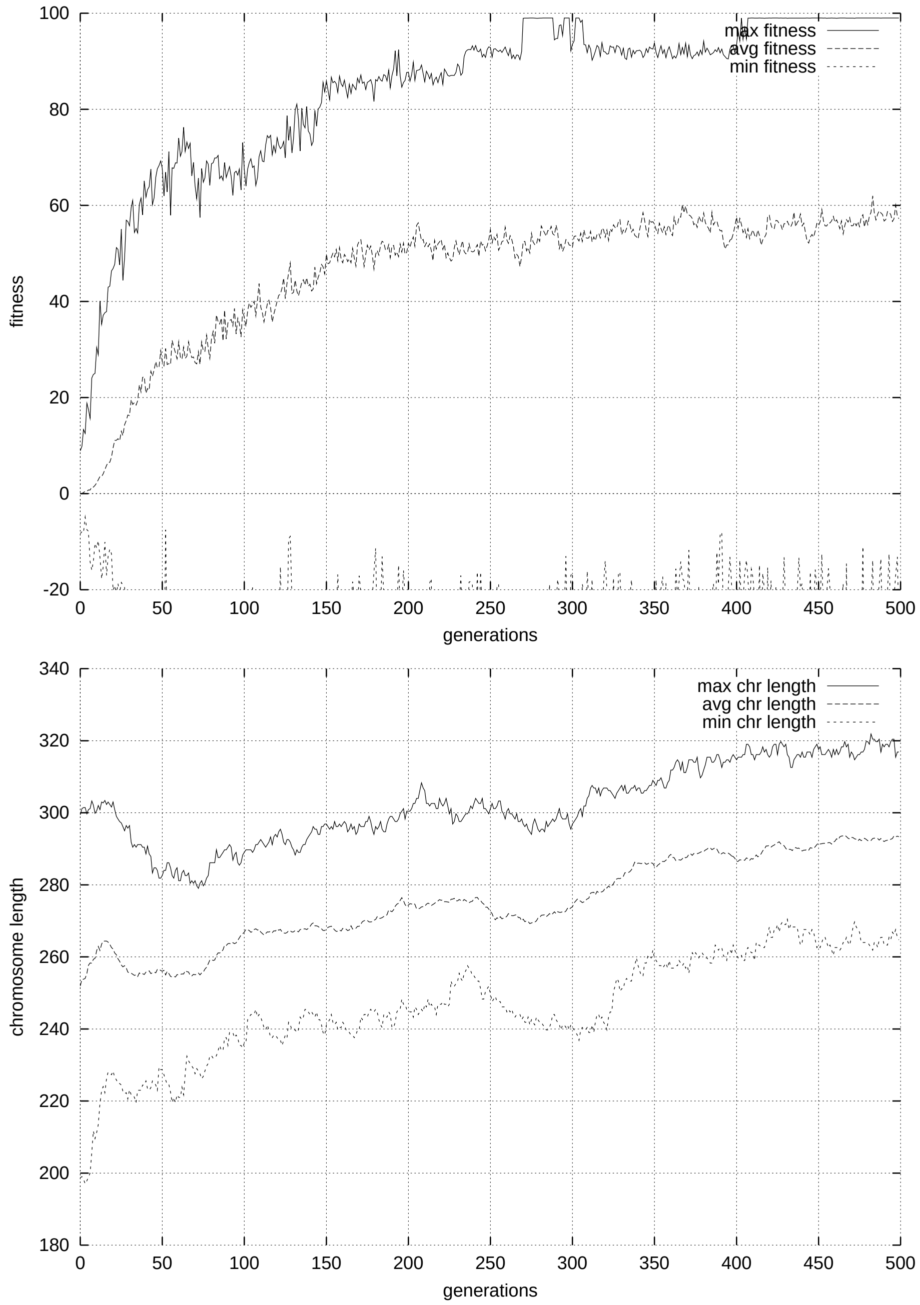


Figure 6: Evolution of the fitness and chromosome length (random insertions and deletions, $r = 1$ %, $p = 20$ %)

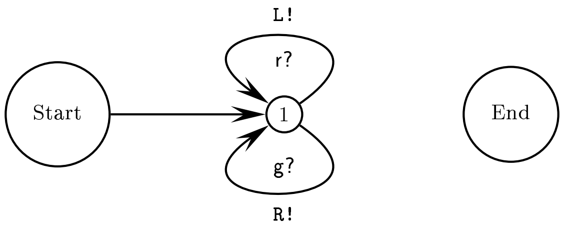


Figure 7: The "optimal" ATN (providing the highest fitness with the simplest structure).

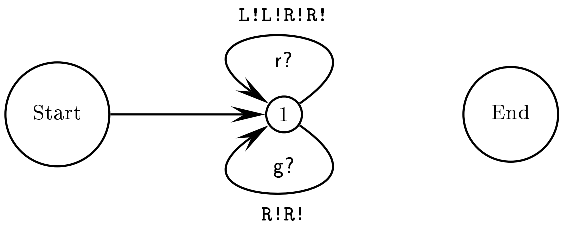


Figure 8: An ATN built by natural selection, implementing the best behavioral strategy with almost the simplest structure.

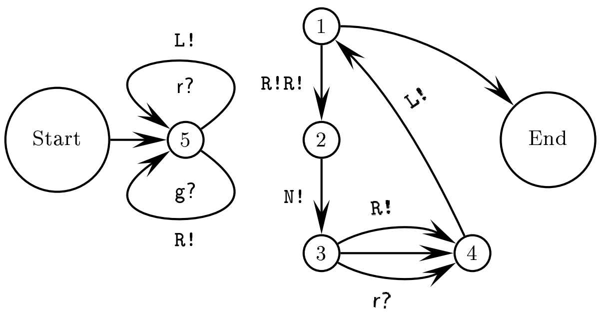


Figure 9: An ATN built by natural selection, implementing the best behavioral strategy.