A Common Genetic Encoding for Both Direct and Indirect Encodings of Networks

Yohannes Kassahun
Robotics Group
University of Bremen
Robert-Hooke-Str. 5,
D-28359, Bremen, Germany
kassahun@informatik.uni-
bremen.de

Mark Edgington
Robotics Group
University of Bremen
Robert-Hooke-Str. 5,
D-28359, Bremen, Germany
edgimar@informatik.uni-
bremen.de

Jan Hendrik Metzen
Robotics Group
University of Bremen
Robert-Hooke-Str. 5,
D-28359, Bremen, Germany
jhm@informatik.uni-
bremen.de

Gerald Sommer
Cognitive Systems Group
Institute of Computer Science
and Applied Mathematics
Christian Albrechts University
Olshausenstr. 40, D-24098,
Kiel, Germany
gs@ks.informatik.uni-
kiel.de

Frank Kirchner
Robotics Group
University of Bremen
German Research Center for
Artificial Intelligence (DFKI)
Robert-Hooke-Str. 5,
D-28359, Bremen, Germany
frank.kirchner@informatik.uni-
bremen.de

ABSTRACT

In this paper we present a Common Genetic Encoding (CGE) for networks that can be applied to both direct and indirect encoding methods. As a direct encoding method, CGE allows the implicit evaluation of an encoded phenotype without the need to decode the phenotype from the genotype. On the other hand, one can easily decode the structure of a phenotype network, since its topology is implicitly encoded in the genotype's gene-order. Furthermore, we illustrate how CGE can be used for the indirect encoding of networks. CGE has useful properties that makes it suitable for evolving neural networks. A formal definition of the encoding is given, and some of the important properties of the encoding are proven such as its closure under mutation operators, its completeness in representing any phenotype network, and the existence of an algorithm that can evaluate any given phenotype without running into an infinite loop.

Categories and Subject Descriptors

I.2.6 [**Computing Methodologies**]: Artificial Intelligence

General Terms

Algorithms, Theory

Keywords

Genetic Encoding, Genotype Phenotype Mapping

1. INTRODUCTION

Neural networks are useful in evolving the control systems of agents [6]. They provide a straightforward mapping between sensors and motors, enabling them to represent directly the policy (control) or the value function to be learned. It has been shown using standard benchmark problems that combinations of neural networks with evolutionary methods (neuroevolution) perform better than traditional reinforcement learning methods in many problem domains, especially in domains which are non-deterministic and only partially observable [2, 9].

In order to design an efficient neuroevolution method that can evolve both the structures and weights of neural networks, a flexible genetic encoding method is needed, and should possess the following important properties: (1) The encoding should be *complete*, in that it should be able to represent all types of valid phenotype networks. (2) The encoding scheme should be *closed*, i. e. every valid genotype represents a valid phenotype. Similarly, the encoding should be *closed under genetic operators* such as structural mutation and crossover that act upon the genotype. (3) It should be possible to apply the encoding to both the direct and indirect encoding of neural networks.

The main contribution of the work presented in this paper is to introduce a genetic encoding of networks that satisfies the above properties, and to formally prove that this genetic encoding fulfills the first two properties. Such proofs are not commonly given for other existing genetic encodings. Additionally, the third property listed above is not satisfied by most existing methods, and its fulfillment by the presented encoding can be regarded as a further important contribution of this work.

The paper is organized as follows: First, a formal definition of the common genetic encoding (CGE) is given, and it is proven that the encoding satisfies the above-mentioned properties. Next, an example of how CGE can be used for the indirect encoding of network structures is provided. After this, a review of representative works in the area of evolution of neural networks is given, and a comparison of CGE to other genetic encodings is made. Finally, we provide some conclusions and a future outlook.

2. COMMON GENETIC ENCODING

In CGE, a genotype consists of a string of genes, whose order implicitly represents the topology of a network. Each gene takes on a specific form (allele): it can either be a *vertex gene*, an *input gene*, or a *jumper gene*. A vertex gene encodes a vertex of a network, an input gene encodes an input to the network (for example, a sensory signal), and a jumper gene encodes a connection between two vertices. A particular jumper gene can either be a forward or a recurrent jumper gene. In addition to the explicitly encoded connections represented by jumper genes, there are also forward connections that are implicitly encoded in the way the genes are ordered in the genotype. A forward jumper gene represents a connection starting from a vertex with higher depth[1] and ending at a vertex with lower depth. A recurrent jumper gene represents a connection between two vertices with arbitrary depths. A vertex gene has a unique identification number, and a number which represents how many input connections it has. A jumper gene stores additionally the global identification number of the starting vertex gene. Depending on whether the encoding is interpreted directly or indirectly, the vertex genes can store different information such as weights (e.g. when the encoded network is interpreted directly as a neural network) or operator type (e.g. when the encoded network is indirectly mapped to a phenotype network).

2.1 Formal Definition

In this subsection, we provide a formal mathematical definition of our encoding for the case where CGE is used as a direct encoding. We begin by defining the set of genotypes, and some functions defined on a genotype's genes. We then discuss the criteria for valid genotypes, the genetic operators used, the development function, and the evaluation function.

2.1.1 The Set of Genotypes $\mathcal{G}$

A genotype $g = [x_1, ..., x_N] \in \mathcal{G}$ is a sequence of genes $x_i \in \mathcal{X}$, where $\mathcal{X} = \mathcal{V} \cup \mathcal{I} \cup \mathcal{J}_{\mathcal{F}} \cup \mathcal{J}_{\mathcal{R}}$. $\mathcal{V}$ is a set of vertex genes, $\mathcal{I}$ is a set of input genes, and $\mathcal{J}_{\mathcal{F}}$ and $\mathcal{J}_{\mathcal{R}}$ are sets of forward and recurrent jumper genes, respectively. For a gene x and a genotype $g = [x_1, \ldots, x_N]$ we say $x \in g$ iff $\exists\, 0 < i \leq N : x = x_i$. Each vertex gene has a unique identity number $id \in \mathbb{N}_0$ and each input gene stores a label $label$[2]. The set of identity numbers and the set of labels are disjoint. Each vertex gene x_i stores a value $d_{in}(x_i)$, which can be interpreted as the number of expected inputs (or the number

of arguments) of x_i. A forward or a recurrent jumper gene stores the identity number of its source vertex gene. Each gene can store different parameters such as, for example, a weight $w_i \in \mathbb{R}$. A subsequence $g_{l,m} = [x_l, x_{l+1}, \ldots, x_{l+m-1}]$ of g with $x_l \in \mathcal{V}$ is said to be a *subgenome* of a genotype g if the number of expected inputs in $g_{l,m}$ is equal to the number of produced outputs in $g_{l+1,m}$ (i.e. is equal to $m-1$)[3].

2.1.2 Functions Defined on the Genes of a Genotype

We define four functions, which will be used in proving the different properties of the genetic encoding. The first function $v : \mathcal{X} \longrightarrow \mathbb{Z}$ is defined as follows:

$$v(x_i) = \begin{cases} 1 - d_{in}(x_i), & \text{if } x_i \in \mathcal{V} \\ 1, & \text{if } x_i \notin \mathcal{V} \end{cases}. \tag{1}$$

The quantity $v(x_i)$ can be interpreted as the number of produced outputs minus the number of expected inputs of the gene x_i. Using this definition, we let

$$s_K = \sum_{i=1}^{K-1} v(x_i), \tag{2}$$

where $K \in \{1, \ldots, N+1\}$. Note that this definition implies $s_1 = 0$. The quantity s_K can be interpreted as the number of produced outputs minus the number of expected inputs in the subsequence $g_{1,K-1}$. We can now define the set of *output vertex genes* as $\mathcal{V}_o = \{x_j \in g \mid x_j \in \mathcal{V} \wedge (s_i < s_j \,\forall\, i : 0 < i < j)\}$ and the set of *non-output vertex genes* as $\mathcal{V}_{no} = \mathcal{V} - \mathcal{V}_o$. The function $parent : \mathcal{X} \longrightarrow \mathcal{V} \cup \emptyset$

$$parent(x_j) = \begin{cases} \emptyset, & \text{if } (s_i < s_j \,\forall\, i : 0 < i < j) \\ x_i, & \text{if } s_i \geq s_j \text{ and } s_k < s_j \,\forall\, k : 0 < i < k < j \end{cases} \tag{3}$$

defines a relationship between the genes in a genotype. Note that for an output vertex gene x_j, $parent(x_j) = \emptyset$. Finally, we define a function $depth : \mathcal{V} \longrightarrow \mathbb{N}$ as

$$depth(x_j) = \begin{cases} 0 & \text{if } parent(x_j) = \emptyset \\ depth(parent(x_j)) + 1, & \text{otherwise} \end{cases}. \tag{4}$$

Table 1 shows an example of a genotype, along with the resulting values of the above-defined functions.

2.1.3 Genotype Validity Criteria

A genotype $g = [x_1, ..., x_N] \in \mathcal{X}^N$ consisting of N genes has to fulfill the following criteria to be considered a valid genotype:

Criterion 1: Each vertex gene $x_i \in \mathcal{V}$ must have at least one input. In other words, $d_{in}(x_i) > 0$.

Criterion 2: There can be no closed loops of forward jumper connection genes in g. A closed loop exists if there is a way to visit a vertex gene more than once when following a series of forward jumpers from a source vertex gene to a target vertex gene.

Criterion 3: There is no forward jumper gene, whose source vertex *depth* is less than the *depth* of its target vertex.

Criterion 4: For a gene $x_k \in g$, $s_k < s_{N+1}, \forall k \in \{1, ..., N\}$.

---

[1] The depth of a vertex is defined as the minimal topological distance (i.e. minimal number of connections to be traversed) from an output vertex of the network to the vertex itself, where the path contains only implicitly defined connections.

[2] Input genes with the same label refer to the same input (see definition of $\mathcal{D}$).

[3] Each gene produces exactly one output. In the case of vertex genes, the output is either an implicit forward connection (see below) or an output of the network.
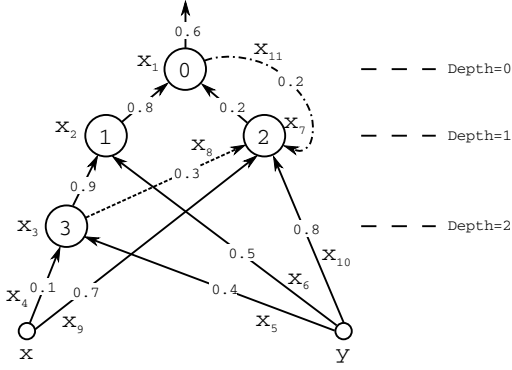
Figure 1: An example of a valid phenotype with one output vertex (0) and two input vertices (x and y).

Table 1: The phenotype in Figure 1 is encoded by the genotype shown in this table. For each gene x_i of the genotype, the gene's defined properties and the values of various functions which operate on the gene are summarized. In the allele row, V denotes a vertex gene, I an input gene, JF a forward jumper gene, and JR a recurrent jumper gene. The source row shows the id of the source vertex of a jumper gene and the *parent* row shows the id of the parent gene.

gene	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}
allele	V	V	V	I	I	I	V	JF	I	I	JR
id	0	1	3	-	-	-	2	-	-	-	-
source	-	-	-	-	-	-	-	3	-	-	0
label	-	-	-	x	y	y	-	-	x	y	-
weight	0.6	0.8	0.9	0.1	0.4	0.5	0.2	0.3	0.7	0.8	0.2
d_{in}	2	2	2	-	-	-	4	-	-	-	-
v	-1	-1	-1	1	1	1	-3	1	1	1	1
s	0	-1	-2	-3	-2	-1	0	-3	-2	-1	0
parent	$\emptyset$	0	1	3	3	1	0	2	2	2	2
depth	0	1	2	-	-	-	1	-	-	-	-

Criterion 5: For every $x_k \in g$: $parent(x_k) = \emptyset \Rightarrow x_k \in \mathcal{V}$.

The first criterion ensures that there are no vertex genes without expected inputs, since a vertex gene with no inputs results in unused substructures in the phenotype. The second and third criteria together guarantee that the developed phenotype contains no loops of forward connections and, as a result, that the evaluation of the phenotype will be completed in a finite amount of time. The last two criteria together ensure that the sum of outputs produced for all genes in g is equal to the sum of all expected inputs. Kassahun[4] describes how one can sample an initial population of genotypes that fulfill the above mentioned criteria.

2.1.4 Genetic Operators

The genetic operators to be used in CGE should be designed so that the genotypes they produce fulfill the criteria of the previous section. In this section, we will give examples of some genetic operators that can be used with CGE.

Parametric mutation: $\mathcal{PA} : \mathcal{G} \longrightarrow \mathcal{G}$. Parametric mutation changes only the values of the parameters included in the genes (e. g. the weights w_i). The order of the genes in g and $\mathcal{PA}(g)$ remains the same.

Structural mutation: $\mathcal{ST} : \mathcal{G} \longrightarrow \mathcal{G}$. An example of a structural mutation operator that fulfills the criteria of Section 2.1.3 follows. When the operator operates on a genotype, it either inserts a *recurrent jumper gene*, or a *subgenome*. If a *recurrent jumper* gene x_k is inserted, it must be inserted after a vertex gene x_i. The source vertex of this recurrent jumper can be chosen arbitrarily. The number of inputs $d_{in}(x_i)$ will be increased by one. If a *subgenome* is inserted, it must be inserted after a vertex gene x_i. The subgenome consists of a vertex gene x_k followed by an arbitrary number $M > 0$ of inputs or forward jumper genes. The source vertex of a forward jumper gene is not allowed to have a depth less than the depth of x_k. The number of inputs of the vertex gene $d_{in}(x_i)$ is increased by 1. Moreover, the number of inputs d_{in} to x_k is set to M and its depth is set to $depth(x_i) + 1$.

Structural crossover: $\mathcal{CR} : \mathcal{G} \times \mathcal{G} \longrightarrow \mathcal{G}$. A good example of a crossover operator that can be used with CGE is the operator introduced by Stanley [8]. This operator aligns two genomes encoding different network topologies, and creates a new structure that combines the overlapping parts of the two parents as well as their different parts. The *id*'s stored in vertex and jumper genes and the labels can be used to align genomes.

2.1.5 Set of Valid Phenotypes $\mathcal{P}_{CGE}$ and Development Function $\mathcal{D}$

Each *valid phenotype* $p \in \mathcal{P}_{CGE}$ is a directed graph structure $p = (V, E)$, consisting of a set of vertices V and a set of (directed) edges E. The set of edges E is partitioned into two subsets: the set of forward connections E_F, and the set of recurrent connections E_R. For each $p = (V, E_F \cup E_R) \in \mathcal{P}_{CGE}$, the subgraph $p_F = (V, E_F)$ is always a directed acyclic graph (DAG). A vertex with no incoming edge is called an *input vertex*.

The *development function* $\mathcal{D} : \mathcal{G} \longrightarrow \mathcal{P}_{CGE}$ creates for every valid genotype $g = [x_1, ..., x_N] \in \mathcal{G}$ a corresponding phenotype $p \in \mathcal{P}_{CGE}$. For each $x_i \in \mathcal{V}$, p contains exactly one vertex $\hat{x}_i$, which has the same identity number as x_i. The set of recurrent connections E_R in p has a direct one-to-one correspondence with $\mathcal{J}_\mathcal{R}$: for each recurrent jumper gene x_i, the vertex whose id equals x_i's source vertex id is connected via an edge $e \in E_R$ to the vertex in p whose id is equal to that of $parent(x_i)$. In the same way, for each $x_i \in \mathcal{J}_\mathcal{F}$ there is a corresponding forward connection in E_F. For each $x_i \in \mathcal{I}$, E_F contains a forward connection from the vertex having x_i's label as id[4] to the vertex with the same id as $parent(x_i)$. Additionally, there are connections in E_F that are not explicitly represented in g. Each non-output vertex gene $x_i \in \mathcal{V}_{no}$ has an *implicit forward connection* with its parent vertex $parent(x_i)$. The closure property of $\mathcal{D}$, i.e. $\forall g \in \mathcal{G}, \mathcal{D}(g) \in \mathcal{P}_{CGE}$, will be proven in Section 2.2. An example of a direct encoding of the neural network shown in Figure 1 is given in Table 1.

2.1.6 The Evaluation Function $\mathcal{E}$

The evaluation function evaluates the developed phenotype $p \in \mathcal{P}_{CGE}$. $\mathcal{D}(g)$ can be interpreted as an artificial neu-

---

[4]There may be several labels possessing the same value for different input vertices, but for each unique label, there exists only one vertex in p whose id corresponds to that label.

ral network in the following way: all input vertices of $\mathcal{D}(g)$ are considered as inputs of the network and all other vertices as neuron nodes. The vertices corresponding to an output vertex gene in g are the output neurons of the network. Each forward and recurrent connection causes the output of its source neuron to be treated as an input of its target neuron. Each artificial neuron stores its last output $o_i(t-1)$. Let $\hat{x}_i$ be a neuron with incoming forward connections from the inputs $\hat{x}_1, ..., \hat{x}_k$ and the neurons $\hat{x}_{k+1}, ..., \hat{x}_l$, and the incoming recurrent connections from neurons $\hat{x}_{l+1}, ..., \hat{x}_m$. For an arbitrarily chosen transfer function φ, the current output $o_i(t)$ of the neuron $\hat{x}_i$ is computed using

$$o_i(t) = \varphi(\sum_{j=1}^{k} w_j I_j(t) + \sum_{j=k+1}^{l} w_j o_j(t) + \sum_{j=l+1}^{m} w_j o_j(t-1)),$$
(5)

where the values of $I_j(t)$ represent the inputs of the neural network. If the network has p inputs and q output neurons, we can define $\mathcal{E}$ as a function which takes the phenotype $\mathcal{D}(g)$ and p real input values, and produces q real output values, i.e. $\mathcal{E} : \mathcal{P}_{CGE} \times \mathbb{R}^p \longrightarrow \mathbb{R}^q$.

It is also possible to implicitly evaluate the encoded phenotype without the need to decode this phenotype from the genotype via $\mathcal{D}$. In order to do this, we traverse the genes in the CGE encoding in reverse (i.e. from right to left) and evaluate it according to the Reverse Polish Notation (RPN) scheme, where the operands (input genes and jumper genes) come before the operators (vertex genes). Using a stack, we can compute the output by moving from right to left through the genotype. If the current gene is an input gene, we push its current value and the weight associated with it onto the stack. If the current gene x_i is a vertex gene, we pop $d_{in}(x_i)$ values with their associated weights from the stack and push the vertex's computed result and its associated weight back onto the stack. If the current gene is a recurrent jumper gene, we retrieve the *previously stored value* of the source vertex gene whose identification number is the same as that of the jumper gene. We then weight this value by the jumper-gene weight, and push it onto the stack. If the current node is a forward jumper node, we first copy the subgenome starting from a vertex gene whose identification number is the same as that of the forward jumper gene. We then compute the response of the subgenome in the same way as we would for a CGE genotype. Finally, we weight the computation result by the forward jumper gene weight, and push it onto the stack. After traversing the genome from right to left completely, we pop the resulting values from the stack.

2.2   Properties of the Encoding

In this section we give a formal proof of the properties of CGE introduced in Section 1. We prove the closure property of the development function $\mathcal{D}$, and the completeness of $\mathcal{G}$ with respect to $\mathcal{D}$. Furthermore, we show that $\mathcal{G}$ is closed under the genetic operators introduced in Section 2.1.4. In addition to this, we will show that $\mathcal{D}(g)$ can be evaluated without ending up in an infinite loop. Although these proofs deal with the case where CGE is used as a direct encoding method, similar results can also be easily proven for the indirect encoding case. We will begin with some propositions that lay a foundation for the subsequent proofs.

Proposition 1. For a valid genotype $g \in \mathcal{G}$, the number of expected inputs by all vertex genes $\sum\limits_{x_i \in g \wedge x_i \in \mathcal{V}} d_{in}(x_i)$ is equal to the number of non-output vertex genes, i.e. $|(\mathcal{V}_{no} \cup \mathcal{I} \cup \mathcal{J}_F \cup \mathcal{J}_R)|$.

PROOF. Assume $\sum\limits_{x_i \in g \wedge x_i \in \mathcal{V}} d_{in}(x_i) > |(\mathcal{V}_{no} \cup \mathcal{I} \cup \mathcal{J}_F \cup \mathcal{J}_R)|$. Since $s_{N+1} = |\mathcal{V}_o| + |(\mathcal{V}_{no} \cup \mathcal{I} \cup \mathcal{J}_F \cup \mathcal{J}_R)| - \sum\limits_{x_i \in g \wedge x_i \in \mathcal{V}} d_{in}(x_i)$, it follows that $s_{N+1} < |\mathcal{V}_o|$. Because of the definition of V_o, there have to be at least $|\mathcal{V}_o|$ indices of s, where s reaches a new maximum. Since $s_1 = 0$ and $s_i \in \mathbb{Z}$, this implies, that there exists an index $i \leq N$ with $s_i \geq |\mathcal{V}_o|$. This further implies that $s_i > s_{N+1}$, which is in contradiction with Criterion 4 for a valid genotype.

Now assume that $\sum\limits_{x_i \in g \wedge x_i \in \mathcal{V}} d_{in}(x_i) < |(\mathcal{V}_{no} \cup \mathcal{I} \cup \mathcal{J}_F \cup \mathcal{J}_R)|$. This implies $s_{N+1} > |\mathcal{V}_o|$. Since $s_1 = 0$ and $s_i - s_{i-1} \leq 1 \, \forall \, 0 < i < N$, it follows that there exists an $i < N$ with $s_i > s_j \, \forall \, j < i$ which is not an output vertex gene (otherwise s_{N+1} would be equal to $|\mathcal{V}_o|$). This contradicts criterion 5 for a valid genotype.

Because both assumptions lead to contradictions, the proposition must be valid. $\square$

Proposition 2. For $g = [x_1, ..., x_N] \in \mathcal{G}$ with N genes, s_{N+1} is equal to the number of output vertex genes $|\mathcal{V}_o|$ in g.

PROOF. For s_{N+1} the following equation is valid:

$$s_{N+1} = \sum_{x_i \in g} v(x_i) = \sum_{x_i \in g \wedge x_i \notin \mathcal{V}} 1 + \sum_{x_i \in g \wedge x_i \in \mathcal{V}} (1 - d_{in}(x_i))$$
$$= \sum_{x_i \in g} 1 - \sum_{x_i \in g \wedge x_i \in \mathcal{V}} d_{in}(x_i) = N - \sum_{x_i \in g \wedge x_i \in \mathcal{V}} d_{in}(x_i)$$

Because of Proposition 1, it is true that

$$\sum_{x_i \in g \wedge x_i \in \mathcal{V}} d_{in}(x_i) = |(\mathcal{V}_{no} \cup \mathcal{I} \cup \mathcal{J}_F \cup \mathcal{J}_R)| = N - |\mathcal{V}_o|.$$

Therefore, s_{N+1} is equal to the number of output vertex genes $|\mathcal{V}_o|$. $\square$

Proposition 3. For a subgenome $g_{l,m} = [x_l, x_{l+1}, \ldots, x_{l+m-1}]$, the sum $s_{l,m} = \sum\limits_{i=l}^{l+m-1} v(x_i)$ is equal to one.

PROOF. The sum $s_{l,m} = \sum\limits_{i=l}^{l+m-1} v(x_i)$ can be written as

$$s_{l,m} = \sum_{x_i \in g_{l,m}} 1 - \sum_{x_i \in g_{l,m} \wedge x_i \in \mathcal{V}} d_{in}(x_i)$$
$$= 1 + \sum_{x_i \in g_{l+1,m}} 1 - \sum_{x_i \in g_{l,m} \wedge x_i \in \mathcal{V}} d_{in}(x_i)$$

Since $g_{l,m}$ is a subgenome, the number of expected inputs in $g_{l,m}$ is equal to the number of produced outputs in $g_{l+1,m}$, and therefore $\sum\limits_{x_i \in g_{l+1,m}} 1 = \sum\limits_{x_i \in g_{l,m} \wedge x_i \in \mathcal{V}} d_{in}(x_i)$ and $s_{l,m} = 1$. $\square$

Subgenomes are an important concept in CGE, because they make it possible to treat developed phenotype structures as a composition of phenotype substructures that correspond to the subgenomes. This allows for the independent evolution of modules that can be used as part of larger structures.

Termination of an Evaluation Strategy

There is an evaluation strategy, which evaluates $\mathcal{E}(\mathcal{D}(g), x)$ for all $g \in \mathcal{G}, x \in \mathbb{R}^n$ without running into an infinite loop.

PROOF. Let $p = \mathcal{D}(g)$ be an ANN with n output neurons. The evaluation of $\mathcal{E}$ requires the computation of the present values of the n output neurons. The value of an output neuron $\hat{v}_i$ can be computed using Equation 5 as

$$o_i(t) = \varphi\left(\sum_{j=1}^{k} w_j I_j(t) + \sum_{j=k+1}^{l} w_j o_j(t) + \sum_{j=l+1}^{m} w_j o_j(t-1)\right)$$

The values of $I_j(t)$ represent the inputs of the neural network, and the values $o_j(t-1)$ are stored in the neuron V_j. Therefore, the computation of $o_i(t)$ requires only the prior computation of $o_j(t)$ for $j \in k+1, ..., l$. These values can be computed by recursively exploiting Equation 5. The only way in which this evaluation strategy can get into an infinite loop is if the computation of $o_k(t)$ for a neuron V_k involves the value of $o_k(t)$ itself. Assuming that this were the case, it would imply that for a $p = (V, E_F \cup E_R) \in \mathcal{P}_{CGE}$, $p_F = (V, E_F)$ contains a cycle (i.e. p_F is not a DAG). From this it follows that $\mathcal{D}(g) \notin \mathcal{P}_{CGE}$, which is in contradiction with the closure of $\mathcal{D}$. Therefore, this evaluation strategy will not run into an infinite loop. $\square$

Completeness of $\mathcal{G}$ with respect to $\mathcal{D}$.

Every possible phenotype can be represented by a genotype, i.e. $\mathcal{D}$ is surjective: $\forall p \in \mathcal{P}_{CGE} \; \exists g \in \mathcal{G} : \mathcal{D}(g) = p$

PROOF. Choose any arbitrary $p = (V, E_F \cup E_R) \in \mathcal{P}_{CGE}$. We want to construct a genotype $g \in \mathcal{G}$, where $\mathcal{D}(g) = p$. First we construct g_F with $\mathcal{D}(g_F) = p_F = (V, E_F)$. Since p_F is a DAG, it can always be transformed into a forest of rooted trees by removing edges from it[5]. The number of trees is the same as the number of outputs of the phenotype network. For each tree we can construct a gene sequence by traversing the tree in a depth-first manner[6]. For each traversed vertex $\hat{v}$, we store the id of the vertex and the number of incoming edges of $\hat{v}$ in p as $d_{in}(v)$. We then concatenate all gene sequences in an arbitrary order. After this, all edges which have been removed from the DAG while transforming it into a forest are inserted into the concatenated gene sequence. If one of those edges goes from a vertex $\hat{v}_j$ (with id j) to a vertex $\hat{v}_k$ (with id k), and $\hat{v}_j$ is an input vertex, we insert an input gene (with label j) after the vertex gene (with id k) into the gene sequence. If $\hat{v}_j$ is not an input vertex, we insert a forward jumper gene (with source vertex id j) after the vertex gene with id k. After constructing g_F, we add all edges in E_R into g_F in a similar fashion. For an edge in E_R going from vertex $\hat{v}_j$ to $\hat{v}_k$, we insert a recurrent jumper gene (with source vertex id j) after the vertex gene with id k. Because of g's construction, $D(g) = p$ is true. An example of this construction is given in Table 2.

---

[5]There are usually many different sets of edges that can be selected for removal, but the described procedure will result in a valid $g \in G$ for only a few of them (see below in the proof of criterion 3).

[6]By performing a depth-first traversal of the tree, it is ensured that all edges of the tree are contained in the genotype as implicit forward connections.

Table 2: The table summarizes the construction of a genotype for the phenotype shown in Figure 1. The type of a gene is indicated by the capital letters and the id/source id/label is denoted by the subscripts. In step I, the tree (which results when removing the forward connections x_5, x_6, x_8 and x_9, and the recurrent connection x_{11} from the phenotype) is represented. In step II, the omitted forward connections are inserted (either as input or as forward jumper gene), and in step III the only recurrent connection x_{11} is added. The resulting genotype is not the same as that shown in Table 1, since the order of the genes is not uniquely defined (i.e. $\mathcal{D}$ is not injective).

Step I	V_0	V_1		V_3		I_x	V_2				I_y
Step II	V_0	V_1	I_y	V_3	I_y	I_x	V_2		I_x	JF_3	I_y
Step III	V_0	V_1	I_y	V_3	I_y	I_x	V_2	JR_0	I_x	JF_3	I_y

We now show that the constructed g is a valid genotype, and must therefore pass the test against the five validity criteria. Criterion 1 is fulfilled because only vertices with at least one child vertex (in the extracted forest) are mapped onto a vertex gene and thus each vertex gene has at least one incoming implicit forward connection ($d_{in} \geq 1$). Criterion 2 is also satisfied because there cannot be a closed chain of forward jumper genes. Each forward jumper gene corresponds to an edge in a DAG which by definition is acyclic. Criterion 3 is fulfilled if the correct edges are removed when extracting a forest f from the DAG p_F. All edges in f result in implicit forward jumpers in g and all removed edges from p_F result in (explicit) forward jumper genes. For each vertex $v \in V$, we choose the edge to be an implicit forward connection in such a way that the depth of the vertex gene in g corresponding to v is maximized. In this case, criterion 3 is satisfied.

Furthermore, criterion 5 is satisfied: Assume that $\exists k < N : parent(x_k) = \emptyset \wedge x_k \notin \mathcal{V}$. This implies that $x_k \in \mathcal{I} \cup \mathcal{J_F} \cup \mathcal{J_R}$. For every $j < k$ and $x_j \in \mathcal{V}$, $s_j < s_k \Rightarrow \sum_{i=j}^{k-1} v(x_i) > 0$. This means that the subsequence $[x_j, ..., x_{k-1}]$ contains a subgenome starting with x_j. Therefore, there is no $x_j \in \mathcal{V}$ to which x_k can be connected. Because of the construction of g, there cannot be any input, forward jumper, or recurrent jumper which is unconnected (since each of them has a corresponding element in p that is connected to a vertex). Hence the original assumption is wrong, and g satisfies criterion 5. The constructed g satisfies also the fourth criterion: assume that $\exists k \in \{1, ...N\} : s_k \geq s_{N+1}$. Choose the minimum k which possesses this property. Since criterion 5 is satisfied (see above) it follows that $x_k \in \mathcal{V}$. Consider now the subsequence $[x_k, ...x_N]$. Because $s_{N+1} \leq s_K$, it follows that $\sum_{j=k}^{N} v(x_j) \leq 0$. This means that this subsequence of the genotype contains less produced outputs than the number of inputs expected by its vertex genes. Due to the construction of g, the number of expected inputs is always equal to the number of produced outputs. Therefore, the assumption is wrong and criterion 4 is satisfied. $\square$

Closure of $\mathcal{D}$.
The development function maps every valid genotype to a valid phenotype: $\forall g \in \mathcal{G} : \mathcal{D}(g) \in \mathcal{P}_{CGE}$.

PROOF. Given any arbitrary $g \in \mathcal{G}$, let $p = \mathcal{D}(g)$. The definition of $\mathcal{D}$ implies that p is a graph structure. In order to prove that $p = (V, E_F \cup E_R) \in \mathcal{P}_{CGE}$, it is sufficient to show that $p_F = (V, E_F)$ is a directed acyclic graph (DAG), i.e. that there are no cycles in p_F. Because of the second genotype validity criterion, the forward jumper connections alone, which are explicitly represented in g, cannot cause a cycle in p_F. We now show that no implicit forward connection can be part of a cycle. Assume that there exists an implicit forward connection from a vertex $\hat{v}_i$ to a vertex $\hat{v}_j$, which is part of a cycle $c = [\hat{v}_l, \ldots, \hat{v}_i, \hat{v}_j, \ldots, \hat{v}_l]$. Because of the definition of implicit forward connections, $depth(\hat{v}_i) = depth(\hat{v}_j) + 1$ (the depth of a vertex $\hat{v}$ is set to the depth of its preimage $\mathcal{D}^{-1}(\hat{v})$ in $\mathcal{G}$). Because cycles have identical start and end vertices with the same depth, the cycle c has to include an implicit or explicit forward connection whose source vertex has a depth less than the depth of its target vertex. Because of the third genotype-validity criterion, this is never the case for explicit forward connections. The same is true for implicit forward connections, since they always connect from a source vertex with depth d to a target vertex with depth $d - 1$. Therefore the assumption is wrong and there is no cycle in p_F. $\square$

Closure of $\mathcal{G}$ under Mutation Operators.
The set of genotypes $\mathcal{G}$ is closed under the mutation operators $\mathcal{PA}(g) \in \mathcal{G}$ and $\mathcal{ST}(g) \in \mathcal{G} \ \forall g \in \mathcal{G}$.

PROOF. Let us first prove that $\mathcal{PA}(g) \in \mathcal{G} \ \forall g \in \mathcal{G}$. Since the parametric mutation operator changes only the parameters included in the genes (such as weights, etc.), and none of the five criteria for a valid genotype is influenced by these parameters, $\mathcal{PA}(g) \in \mathcal{G}$ is true because $g \in \mathcal{G}$.

Next, we show that $\mathcal{ST}(g) \in \mathcal{G} \ \forall g \in \mathcal{G}$. If $\mathcal{ST}$ inserts a recurrent jumper gene x_k after a vertex gene x_i, the number of inputs of the vertex gene x_i is increased by 1. Therefore, $s_k^{new} = s_{i+1}^{old} - 1$. Since $v(x_k) = 1$, $s_{k+1}^{new} = s_k^{new} + 1 = s_{i+1}^{old}$. Because $s_{i+1}^{old} < s_{N+1}$, both s_k^{new} and s_{k+1}^{new} are smaller than s_{N+1}, and hence criterion 4 is satisfied for $\mathcal{ST}(g)$. Furthermore, x_k could only violate Criterion 5 if $\forall j < k = i + 1 : s_j < s_k^{new}$. This is not the case, since x_i is a vertex gene with at least one input (the newly added recurrent jumper) and therefore $v(x_i) \leq 0$ and $s_i \geq s_k^{new}$. Criterion 1 is satisfied because no new vertex genes are introduced and no jumper and input genes are removed, and therefore $d_{in}(x) \geq 1$ remains true. Criterion 2 is satisfied because no new forward jumpers are inserted. As a result, the depths of vertices do not change, and criterion 3 is also satisfied.

If $\mathcal{ST}$ adds an entire subnetwork $[x_k, \ldots, x_{k+M}]$ with $M+1$ genes after a vertex gene x_i, $\mathcal{ST}(g) \in \mathcal{G}$ is also satisfied: since $\sum_{j=k}^{M} v(x_j) = 1$, all the considerations concerning Criteria 4 and 5 from above remain the same. Criterion 1 is satisfied because there is only one newly introduced vertex gene, x_k, with $d_{in}(x_k) = M > 0$. Furthermore, Criterion 2 cannot be violated, because each newly added forward jumper gene has x_k as its target vertex. Since x_k is newly introduced, there is no forward jumper with x_k as its source vertex. Therefore, the new forward jumper cannot be part of a closed chain. Criterion 3 is satisfied because the source

vertices of all newly introduced forward jumpers do not have a depth less than the depth of their target vertices. $\square$

Similarly, one can prove the closure of $\mathcal{G}$ under the crossover operator.

3. CGE FOR INDIRECT ENCODING

We will use the edge encoding of Luke and Spector [5] to illustrate how CGE can be used for the indirect encoding of network structures. The authors present an alternative to Gruau's cellular encoding technique [3] for evolving graph and network structures via genetic programming. According to the authors, edge encoding differs from cellular encoding in the following ways: (1) Edge encoding grows a graph by modifying the *edges* in the graph, whereas cellular encoding grows a graph by modifying the *nodes* in the graph. (2) Edge encoding traverses its chromosome in a *depth-first* manner, whereas cellular encoding traverses its chromosome in *breadth-first* manner. (3) The leaf nodes in an edge-encoded chromosome represent unique *edges* in the resultant graph, while in cellular encoding, leaf nodes represent *nodes* in the resultant graph.

The developmental process when using CGE with an edge encoding is analogous to the implicit evaluation of a genotype in the direct encoding case. In contrast to the the implicit evaluation process in the direct encoding case presented in Section 2, while developing an edge encoding phenotype, the CGE genotype will be executed from the root node towards leaf nodes.

Edge Encoding Operators. An edge encoding operator (i.e. a gene in the chromosome) always receives a single edge as an input from its parent, as well as a (possibly empty) stack of graph nodes. This permits a depth-first tree traversal, which better preserves the semantics of building blocks after the application of a crossover operator. Luke and Spector argue in [5] that the development operators in Table 3 are sufficient for describing the topology of all connected graphs of two or more nodes. The *Double* operator duplicates an edge connection between two nodes of the developing graph structure. The *Loop* operator takes an edge e_{ab} (connecting from node a to b), and creates a new edge e_{bb} (from node b to itself). The *Reverse* operator takes an edge e_{ab}, and replaces it with an edge e_{ba}, in which the direction is reversed. *Cut* eliminates the edge that is passed to it. *Push* modifies the node-stack (NS), adding to it a new node. The *Attach* operator takes an edge e_{ab}, pops a node c off of the NS, and creates two new edges, e_{ac} and e_{bc}. The last two node types listed in Table 3 do not operate on edges, but only affect the execution-flow during the developmental process. The *Input* node accepts an edge as its input, and does nothing with it, terminating the modifications to be made on this edge. Finally, the *Jumper* node allows for recursive calls to subgenomes.

CGE Encapsulation. In order to encapsulate edge encoding with CGE, we make use of vertex genes, input genes, and forward jumper genes (recursive jumper genes are not needed). Each vertex gene in the CGE genotype is assigned

---

[7]Jumper connections act differently depending on how many times they have iterated. If the recursion depth has been reached, a jumper node pops from the ES and NSS. Otherwise it does nothing, passing execution to its source node.

Table 3: Operator types used in edge encoding for generating graphs, the number of output edges they generate, and the operations performed on the different stacks. Pop operations are represented as ⇑, and push as ⇓. The upper group are *development* operators, and the lower are *execution* operators.

| Operator Type	Outputs	ES Operations	NS Operations	NSS Operations
Double	2	$\Uparrow (e_{ab})$, $\Downarrow (e_{ab}, e_{ab})$	-	⇑⇓⇓
Loop	2	$\Uparrow (e_{ab})$, $\Downarrow (e_{bb}, e_{ab})$	-	⇑⇓⇓
Reverse	1	$\Uparrow (e_{ab})$, $\Downarrow (e_{ba})$	-	⇑⇓
Cut	0	$\Uparrow (e_{ab})$	-	⇑
Push	1	-	$\Downarrow (n_{new})$	⇑⇓
Attach	3	$\Uparrow (e_{ab})$, $\Downarrow$ (e_{ax}, e_{bx}, e_{ab})	⇑	⇑⇓⇓⇓
Input	0	$\Uparrow (e_{ab})$	-	⇑⇓
$Jumper'$	0	- / $\Uparrow (e_{ab})$	-	- / ⇑

Table 4: An example of a CGE genotype for indirect encoding of graphs and networks using edge encoding. *P*, *D*, *A* and *L* indicate the *Push*, *Double*, *Attach*, and *Loop* operators, respectively. *I* and *JF* indicate Input and Forward-Jumper genes.

gene	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
allele	V	V	V	I	JF	V	I	I	I
oper. type	P	D	A	-	-	L	-	-	-
id	0	1	2	-	-	3	-	-	-
source	-	-	-	-	1	-	-	-	-
recursion depth	-	-	-	-	1	-	-	-	-
d_{in}	1	2	3	-	-	2	-	-	-
v	0	-1	-2	1	1	-1	1	1	1
s	0	0	-1	-3	-2	-1	-2	-1	0
parent	∅	0	1	2	2	2	3	3	1
depth	0	1	2	3	3	3	4	4	2

an *operator-type* attribute which represents one of the development operators shown in Table 3. In the case of edge encoding, a forward jumper gene represents a recursive call to the subgenome that begins with the jumper gene's source vertex gene. The *recursion depth* is stored as an attribute of the jumper gene. Recursion of this sort allows for the development of repetitive parts of a phenotype network, and results in a compact genotype whose decoding algorithm terminates. The input genes terminate the further development of the edges they receive as inputs. An example of a genotype is shown in Table 4.

Development of Networks. One key advantage of using CGE for representing other encodings is that it allows *implicit development* to be performed on it. In other words, there is no need to first decode the genotype before developing it. To facilitate this development, three separate stacks are used. The Edge Stack (ES) is responsible for storing edges, and passing them from parent nodes to child nodes. The Node Stack (NS) stores a list of nodes that are used by the *Attach* operator and the *Push* operator. The Node-

Figure 2: The first three steps in the developmental process and a fully developed phenotype network for the example genotype shown in Table 4. Note that ES represents the stack of graph edges and NS represents the stack of graph nodes.

(a) After *Push* (b) After *Double*

(c) After *Attach* (d) Final network
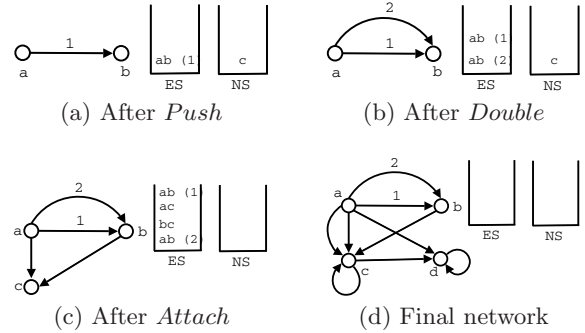
Stack Stack (NSS) stores the NS state at various points during the decoding of the chromosome. Table 3 shows the sequence of operations performed on each stack by the different operators. Important to note is that, in general, the operators first pop a NS off of the NSS, then make any needed changes to this NS, and finally push this modified NS onto the NSS zero or more times.

The implicit development function $\mathcal{D}_{im}$, which directly decodes the genotype into the final network, carries out the following important steps: (1) Initialize the stack of graph edges with a single edge. (2) Start from the leftmost gene of the genotype, and move towards the right while developing the phenotype. (3) If the current gene is a *vertex* gene, pop an edge off the stack of graph edges and push n edges, where n is the number of operator outputs shown in Table 3. If the current gene is an *input* gene, pop an edge from the stack of graph edges and terminate modifications for the popped edge. If the current gene is a *forward jumper* gene, either (a) copy the subgenome whose root node is the jumper gene's source, and execute this subgenome, or (b) pop an edge from the ES and a node-stack from the NSS. Figure 2 shows the first three steps in the developmental process and the final phenotype network of the example genotype shown in Table 4.

4. COMPARISON TO OTHER GENETIC ENCODINGS

In this section we will give a review of *representative* works and compare other existing genetic encodings with CGE. For a detailed review of the works see [10]. Angeline et al. developed a system called GNARL (GeNeralized Acquisition of Recurrent Links) which uses only structural mutation of the topology, and parametric mutations of the weights as genetic search operators [1]. The main problem with this method is that genomes may end up in many extraneous disconnected structures that have no contribution to the solution. The Neuroevolution of Augmenting Topologies (NEAT) [8] evolves both the structure and weights of neural networks. It starts with networks of minimal structures and increases their complexity along the evolution path. The algorithm keeps track of the historical origin of every gene that is introduced through structural mutation. This history is used

Table 5: Comparison between some representative genetic encodings and CGE. G, N, CE, and EE stand for GNARL, NEAT, Cellular Encoding, and Edge Encoding, respectively.

Property	G	N	CE	EE	CGE
Completeness	√	√	√	√	√
Closure		√	√	√	√
Modularity			√	√	√
Support direct encoding	√	√			√
Support indirect encoding			√	√	√
Evaluation without decoding (direct encoding case)					√

by a specially designed crossover operator to match genomes which encode different network topologies. Unlike GNARL, NEAT does not use self-adaptation of mutation step-sizes. Instead, each connection weight is perturbed with a fixed probability by adding a floating point number chosen from a uniform distribution of positive and negative values.

Gruau's Cellular Encoding (CE) method is a language for local graph transformations that controls the division of cells which grow into an artificial neural network [3]. The genetic representations in CE are compact because genes can be reused several times during the development of the network and this saves space in the genome since not every connection and node needs to be explicitly specified in the genome. Defining a crossover operator for CE is still difficult, and it is not easy to analyze how crossover affects subfunctions in CE since they are not explicitly represented.

A comparison between some representative genetic encodings developed so far and CGE with respect to their completeness, closure, and modularity properties, as well as some additional features, is given in Table 5.

For the direct encoding case, the "evaluation without decoding" feature of CGE eliminates a step in the phenotype-development process that would otherwise require a significant amount of time, especially for large and complex phenotype networks. NEAT has been adapted to evolve modular networks in a system called Modular NEAT [7], but NEAT in its original form does not inherently support the evolution of modular networks.

5. CONCLUSIONS AND OUTLOOK

A flexible genetic encoding that is suitable for both direct and indirect genetic encoding of networks has been presented. We have shown that this encoding is both complete and closed. Additionally, we have illustrated how the encoding allows, in the direct encoding case, for a phenotype to be evaluated without the need to first decode it from the genotype. Moreover, because the encoding's genotypes can be seen as having several subgenomes, it inherently supports the evolution of modular networks in both direct and indirect encoding cases.

In the future, we will investigate the design of indirect encoding operators that can achieve compact representations and significantly reduce the search space. There is much work to be done in designing genetic operators. In particular, we would like to develop genetic operators whose

offspring remain in the locus of similarity to their parents in both structural and parametric spaces. We believe that more efficient evolution of complex structures would be facilitated by such operators. Furthermore, we will conduct experiments to empirically compare CGE with other encodings.

6. REFERENCES

[1] P. J. Angeline, G. M. Saunders, and J. B. Pollack. An evolutionary algorithm that constructs recurrent neural networks. *IEEE Transactions on Neural Networks*, 5:54–65, 1994.

[2] F. Gomez, J. Schmidhuber, and R. Miikkulainen. Efficient non-linear control through neuroevolution. In *Proceedings of the European Conference on Machine Learning (ECML 2006)*, 2006.

[3] F. Gruau. *Neural Network Synthesis Using Cellular Encoding and the Genetic Algorithm*. PhD thesis, Ecole Normale Superieure de Lyon, Laboratoire de l'Informatique du Parallelisme, France, January 1994.

[4] Y. Kassahun. *Towards a Unified Approach to Learning and Adaptation*. PhD thesis, Technical Report 0602, Institute of Computer Science and Applied Mathematics, Christian-Albrechts University, Kiel, Germany, February 2006.

[5] S. Luke and L. Spector. Evolving graphs and networks with edge encoding: Preliminary report. In *Late-breaking papers of Genetic Programming 1996*. Stanford, CA, 1996.

[6] S. Nolfi and D. Floreano. *Evolutionary Robotics. The Biology, Intelligence, and Technology of Self-Organizing Machines*. MIT Press, Massachusetts, London, 2000.

[7] J. Reisinger, K. O. Stanley, and R. Miikkulainen. Evolving reusable neural modules. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2004)*, pages 69–81, 2004.

[8] K. O. Stanley. *Efficient Evolution of Neural Networks through Complexification*. PhD thesis, Artificial Intelligence Laboratory. The University of Texas at Austin., Austin, USA, August 2004.

[9] M. E. Taylor, S. Whiteson, and P. Stone. Comparing evolutionary and temporal difference methods in a reinforcement learning domain. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1321–1328, New York, NY, USA, 2006. ACM Press.

[10] X. Yao. Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9):1423–1447, 1999.