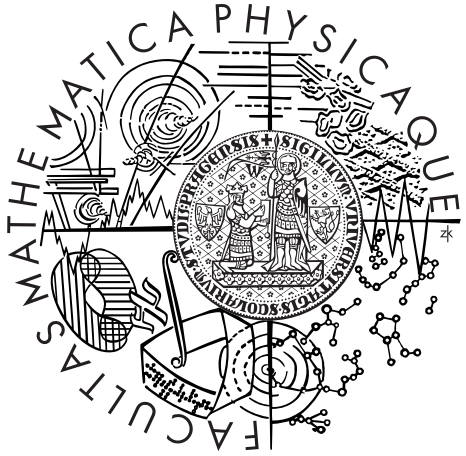
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Marek Leibl

Modular and Ontogenetic Evolution of Virtual Organisms

Department of Software and Computer Science Education

Supervisor of the master thesis: RNDr. František Mráz, CSc.
Study programme: Informatics
Specialization: Theoretical computer science

Prague 2014

I would like to express my sincere gratitude to my supervisor RNDr. František Mráz, CSc.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In ..... date ..... signature of the author

Název práce: Modular and ontogenetic evolution of virtual organisms

Autor: Marek Leibl

Katedra: Kabinet software a výuky informatiky

Vedoucí diplomové práce: RNDr. František Mráz, CSc., Kabinet software a výuky informatiky

Abstrakt: Růst vypočetního výkonu umožňuje v současné době automatizovat mnoho praktických problémů pomocí počítačových programů. Automatizace zahrnuje i problémy jako je návrh virtuálních chodích robotů, na základě kterých je v některých případech možné zkonstruovat reálného robota. Tato práce porovnává dva odlišné přístupy k vývoji virtuálních robotických organismů: umělou ontogenezi (artificial ontogeny), kdy virtuální organismus nejprve vyroste umělým ontogenetickým procesem, a přímé metody bez ontogenetického procesu. Dále je na základě srovnání různých přístupů navržena nová metoda pro vývoj virtuálních robotických organismů: Hypercube-based artificial ontogeny, která je kombinací umělé ontogeneze a Hypercube-based neuroevolution of augmenting topologies (HyperNEAT).

Klíčová slova: Evoluce, Virtuální organismy

Title: Modular and Ontogenetic Evolution of Virtual Organisms

Author: Marek Leibl

Department: Department of Software and Computer Science Education

Supervisor: RNDr. František Mráz, CSc., Department of Software and Computer Science Education

Abstract:

Increase of computational power and development of new methods in artificial intelligence allow these days many real-world problems to be solved automatically by a computer program without human interaction. This includes automatized design of walking robots in a physical virtual environment that can eventually result in construction of real robots. This work compares two different approaches to evolve virtual robotic organisms: artificial ontogeny, where the organism first grows using an artificial ontogenetic process, and more direct methods. Furthermore, it proposes a novel approach to evolve virtual robotic organisms: Hypercube-based artificial ontogeny that is combination of artificial ontogeny and Hypercube-based neuroevolution of augmenting topologies (HyperNEAT).

Keywords:

Evolution, Virtual organisms

Contents

1. Introduction

Increase of computational power and development of new methods in artificial intelligence allow these days many real-world problems to be solved automatically by a computer program without human interaction. This includes even some problems that require creativity such as art or design. Another example of such problems is designing walking robots capable of effective movement in various environments.

The evolutionary algorithms, method inspired by natural selection, are promising for automatized design of walking robots in a physical virtual environment that can eventually result in construction of real robots. They can often provide an effective solution that would not be discovered by a human. The methods for evolving walking robots can be also used for different applications that include designing physically realistic creatures in computer games or movies. Some more biologically accurate methods may provide further understanding of processes in biology.

Several approaches to evolve virtual robotic organisms have been introduced in previous works. The first important work that evolved virtual robots was Sims's Evolving virtual creatures [7] 1994 that inspired many later works. More recent works [2, 4] enhanced original method using advanced neuroevolution methods NEAT [1] and HyperNEAT [8]. More biologically accurate approach, artificial ontogeny was proposed by Bongard and Pfeifer [5] that developed method, where a mapping from genotype to the phenotype is realized using an artificial ontogenetic process.

This work combines results from several works related to evolving virtual organisms to develop two new methods that are compared through series of experiments. The first method, here referred as *Modular representation*, is based on bachelor's thesis of the author of this thesis [12], where morphology of organisms is represented using directed graphs as proposed by Sims and control system is evolved using HyperNEAT generative encoding inspired by Haasdijk, Rusu and Eiben [4]. The proposed method extends control system of the previous work enhancing generative encoding of neural networks, while method of evolving morphology is adopted from the previous work.

The second method, referred as *Hypercube-base artificial ontogeny* (*HyperAO*), shares method for evolving control system with the first proposed method. It introduces a novel concept to evolving morphology of virtual organisms that combines two different approaches: artificial ontogeny and HyperNEAT generative representation. The proposed method is designed to combine advantages of both approaches: (1) a developmental process provides possibility of additional adaptability, (2) HyperNEAT generative encoding reduces dimension of a search space and allows evolving regularities and most importantly (3) representation of morphological structure using artificial neural networks allows using various neuroevolution methods to evolve morphology of virtual organisms.

This work includes a distributed implementation of both proposed methods and experiments, where the proposed methods are compared. In the experiments, organisms are evolved for walking in a virtual physical 3D-world, where the quality of evolved organisms is measured as the walked distance for a fixed time. The

distributed solution for evaluating quality of organisms reduced duration of experiments from several weeks to a few days.

In summary, the primary goals of this thesis are: (1) compare developmental methods, artificial ontogeny in particular with other methods that evolve virtual organisms and (2) develop a new method that combines two different approaches: artificial ontogeny and Hypercube-based neuroevolution of augmenting topologies (HyperNEAT) to represent morphological structure using artificial neural networks that allows applying neuroevolution method to the morphology of virtual organism.

1.1 Structure of The Thesis

The thesis is divided into five main chapters. First, Chapter 2 overviews existing works and provides detailed description of selected methods that are related to the methods in the following chapters. Next, Chapter 3 describes the first proposed method based on the previous work of the author of this thesis. The chapter includes description of the control system that is common for both proposed methods. Chapter 4 summarizes results from previous works that are used as a motivation for the second proposed method, *HyperAO*, followed by a description of the method and formulation of hypotheses. Chapter 5 provides a description and analyzes distributed evaluation in a physical simulation and provides suggestions to effective distributed evaluation. Chapter 6 compares the proposed methods in series of experiments and shows that *HyperAO* is outperformed by the the first proposed method based on bachelor's thesis. Finally, Chapter 7 discuss results and argues that *HyperAO* still provides several advantages and suggest possible extensions.

2. Background

This chapter describes some of existing projects and methods for evolving virtual organisms. Note that different projects use different term for virtual organism (e.g. virtual creatures, robotic organisms, etc.), we will use these terms interchangeably in the following text (however sometimes the used term can suggest possible application in virtual reality, animations or robotics).

First, Section 2.1 gives an overview of some existing projects. The following sections focus deeper on methods used in these projects. Section 2.2 describes basic principles for evolving morphology and control system of virtual organism in simulated environment. Next, we describe in-depth the *NEAT* algorithm in Section 2.3 and the algorithm based on *NEAT* — the *HyperNEAT* algorithm in Section 2.4. Finally, Section 2.5 focuses on *artificial ontogeny* — combination of evolutionary algorithms with ontogenetic development and differential gene expression.

2.1 Overview of existing projects

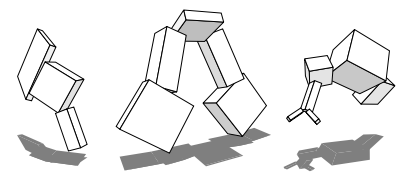
The primary inspiration for this work is Evolving virtual creatures (1994) [7] by Karl Sims, which inspired many other later projects. In this work virtual creatures are evolved to perform various tasks (e.g. walking, swimming, jumping, following light, etc.) in various simulated environments in three-dimensional physical world. Both control system and body is represented together as a directed graph, which is evolved using an evolutionary algorithm. Figure 2.1 [7] shows some examples of evolved creatures in Evolving virtual creatures.

More recent works [2, 4] use the NeuroEvolution of Augmenting Topologies (*NEAT*) algorithm (see Section 2.3) instead of or combined with original method by Sims. As suggested by Krčah, [2] *NEAT* is suitable for evolving neural network control system of virtual organisms. Krčah created modification of *NEAT* to evolve virtual creatures capable of performing tasks similar to Evolving virtual creatures. In his work [2] he showed that *NEAT* improves performance of the original method by Sims. Figure 2.2 shows some robotics organism evolved in his work.

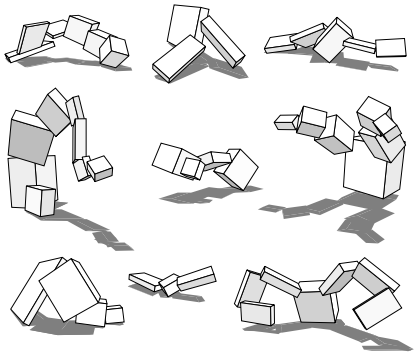
Haasdijk, Rusu and Eiben [4] evolved robot control for multi-robot organisms using the *HyperNEAT* algorithm (see Section 2.4). Here, each body part is considered as a single robot or agent, where each robot has its own controls system, which can communicate only with other directly connected robots (i.e. no central control system is used). Control system of all robot units is represented indirectly by a single neural network (*CPPN*), which generates control system for each robot based on its position. Therefore, control system of robots forming a multi-robot organism often have some regularities generated with *CPPN*.[1]

Bongard and Pfeifer [5] used an evolutionary algorithm combined with ontogenetic development and differential gene expression to evolve virtual organism

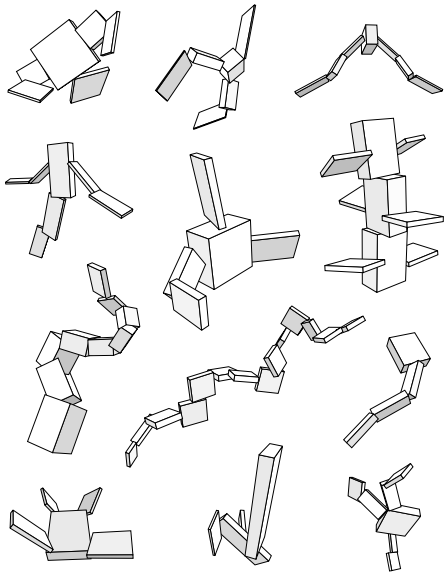
[1] For more details of regularities generated by *CPPN* see Section 2.4.



(a) Creatures evolved for jumping.



(b) Creatures evolved for walking



(c) Creatures evolved for swimming.

Figure 2.1: Examples of evolved virtual creatures in various environments by Sims [7].

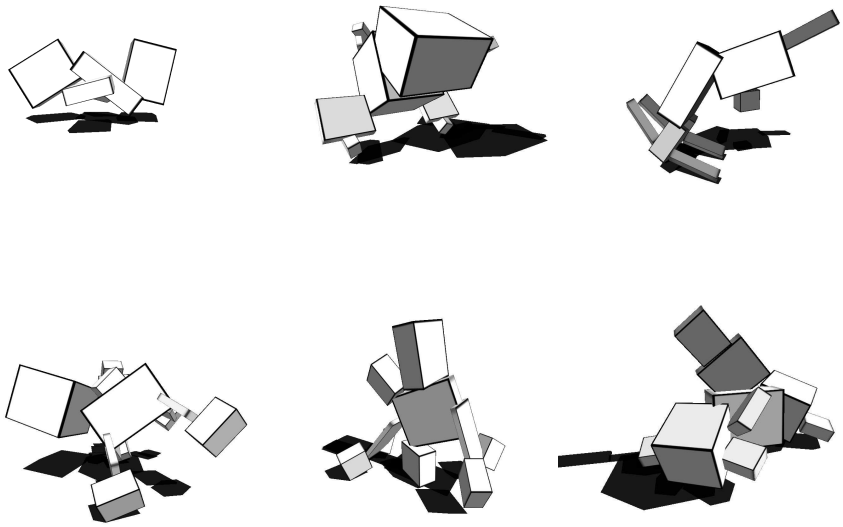


Figure 2.2: Examples of virtual creatures evolved for running by Krčah [2].

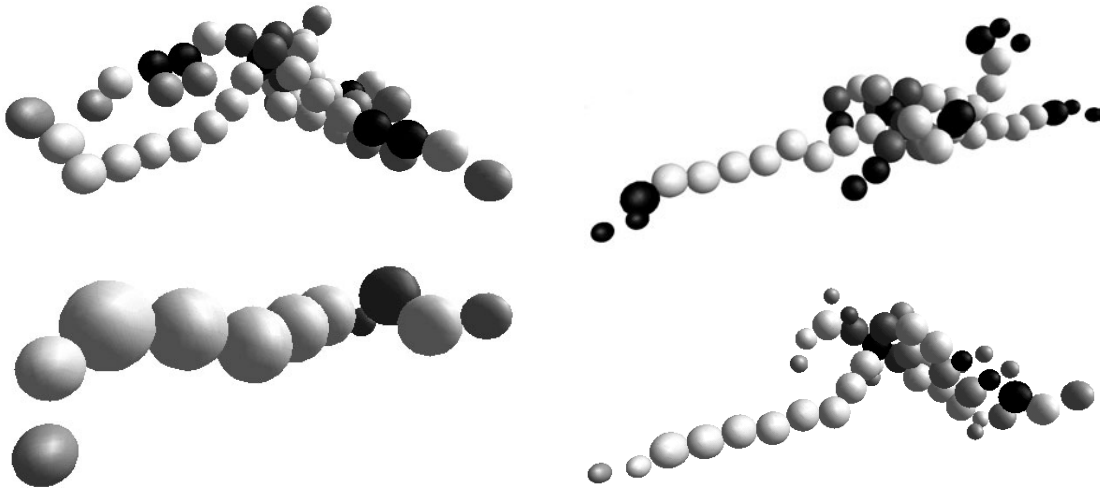


Figure 2.3: Examples of virtual organisms evolved for block-pushing task by Bongard and Pfeifer [5]. The block is not shown, but lies to the left of the organism. The color of spheres indicates presence of sensor and motor neurons. The white and light grey units have both sensors and motors, but the light grey units do not use their motors (either because there are no input connections to the motor neuron, or because there is no joint within this unit). The dark gray units indicate the presence of sensor neurons, but no motor neurons. The black units indicate the unit contains neither sensor nor motor neurons.

with ability to "grow" from a single unit. Unlike other mentioned methods, in their method refered as *artificial ontogeny* (AO) both morphology and control system is represented by a single genetic regulatory network (GRN).

GRN controls amount of generated *gene products* based on current amounts simulated in continuous manner. *Gene products* are simultaneously diffused inside each body parts and into other body parts. Some *gene product* stimulate or inhibit organism's growth or growth of some specific phenotype parts (e.g. sensors, effectors). The growth is simulated for a given time interval and when growth is finished, control system of the organism is started to perform a block-pushing task.

All these works evolve virtual organisms in 3D virtual physical world, but often with different motivation. Sims evolved creatures for animations, Krčah and Haasdijk, Rusu and Eiben focus more on applications in robotics, Bongard and Pfeifer focus on repeated structure and complexity of resulting phenotype. The following sections describe the methods used in these papers.

2.2 Evolving virtual organisms

In this section we describe original method introduced by Karl Sims [7] for evolving both morphology and robot control simultaneously using evolutionary algorithm. This method is later used as reference method and basic for some modifications or extensions.

First, Section 2.2.1 gives an overview of basic principles in evolving virtual organisms. Next, Section 2.2.2 describes robots body and its representation in genotype and finally, Section 2.2.3 focuses on robot control.

2.2.1 Introduction

Virtual organism consists of multiple interconnected rigid elements (modules). Modules can have sensors (e.g. touch sensor, light sensor, velocity sensor, etc.) and effectors (e.g. applying physical force to the module). Robot movement is realized by applying forces to its modules. Neural network control system is placed into each module to coordinate robot movement, additionally cental control system is used to coordinate local control systems.

Both robot body (morphology) and control system is encoded in the genotype. Robot morphology is represented by directed graph and control system by nested graph in this graph. Each node has instructions for creating corresponding module or neuron (for node in nested graph). Phenotype is then constructed by recursive visiting of connected nodes.

When the phenotype is complete, robot is placed into virtual environment designed for a particular task. The environment is usually simple virtual 3D physical world containing only objects needed for the current task. The robot receives a score (fitness) based on its behavior in virtual environment. Common tasks are robot locomotion in different environments, swimming, jumping, following light, etc. Simulation is typically restricted to some fixed time, while physical simulation is usually computationally-expensive.

2.2.2 Evolving morphology

Robot body is represented in the genotype as a directed (multi)graph with one starting node (*morphological graph*). Each node describes rigid part properties (e.g. shape) and each edge describes parameters of connection between parts (e.g. joint type, degrees of freedom, placement of child module etc.).

To create a phenotype from the graph, first, the starting node is used to create the root body part. Then, for each connected node in *morphological graph* new rigid part is created and connected to the organism. If the graph contains reachable directed cycles, the organism could have infinite body, therefore, limits for maximum number of body part created from a node are added (*recursive limit*).

Encoding an organism body as a graph allows to evolve organisms with repeated body parts, while nodes can be used multiple times. Resulting phenotype robot consists of three-dimensional rigid parts, which form a tree structure. Additionally each edge can have a *terminal-only* flag, which indicates that, edge will be used only when *recursive limit* is reached.[2] Figure 2.4 shows some sample genotypes and corresponding phenotypes.

Mutating *morphological graphs* includes (1) structural mutations (e.g. adding or removing nodes and connections, reconnecting nodes) of the graph and (2) uniform mutation of parameters of each node and connection. Boolean parameters reverts their values, scalar parameters add a small number from a normal distribution when mutation occurs.

There are two alternative ways of mating – *crossover* and *grafting*. When mating is applied, one of them is chosen randomly (with the same probability).

[2]This allows to evolve for instance structures similar to a human arm.

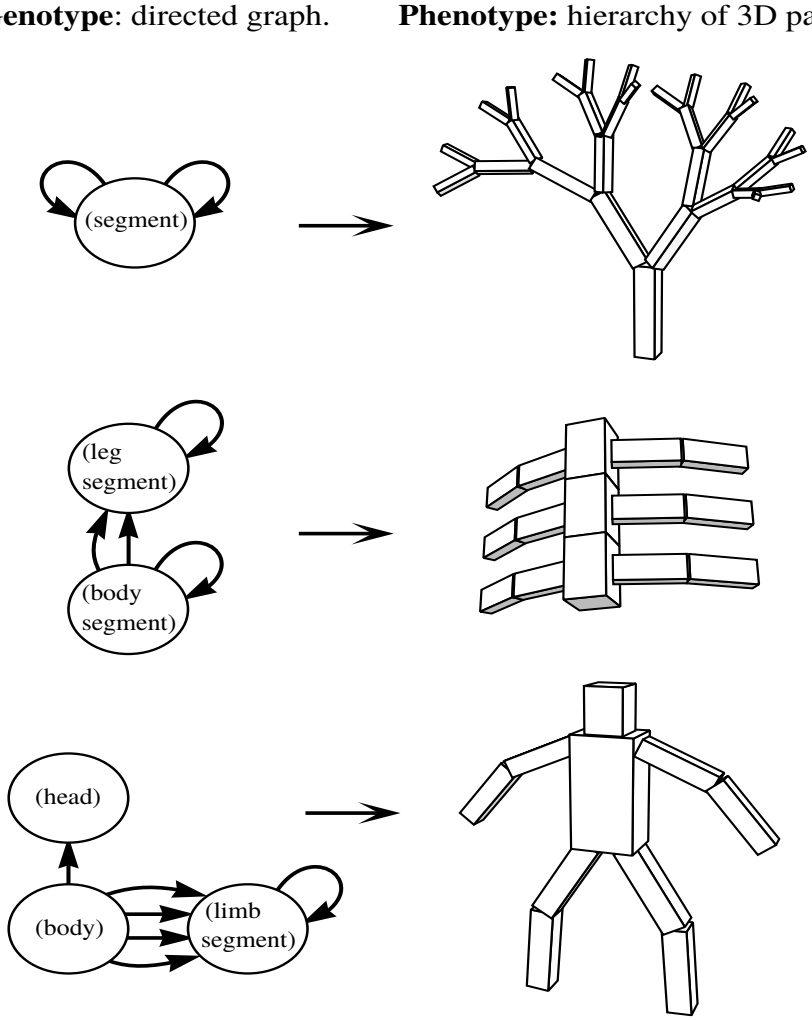


Figure 2.4: Designed examples of morphological graphs and corresponding morphologies. Paremeters of body parts and control system are not shown. Texts in the nodes are not included in the genotype, they only suggest possible interpretation of the body part type.

When the first mating type, *crossover*, was chosen, nodes of the two parents are first aligned in a row. Then, one or more crossover points are made to switch the source, which is copied to the offspring.[3] All nodes and connection reachable from the copied nodes are then also copied (i.e. the transitive closure is made). Finally, if some part of the graph becomes unreachable from starting node, it is removed.

Second mating type, *grafting* is similar to a mutation. The parent with higher fitness is copied into an offspring. Next, random connection is chosen and reconnected to a random node of the second parent. All nodes and connections that become reachable are also copied and those, which becomes unreachable are removed in the same way as in *crossover*.

Note that (unlike crossover in GA) only one child is produced during both types of mating. After mating, mutation with lower magnitude is applied to the *offspring*. Example of *crossover* and *mating* is shown in Figure 2.5.

[3]This approach is similar to crossover in genetic algorithms.

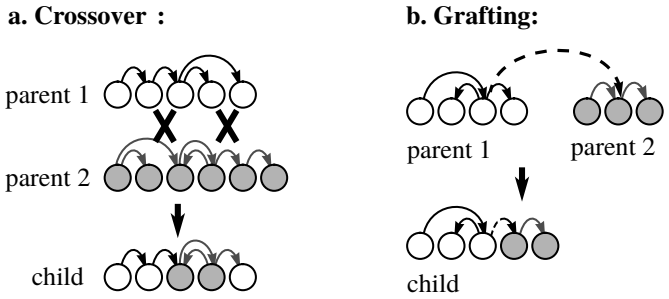


Figure 2.5: Two methods for mating directed graphs — *crossover* and *grafting*.

2.2.3 Evolving control system

The control system of the virtual organism consists of multiple artificial neural networks (*ANNs*) distributed over the organism's body. Two types of networks are used — the network placed in each *module* (*local ANN*) and the network for coordination and synchronization of local controls (*central ANN*).

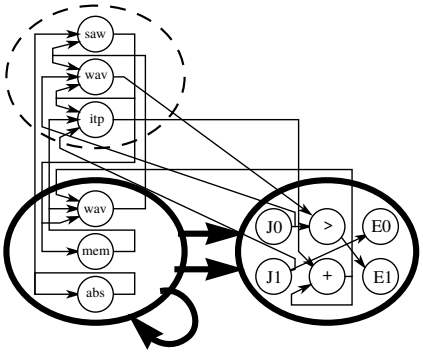
Instead of summed input and sigmoid activation function, here it is used set of predefined functions (e.g. $\text{sum}, \sin, \cos, \min, \max, \text{divide}, \text{if}$, etc.). Each neuron has the number of inputs corresponding to its activation function. Activation function of each neuron is evaluated in each time step.

Local ANNs process inputs from sensors in the module and outputs from neighboring *local ANNs* and *central ANN*. Their output is used to control effectors in the module (e.g. applying force to the module) and as input value to neighbouring local *ANNs* and *central ANN*. *Local ANNs* are represented in genotype as nested graphs for each *module*. Since recurrent connection are allowed, organism can use them as short-term memory, which is suitable for some tasks (e.g. robot locomotion).

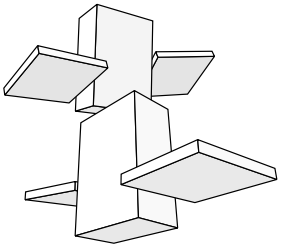
Unlike *Local ANNs*, *Central ANN* is not associated with any module and its sensors or effectors. Thus, it is not represented as nested graph, but separate graph, which is used in the phenotype only once. It is used to coordinate distant modules as an analogy of a central brain. Figure 2.1 [7] shows example of evolved genotype and corresponding phenotype morphology and control system. Haasdijk, Rusu and Eiben [4] shown that *Central ANN* is not necessary to evolve control system of the virtual organism, however, evolving control system without central control is more difficult task.

2.3 NeuroEvolution of Augmenting Topologies

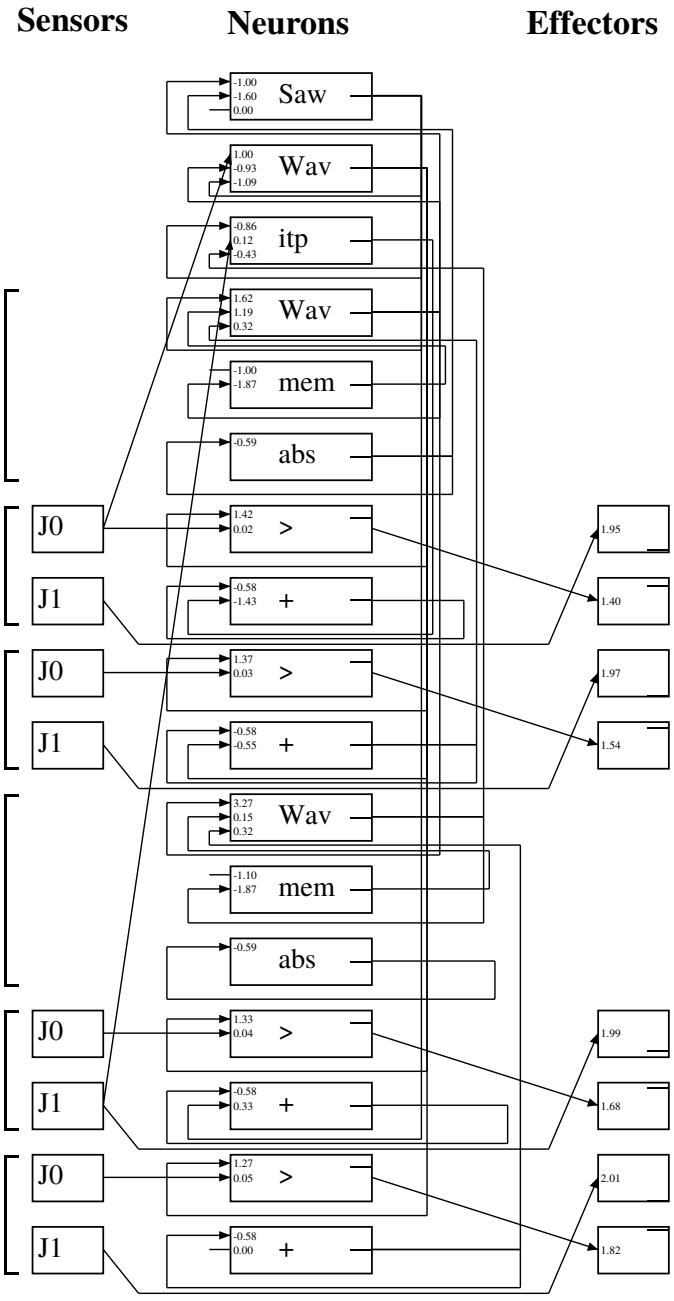
Evolving network topology removes the need for adjusting it experimentally, ideally the most suitable topology is discovered by the evolution. On the other hand, when evolving topology it can be difficult to find a proper way for representing networks in genotype and their mating. Therefore, evolving neural network topology requires more complex and sophisticated solution. Here, we describe popular method for evolving neural network topology called NeuroEvolution of Augmenting Topologies (*NEAT*) [1], which evolves neural networks topologies together with connection weights.



(a) The nested graph genotype.



(b) The phenotype morphology.



(c) The phenotype control system.

Figure 2.6: Example of genotype and corresponding phenotype of evolved virtual organism by Sims [7]. (a) shows evolved nested graph genotype. The outer graph (*morphological graph*) in bold describes a creature's morphology (picture (b)). The inner graph describes its neural control system (picture (c)), where $J0$ and $J1$ are joint angle sensors, and $E0$ and $E1$ are effector outputs. The dashed node represents central control, which is not associated with any body part. (b) shows the phenotype morphology generated from the evolved genotype. (c) describes the phenotype control system generated from the evolved genotype.

The *NEAT* algorithm is based on three principles — *historical marking* of genes, protecting innovations through *speciation* and *incremental grow* from minimal structure[4]. It has been shown that all these principles are dependent on each other [1, 2]. When one of them is removed, the algorithm will most likely have a lower performance compared to using all three principles simultaneously.

This section describes in-depth the *NEAT* algorithm. First, Section 2.3.1 focuses on representation of a network in genotype using *historical marking*. Next, Section 2.3.2 and Section 2.3.3 describe mutations and mating of networks using the *historical marking*. Section 2.3.4 introduces measuring similarity of network genotypes and protecting innovation through *speciation*. Finally, Section 2.3.5 describes incremental grow from the minimal structure.

2.3.1 Historical marking

Unlike evolving neural network weights with a fixed topology, when we evolve neural network topology there are several possibilities for representing structure and weights in genotype. Two different approaches can be used — *direct encoding*, when information about each connection and weight is stored in the genotype, and *indirect encoding*, when genotype consists of rules for construction of the network.

The main advantages of *indirect encoding* are (1) compact representation (i.e. lesser search space compared to *direct encoding*), (2) possible reuse of genetic information to build a phenotype network with some repeated or symmetric structure and (3) often simple realization of mating. However, according to Braun and Weisbrod (1993) [3] *indirect encoding* might restrict phenotype networks to some suboptimal class of topologies.

NEAT uses a *direct encoding*, where each part of the network is stored in a genotype. The genotype of a network consists of lists of neurons and connections. Each neuron contains information about type of the neuron (input, output or hidden), connections are represented using connected neurons, a connection weight, *enabled bit* and an *innovation number*. The *enabled bit* determines whether the connection in genotype is used in the corresponding phenotype. The *innovation number* (or *historical mark*) can be interpreted as historical origin of the gene.

When a new structural element (i.e. node or connection) arises, it receives a unique number — the *innovation number*. Genes of the same historical origin (i.e. with a common ancestor where the genes arose by a mutation) most likely have the same or similar role. Therefore, they are suitable to be combined by mating. *Innovation numbers* are then used for multiple purposes in the algorithm (e.g. mating and measuring similarity of two genotypes), which are described in the further text. The example genotype together with the corresponding phenotype is shown in Figure 2.7 [1].

[4] These principles are also used in some methods extending or similar to the *NEAT* algorithm (e.g. *HyperNEAT* — see Section 2.4)

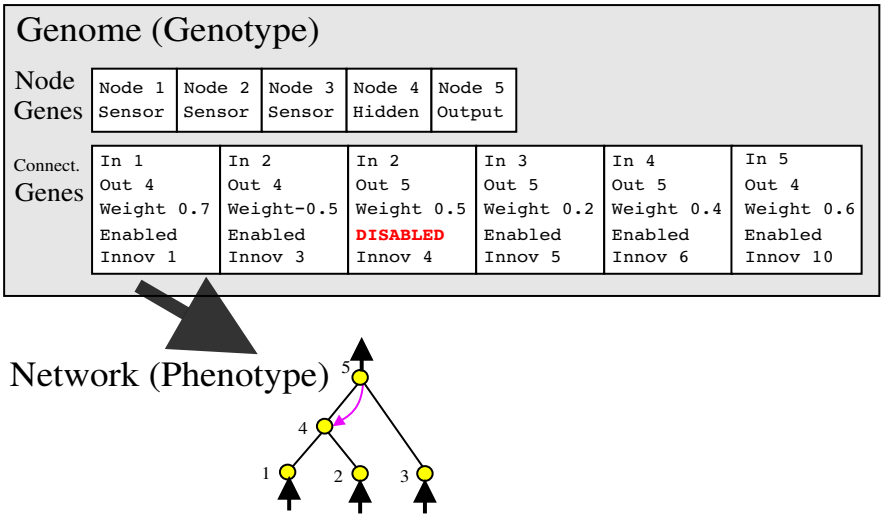


Figure 2.7: Example of a genotype and the corresponding phenotype network in *NEAT*. The network consists of three input (sensory) neurons, one hidden and one output neuron (the first list — node genes). The second list (connection genes) stores connections and their properties sorted by *innovation numbers*. The genotype contains altogether six connections, one of them is disabled, thus not used in the phenotype.

2.3.2 Mutating networks

Mutating networks includes both changes of weights and structural changes. Mutating weight is realized by adding small random number with normal distribution to its value, for each weight independently. Two structural mutations are used — *add neuron* and *add connection*.

Mutation *add neuron* splits randomly selected connection into two connections with added neuron between them. During the mutation the original connection is deactivated (but not removed from the genotype), the connection to newly added neuron is initialized with unit weight and the connection from added neuron has the same weight as the original connection. Thus, the final computed function is changed only slightly by adding a non-linear transformation. Mutation *add connection* connects two random previously unconnected neuron. Connection can be added only if the network remains non-recurrent after adding the connection. Both types of structural mutations are shown in Figure 2.8 [1].

2.3.3 Mating networks

Common problem of *direct encoding* is mating genotypes of network with different topology. In this case, it might be difficult to recognize which connections carry similar functional role to transfer genetic information of both parents into possibly better children. *NEAT* solves this problem by using *innovation numbers* (see Section 2.3.1). The following text describe mating of two genotypes in the *NEAT* algorithm.

Mating two parental network from the same species (see Section 2.3.4) produces a single network. First, the connections with matching *innovation numbers* are selected with one or more random crossing points (similar to mating in GA) and copied into the child network.

Next, *disjoint* connections (non-matching connections with *innovation num-*

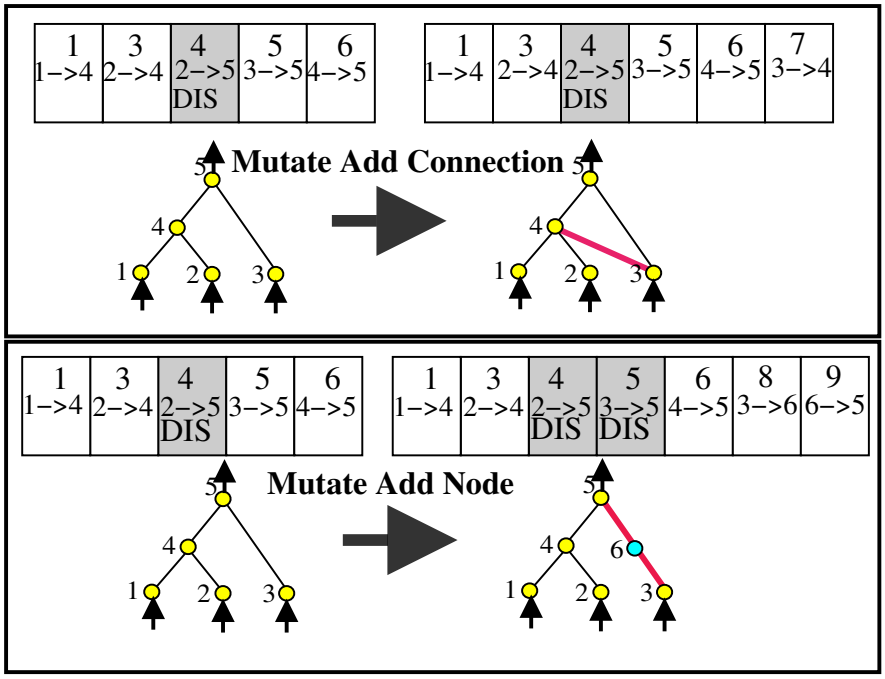


Figure 2.8: Two types of structural mutations in NEAT. Top figure shows *add connection* mutation creating new connection (7) and appending it on the end of the list of connections. Bottom figure demonstrates *add neuron* mutation creating new neuron (6), where the split connection (5) is disabled and two new connections are added (8 and 9) to replace the previous connection. Note that only the *innovation numbers* and the connected neurons (but not the weights) are shown here.

ber not larger than maximum *innovation number* in the genotype with smaller maximum *innovation number*) from parent with higher fitness[5] and all *excess* connections (remaining non-matching connections) are copied into the child network. Additionally some disabled connection are enabled — each with 25% probability. Figure 2.9 [1] shows an example of mating two networks.

2.3.4 Speciation

When a network with a new topology arises through mutation or mating it will have most likely low fitness until its weights are optimized. To prevent extinction of networks with some innovative topology before their weights are optimized, *NEAT* uses a *speciation*. Population is divided into multiple subpopulations with similar individuals. The difference between two individuals is measured on network genotypes — connections and their weights and it is defined as a weighted sum of the three factors, given by the following formula:

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 \cdot \overline{W}, \tag{2.1}$$

where

[5] Alternatively one of parents is chosen randomly, where parent with higher fitness is more likely to be chosen.

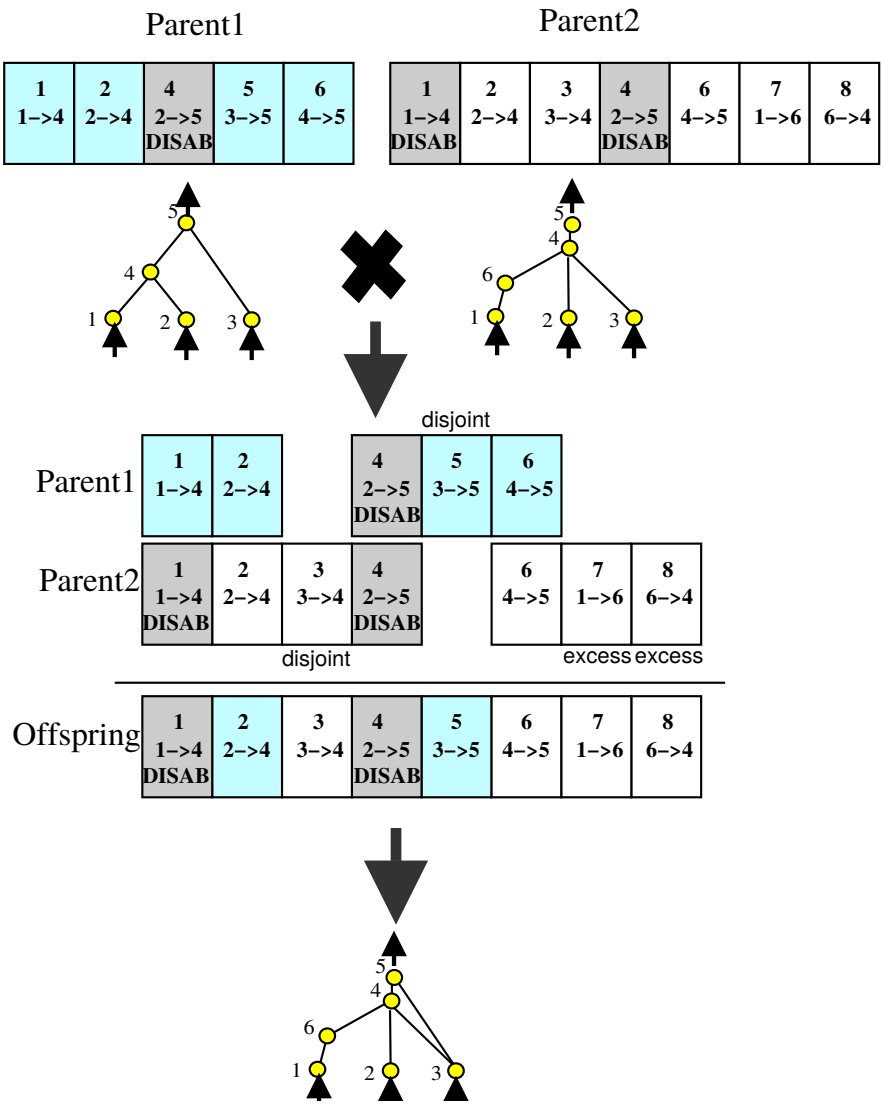


Figure 2.9: Mating two networks in *NEAT*. First, top figure shows two parent networks — both genotype and phenotype are shown. Next, the creation of the child genotype is shown. The connections with matching *innovation numbers* are placed under each other. Rest of the connections are marked with *disjoint* or *excess* signs. Color of each connection indicates from which of the two parents was the connection used. Finally, the resulting phenotype network is shown on the bottom figure. Note that only connections and their *innovation numbers* are shown here, connection weights are copied together with the connections.

E	is the number of *exess* genes,
D	is the number of *disjoint* genes,
$\overline{W}$	is the average absolute weight difference of matching genes,
N	is the size of larger genotype,
c_1, c_2, c_3	are positive constants (importance of particular factors).

Lower difference δ means that the measured genotypes are more similar.

When the next generation is produced it is divided into multiple species according to the following rules. (1) Initial population forms a single species. (2) When new genotype is created it is compared with random genotypes — each from different existing species and placed to the first species with the random genotype being less different than some threshold. The threshold value is adjusted each generation (increased if too many species was created and decreased if too few species exists).

When no similar genotype is found, new species containing the genotype is created. The species is then protected for some number of generations through *species elitism*, when the best genotype within the species is copied into the next generation. When the protection expires, species can be removed if its genotypes was not able to optimize itself during the time.

To maintain diversity in population, fitness of genotypes is scaled to reduce the number of similar genotypes in the population. Scaling is realized by *explicit fitness sharing*, where similar genotypes share their fitness.[6] The scaled fitness is given by the following formula:

$$\tilde{f}_j = \frac{f_j}{\sum_i sh(\delta(i,j))},$$

where

f_j	is the fitness value of the jth genotype,
$\tilde{f}_j$	is the scaled fitness value of the jth genotype,
$\delta(i,j)$	is the difference between the genotypes i and j, computed according to 2.1,
$sh(x)$	is the indicator function for similar genotypes, $sh(x) = \begin{cases} 1 & \text{for } x < \theta, \\ 0 & \text{for } x \geq \theta, \end{cases}$
θ	is the threshold indicating how different genotypes are still considered similar.

2.3.5 Incremental growth

When we evolve topology of the network, it is usually desirable to find the network with minimal size capable of solving some particular problem since larg-

[6] This is analogy of sharing resources by a species in the nature.

er networks are more computationally-intensive and have larger weights search space. Therefore, it is more difficult to optimize their weights.

NEAT uses approach suitable for finding network of a minimal size. First, an initial population is formed by networks of minimal possible size (no hidden neurons are used). Then, during the evolution, the size of networks is incrementally growing by mutations until larger networks have some evolutionary advantage (i.e. they are capable of performing better than networks with lower size, therefore they have larger fitness value). In an ideal case the network of minimal size capable of solving the problem is found.

Another advantage of the incremental growth is lower weights search space from begining. Therefore, if a solution requiring only a small network exists it is likely to be found in first few generations.

2.4 Hypercube-based NEAT

The *NEAT* algorithm is suitable for evolving and finding the suitable topology for the networks with small or medium size. However, some problems might require larger networks, which would be difficult evolve using the *NEAT* algorithm if search space becomes too large. Therefore, the possible solution for large networks is to use an indirect encoding, which would reduce the size of the search space.

This section describes the Hypercube-based *NEAT* (*HyperNEAT*) algorithm[7] [8] that uses an indirect generative encoding, which allows evolving large neural networks with fixed topology. First, generating weights of evolved network using Compositional Pattern Producing Network is described in Section 2.4.1. Next, advantages of generative encoding and its possible applications are discussed in Section 2.4.2.

2.4.1 Generative encoding

Gauci and Stanley [8] argued that phenotypes in nature typically exhibits higher complexity than their genotypes. They suggested a method that could discover high complexity phenotypes (e.g. large-scale neural networks) through a mapping from genotype to the phenotype that translates few dimensions into many dimensions.

In the *HyperNEAT* algorithm, a large-scale neural network with fixed topology (*substrate*) is represented in genotype using smaller neural network — Compositional Pattern Producing Network (*CPPN*), which is evolved using the *NEAT* algorithm. The following text describes the representation of *substrate's* weights using a *CPPN* network.

The *substrate* network is placed into p-dimensional space, where each neuron has an unique position. The positions of the neurons should reflect the geometry of the problem, then as argued by Gauci and Stanley [8], this information could

[7] "Hypercube-based" means that *NEAT* is used here to evolve a mapping from k-dimensional hypercube to 1-dimensional hypercube (i.e. interval $[-1, 1]$), which represents a phenotype network using an indirect encoding.

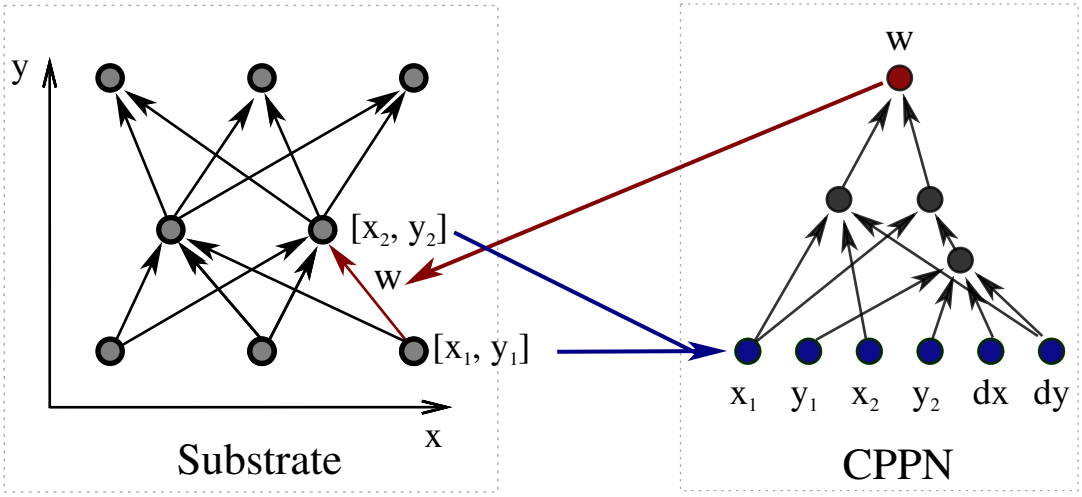


Figure 2.10: An example of computing connection weight in *HyperNEAT*. In this example the *substrate* is placed into two-dimensional space, thus, each neuron has two coordinates x and y. The weight of the connection from the neuron with coordinates $[x_1, y_1]$ to the neuron with coordinates $[x_2, y_2]$ in *substrate* is computed using *CPPN*. The following values are used as *CPPN* inputs: positions of two neurons (x_1, y_1, x_2 and y_2) and differences in each coordinate (dx and dy, where $dx = x_2 - x_1, dy = y_2 - y_1$). The output of *CPPN* is then used to set the connection weight.

be an advantage for an evolution, when regularities in the solution could be exploited.

To determine weights in the *substrate* network, *CPPN* query each potential connection in *substrate* and determine its presence and weight if connection is used.This is realized by, first, using coordinates of two neurons (and additionally their differences or other parameters) as inputs of *CPPN*. Then, the output of *CPPN* is computed and used as the weight of the connection. If the absolute value of the weight is below some predefined threshold, the connection is not used. This can reduce number of connections of the *substrate* without significant change of computed function. Figure 2.10 shows computing of a single connection weight in an example *substrate* network.

2.4.2 Advantages of generative encoding and its applications

HyperNEAT inherits advantages from *NEAT* such as reduced search space in initial generations and increasing complexity of the phenotype during the evolution. Furthermore, the size of *CPPN* is independent on the size of *substrate* (i.e. a single *CPPN* can be used to generate weights for neural networks of different sizes). Thus, *CPPN* can be first evolved using smaller *substrate* (which is computed is faster) to compute the fitness of the *CPPN* and then, when *CPPN* is evolved, it can by used for larger *substrate*.

Gauci and Stanley [8] argued that *HyperNEAT* is suitable for discovering regularities (e.g. repeating motifs) in the phenotype and it is able to generalize

significantly better than direct encoding for some types of problems. Moreover, as shown by Haasdijk, Rusu and Eiben [4], *HyperNEAT* can develop a control system for virtual organisms, where a single *CPPN* is used to generate local neural control of each module. The *CPPN* then generates similar controls with variations based on the modules positions.

2.5 Artificial Ontogeny

The methods described in the previous text used a directed graph to represent and evolve morphology of the virtual organism. This approach allowed to evolve several interesting virtual creatures. However, Cheney, Nick, et al. [16] argued that despite nearly two decades of work since the method proposed by Sims [7], evolved morphologies are not obviously more complex or natural, and the field seems to have hit a complexity ceiling. Therefore, new approaches should be considered to develop methods that will produce more complex organisms capable of solving complex real-world tasks.

This section provides a detailed description of a developmental method for evolving virtual organisms — *artificial ontogeny* that is the main inspiration for further methods described in this work. *Artificial ontogeny* was introduced by Bongard and Pfeifer in their work [5,15] that combines an evolutionary algorithm, ontogenetic development and differential gene expression.

Unlike directed graphs, *artificial ontogeny* uses more indirect representation of an organism body compared to the directed graphs. A mapping from a genotype to the phenotype is realized via an ontogenetic process, where an organism grows from a single structural unit (that is an analogy of a cell) to a complete organism. Main role in ontogenetic process plays *gene product* that are abstraction of gene product in the biology. They have two roles — they affect construction of phenotypic elements and affects expression of other genes.

The section is divided into six subsections that describe *artificial ontogeny* from the structure of the phenotype to the representation in the genotype. First, the phenotype of the organism is described — Section 2.5.1 describes its morphology and Section 2.5.2 its control system. Next, Section 2.5.3 focuses on the growth of organism's body based on the actual concentrations of *gene products*. Further, Section 2.5.4 describes growth of neural control system using a *cellular encoding* [5]. The following section, Section 2.5.5 describes *genetic regulatory network* and related diffusion of *gene products* over organism's body. Finally, the structure of the genotype is described in Section 2.5.6.

2.5.1 Morphological Structure

Similarly to the other works [2, 7], the organism in *artificial ontogeny* consists of multiple body parts — modules. Here, due to a biological analogy, it is more accurate using term "cells" instead of "modules". The architecture of the organism's phenotype described in the following text is not used in this work, thus some details are skipped to provide a basic overview of organism's morphology

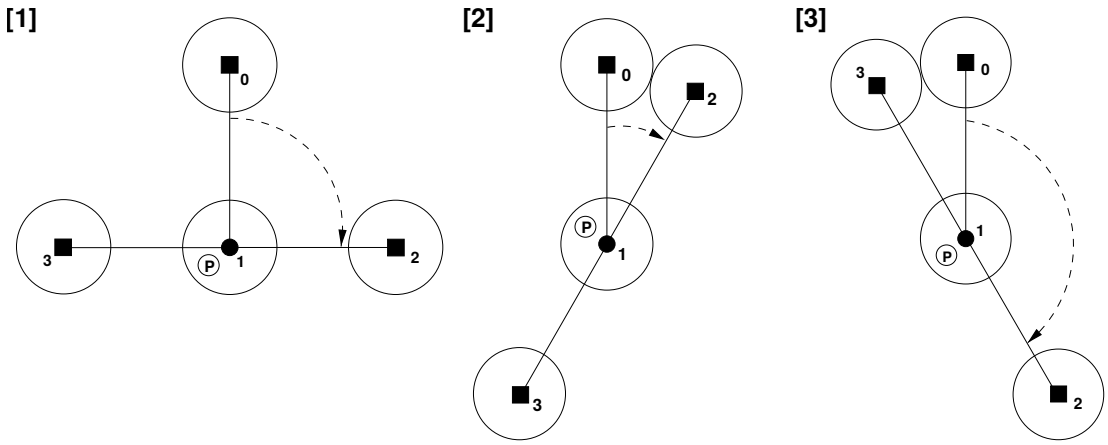


Figure 2.11: **Rotational joints in artificial ontogeny.** The figure shows cell 1 attached to cell 0 and cells 2 and 3 attached to cell 1. Subfigures [1] – [3] show different rotations of cell 1. The rotation occurs through the plane described by cells 0,1 and 2. Cell 1 contains proprioceptive sensor that emits a value negatively correlated to the angle described by cells 0,1 and 2 (dashed arc). The emited value of the sensor is zero for Subfigure [0], positive for Subfigure [2] and negative for Subfigure [3].

and control system.

An organism consists of one of more cells of a spherical shape that are connected using rotational joints. Each cell contains 6 places (located to the north, south, west, east, up and down from the center of the cell), where another cell can be attached using a rotational joint with one degree of freedom. Figure 2.11 [5] shows possible rotation of an attached cell.

Each place, from which another cell can be attached, has associated a *diffusion site*, where the concentrations of *gene products* are simulated. *Diffusion sites* are located at a half distance between the center of the cell and the position on the cell boundary, where new cells can be attached.

During the growth, each cell can grow in size depending on the concentrations of *gene products*. When a cell reaches twice radius of an initial size, it is split into two cells with the initial size. Splitting the cell is realized by resetting radius of the original cell to the initial value and attaching a new cell to one of 6 positions.[8]

2.5.2 Neural Control system

Organism's control system is realized using a single neural network with its neurons distributed over the organism's body. Each neuron is associated with a particular *diffusion site* and each *diffusion site* can contain zero or more neurons. Synaptic connections can connect two neurons (1) within a *diffusion site*, (2) between two different *diffusion sites* within a cell or (3) between two *diffusion sites* located in different cells.[9]

The neural network consists of three types of neurons — (1) sensory neurons

[8] In case that the attachment place is already used (i.e. another cell is connected to this place), existing cell is reconnected to the newly created cell.

[9] Note that the synaptic connections can connect also neurons that may be located in very distant cells (e.g. a sensory neuron and a motor neuron).

(propriceptive, touch and light sensor), (2) internal neurons and (3) motor neurons. There is no restriction on the number of the neurons, thus each type of neurons may be present in a *diffusion site* zero of more times.

Motor neurons control movement of the cells, where they are placed. When no motor neuron is present in a cell, the associated rotational joint is passive (i.e. fixed). When more than one motor neuron is included in a cell, the average output is used to control movement of the rotational joint. The organism's movement is achieved by applying a torque to its cells to a desired angle that is determined by outputs from motor neurons and scaled to range $[-\frac{\pi}{2}, \frac{\pi}{2}]$.

Internal neurons are further divided into three subtypes. They are used to (a) propagate a signal from sensory to motor neurons (*hidden neurons*) or (b) they emit a constant maximum positive value (*bias neurons*) or (c) they emit a sinusiodal output (*oscilatory neurons*). *Bias neurons* have no inputs, *oscilatory neurons* have one or more inputs that determine frequency[10] of their outputs.

The output of each neuron is computed at each time-step, when the neural network is evaluated. The outputs are then stored and used as inputs in the next time-step. Thus, recurrent connections can be evolved and used as a short-term memory that can be useful for some tasks (e.g. locomotion in the virtual environment). Output of the neurons (except for *bias, oscillatory and input neurons*) is computed using the following activation function: $\sigma(x) = \frac{2}{1+e^{-\lambda x}} - 1$. Thus, outputs of the neurons range from -1 to 1. The control system is activated in the moment when its growth is complete to initiate organism's movement. Examples of organisms evolved using *artificial ontogeny* are shown in Figure 2.3 [5].

2.5.3 Growth from a Single Cell

Both morphology and control system described in the previous text are simultaneously constructed during the growth of the organism. The growth is realized by T discrete time-steps, where in each step (1) concentrations of *gene products* are changed and (2) new phenotypic elements can be created. This subsection describes construction of the phenotype based on concentration of *gene products* concentrations during the growth process.

Gene products can be viewed as an abstraction of *gene products* in nature. Amounts of *gene products* that are present in *diffusion site* are represented using values from range $[0, 1]$. These values can be interpreted as concentrations of *gene products* in the nature linearly scaled to range $[0, 1]$, where 0 indicates that a *gene product* is not present in a *diffusion site* and 1 indicates maximum possible concentration.[11]

There is totally 24 different types of *gene products*, where each of them has a predefined role. Two *gene products* affect growth of cells (*grow enhancer*, *grow repressor*), more *gene products* are used for development of control system and

[10] The frequency is given by the sum of inputs, where higher input values cause higher frequency.

[11] Note that the sum of the *gene product* concentrations can exceed value 1. We assume that (not scaled) maximum possible concentration are much lower than 1. Therefore, concentration of a single *gene product* is not directly affected by concentration of different *gene product* (however, they can affect each other indirectly via *genetic regulatory network* described in further text).

rest of them have no direct phenotypic effect. All types of *gene products* are also used as gene regulators (i.e. they affect production of other *gene products* as described in Section 2.5.5).

Concentrations of *gene products* are simulated in discrete time-steps within the organism during its growth. Each *gene product* is simulated in each *diffusion site* with only local interactions between neighbouring *diffusion sites* that are (1) all pairs of *diffusion sites* within a cell except opposite sites (e.g. north and south *diffusion sites* are not considered neighbors) and (2) nearest *diffusions sites* of two connected cells.

The growth starts with creating a single cell. During the growth new phenotypic elements (i.e. cells and neural structure) are added to the existing phenotype or existing phenotypic elements are altered, where each new part of the phenotype is associated with a particular *diffusion site*. As a result, the phenotype grows in size during the process of development.

When a new cell is created, it can grow in size depending on concentrations of *grow enhancer* and *grow repressor gene products* within the cell. As mentioned in the previous text, when a cell reaches twice a size of its initial radius, a new cell is created and size of the original cell is set to the initial size. The new cell is attached in the direction corresponding to the *diffusion site* with maximum concentration of *grow enhancer* within the parent cell. Growth of the organism is terminates after predefined T time-steps, when the organism is considered to be "adult".

2.5.4 Neural Growth

Bongard and Pfeifer [5] intercorporated *cellular encoding* by Gruau at al. [13] into their method to evolve and grow neural control system of the organism simultaneously with the morphological structure. The *cellular encoding* is a developmental encoding that allows evolving both topology and weights of a neural network. This work uses a *CPPN* encoding instead of *cellular encoding* that is used in *artificial ontogeny*, thus some technical details are skipped to provide description of basic concepts that can be compared with other approaches in further chapters.

As described in the previous text, the neural control system has neurons placed inside *diffusion sites* within the organism, where synapses can connect neurons between different *diffusion sites* and also between different cells. The phenotype neural control is constructed using *cellular encoding operations* [15] during the growth of the organism that alter neural structure, synaptic weights, types of neurons and their placement inside *diffusion sites* within the organism.

Cellular encoding operations are related to *diffusion sites* inside the organism. They are applied simultaneously to each *diffusion site* at each time-step based on the concentrations of *gene products* within the *diffusion site*. Completely 17 *cellular encoding operations* is used, where each operation is triggered by different *gene product*. Thus, each *cellular encoding operation* has asociated its *gene product*. An operation is triggered when the corresponding *gene product* concentration reaches a treshold value 0.8. When more than one operations are triggered for a particular *diffusion site* in a single time-step, they are applied in a predefined order.

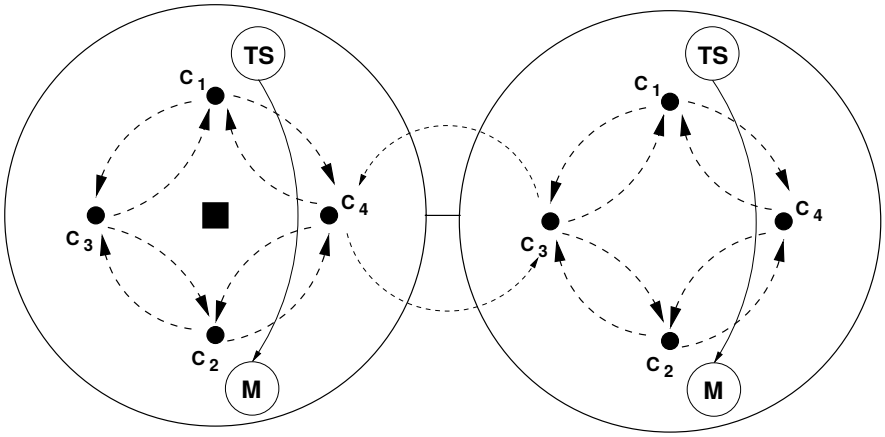


Figure 2.12: **An example of two neighboring cells in artificial ontogeny.** Only 4 of 6 *diffusion sites* are shown (denoted as $C_1 - C_4$) in the cells. Dotted lines indicate diffusion between neighboring *diffusion sites*. Both cells contain touch sensor neuron (TS) and motor neuron (M) that are located in *diffusion sites* C_1 and C_2 respectively and the synapse from the sensor to the motor with an unit weight.

There are two types of *cellular encoding operations*: (1) operations applied to neurons and (2) operations applied to synapses within a *diffusion site*. Operations applied to neurons include serial and parallel duplication, deletion, migration of a neuron between *diffusion sites* and changing type of a neuron. Operations applied to synapses includes duplication and deletion of synapse, increase and decrease of a synaptic weight and reconnecting a synapse to a different neuron.

As described in the previous text, a *diffusion site* can contain more than one neurons or synapses. Thus, two additional pointers are maintained for each *diffusion site* that indicate which neuron and the synapse will be used for an operation. *Cellular encoding operations* also include moving these pointers. Table 2.1 [15] shows complete list of used *cellular encoding operations* in order they are applied.[12]

Initially, when a new cell is created, it contains two neurons — a touch sensor neuron (TS) and a motor neuron (M) as shown in Figure 2.2. These neurons are placed in opposite *diffusion sites* (*diffusion sites* 1 and 2 in Figure 2.2) and connected using a synapse from TS to M with a unit weight. Additionally, when a new cell is created from an existing cell, all neurons from the *diffusion site* of the parent cell, from which the new cell was created, are moved to the neighboring *diffusion site* of the new cell, where all existing connections are kept. This may lead to a directed migration of neurons across the cells, where connections even between distant cells remain maintained. Furthermore, this method allows evolving recurrent neural networks that propagate a signal from sensors to motors.

2.5.5 Genetic Regulatory Network

This text describes a *genetic regulatory network* and related diffusion of *gene products* within the organism that affect concentrations of *gene products* inside the *diffusion sites* during the growth of the organism. *Genetic regulatory net-*

[12]Note that if a *diffusion site* has no neural structure, no operation is applied.

Table 2.1: **Cellular encoding operations in artificial ontogeny.**

Operation	**Description**
Serial duplication of a neuron	Split the current neuron into two neurons. Move the output synapses of the original neuron to the new neuron. Connect the original neuron to the new neuron with a synapse of positive weight.
Parallel duplication of a neuron	Split the current neuron into two neurons. Copy the input and output synapses to the new neuron. Connect the two neurons to each other using two synapses of positive weight.
Neuron migration to the previous *diffusion site*	Move the current neuron to the previous *diffusion site* within the cell.
Neuron migration to the next *diffusion site*	Move the current neuron to the next *diffusion site* within the cell.
Change target of a synapse	Move the head of the current synapse to the current neuron.
Change source of a synapse	Move the tail of the current synapse to the current neuron.
Increase a synaptic weight	Increment the weight of the current synapse by 0.01.
Decrease a synaptic weight	Decrement the weight of the current synapse by 0.01.
Duplication of a synapse	Duplicate the current synapse.
Delete of a neuron	Delete the current neuron, including any ingoing and outgoing synapses.
Delete of a synapse	Delete the current synapse.
Move the neuron pointer to the next neuron	Move the neuron pointer to the next neuron at the current *diffusion site*.
Move the neuron pointer to the previous neuron	Move the neuron pointer to the previous neuron at the current *diffusion site*.
Move the synapse pointer to the next synapse	Move the synapse pointer to the next synapse at the current *diffusion site*.
Move the synapse pointer to the previous synapse	Move the synapse pointer to the previous synapse at the current *diffusion site*.
Change the type of the current neuron	Change the type of the current neuron to the next neuron type in the following order: (1) Touch sensor, (2) Propriceptive sensor, (3) Light sensor, (4) Bias neuron, (5) Oscillatory neuron, (6) Internal neuron and (7) Motor neuron. Note that the last type is changed to the first type.
Change the type of the current neuron in reverse order	Change the type of the current neuron to the previous neuron type in the following order: (1) Touch sensor, (2) Propriceptive sensor, (3) Light sensor, (4) Bias neuron, (5) Oscillatory neuron, (6) Internal neuron and (7) Motor neuron. Note that the first type is changed to the last type.

work abstracts some properties of biological genetic regulatory networks such as interaction between DNA segments. It can be also viewed as an intermediate step in transition from genotype to the phenotype and it plays a fundamental role in *artificial ontogeny*.

There are three factors affecting gene product concentrations during the growth — (1) genetic regulatory network (*GRN*), (2) diffusion to the neighboring *diffusion sites* and (3) diffusion outside the organism. Diffusion to the neighboring *diffusion sites* and outside the organism is not directly affected by the genotype. Genetic regulatory network increases concentrations of some *gene products* (i.e. it produces new *gene products*) at each time-step and each *diffusion site* simultaneously based on the current concentrations of *gene products* in the *diffusion sites* within the cell.

Genetic regulatory network consists of rules for production of *gene products* that we denote as *GRN rules*. Unlike the developmental rules described in the previous text that are related to *diffusion sites*, *GRN rules* are applied to cells, where at each time-step for each cell a subset of *GRN rules* is applied to the cell.

GRN rules have the following structure: Each *GRN rule* has specified its regulating *gene product*, concentration domain and target *diffusion site* that are encoded in the genotype. A rule is applied if its target *diffusion site* has the regulating *gene product* concentration within the concentration domain. Each *GRN rule* has also specified produced *gene product* and amount of concentration which is produced to the target *diffusion site* when the rule is applied. The concentration domain is either: (1) an interval $[l, u] \subseteq [0, 1]$ or (2) its complement (i.e. $[0, 1] \setminus [l, u]$), where l indicates lower and u indicates upper bound for a *gene product* concentration.

Diffusion to neighboring *diffusion sites* is realized by moving some amount of *gene products* from the site with higher concentration to the site with lower concentration that is done at each time-step of the growth. The transfered amount of *gene product* is linearly correlated to the difference of two concentrations. A higher amount is diffused to neighboring sites within the cell and a smaller amount to neighboring *diffusion site* in the neighboring cell.

Diffusion out of the organism decreases concentration of each *gene product* by a small ratio. Thus, a diffused amount of *gene product* is linearly correlated to the absolute concentration. This causes an exponential descease of the concentrations during the growth of the organism.

2.5.6 Representation in the Genotype

A genotype in *artificial ontogeny* encodes a *genetic regulatory network* described in the previous text.[13] The genotype consists of a sequence of values from range $[0, 1]$ that is evolved using a real-valued genetic algorithm. The genetics algorithm uses an asymmetric crossover, thus it evolves genotypes with a variable length and allows evolving genotypes with increasing length during the evolution. The following text describes representation of *genetic regulatory network* in the genotype.

[13] Therefore, the genotype in *artificial ontogeny* is also referred as a *GRN genotype*.

Table 2.2: Structure of a single gene in artificial ontogeny

Pr	is *promotor site* indicator, it determines if the following 7 values form a gene,
P_1	determines if the concentration domain for the regulation *gene product* is an interval (values higher that 0.5) or its complement (values lower that 0.5),
P_2	represents index of the regulating *gene product*,
P_3	represents index of the *gene product* that is produced in target *diffusion site*,
P_4	represents index of target *diffusion site*,
P_5	determines an amount of the produced *gene product*,
P_6	is the lower bound for the concentration domain of the regulating *gene product*,
P_7	is the upper bound for the concentration domain of the regulation *gene product*.

The values in the genotype either encode *GRN rules* or have no phenotypic effect. *GRN rules* are encoded using continuous blocks (*genes*) in the genotype, where each *gene* consists of 7 values that encode a single *GRN rule*. *Genes* can start on any position in the genotype (except of positions at the end of the genotype). Starting position of a *gene* is determined by a *promotor site* that is a position in the genotype, which indicates that the following 7 positions form a *gene*.

To construct *genetic regulatory network* from the genotype, first *promotor sites* are found. A position in the genotype is a *promotor site* if both (1) it is not part of a previous *gene* or at the end of the genotype and (2) its value is below $\frac{n}{l}$, where n is a predefined average number of *genes* that should be used in each initial random *genotype* and l is predefined length of *genotypes* in the initial population. Thus, each *genotype* in the initial random population will have on average approximately n *genes*.

The structure of a single *gene* is following. We denote parameters that form a *gene* as $P_1, \ldots, P_7$ and a value on the *promotor site* position as Pr. Each parameter in the *gene* corresponds to a single parameter of the *GRN rule*. Lower and upper bounds for the concentration domain are represented directly by the value in the *gene* (P_6, P_7). A value at the position that encodes produced amount of a *gene product* (P_5) is linearly scaled to range $[0, 0.01]$.

To determine both regulating and target *gene product*, the corresponding value in the *gene* (P_2, P_3) is multiplied by the number of used *gene products* and then rounded to an integer value that is treated as an index of a *gene product*.[14] As a result, each *gene product* has an equal probalility in the initial random population.

A similar approach is used to encode indexes of *diffusion sites* in the *gene* (P_4). One more parameter (P_1) is used to indicate wheter the *concentration domain* is an interval determined by lower and upper bounds (values higher than 0.5) or its complement (values lower than 0.5). Figure 2.13 [5] shows an example of a genotype and Table 2.2 includes complete list of parameters forming a *gene*.

[14] *Gene products* are indexed as $0, \ldots, p-1$, where p indicates the number of used *gene products*.

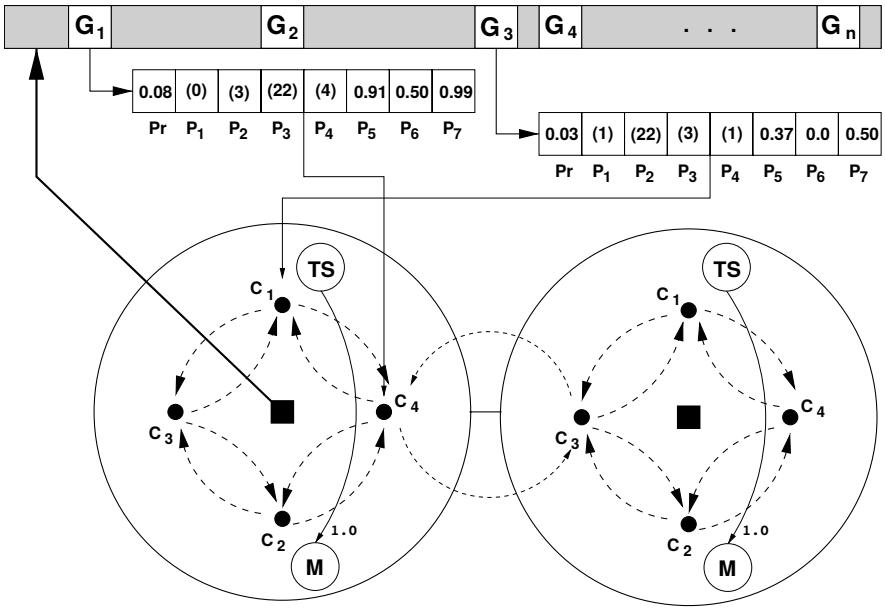


Figure 2.13: **An example of a genotype in artificial ontogeny.** The figure shows the situation as in Figure 2.12. Additionally, a genotype is included (on the top) that consists of n genes $G_1, \ldots, G_n$ (only genes G_1 and G_3 are shown in details). The values in parentheses indicate integer or binary values (e.g. indexes of *diffusion sites* or *gene products*). Light grey color indicates positions in the genotype that do not encode genes.

3. The First Proposed Method: Evolving Morphological Structure using Directed Graphs

This chapter describes a proposed method for evolving virtual organism that includes evolving morphological structure and control system of the organism. The proposed method is based on the previous work of the author of this thesis [12], where evolving morphological structure is adopted without any modifications. On the opposite, control system was improved and extended to evolve organisms capable of more efficient locomotion that exploits potential of the morphology.

The proposed method evolves morphology and control system of the organism simultaneously, where the morphological structure is represented using directed graphs as proposed by Sims [7]. Unlike Sims, the control system is evolved using the HyperNEAT algorithm [8] with inspiration by the method proposed by Haasdijk, Rusu and Eiben [4]. In addition to Haasdijk, Rusu and Eiben, a central control system is used to coordinate organisms movement.

The HyperNEAT generative encoding provides several advantages in addition to the original Sim's method: (1) it reduces search space of the control system by using a single compositional pattern-producing network (CPPN) that represents weights of local networks, moreover, (2) additional CPPN inputs also allow module differentiation based on the module position within the organism, as a result, evolved neural network most likely exhibits similarities such as repetition with variations, finally (3) HyperNEAT inherits advantages of NEAT such as sensible crossover, speciation and growth from a minimal structure that increase efficiency of the evolution of virtual organisms as shown by Krčah [2].

This chapter is divided into two sections: First section, Section 3.1 describes morphological structure of the organisms and its representation in the genotype that are adopted from the previous work of the author of this thesis. Second section, Section 3.2 describes the neural networks that form the control systems of the organism including sensor, effectors and HyperNEAT-generative representation.

3.1 Evolving Morphology

The morphological structure of virtual organisms in this work is similar to organisms evolved in the Sim's work [7]. However, unlike Sims, organisms in this work consist of homogeneous modules (i.e. modules of identical shape and physical properties) connected to each other using joints that have a single degree of freedom or are fixed. Moreover, unlike most of previous works, modules are here allowed to intersect each other for reasons discussed further in this section.

This section describes a morphological structure of virtual organisms and its representation in the genotype. The section includes following two subsections:

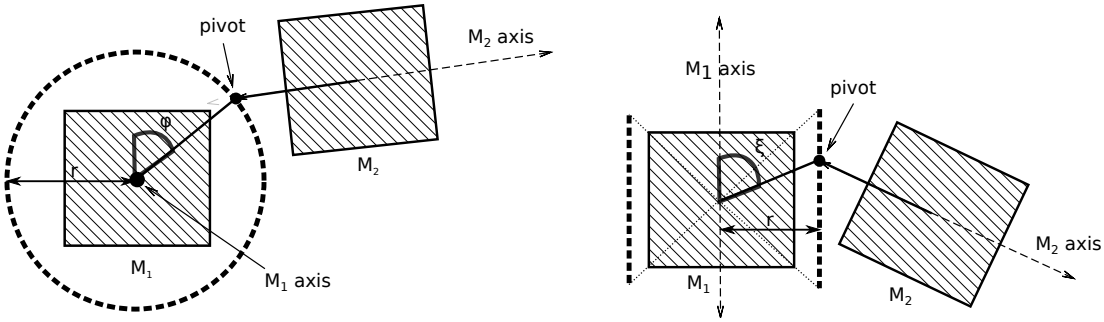


Figure 3.1: **Two 2-dimensional projections of possible pivot positions.** Both pictures show the situation, where module M_2 is attached to module M_1. Bold dashed circle (left picture) and lines (right picture) indicate possible pivot positions. Plane of the left picture is perpendicular to the axis of module M_1 and the plane of the right picture is parallel to the module axis and one of module faces. The pivot position has constant distance r from M_1 axis. The angle ξ between the line from the center of M_1 to the pivot and M_1 axis is from the range $[\frac{\pi}{4}, \frac{3\pi}{4}]$ (dotted lines on the right picture correspond to the minimal and maximal angle). Second angle ϕ determines rotation around M_1 axis (left picture). Both angles are stored in the organism's genotype. The axis of module M_2 is determined by the pivot position and the center of module M_2.

First, Section 3.1.1 describes a structure of organism body, connections of attached modules and their movement. Next, Section 3.1.2 describes representation of morphological structure in the genotype including construction of the phenotype morphology for a given genotype.

3.1.1 Morphological Structure

The organism consists of *modules* connected to each other that form a tree structure (*morphological tree*). *Modules* are connected using a fixed joint or a rotational joint with one degree of freedom. Modules that form an organism are homogeneous — they have the same physical properties (e.g. weight, friction, etc.) and an identical cube shape. However, they may differ in their control systems as described in the next section. The following text describes attaching modules using fixed and rotational joints.

Attaching a module to its parent in the *morphological tree* includes the following procedure. For better clarity we denote attached module as M_2 and its parent module as M_1. To determine module M_2 position, first, position of the pivot that is the center of the rotation of the attached module, is determined. The pivot is placed in constant distance r from the *module axis* that is one of 3 lines crossing the center of the module and perpendicular to its 2 faces. The final pivot position is determined by two angles ϕ and ξ encoded in the genotype. The possible pivot positions are shown in Figure 3.1.

Once position of the pivot is determined, the attached module M_2 is placed in the same direction as the pivot (from the center of module M_1), but in double distance (i.e. pivot is placed in the middle between the centers of two modules). Additionally, if a rotational joint is used, the attached module can be rotated around the pivot by applying force to the module, where the plane of rotation is determined by an angle encoded in the genotype. Figure 3.2 shows rotation of the attached module. Desired rotation of the attached module is determined

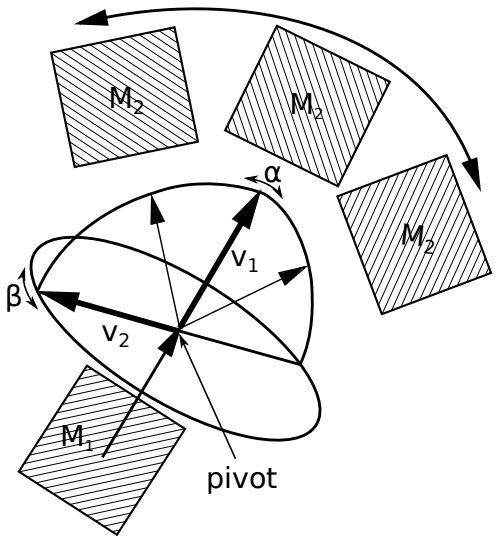


Figure 3.2: **Rotation of attached module around its pivot.** Picture shows module M_2 attached to its parent module M_1 using a rotational joint. Two unit vectors v_1 and v_2 together with the pivot position and fixed distance from the pivot defines a circle, where is module M_2 (its centroid) rotated. Vector v_1 is determined by the center of M_1 and the pivot position. Vector v_2 is determined by angle β, which is stored in the genotype, and determines a plane in which M_2 is rotated. Angle α then determines rotation of module M_2 around the pivot ($\alpha = 0$ if the center of M_2 has a common line with the pivot and the center of M_1). For a fixed joint, α has a constant value $\alpha = 0$. Note that length of the joint is increased here for better clarity.

by local control system of the module and it is given by the following formula:

$$\alpha[A, \omega, \phi](t) = Az \sin(\omega t + \phi), \tag{3.1}$$

where

t	is time in the simulation,
A	is an amplitude (determined by the control system),
z	is a bipolar value (i.e. 1 or -1) inverting movement of a cloned module (see Section 3.1.2) that is determined by the genotype,
ω	is an angular speed (determined by the control system),
ϕ	is a phase (determined by the control system).

When a desired angle for a rotational joint is determined, the force is applied to the module in the direction to the desired angle. The desired angle is determined by outputs from the control system (amplitude, angular speed and phase) and current time in the simulation. For fixed outputs from the control system is the desired angle a sinusoidal function of time in the simulation. As a result evolved organisms exhibit a periodical motion that is suitable for tasks such as locomotion or swimming.[1]

Unlike other works [2, 5, 7] that evolve virtual organisms, modules are allowed to intersect themselves during motion of the organism . This might not be practical for construction of real-word robots. However, the modules intersections can be often reduced by a slight modification of the phenotype (e.g. replacing intersected modules with a single larger and heavier module). Moreover, it is believed [12] that more freedom to the morphology could allow evolving various types of organisms that could not be evolved otherwise. To prevent massive intersections of the modules, some mechanisms can be used to penalty organisms with a high amount of module intersections in the fitness.

[1] However, this might prevent evolution from finding successful organisms with different types of motion.

3.1.2 Representation in The Genotype

Morphology of the organism is represented using the *morphological graph* described in the previous chapter. The *morphological graph* indirectly encodes morphology of the organism that is constructed using breadth-first search applied on the graph followed by depth-first search to create symmetries in the organism's body. The following text provides more detailed description of the method including a complete list of used genetic operators.

The used representation is designed to evolve organisms that exhibit symmetries[2] to reduce search space and allow evolving larger organisms. The used symmetries was proposed in the previous work of the author of this thesis [12] with inspiration by living organisms in the nature that often exhibit a mirror (plane) symmetry (e.g. two symmetrical arms and legs), which allows them an effective movement in their natural environment.

The following text describes construction of the phenotype morphology based on the *morphological graph* stored in the genotype. First, the module corresponding to the starting vertex in the *morphological graph* is created (*root module*). Then, breadth-first search (*BFS*) from the starting vertex is used to create corresponding *morphological tree* that forms body of the organism, where each vertex can be visited multiple-times (determined by *module-limit* for each vertex).

To evolve organisms that exhibits symmetries two types of edges are included in the *morphological graph* — *simple edges* and *double edges*. The edges marked as a *double edge* (that is a symmetry indicator) are treated during the breadth-first search as two separate edges — the original edge and its clone that has exactly the same parameters as an original edge only with a difference that it is marked as a *clone edge*.

When *module-limit* is reached for a vertex during *BFS*, the vertex can be still visited until the height of the constructed *morphological three* is increased. This approach has advantage that the resulting organism's morphology is not dependent on order in which vertexes are visited. Furthermore, the clone of a subtree (i.e. the subtree of *morphological tree* that was created during *BFS* using a *clone edge*) has always the same number of modules as the original subtree.

During the construction of a *morphological tree*, each attached module is placed to the position relative to its parent that is determined by parameters in the corresponding edge as described in the previous text. When the *morphological tree* is constructed, it is searched using depth-first search (*DFS*) from *root module*, where each cloned subtree is mirrored using a plane determined by *symmetry-type* stored in the genotype. Complete list of parameters of vertexes and edges in *morphological graph* is shows in Table 3.1.

Morphological graph is evolved during the evolution using genetic operations described in the previous chapter that include two types of mating (crossover and grafting) and mutations. Genotypes are produces by (1) first mating followed by mutation or by (2) mutation with higher mutation rate. Probability of mating starts at 20% and it is continuously increased during the evolution to 50%[3]. When

[2] This does not limit possible evolved organisms to only organisms with symmetries. However, organisms with symmetries are more likely to be evolved.

[3] From the beginning of the evolution mutations usually performs better compared to matting

Table 3.1: Parameters of vertexes and edges in *morphological graph*.

Vertex param.	**Description**
module-limit	determines the maximum number of modules that can be created from a vertex (note that this limit can be exceeded as described in the text above),
Edge param.	**Description**
location	determines the position of the pivot relative to the parent module,
rotational-joint-indicator	determines whether a joint is rotational or fixed,
connection-type	determines whether a plane symmetry is used,
symmetry-type	determines one of three possible planes of symmetry,
movement-symmetry	determines (1) whether rotation of the root module of the cloned subtree will be inverted (i.e. $z = -1$ in Formula 3.1) and (2) whether movement of non-root modules in the cloned subtree will be inverted,
axis-rotation	determines the angle (β in Figure 3.2) that defines a plane, where the module is rotated.

two genotypes are mated, crossover or grafting is randomly selected with equal probability.

Applied mutations include mutation of parameters and structural graph mutations. Structural mutations adds or remove vertexes in *morphological graph* or alter connections between them. Mutations of parameters of vertexes and edges correspond to the type of the parameter. Table 3.2 includes all used structural mutations and Table 3.3 includes all used mutations of parameters.

3.2 Evolving Control System

Evolving morphology of the virtual organism usually also requires co-evolving control system to allow (1) coordinated movement of the modules and (2) interaction with objects in the virtual world. Additionally, some methods to evolve robot controllers that are described in this section can be also used to encode and evolve morphology of the organism as shown in the next chapter.

This section describes proposed neural control system of the organism inspired by Haasdijk, Rusu and Eiben [4]. In addition to Haasdijk, Rusu and Eiben, two extension to the control system are added: (1) connections between local module controllers and (2) a central controller to coordinate organism movement. The connections between modules and central controller includes a short memory that can by useful for some tasks such as evolving robot locomotion. As argued by Haasdijk, Rusu and Eiben, this should make evolving control system easier than using isolated agents (i.e. modules) with reactive controller.

that is equivalent to a large mutation. On the other hand, at further generations mating allows gene exchange of evolved genotypes that may include an useful information.

Table 3.2: Structural mutations of *morphological graph*.

Mutation	**Description**
add-random-connected-vertex	creates a new random vertex that is connected to an existing vertex using a new random edge,
remove-random-vertex	removes a randomly chosen existing vertex including its incident edges (note that *starting vertex* can not be removed),
remove-random-unreachable-vertexes	uniformly removes unreachable vertexes,
swap-vertexes	swaps parameters of two randomly selected existing vertexes (except *starting vertex*),
add-random-edge	creates a new random edge between two randomly selected existing vertexes (note that source and target vertexes can be already connected or the same vertex),
remove-edge	removes a random existing edge,
change-edge-target	selects a random existing edge and changes its target vertex to another randomly selected existing vertex.

Table 3.3: Mutations of vertexes and edges.

Mutation	**Description**
increase-module-limit	increases *module-limit* by 1,
decrease-module-limit	decreases *module-limit* by 1 (applied only if *module-limit* is at least 2),
move-pivot	moves the pivot alongside *module axis* by a random value from normal distribution (i.e. adds a random value to ξ in Figure 3.1),
rotate-pivot	rotates the pivot around *module axis* by a random angle from normal distribution (i.e. adds a random value to ϕ in Figure 3.1),
rotate-movement-plane	adds a random value from normal distribution to the angle determining the plane of the module rotation (β in Figure 3.2),
change-edge-type	changes *simple edge* to *double edge* and *double edge* to *simple edge*,
change-joint-type	changes joint type (i.e. whether the joint is rotational or fixed),
change-symmetry	randomly changes the plane of symmetry,
change-movement-symmetry	changes the inverse movement indicators.

This section is further divided into three following subsections: First, Section 3.2.1 describes all used sensors of the organisms and its effector that realize movement of the modules. Next, Section 3.2.2 describes structure of the control system that includes description of communication between particular controllers and evaluation of the neural networks. Finally, Section 3.2.3 describes HyperNEAT generative representation of the neural networks including placing of neurons in the substrate space.

3.2.1 Sensors and Effectors

Sensors of the organism are distributed over its body to sense collision with objects in the virtual environment and its own state. Some sensors are designed for a particular task (e.g. light following) and they are disabled for other tasks. There are two types of sensors — (1) locals sensors stored in each non-root module and (2) central sensors associated with whole organism.

Local sensors include collision detection, relative module rotation sensor, module velocity sensor and optional light sensor for light-following task. Central sensors include a velocity sensor and optional light sensor that are an analogy of the local sensors related to whole organism. Outputs from sensors are directly connected to corresponding neural network controllers, where output values range from -1 to 1. This is achieved by linear scaling of inputs with bounded range (e.g. rotation of the module) and scaling using a sigmoidal function inputs with unbounded values (e.g. velocity). The following sigmoidal function is used:

$$\sigma(x) = \frac{2}{1 + e^{-\lambda x}} - 1. \tag{3.2}$$

Complete list of local sensors is included in Table 3.4.

Effectors of the organism determine parameters of module periodical motion, where desired angle is achieved by applying physical forces to modules as described in the previous section. In some cases a periodical motion of the modules may produce successful locomotion even using an initial random control system. As a result initial random population often contains organisms that are capable of a simple movement. However, more efficient locomotions usually require setting parameters of the periodical motion during the locomotion.

3.2.2 Structure of Control System

Haasdijk, Rusu and Eiben [4] used HyperNEAT generative encoding to evolve distributed control system for multi-robot organism consisting of simple connected robots (that correspond to modules in this work), where a single CPPN encodes control system of all robots within the multi-robot organism. Using additional inputs to the CPPN for the module positions within the organism allows modular differentiation, where local neural networks in two modules may differ in their weights.

In contrast to Haasdijk, Rusu and Eiben, evolving distributed control system is not goal of this work. Thus, additional central neural network connected to

Table 3.4: Organism's local sensors.

Sensor	Output values
rotation-sensor	module rotation around its pivot (α in Figure 3.2) linearly scaled to the range $[-1, 1]$,
collision-sensor	the value $\mathrm{s}(t)$ determined by the time from the last collision. The value is computed as follows: $$\mathrm{s}(t) = \begin{cases} -\left(\frac{t_{\max}-t}{t_{\max}}\right)^2 & \text{for } t \in [0, t_{\max}], \\ 0 & \text{otherwise,} \end{cases}$$ where t indicates the time from from the last collision and $t_{\max}$ is a predefined constant value,
velocity-sensor	values $\sigma(x), \sigma(y), \sigma(z), \sigma(v)$, where (x, y, z) indicates the velocity vector of the module, v indicates size of the velocity vector and σ is sigmoidal function given by Formula 3.2,
light-sensor (optional)	values $\cos(\alpha), \sin(\alpha)$, where α indicates angle between actual module velocity vector and direction to an object specific for a particular task (e.g. followed object or an obstacle), note that this sensor is enabled only for some specific tasks (otherwise it returns zero values).

each local network and further connections between local neural networks are added that should allow (1) faster propagation of information between distant modules and more importantly (2) possibility of evolving centralized organism control.

The following text describes realization of the control system that consists of local neural networks for each non-root module and a single central neural network. Local and central neural networks are layered neural networks that have a single hidden layer and sigmoidal activation function given by formula 3.2. The hidden layer includes a fixed predefined number of neurons, number of neurons in input and output layers depends on number of neighboring modules.

The system of connected local and the central neural networks is equivalent to a single recurrent network. The communication between networks is following: Similarly to Haasdijk, Rusu and Eiben [4], each local control receives output from collision sensors of its neighbor non-root modules. Additionally, three more input and output neurons for each neighboring local control are used to propagate signal between neighboring modules.

As indicated in the previous text, local networks are connected to the central network. The communication with the central network is realized using the same number of neurons as with a neighboring local network. Therefore, the number of neurons within the central network is linearly correlated to the number of non-root modules.

The neural networks are evaluated in predefined equidistant time-steps of the simulation. The evaluation consists of three substeps that are performed

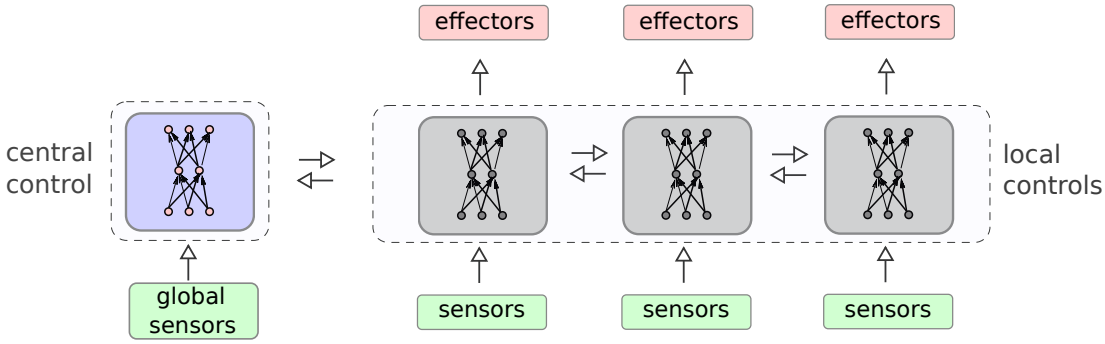


Figure 3.3: **Structure of the neural control system.** The figure shows control system of an organism that consists of 4 modules (a single root module and three non-root modules). Each non-root module has its local control (gray squares on the right). The root module is associated with the central control (blue square on the left). The arrows indicate communication between modules, sensors and effectors. Note that for clarity only a single couple of arrows in shown for communication between the central network and each local network.

in the following order: First, inputs of local networks are initialized by evaluating local sensors and from stored memory that contains output values from the previous time-step (zero values in the first time-step). Next, when inputs of local networks are initialized, their outputs can be computed. The outputs of neurons that are associated with input neurons of the neighboring local networks and the central network are stored for the next time-step, other outputs determine parameters of the module rotation. Finally, central network is evaluated and its outputs stored in the memory of local networks. The described communication between neural networks, sensors and effector is shown in Figure 3.3.

3.2.3 Generative Representation

The control system of the organism is represented in the genotype using HyperNEAT generative encoding, where a separate CPPN is used to represent each type of networks — *local CPPN* represents the local neural network and *central CPPN* represents the central neural network. As a result, the genotype of the organism consists of completely 3 components — (1) *morphological graph* that represents a robot body, (2) *local CPPN* that represents weights of the local networks and (3) *central CPPN* that represents weights of the central control, where each is mutated and mated separately. The following text describes placing neurons in the substrate spaces.

This work uses more dimensions for neuron positions in substrate space compared to Haasdijk, Rusu and Eiben [4]. This increases a freedom of the evolved controllers by adding more inputs to the evolved CPPN. Substrate space of local neural networks includes completely 5 following dimensions: First dimension corresponds to the layer of the network. Next dimension is associated with a group of neurons in the layer. Last three dimensions store additional information specific for the group of neurons that usually corresponds to a position or a direction in the 3D-world. For instance, vector inputs such as velocity are represented by a single group of neurons, where each input corresponds to a neuron with a different position (i.e. they differ in the last three coordinates).

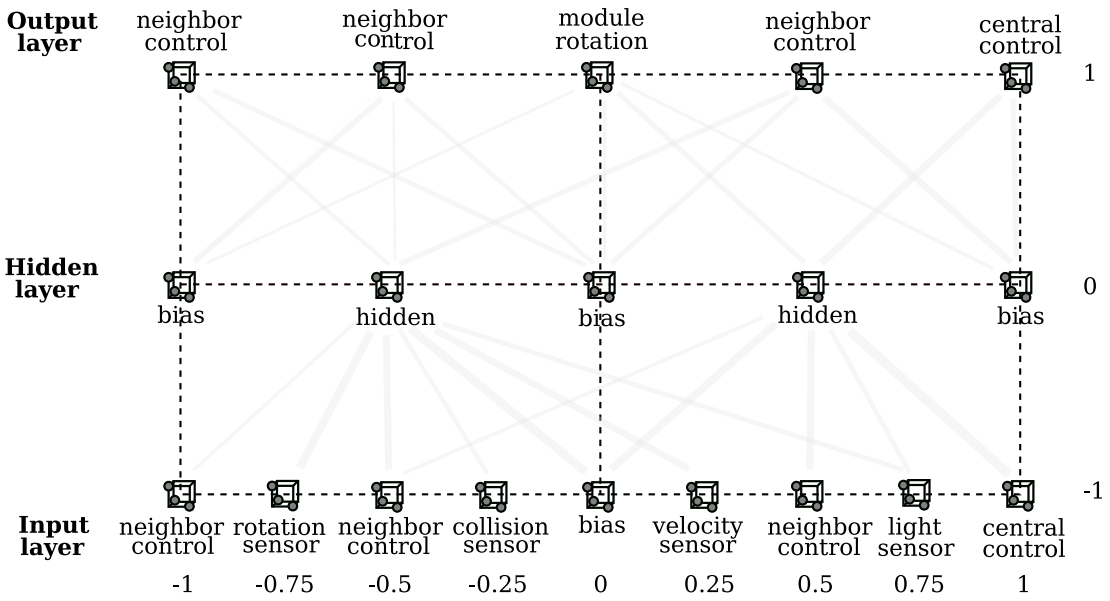


Figure 3.4: **Local neural network control placed into 5-dimensional space.** Only first two coordinates (x_1, x_2) that determine group or neurons are shown, where x_1 corresponds to the layer and x_2 corresponds to the position of group in the layer. Each group of neurons may contain one or more neurons that differ in other three coordinates. Note that drawn synapses show only an illustration of possible connections between groups of neurons.

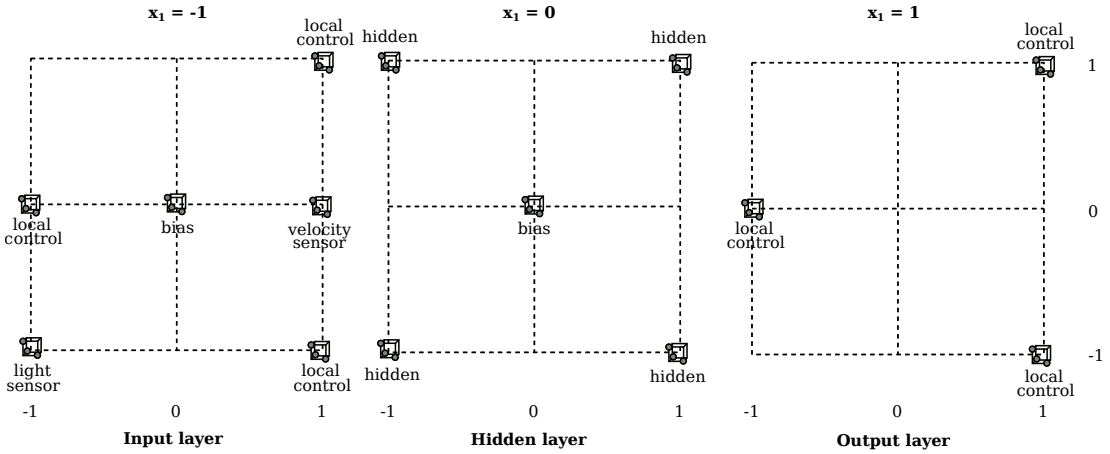


Figure 3.5: **Central neural network control placed into 6-dimensional space.** Only first three coordinates are shown here. First coordinate (x_1) corresponds to the layer and second two coordinates (x_2, x_3) correspond to group of neurons within the layer. Note that synapses between layers are not shown for clarity.

The number of CPPN inputs is linearly correlated to the substrate dimension. Each dimension in the substrate corresponds to the following 3 inputs in *local CPPN*: (1) source neuron coordinate, (2) target neuron coordinate and (3) their difference (i.e. $y_i - x_i$, where x indicates source neuron position and y indicates target neuron position).

To allow differentiation of local controllers, an approach adopted from Haasdijk, Rusu and Eiben [4] is used, where *local CPPN* includes additional 4 inputs for module positions: a normalized relative module position (i.e. an unit vector) related to the root module position (in the moment the phenotype was created) and the distance from the root module. Positions of neurons in the local neural network are shown in Figure 3.4 and further described in Table 3.5.

When organism's morphology exhibits symmetries (e.g. bilateral symmetry) the control system should reflect these symmetries in order to allow effective locomotion. Therefore, in addition to regularities in the local neural controls that are result of HyperNEAT generative encoding, control system includes a mechanism to reflect symmetries of the morphology in the control system: when a subtree of the *morphological tree* is cloned to create a symmetrical part of the organisms, each local control within the subtree is also cloned, thus the clone of the subtree includes identical local neural controllers with possibility of inversion of movement as described in the previous section (see Section 3.1.2). This produces symmetrical organisms that exhibit a symmetrical locomotion of the symmetrical parts.

The central neural network is usually larger compared to the local neural networks, it has been also allowed more freedom to the evolve more complex high-level control. This is achieved by adding one more dimension to its substrate space compared to local neural networks. The substrate space includes completely 6-dimensions, where each dimension corresponds to 3 inputs as in local neural networks. Unlike local networks, no CPPN inputs for differentiation are needed. Position of the neurons in the central control substrate space are shown in Figure 3.5 and further described in Table 3.6.

The described configuration of neurons in the substrates is result of several augmentations of the initial configuration of the author of this thesis [12]. Resulting substrates configuration significantly improves evolved organisms compared to previous method of the author. Two additional enhancements have been realized from the previous work: (1) CPPN neurons includes a parameter that determines used activation function to evolve control systems that exhibits regularities as proposed in the HyperNEAT algorithm [8] and (2) local neural networks in neighboring modules are connected to propagate information through the organism as described in the previous section. List of activation functions is included in Table 3.7.

Table 3.5: Structure of local neural networks.

Neuron group	**Position in the substrate** (x_3, x_4, x_5 coordinates)
rotation-sensor	single neuron with zero coordinates,
collision-sensor	single neuron with zero coordinates for collisions of the module of this local control and one additional neuron for each neighbor module collision, where position of the neuron corresponds to the relative position of the neighbor module,
velocity-sensor	four neurons with coordinates $[1,0,0]$, $[0,1,0]$, $[0,0,1]$ for a velocity vector and $[0,0,0]$ for its size,
light-sensor	two neurons with coordinates $[1,0,0]$ and $[0,0,1]$ for $\cos(\alpha)$ and $\sin(\alpha)$ (see Table 3.4), respectively,
bias, hidden	8 neurons with coordinates $[\pm 1, \pm 1, \pm 1]$ (i.e. vertexes of a 3D-cube)
module-rotation	three neurons with coordinates $[1,0,0]$, $[0,1,0]$ and $[0,0,1]$ that correspond to phase, angular velocity and amplitude, respectively (i.e. ϕ, ω and A in Formula 3.1),
central-control	three neurons with coordinates $[1,0,0]$, $[0,1,0]$ and $[0,0,1]$ for communication with the central neural control,
neighbor-control	a single neuron for each neighbor module local neural control, where position of the neuron corresponds to the relative position of the neighbor from this module.

Table 3.6: Structure of the central neural network.

Neuron group	**Position in the substrate** (x_4, x_5, x_6 coordinates)
velocity-sensor	four neurons with coordinates $[1,0,0]$, $[0,1,0]$, $[0,0,1]$ for a velocity vector and $[0,0,0]$ for its size (note that unlike local networks this sensor is related to the organism‘s center of the mass),
light-sensor	two neurons with coordinates $[1,0,0]$ and $[0,0,1]$ for $\cos(\alpha)$ and $\sin(\alpha)$ (see Table 3.4), respectively (note that unlike local networks this sensor is related to the organism‘s center of the mass),
bias, hidden	8 neurons with coordinates $[\pm 1, \pm 1, \pm 1]$ (i.e. vertexes of a 3D-cube)
local-control	a single neuron for each non-root module, where position of the neuron corresponds to the relative position of the corresponding module (i.e. completely three neurons are used for communication with local modules).

Table 3.7: CPPN activation functions.

Activation function	**Definition**
sigmoidal function 1	$\mathrm{f}(x) = \dfrac{2}{1+e^{-x}} - 1$
sigmoidal function 2	$\mathrm{f}(x) = \dfrac{1}{1+e^{-x}}$
linear function	$\mathrm{f}(x) = \begin{cases} -1 & \text{for } x \in (-\inf, -1), \\ x & \text{for } x \in [-1, 1], \\ 1 & \text{for } x \in (1, \inf), \end{cases}$
harmonic function	$\mathrm{f}(x) = \sin(x)$
exponential function	$\mathrm{f}(x) = \begin{cases} e^x & \text{for } x \in (-\inf, 1), \\ e & \text{for } x \in (1, \inf), \end{cases}$
gaussian function	$\mathrm{f}(x) = e^{-x^2}$
absolute value	$\mathrm{f}(x) = \begin{cases} \lvert x \rvert & \text{for } x \in (-1, 1), \\ 1 & \text{otherwise}, \end{cases}$

4. The Second Proposed Method: Hypercube-based Artificial Ontogeny

Existing methods using *artificial ontogeny* seem to be outperformed by methods using more direct encoding with higher level of abstraction such as *NEAT* or *HyperNEAT* for evolving neural structure and directed graphs for evolving morphology. However, even very successful methods such as *HyperNEAT* are not capable of evolving complex structures that can be seen in the nature. Thus, more biologically inspired methods could provide a solution to develop new advanced methods for both neuroevolution and evolving morphology.

This chapter introduces a new method for evolving virtual organism denoted as *Hypercube-based artificial ontogeny* (*HyperAO*) that combines two methods described in the previous text: artificial ontogeny and Hypercube-based neuroevolution of augmenting topologies (HyperNEAT). The new method focuses primary of morphology of the organism. The main innovation of the new method is using a powerful method of neuroevolution — HyperNEAT to evolve genetic regulatory network (GRN) that determines organism's morphology through an ontogenetic process. This should take an advantage of HyperNEAT generative encoding to evolve regularities in GRN that might improve evolved phenotypes.

The chapter is divided into five sections. Section 4.1 compare results from various works related to both developmental methods (in particular artificial ontogeny) and other methods for evolving virtual organisms. Section 4.2 describes a high-level overview of the proposed method and provides a comparison with other methods. Section 4.3 describes a mapping from a genotype to the phenotype that is realized via a developmental process. Section 4.4 focuses on genetic regulatory network represented using HyperNEAT generative encoding. Section 4.5 formulates hypotheses related to the proposed method.

4.1 Motivation to the Proposed Method

There is a trade-off between developmental systems such as *artificial ontogeny* and methods using more direct representation. Even though several studies have been made [6, 10, 11] in developmental systems, it seems that they are still poorly understood. This section discusses and summarizes results from several studies related to developmental systems (in particular artificial ontogeny) and more direct methods for evolving virtual organisms that are motivation for the proposed method.

The section is divided into four subsections. First section, Section 4.1.1 compares direct and indirect encoding. Next section, Section 4.1.2 focuses on level of abstraction of biologically inspired methods and Section 4.1.3 overviews a recent developmental method for evolving morphology that skips simulation of a developmental process. Finally, Section 4.1.4 discusses advantages of a simulation of developmental process.

4.1.1 Indirect Encoding and Regularities

Roggen and Federici [10] argued that for smaller search spaces the direct encoding usually performs better compared to indirect encoding (e.g. developmental encoding) that typically suffer from gene reuse (pleiotropy), when one gene affects two or more unrelated phenotypic traits. Thus, improving a particular phenotypic trait may corrupt another phenotypic trait that is affected by the same gene. Unlike indirect encoding, direct encoding can evolve each part of the solution independently.

As a number of evolved parameters increases, due to "combinatorial explosion" of possible phenotypes, evolving a solution using a directed encoding becomes much harder. In this case, indirect encoding has an advantage of reduced search space. Indirect encoding may be also specialized for some particular problem domain, when an initial knowledge of a problem may be useful for designing suitable encoding of the solution. On the other hand, it may restrict possible phenotypes to some suboptimal subset of the all phenotypes that could be expressed using a direct encoding.

For some problems a solution that exhibits some regularities is necessary to solve the problem. Such problems include evolving complex structures similar to structures seen in the nature (e.g. human brain). Gauci and Stanley [8, 9] argued that most important regularities are: symmetry (in particular bilateral symmetry), repetition, imperfect symmetries, imperfect repetition and hierarchical structures. Regularities are result of gene reuse, where one or more genes are used several-times during the construction of the phenotype. Obviously, it is not possible (or unlikely) to evolve phenotypes that exhibit such regularities using a direct encoding. Therefore, indirect encoding has to be used to evolve complex structures that exhibit such regularities.

4.1.2 Level of Abstraction

To design new evolutionary methods capable of producing complex phenotypes that exhibit regularities, biology can provide a valuable inspiration. Similarly to direct and indirect encoding, there is a trade-off between "less indirect" methods (i.e. higher level of abstraction) and "more indirect" methods (i.e. lower level of abstraction). It may be difficult to determine suitable level of abstraction for a problem. Low level of abstraction usually results in computationally-intensive simulation (e.g. developmental process, simulation of cells). On the other hand, high level of abstraction might loose some beneficial biological properties including lack of regularities and other properties discussed further in this section.

Level of abstraction in artificial ontogeny proposed by Bongard and Pfeifer [5] is lower (i.e. it is more biologically realistic) compared to the most of the methods that evolve virtual organisms that use more direct parametric encoding schemes (e.g. Sims [7]). Unlike these more direct methods, artificial ontogeny includes a developmental process that is realized through repeated division of units that form its body, which is similar to biological cellular development.

On the other hand, artificial ontogeny uses a higher level of abstraction compared to more biologically accurate models of cellular developments. Model of growth used in AO is inspired by cellular development to create relatively continuous transition from genotype to the phenotype. It was shown by Bongard and Pfeifer that *artificial ontogeny* is capable of evolving organisms with hierarchical repeated structures that are part of fundamental properties for biological systems.

4.1.3 Skipping Developmental Process

Creating a large number of cells during the process of development in *Artificial ontogeny* may result in computationally-intensive simulation. However, new developmental methods [16, 17] have been recently introduced that can produce phenotype directly — without simulation of the development.

The more recent method (Cheney, Nick, et al. [16]) evolves phenotypes that consist of soft material voxels inside a 3D grid. The phenotype in created from multiple types of voxel, where some of them undegro periodic volumetric actuations to realize movement of the organism. The method represents genotypes using a single CPPN that for each voxel in the grid directly determines whether a voxel will be included in the organism and its type (i.e. material). Thus, construction of the phenotype is significantly less computationally-intensive compared to simulation of developmental process.

Furthermore, this approach allows resolution of the organisms to be easily scaled. Then, it is possible to first evolve organisms with lower resolution which is less computationally-intensive and in later generations increase the resolution to optimize organisms for the final resolution. Figure 4.1 [16] illustrates construction of the phenotype for a given *CPPN* and shows examples of evolved organism by this method.

4.1.4 Advantages of the Developmental Process

The described method that skips process of growth provides further abstraction over a developmental process. Yet it is still capable of producing phenotypes that exhibit some properties of developmental methods (e.g. repeated structures with variations, symmetries, etc.). Despite the ability of evolving regularities, simulating of developmental process might still provide some advantages.

An important property of developmental process in the nature is ability to express a single genotype into different phenotypes for different environments. Both artificial ontogeny and the CPPN representation can be modified to be adaptive on the environment. However, adaptive extension of artificial ontogeny is (1) more straightforward compared to the *CPPN* representation that seems to be more limited as argued in the following text.

The CPPN representation of the organism uses only a local information (i.e. position on the grid) to determine a phenotypic element at the corresponding position. This allows to skip a developmental process, on the other hand, it also limits possible adaptation that could be realized during the growth. For instance, AO has an ability to alter trajectory of the growth when it discover some physical constraint (e.g. obstancle in the direction of the growth). This is not, however,

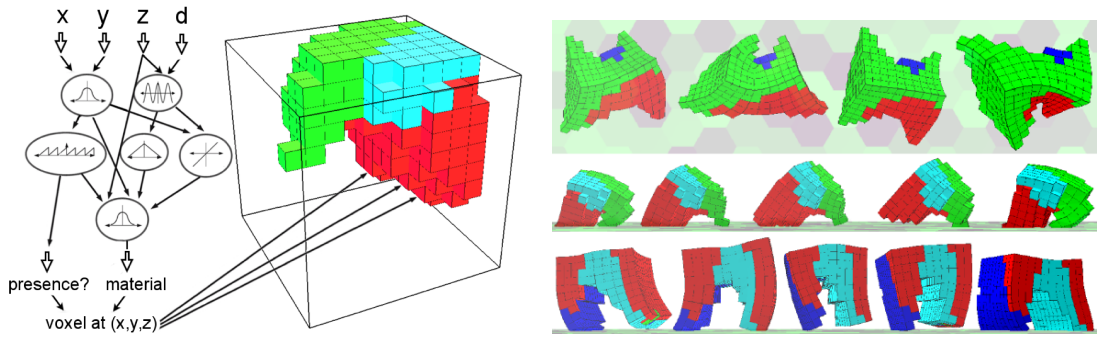


Figure 4.1: **Organisms evolved by Cheney, Nick, et al. [16]** The figure shows organisms evolved using the *CPPN-NEAT* encoding. Left picture illustrates mapping from genotype (i.e. *CPPN*) to phenotype. To create a phenotype a *CPPN* is queried for each voxel and its output values determine presence of voxels and their material. Right picture shows examples of evolved organisms.

possible with *CPPN* representation that uses only a local information.

Morphological adaptation to the environment is not limited to the initial growth of the organism. Bongard [14] showed that if an organism has an ability to exhibit morphological change during its lifetime, it can accelerate the evolution and evolved organisms are more robust compared to organisms evolved without this ability. Extending AO with such ability could be very beneficial. However, developing such method capable of morphological change during the organism‘s lifetime combined with a developmental process would require further investigation.

4.2 Structure of the Proposed Method

This section provides an overview of the proposed method that is a combination of two approaches described in the previous text: (1) HyperNEAT developmental encoding [8] — powerful developmental method without developmental process that is used in the first proposed method to evolve control system of the organism that include module differentiation based on module position and (2) artificial ontogeny [5] — more biologically accurate developmental method that uses an ontogenetic development to grow morphology and neural network control of the organism. The following text describes main principles integrated into the proposed method and differences from the artificial ontogeny proposed by Bongard and Pfeifer.

The proposed method aims to combine advantages of several approaches: possibility of adaptation during the growth process (or additional morphological adaptation during the lifetime), evolving high degree of regularities using Hyper-

NEAT generative encoding (that also includes advantages of NEAT: speciation, growth from minimal structure and sensitive mating) and hierarchical repeated structure using genetic regulatory network.

To achieve this, the proposed method is combination of: (1) control system of the first proposed method that consists of local and a single central neural networks, (2) ontogenetic process, where organisms grows based on concentrations on gene products and (3) HyperNEAT generative encoding for evolving genetic regulatory networks. The following text provides more detailed description of used methods.

The main difference from the original artificial ontogeny by Bongard and Pfeifer [5] is realization of the genetic regulatory network (GRN). Instead of "if-then" rules (i.e. *GRN rules*) that are applied in case that their regulating gene product concentration lies within a given range, GRN in *HyperAO* is realized by a system of neural networks, where each module includes a single neural network. Furthermore, unlike AO, the neural net woks may differ more two modules, this should give evolved GRN more flexibility compared to AO.

In addition to the neural network placed in modules, GRN includes an another neural network that can be thought of as an analogy of the central neural control in the control system described in the previous chapter. To distinguish between these two types of networks, neural networks placed in the modules will be further denoted as *local GRNs* and the additional neural network as *central GRN*. The *central GRN* is an experimental enhancement of the genetic regulatory network that might contribute to evolving high-level components or other regularities in the morphology.

Each type of the neural networks that realize genetic regulatory network are represented using a Compositional pattern-producing network (CPPN) that encode their weights and allows differentiation of *local GRNs* that is similar to module differentiation in the control system. As a result genotype of the organism consists of four CPPNs that are mutated and mated independently. Generative representation of *local GRNs* and *central GRN* is further described in Section **??**.

The next important difference from the original artificial ontogeny is using *morphological graph* from the first proposed method that allows evolving additional symmetries in the phenotype morphology. In the process of growth, first *morphological graph* is constructed that is continuously expressed during the growth to the phenotype, where the *morphological graph* is expressed to the final organism using the same method as described in the previous chapter. This allows evolving all symmetries realized in the first proposed method that include bilateral symmetry and other plane symmetries. More detailed description of the developmental process in included in Section 4.3. Figure **??** shows comparative shema of three representations: artificial ontogeny, directed graphs and HyperAO.

4.3 Developmental Process

This section describes the developmental process that uses genetic regulatory networks to create *morphologial graph*, which is translated to the complete organism using the method described in the previous chapter (see Section **??**).

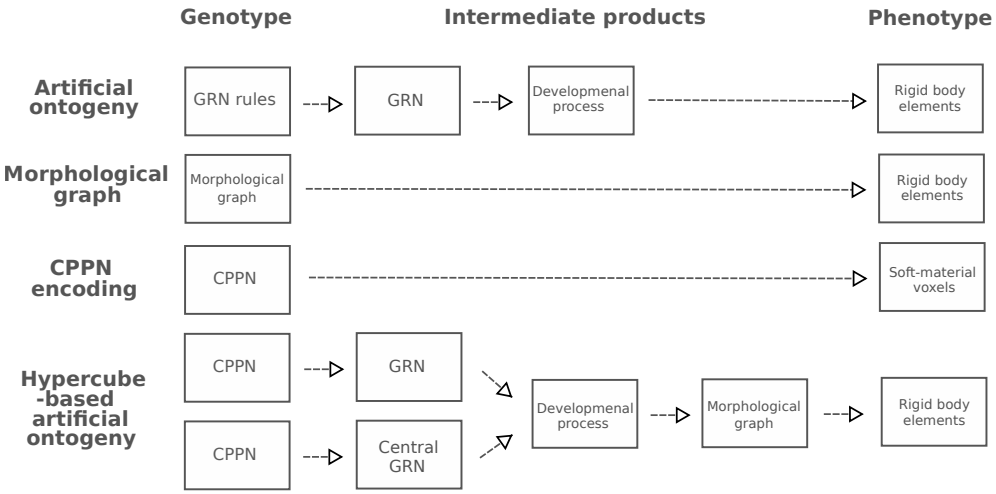


Figure 4.2: **Comparing different approaches of evolving morphology.** This figure provides comparison of four approaches to evolve morphology of the virtual organisms: Artificial ontogeny by Bongard and Pfeifer [5], Morphological graph used by Sims [7], CPPN-NEAT encoding proposed by Cheney, Nick, et al. [16] and Hypercube-based artificial ontogeny proposed in this work. Columns compare particular aspects of these methods: Column Genotype indicates structure that is directly encoded in the genotype, column Phenotype indicates a physical realization of the phenotype morphology and column Intermediate products indicates additional substeps that are used to create a phenotype. Note that some details of described methods are omitted for clarify.

The developmental process is an interative process that consists of a number of substeps — *growth iterations*, where the number of *growth iterations* is stored in the genotype.[1] During the growth *morphological graph* is constructed from a single initial (root) vertex. Unlike the method proposed in the previous chapter, *morphological graph* in *HyperAO* is always a tree. Thus, no module limits are needed to for infinite phenotypes. Each vertex has a single associated module — non-clone module that is expressed from the vertex. To reduce computational intensity of the growth, gene products are simulated only for non-clone modules denoted as *GRN modules*. Each *GRN module* (and there fore each associated *morphological vertex*) includes completely 27 diffusion sites with positions the following positions on a unit sphere:

$$\left\{ \frac{(x,y,z)}{|(x,y,z)|} : x,y,z \in \{-1,0,1\} \right\}$$

in the local module coordinates, where $[0,0,0]$ is centroid of the module.

Each diffusion site has its *growth potential* computed as difference of two gene products concentrations described further in this section. When some of diffusion sites within the module reaches a predefined threshold a new vertex is attached to the morphological graph, where parameters of created edge are determined based on the concentration of gene products. The diffusion site with maximum growth potential is then associated with the attached vertex and it is connected with a predefined diffusion site (diffusion site with coordinates $[-1,0,0]$) in the

[1] Note that alternatively other method could be used to determine whether the growth should be terminated. This would require further investigation.

new vertex. Note that diffusion site in the center of the module is not allowed to create new vertexes, furthermore each diffusion site not allowed to create more that one vertex.

In addition to diffusion sites, each *GRN module* includes *local GRN* that affects gene products concentrations within the module. The *local GRN* is created when new vertex is attached to *morphological graph*. To determine synapses weight of *local GRN* using a CPPN, position of the module must be known, therefore a phenotype based on the actual *morphological graph* is created.[2] When a new vertex is attached *central GRN* is updated, when a single input and output neurons are added for each gene products. Both types of GRNs are further described in the next section.

The diffusion of gene products within the organism is similar to diffusion in artificial ontogeny, it includes the following factors: (1) production of gene products by local GRN and the central GRN based on the actual concentration, (2) diffusion between neighboring diffusion sites and (3) diffusion out of the organism. Unlike artificial ontogeny, here concentrations are allowed to be negative, where concentration of each gene product ranges from $[-1, 1]$. Negative concentrations can be thought of as a positive concentration of an another gene product with exactly opposite effect. As a result diffused amount of gene product can be negative and diffusion out of the organisms is realized by decreasing of absolute concentration by a small ratio.

Diffusion between neighboring sites depends on distance of two diffusion sites. Higher amount is diffused between diffusion sites withing a module and smaller amount is diffused between diffusion sites placed in two different modules. A neighboring diffusion sites within the module forms a 3D grid that is transformed into a sphere in order to have same distances of diffusion site of the centroid of the module. Diffusion of gene products within the organism is show in Figure **??**.

During construction of *morphological graph* gene product concentrations are used to: (1) determine whether a new vertex will be attached and (2) in case when a new vertex is created they determine parameters of the edge that connects the vertex to its parent. Completely 15 gene products are used, where some of them encode edges properties and rest of them have no direct phenotypic effect. To describe encoding of edges parameters using the gene product concentrations, we use the following notation: We denote gene product concentrations as [*gene product name*] (e.g. [*growth enhancer*] indicate concentration of the *growth enhancer* gene product that is one of 15 used gene products). Complete list of edges parameter encoded using gene product concentrations is included in Table 4.1. The main steps of the described developmental process are summarizes in the following pseudocode.

[2]Constructing a phenotype morphology based on the actual *morphological graphs* can be also used for visualization of the growth that is useful for analysis of the growth process and adjusting its parameters.

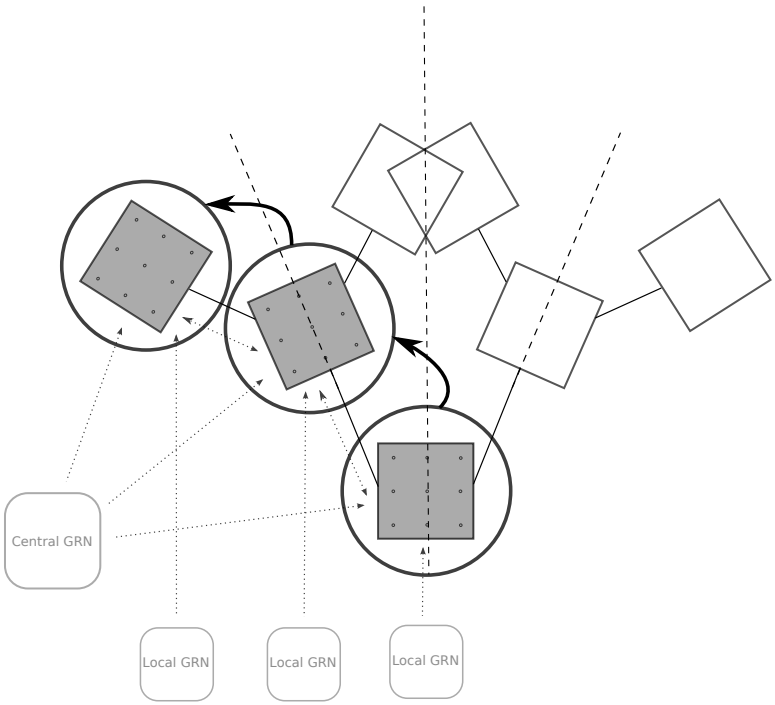


Figure 4.3: **Diffusion in the organism.** Figure shows a sample organism with two plane symmetries. Grey modules indicate *GRN modules*, where a diffusion process is realized. Bold cycles around modules indicate corresponding vertexes in the *morphological graph*.

Algorithm 1: DevelopmentalProcess

```
morphGraph ← create a morphological graph with a single vertex;
iteration ← 0;
while iteration < growthIterations do
    GrowthIteration;
create a physical organism and its control system;
```

Procedure GrowthIteration

```
/* Updating gene products concentrations.                    */
compute local GRNs outputs;
compute central GRN outputs;
apply changes to gene products concentrations;
compute amount of diffused gene products concentrations;
perform diffusion;
foreach vertex ∈ morphGraph do
    diffSites ← DiffusionSites (vertex);
    maxGP ← max{GrowthPotential(ds) : ds ∈ diffSites};
    if maxGP > threshold then
        /* Adding new vertex to the morphological graph. */
        attach a new vertex to the vertex;
        determine parameteres of the edge depending on
        concentrations in the diffusion site;
        connect corresponding diffusion sites of two vertexes;
        compute position of module that corresponds to the new
        vertex;
        create local GRN for the new vertex;
        add new neurons to the central GRN;
```

Table 4.1: Representation of *morphological edges* in *HyperAO*.

Edge param.	**Encoding by gene products concentrations**
location	direction of an attached module is given by the unit vector defined by the following formula: $\frac{\sum_{d\in \mathrm{DS}} \mathrm{position}(d) \cdot \mathrm{growpotential}(d)}{\sum_{d\in \mathrm{DS}} \mathrm{growpotential}(d)}$, where growpotential = [*growth enhancer*] − [*growth repressor*],
rotational-joint-indicator	the joint is rotational if [*rotational enhancer*] ≥ -0.1, it is fixed otherwise,
connection-type	a plane symmetry is used in case that [*symmetry enhancer*] − [*symmetry repressor*] + $\alpha - \beta n > 0$, where α, β are predetermined constant values n indicates number of edges that uses a plane symmetry,
symmetry-type	three gene products are used to encode symmetry-type, chosen symmetry corresponds to the gene product with maximum concentration,
movement-symmetry	two indicators of movement inversion are determined by two gene products: *root movement inv.* and *subtree movement inv.*, inversion is used in case that the corresponding gene product concentration is negative,
axis-rotation	axis-rotation angle is determined by an angle between vector: ([*rotation-x*], [*rotation-y*]) and unit vector $(1, 0)$.

4.4 Genetic Regulatory Networks

As indicated in the previous text, in this method GRNs are realized using neural networks that are encoded by two CPPNs. This approach has several advantages: (1) representation of GRN using a neural network allows use of neuroevolution method to evolve GRNs, (2) it also allows evolving more complex GRNs than "if-then" rules and finally (3) using HyperNEAT allows differentiation of GRNs based on their position in the organism that can be interpreted as analogy of chemical concentrations in the organism.

Central GRN and *Local GRNs* are layered networks with a single hidden layer. Architecture of Local GRNs is identical (except that synapses with small weights are removed), where for each *diffusion site* and each gene product within the *diffusion site* a single input and output neuron is used. Unlike Local GRNs that are fixed, Central GRNs grows during a developmental process. Central GRN includes a single input and output neuron for each gene product of the central diffusion site (i.e. difuusion site with position $[0, 0, 0]$) within each GRN module. Therefore, size of the Central GRN is smaller than Local GRNs until size of the organism reaches 27 modules.[3]

Generative representation of GRNs is similar to neural network in the control system. Local GRNs are placed into 7-dimensional substrate space, where the first coordinate indicates layer of the network, next three dimensions indicate position of a gene product and the last three dimensions corresponds to the position of diffusion site in the module.

Positions of gene products are identical within the organism and they are evolved together with GRNs. In the initial population a position is chosen randomly for each gene products from interval $[-1, 1]^3$. Mutations and mating gene products positions are performed independently for each gene product. Mutation is realized by adding a random vector from a 3D-normal distribution to the position. Mating is realized by computing a random (uniform) convex combination of two positions.

Central GRN is placed into 6-dimensional substrate space, where first three coordinated correspond to positions of gene products and last three positions to the relative position of module from the root module.

The resulting genotype in *HyperAO* consists of: (1) four CPPNs (from which two represent control system and two morphology), (2) a single integer value that represents number of growth iterations and (2) positions of gene products (i.e. 27 vectors from range $[-1, 1]^3$), where each component is mutated and mated independently.

[3] Note that the Central GRN is inspired by both the central neural control described in the previous chapter and CPPN-NEAT encoding [16] that produce phenotypic elements based on their position in the organism. However, unlike CPPN-NEAT encoding, where CPPN directly determine phenotypic elements based on positions in the organism, *central GRN* affects phenotypic elements more indirectly — through the following intermediate products: (1) CPPN is used to produce substrate (i.e. *central GRN*) and (2) the substrate that produce *gene products* that are responsible for growing new phenotypic elements.

4.5 Hypotheses to the Proposed Methods

It is believed that this modification to the initial artificial ontogeny might improve original method. Furthermore, *HyperAO* might also outperform method proposed by Sims [7] for some tasks. This leads to the following hypotheses:

Hypothesis 1. *Hyper-cube based artificial ontogeny produces significantly better organisms than the original artificial ontogeny.*

Hypothesis 2. *Hyper-cube based artificial ontogeny is capable of producing comparable or better organisms compared to the first proposed method that evolves morphological structure using directed graphs.*

Note that the stated hypotheses are tested in later chapter (see Section **??**) through performing series of experiments.

5. Distributed computing

Fitness evaluation in evolutionary robotics is in most cases computationally demanding task. However, population to be evaluated can be easily divided into multiple subpopulations and the fitness evaluation for each subpopulation can be realized independently. Therefore, the developed application uses distributed fitness evaluation to decrease evaluation time and allow more complex simulations in shorter time.

In this chapter we focus on how the time for an evaluation can be decreased with various degrees of distribution. First, we describe structure of the developed application in Section 5.1, which is the basis for the following sections. Next, in Section 5.2 we develop a theoretical model of the fitness evaluation time with varying degree of distribution. The final Section 5.3 compares this theoretical model with the performance obtained in real experiments.

5.1 Modular architecture overview

This section introduces two main components of the developed application — an *evolution application* and *runners*.

The *runner* is a program designed for a fitness evaluation of virtual organisms. It receives a genotype from the *evolution application*, creates a phenotype and then evaluates a fitness value in a simulated physical environment. The evaluation of a single genotype consists of the following steps:

(a) the *runner* receives a genotype from the *evolution application*,

(b) a phenotype is created,

(c) the complete organism is simulated in a physical virtual environment,[1]

(d) the *runner* sends score to the *evolution application*, and

(e) waits for a next genotype.

Multiple instances of the *runner* is usually used (therefore multiple genotypes can be evaluated simultaneously). To run *evolution application* at least one *runner* is required.

The *evolution application* is the application, where the core evolutionary algorithm is implemented. It has two logical phases:

(i) producing a new generation (i.e. selection, mating and mutations) and

(ii) the fitness evaluation.

[1] Genotype receives a score based on its behavior on a specific task, fitness value is usually weighted sum of score and some other organism properties (see [[odkaz na fitness evaluation]]).

The first phase is not distributed (i.e. no *runners* are used) and it takes usually less than a second. The second phase takes usually from several minutes to a few hours.[2] Therefore, we focus on the second phase – the fitness evaluation, which is in most cases far more computationally-intensive than producing a new generation.

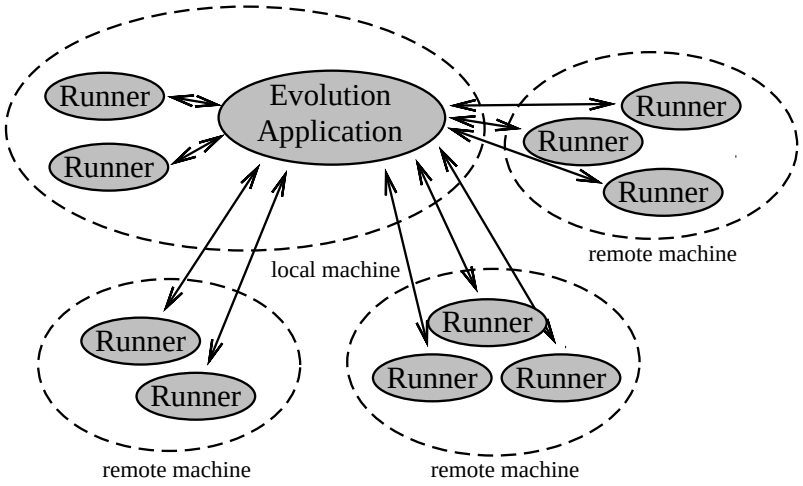


Figure 5.1: Communication between the *evolution application* and *runners* on local and remote machines.

To evaluate a fitness value of each genotype, the *evolution application* uses the *runners*. It sends a genotype to a *runner* and waits for a score obtained for a particular task. When a score is received from a *runner*, the *evolution application* sends another genotype to this *runner* to be evaluated. When scores from all genotypes from the current generation are received, a new generation is created. Figure 5.1 illustrates the communication between the *evolution application* and the *runners*.

The following sections focus on approximating the evaluation time with respect to the number of *runners* — both theoretical models and experimental results are considered. We show that performance increases almost linearly with the number of *runners*.

5.2 Theoretical approach to performance increase with multiple runners

Ideally fitness evaluation speed increases linearly with the number of *runners*. However, there are some limitations and factors that might reduce the efficiency of an evaluation with multiple runners (e.g. limited communication bandwidth, an overhead associated with distributing genotypes to the *runners*).

[2]The time depends on degree of parallelization, performance of used computers and properties of the experiment (e.g. complexity of simulated environment, time duration of simulation, number of repeating etc).

In this section we describe three different approaches to analyze effectivity of distributed fitness evaluation. First, Section 5.2.1 develops a simple model of distributed fitness evaluation with certain simplifying assumptions. Further text focuses on various properties and special cases of this model — Section 5.2.2 focuses on asymptotic performance of an evaluation with large populations, Section 5.2.3 applies results from the previous section on *runners* with the same performance. Finally, in Section 5.2.4 we analyze parralel fitness evaluation for running time of a population with fixed size and various number of *runners* with the same performance.

5.2.1 Preliminary assumptions and definitions

First, we define the following properties:

T_{gen}	the time for processing one generation
T_{total}	total running time for N generations
N	the number of generations

We assume that the running time for each generation is the same,[3] therefore

$$T_{\text{total}} = N \cdot T_{\text{gen}}.$$

Now we can focus on running time of a single generation.

We further divide T_{gen} into the fitness evaluation time and the time for producing a new generation.

$$T_{\text{gen}} = T_{\text{eval}} + T_{\text{evol}}.$$

T_{eval}	the fitness evaluation time
T_{evol}	the time for producing a new generation (without fitness evaluation)

As mentioned in the previous section, producing the next generation takes far less time in comparison to the fitness evaluation.[4] Therefore, we do not distribute it thus, producing a new generation is not influenced by the number of *runners*. We focus primary on fitness evaluation time in the following text.

We denote:

[3] As organisms become larger or more complex during the evolution, physical simulation might become more computationally-intensive. Therefore, it could take more time to evaluate whole generation than takes it to evaluate the initial population.

[4] This might not be satisfied for very large population sizes. Later, we show that $T_{\text{eval}} = \Theta(\frac{n}{p_{\text{total}}})$, where p_{total} is the total performance of the *runners*. However, computing similarity of genotypes has quadratic asymptotic complexity i.e. $T_{\text{evol}} = \Theta(n^2)$ (for more details see [[odkaz]]). Hence, producing a new generation could be more computationally-demanding than a fitness evaluation for a large population size. However, while population size is lower than 500 genotypes (maximal population size in our experiments), the time for producing a new generation remains insignificant in comparison to the time for a fitness evaluation. Thus, it is not considered in further text.

m	the number of *runners*
n	the number of genotypes in a single population (i.e. population size)
n_i	the number of genotypes evaluated by ith *runner*, obviously $\sum_i n_i = n$
t_i	total time for sending, evaluating and receiving the statistics of evaluation of a single genotype by the ith *runner*
T^i_{eval}	the time for an evaluation of n_i genotypes by the ith *runner* (including the time for communication)

We assume the times for an evaluation of all genotypes by the ith *runner* to be equal.[5] Therefore, the evaluation time for n_i genotypes by the ith *runner* is the time for an evaluation of single genotype multiplied by the number of genotypes evaluated by the *runner*

$$T^i_{\text{eval}} \quad = \quad n_i t_i \ .$$

Total running time for a fitness evaluation of a single generation is equal to the maximum time of evaluation of a single *runner*

$$T_{\text{eval}} = \max_{i \in \{1,\ldots,m\}} T^i_{\text{eval}} \ .$$

The time for an evaluation of a single genotype can be further divided into evaluation time in simulation and communication time (i.e. sending genotype to the *runner* and receiving statistics from the *evolution application*).

$$\begin{aligned} t_i &= t^i_{\text{eval}} + t^i_{\text{comm}} \ , \\ T^i_{\text{eval}} &= n_i \cdot t_i = n_i \cdot (t^i_{\text{eval}} + t^i_{\text{comm}}). \end{aligned}$$

t^i_{eval}	the time for an evaluation of a single genotype by the ith *runner* (time for sending genotype is not included)
t^i_{comm}	total time for sending a single genotype from *evolution application* to the ith *runner* and sending an evaluation score from the ith *runner* to *evolution application*

We define performance as the average number of genotypes evaluated by a *runner* in a fixed time.[6]

$p_i = (t_i)^{-1}$	the performance of the ith *runner*
$p_{\text{total}} = \sum_i p_i$	the total performance
$\tilde{p_i} = \frac{p_i}{p_{\text{total}}}$	the proportional performance of the ith *runner* (i.e. normalized performance to the total performance)

[5] As mentioned before, when organisms become more complex during an evolution, it could take more time to evaluate them in comparison to genotypes from the initial population.

[6] If the time t_i is measured in seconds, p_i can be interpreted as an average number of genotypes evaluated per second.

Note that n_i (the number of evaluated genotypes by the ith *runner*) can be viewed as function of n (i.e. $n_i = n_i(n)$). As mentioned in the previous section, when an evaluation score is received from a *runner*, *evolution application* sends back next genotype immediately (whenever there is any genotype to be evaluated). Therefore, if the population size is increased by one genotype, only n_i with the lowest T^i_{eval} will be incremented, other n_i will remain unchanged.

Now, we show a simple example for clarification of the previous definitions. We consider the following situation. We have population of 100 genotypes and we use 80 *runners* with the same performance $c > 0$ (i.e. $n = 100, m = 80, p_i = c, i = 1, \ldots m$). Therefore, first 80 genotypes is evaluated — each *runner* evaluates one genotype. Next, the rest of 20 genotypes is evaluated by 20 *runners* (we assume that *runners* $1, \ldots, 20$ are chosen) and other 60 *runners* are waiting.

Our parameters have the following values

$$\begin{aligned} p_{\text{total}} &= mc, \\ t_i &= \frac{1}{c},\ i = 1, \ldots m, \\ \tilde{p}_i &= \frac{1}{m},\ i = 1, \ldots m, \end{aligned}$$

and

$$n_i = \begin{cases} 2 & \text{for } i = 1, \ldots 20, \\ 1 & \text{for } i = 21, \ldots m. \end{cases}$$

Therefore,

$$T_{\text{eval}} = \max_{i \in \{1,\ldots,m\}} T^i_{\text{eval}} = T^1_{\text{eval}} = n_1 t_1 = \frac{2}{c}.$$

It is apparent that the number of *runners* $m = 80$ is not appropriate for the population size of $n = 100$ genotypes. We could use only 50 *runners* to achieve the same total running time or use population size of 180 genotypes instead.

In the following text we analyze our model for various population sizes and various numbers of *runners* with the same or different performances.

5.2.2 Asymptotic analysis of distributed computing with a large population

Now, we focus on asymptotic time for an evaluation of large populations in general case, when *runners* can have different performances. Later in this chapter,[7] we use these results to determine an optimal population size for a distributed fitness evaluation.

First, we show that for a large population of size n and fixed *runners* (with fixed performances $p_i, i = 1, \ldots, m$, where in general $p_i \neq p_j$ for $i \neq j$) the proportion of evaluated genotypes by ith *runner* is asymptotically equal to the proportion of ith *runners* performance

$$\frac{n_i}{n} \sim \tilde{p}_i, \text{ for } i = 1, \ldots, m.$$

[7] See Section 5.3.3 in the following section.

Proof. Without loss of generality, assume that $\tilde{p}_i \in \mathbb{Q}$ (i.e. $\tilde{p}_i$ is rational). Thus, there is a smallest number of *runners* $n^* \in \mathbb{N}$ such that

$$\tilde{p}_i n^* \in \mathbb{N}, i = 1, \ldots, m.$$

Then we have

$$n_i^* = \tilde{p}_i n^*.$$

and

$$\frac{n_i}{n} = \tilde{p}_i \text{ if and only if } n = kn^*, k \in \mathbb{N}.$$

Note that the population size n^* can be also interpreted as the time for an evaluation of n^* genotypes, where the evaluation time for a single genotype is one time unit (i.e. $t_i = 1, i = 1, \ldots, m$).

Now, we have

$$n = k_n n^* + c_n$$

where

$$k_n \in \mathbb{N} \cup \{0\}, c_n \in \{0, \ldots, n^*\text{-}1\}$$

The number of evaluated genotypes by ith *runner* $n_i = n_i(n)$ is apparently non-decreasing function of n, thus:

$$k_n n_i^* \leq n_i \leq (k_n + 1) n_i^*$$

i.e.

$$k_n n^* \tilde{p}_i \leq n_i \leq (k_n + 1) n^* \tilde{p}_i$$

thus

$$\frac{k_n n^*}{n} \tilde{p}_i \leq \frac{n_i}{n} \leq \frac{(k_n + 1) n^*}{n} \tilde{p}_i.$$

Now, it is sufficient to show that

$$\frac{k_n n^*}{n} \to 1 \text{ and } \frac{(k_n + 1) n^*}{n} \to 1 \text{ as } n \to \infty.$$

We have

$$\lim_{n\to\infty} \frac{k_n n^*}{n} = \lim_{n\to\infty} \frac{k_n n^*}{k_n n^* + c_n} = \lim_{n\to\infty} \frac{1}{1 + \frac{c_n}{k_n n^*}} = 1$$

and

$$\lim_{n\to\infty} \frac{(k_n + 1) n^*}{n} = \lim_{n\to\infty} \left(\frac{k_n n^*}{n} + \frac{n^*}{n} \right) = \lim_{n\to\infty} \frac{k_n n^*}{n} + \lim_{n\to\infty} \frac{n^*}{n} = 1 + 0 = 1.$$

Therefore

$$\frac{n_i}{n} \to \tilde{p}_i \text{ as } n \to \infty$$

□

Consequently, we get that the time for an evaluation of n_i genotypes by ith *runner* is asymptotically equal to the number of evaluated genotypes divided by performance of the *runner*

$$T_{\text{eval}}^i = n_i \cdot t_i = \frac{n_i}{p_i} \sim \frac{n}{p_{\text{total}}}$$

and therefore the total time for an evaluation is asymptotically equal to the population size divided by the total performance of runners (i.e. for a large population size evaluation speed is approximately linear function of runners performance)

$$T_{\text{eval}} \sim T^i_{\text{eval}} \sim \frac{n}{p_{\text{total}}}.$$

5.2.3 Asymptotic analysis of distributed computing with a large population and homogeneous runners

Now, we focus on a typical case, when *runners* have equal performance (e.g. *runners* are distributed uniformly on computers of equal configuration[8]).

Under these assumptions, we have

$$p_i = p \text{ and } t_i = t, i = 1, \ldots, m.$$

Therefore, the total performance is $p_{\text{total}} = mp$ and we get, that

$$T_{\text{eval}} \sim \frac{n}{m} t$$

i.e. the total time for an evaluation of population with n genotypes is asymptotically equal to the time for an evaluation of n genotypes on a single *runner* divided by the number of the *runners*.

5.2.4 Analysis of distributed computing with fixed population size

Now, we focus on the time for an evaluation of a population with fixed size and various number of *runners*. We continue assuming homogeneous *runners* (i.e. *runners* with equal performance).

If the population size is divisible by the number of *runners* (i.e. $n = km, k \in \mathbb{N}$) each of m *runners* evaluates $k = \frac{n}{m}$ genotypes, thus

$$T_{\text{eval}} = \frac{n}{m} t.$$

Otherwise, if the population size is not divisible by the number of *runners*

$$\text{i.e. } n = km + c, k \in \mathbb{N}, 0 < c < m$$

then $m - c$ *runners* evaluate each $k = \lfloor \frac{n}{m} \rfloor$ genotypes and c *runners* evaluate each $k + 1 = \lceil \frac{n}{m} \rceil$. Therefore, generally we have

$$T_{\text{eval}} = \left\lceil \frac{n}{m} \right\rceil t.$$

[8] Note that using multiple *runners* on one computer is mostly more effective than using only one *runner* per computer. The optimal number of *runners* can vary on various hardware and can be determined experimentally.

We get that the optimal numbers of *runners* with equal performance are numbers such that the population size is divisible by this numbers of *runners*, i.e. the population size is a multiple of the number of *runners*.

Figure 5.2 shows the evaluation time T_{eval} with several fixed population sizes and various numbers of *runners*.

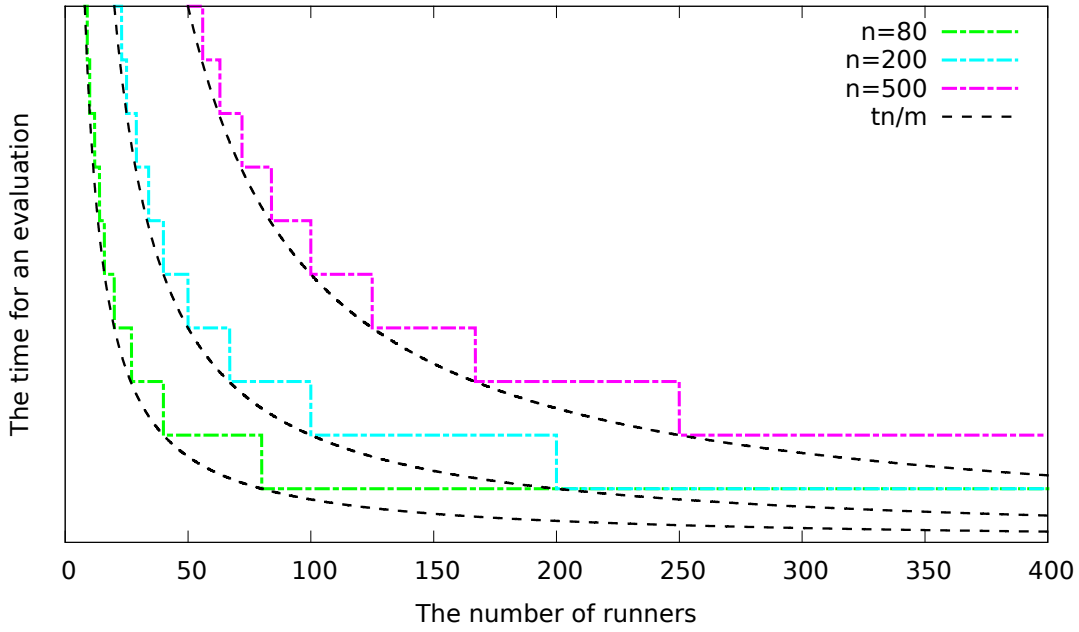


Figure 5.2: The total time for an evaluation of populations with typical sizes: 80, 200 and 500 genotypes and varying numbers of *runners*. The black dashed hyperbolic curve represents a lower bound for a time, which is achieved if the population size is divisible by the number of *runners*.

Note that points on the graph in Figure 5.2, where function decreases, forms a Pareto frontier[9] with points of two types. First, there are points which represent the fitness evaluation time for the number of *runners* such that population size is divisible by the number of *runners* (i.e. $m \mid n$). And second, there are points such that population size is not divisible by the number of *runners* (i.e. $m \nmid n$).[10]

In the following text, we compare the above theoretical results with data from real experiments.

5.3 Experiments

In this section we measure the time for a fitness evaluation in real experiments. Measured data are then compared with the theoretical results from the previous

[9]The Pareto frontier are points (i.e. pairs (T_{eval}, m)) such that there is no other point with $(T'_{\text{eval}}, m') < (T_{\text{eval}}, m)$ (i.e. $T'_{\text{eval}} \leq T_{\text{eval}} \wedge m' \leq m \wedge (T'_{\text{eval}} < T_{\text{eval}} \vee m' < m)$) for a given population size.

[10]Examples of these points could be the following. Let us have a population of $n = 80$ genotypes. Then, point of the first type is $(T_{\text{eval}} = 20t, m = 4)$, point of the second type is $(T_{\text{eval}} = 27t, m = 3)$, where t is the time for an evaluation of a single genotype. Both these points lie on the Pareto frontier, but for example a point $(T_{\text{eval}} = 2t, m = 41)$ is not in the Pareto frontier, it is dominated by $(T_{\text{eval}} = 2t, m = 40)$.

section. Experiments show that the theoretical model is suitable for the evaluation time with a typical population size and number of *runners*.

First, Section 5.3.1 describes the method of performing experiment, used setting and properties of used machines. Next, Section 5.3.2 analyzes measured data — an evaluation time and performance. Finally, Section 5.3.3 summarizes results from this chapter and applies them to determine appropriate population size and *runners*.

5.3.1 Method used in performance test

In our experiment we measure the time for an evaluation of a single generation of 300 random genotypes. Each genotype is simulated in a virtual environment for 90 second of simulated time. Table 5.1 shows mentioned properties used in the experiments.

Measuring the time for a fitness evaluation starts after creation of initial random population and terminates when an evaluation score from the last genotype is received. Measured time includes:

- the time for creating connection between the *evolution application* and *runners*,
- sending genotypes from *evolution application* to *runners*,
- simulation in a virtual environment, and
- sending statistics from *runners* to the *evolution application*.

Initial creation of random population and processing evaluated population is not included in the measured time.

Multiple experiments with various numbers of *runners* were performed to estimate evaluation time for a given number of *runners*. For each *runner*, a single remote computer was used. Therefore, the performance of a *runner* is not affected by another *runner* running on the same computer.

Used computers are divided into groups with an identical hardware configuration. Totally 8 groups of equal configured computers were used. We used incrementally from 1 to 4 computers from each group (i.e. from 8 to 32 *runners*). Each experiment for a given number of *runners* was repeated five times to minimize random factors affecting the measured time. Hardware configurations of the used computers are described in Table 5.2.

5.3.2 Experimental results

The Figure 5.3 shows both the measured times for an evaluation and the corresponding performance for different numbers of *runners*. The evaluation times are fitted with a hyperbolic curve, performances (i.e. average numbers of genotypes

Table 5.1: Setting used in the experiment.

Parameter	Value	Description
Population size	300	the number of evaluated genotypes
Simulation time	$30s$	the simulation time for a single evaluation
Number of evaluations	3	the number of iterations of an evaluation for a given genotype (i.e. total simulation time for a genotype is $90s$)

Table 5.2: Hardware configuration of used computers in the experiment.

Type	PC (x86-64)	PC (x86-64)	PC (x86-64)
Processor	Intel Core i7 920 (4x 2.66 GHz)	Intel Core2 Quad Q9550 (4x 2.83 GHz)	AMD Athlon II X4 640 (4x 3 GHz)
Memory	6 GB RAM	4 GB RAM	4 GB RAM
Graphics	Nvidia GeForce 210	Nvidia GeForce 9400GT	AMD Radeon HD 4250
OS	Gentoo Linux	Gentoo Linux	Gentoo Linux
Groups	1-4	5-7	8
Total Count	16 (4x4)	12 (3x4)	4 (1x4)

evaluated in a constant time) are fitted with a line.

We focus primary on evaluation times. As we can see from the graph, differences between measured times within a group of experiments with the same number of *runners* are very small (i.e. variance is very low).

The measured times are fitted[11] with a following hyperbolic function

$$y = \frac{a}{x - b} + c.$$

Table 5.3 shows estimated parameters. According to the model from the previous section, the time for an evaluation of a single generation should decrease with the number of *runners* according to the following formula

$$T_{\text{eval}} = \frac{\alpha}{m},$$

where $\alpha = nt$. From Table 5.3 we can see that estimated values b and c are close to zero (the difference is not statistically significant). Therefore, the experimental data are not in a conflict with the model.

Now we can estimate value α from experimental data. We have

$$\hat{\alpha} = a = 4681.18,$$

i.e. estimation for the time for an evaluation of a generation with a single *runner*.

[11] Method of least squares is used to estimate parameters.

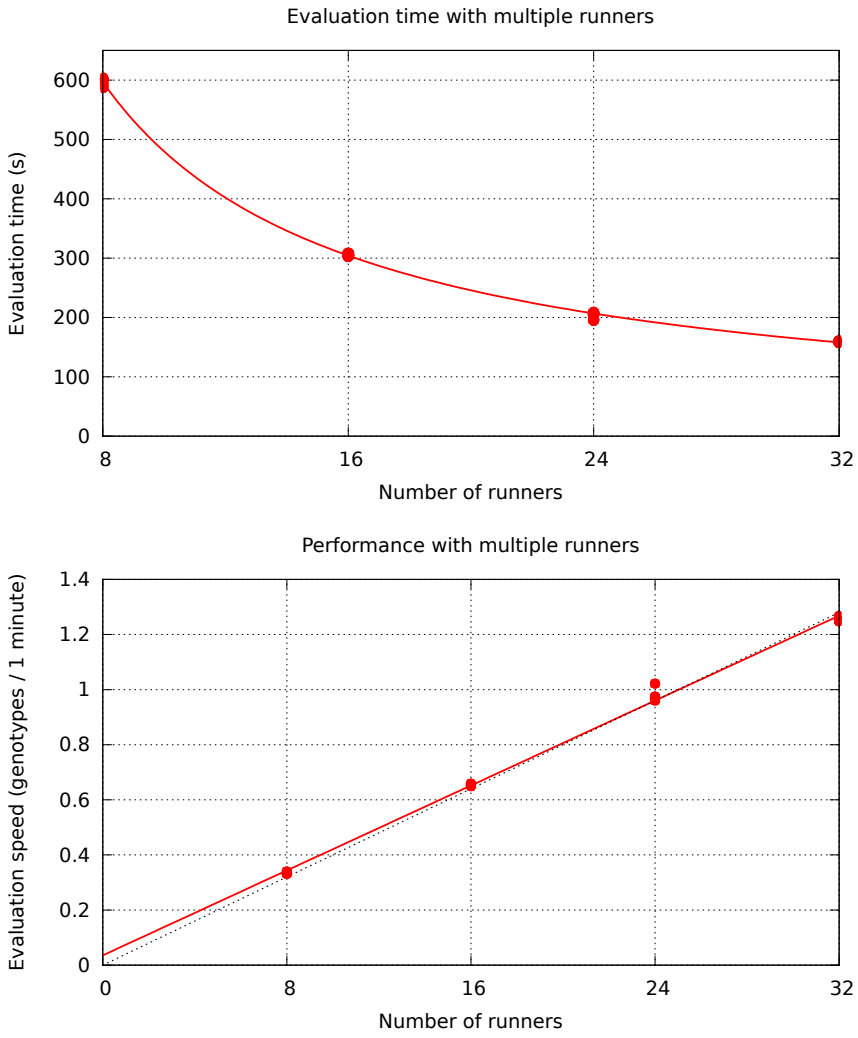


Figure 5.3: Evaluation time and performance with multiple runners.

We can also estimate the time for an evaluation of a single genotype:

$$\hat{t} = \frac{\hat{\alpha}}{n} = \frac{4681.18}{300} \doteq 15.6s,$$

i.e. in this experiment 90 seconds of simulated time corresponds to approximately 15.6 second of real time.

The following section summarizes results from this chapter and determines appropriate number of *runners* and the population size.

5.3.3 Conclusions

In this section we first summarize both theoretical and experimental results on distributed computing and we use the results to determine the optimal number of *runners* and the population size for distributed fitness evaluation.

We have shown that for a typical population size and computers with the same performance, evaluation of genotypes is optimally distributed if the population

Table 5.3: Estimated values in experimental data.

Param.	Estimated value	Standard Error	Value according to the model	Description
a	4681.18	+-321.9 (6.875%)	$t > 0$	the time for an evaluation with a single runner
b	11.67	+-10.26 (87.88%)	0	
c	-0.0015	+-0.42 (28670%)	0	

size is a multiple of the number of *runners*. In this case, the performance increases linearly with the number of *runners*.

Generally, when computers with different performances are used, it can be more difficult to determine the optimal numbers of *runners* and population size. However, we can use similar approach as in the proof of linearity of asymptotic evaluation time in the Section 5.2. Assuming we have multiple computers with varying performances and each computer is running one or multiple *runners*, we can use the following method to determine the optimal population size:

- We estimate the proportional performance of *runners* on each computer $\tilde{p}_i$.
- We choose population size as n^* satisfying $\tilde{p}_i \cdot n^* \in \mathbb{N}$, for all $i = 1, \ldots, m$.[12]
- If the population size n^* would be too large, we can round normalized performances and repeat the previous step to get lesser population size.

Note that if n^* is the optimal population size, then each multiple $kn^*, k \in \mathbb{N}$ is also optimal population size for a distributed computing.

Now we can determine the optimal population size for given *runners* with some particular performances or vice versa. However, the assumptions we have made in the previous section might not be always satisfied.

We can assume connection speed to be fast enough most of the time in a typical experiment since genotypes are relatively small. Another assumption that evaluation of a single genotype requires a fixed time might not be satisfied in some experiments (e.g. evaluation of more complex genotypes could require more time than evaluation of initial genotypes).

[12] We can assume that $\tilde{p}_i$ is a positive rational number (we can measure performances only with finite precision). Thus, we have $\tilde{p}_i = \frac{a_i}{b_i}$ for some $a_i, b_i \in \mathbb{N}$ and the least common multiple of $b_i, i = 1, \ldots, m$ is the least number n^* satisfying the condition.

6. Experiments

This chapter compares two proposed methods on a sample task, where organisms are evolved for locomotion in a virtual environment. The methods are compared based on (1) fitness of the best evolved genotypes and (2) strategy and appearance of evolved organisms. The second proposed method, *HyperAO* is also compared with original artificial ontogeny proposed by Bongard and Pfeifer [5]. To reduce duration of performed experiments, the distributed fitness evaluation is used that is described the previous chapter.

The chapter consists of four sections. Section 6.1 describes limitations of a physical simulation and physical validations for evolved organisms. Section 6.2 provides a description of the locomotion task including measuring quality of organisms and experimental results. Section 6.3 describes advanced tasks designed to evolve organisms for light following and block pushing. Section 6.4 summarizes results from this chapter and provides conclusions for the stated hypotheses.

6.1 Physical Validation

As described in the previous text, virtual organisms are evolved in a physical virtual 3D-world. The physical simulation is realized using a game engine — jMonkeyEngine [?] that includes JBullet [?], a java port of open source Bullet Physics Library [?]. Simulation of virtual organisms includes several random factors that effect a measured quality of the genotypes (i.e. fintess). Thus, estimating quality of organisms requires more that one repeated evaluation in the simulation.

In addition to random factors, the physical simulation includes other limitations. The limited accuracy of the simulation often causes an evolution to discover a class of organisms that exploit a physically-unrealistic or unwanted behavior such as resonating modules caused by a harmonic oscillation. This behavior often leads to a high fitness, thus, it dominates the population in a few generations.

To prevent evolution from evolving such organisms, a physical validations are used that penalize physically-unrealistic organisms in fitness. The physically-unrealistic behavior is detected based on velocity of the modules. When a maximum measured velocity of some module exceeds a predefined threshold, an organisms is penalized in the fitness accordingly. Velocities close to the threshold are only slightly penalized, on the opposite, organisms with extremely high velocities of their modules are given a minimum possible fitness.

6.2 Evolving Locomotion

To compare proposed methods a sample task is used, where organisms are evolved for locomotion in a virtual environment. The virtual environment consists of a flat surface and a virtual organism, no other objects are present in the environment. The following text provides a description of the performed experiments.

6.2.1 Measuring Quality of Organisms

Evolved organisms in preliminary experiments often exhibit (1) a large number of intersecting modules and (2) low center of mass that result in poor locomotion. To increase locomotion capabilities of organisms, additional two factors are incorporated into the fitness — variability of module positions during the simulation and average height of the organism‘s center of mass. The quality of organisms is computed as:

$$\text{fitness} = \left(\alpha \cdot \text{distance}^2 + \beta \cdot \text{path} + \gamma\right)(\text{height} + \text{variability} + \delta),$$

where

distance	indicates maximum distance from an initial position during the simulation,
path	indicates length of the walked path during the simulation (note that path $\geq$ distance),
height	indicates average height of the organism‘s center of mass during the simulation,
variability	indicates variability of modules positions during the simulation computed as: $\sqrt{\widehat{\sigma_x^2} + \widehat{\sigma_y^2} + \widehat{\sigma_z^2}},$ where $\widehat{\sigma_x^2}, \widehat{\sigma_y^2}$ and $\widehat{\sigma_z^2}$ indicate average variance of module positions in coordinates x, y, z during the simulation,
$\alpha, \beta, \gamma, \delta$	indicate constants values ($\alpha, \beta = 0.5, \gamma, \delta = 0.1$).

Note that the position of the organism indicates its center of the mass.

In the initial generations, including path into fitness helps evolving moving organisms. However, evolved organisms may exhibit walking around the initial position. In later generations, second power of distance exceeds length of the walked path. As a result organisms walking in the direction from the initial position are preferred.

6.2.2 Experimental Setting

The following text describes parameters of the performed experiments. For each proposed method a single series of experiments is performed to evolve organisms locomotion in the virtual environment. Each series consists of 6 evolution runs that last for 100 generations and then are terminated. While genotypes include a relatively large number of parameter, the population size is set to 500 genotypes that provides a high probability of evolving successful organisms. To reduce random factors in fitness each evaluation is repeated.

A simulation of organisms in a virtual environment have duration of 30 second for each evaluation (this does not include growth in *HyperAO* method), when the walked distance and other factors that contribute to the final fitness are measured. These factors are then averaged from all evaluations and the final fitness

Table 6.1: Parameters of performed experiments

Parameter	**Value**
Number of generations in a single evolution	100
Genotypes in a single generations	500
Number of repeated evaluations	2
Duration of a single simulation	30 seconds
Evolution runs for each series of experiments	6

is computed. Described parameters are listed in Table 6.1.

To reduce time duration of the performed experiments, distributed fitness evaluation described in the previous chapter is used. The distributed evaluation allows performing each series of 6 evolution runs in approximately 10 hours using completely 22 computers. Thus, each evolution run takes less than 2 hours.[1] Computers used for the distributed fitness evaluation are listed in Table 6.2.

Table 6.2: Hardware configuration of used computers in the experiments.

Type	PC (x86-64)	PC (x86-64)	PC (x86-64)
Processor	Intel Core i7 920 (4x 2.66 GHz)	Intel Core2 Quad Q9550 (4x 2.83 GHz)	AMD Athlon II X4 640 (4x 3 GHz)
Memory	6 GB RAM	4 GB RAM	4 GB RAM
Graphics	Nvidia GeForce 210	Nvidia GeForce 9400GT	AMD Radeon HD 4250
OS	Gentoo Linux	Gentoo Linux	Gentoo Linux
Total Count	10	7	5

6.2.3 Results

The results of performed experiments are shown on Figure ?? and Figure ??, where the first figure shows fitness of best genotypes in generations for each evolution run and the second figure shows average fitness in generations for each evolution run.

Based on experimental results, *HyperAO* performs slightly better on the locomotion task in the initial generations. However, in later generations *HyperAO* exhibits stagnation, while *Modular representation* produces increasingly better organisms. As a result, best genotypes produced by *Modular encoding* significantly outperform best genotypes produced by *HyperAO*.

The second figure indicates that *HyperAO* produces higher average finesses. This is most likely caused by a single evolution run with extremely high average finesses. This might of losing variability and early convergence of the population

[1] Note that performing both series of experiments without distributed fitness evaluation would require approximately 20 days of computing.

Figure 6.1: **Fitness of best genotypes in generation.** The figure shows fitness of best genotypes in generation for each evolution run (red and blue points), where the color corresponds to the used method — red color indicates *Hypercube-based artificial ontogeny* and blue color indicates *Modular representation*. Two fitted curves estimate average fitnesses of best genotypes, where the averages are estimated using local polynomial regression (loess). Dark gray areas indicate confidence intervals for both methods. We can see that *Modular representation* performs significantly better in later generations.

in later generations. The following text focuses on locomotion strategy produced by these two methods.

The evolved organisms in the experiments differ in several aspects. *Modular representation* produces organisms with a higher speed of locomotion, thus they achieve higher fitness compared to the *HyperAO*. Organisms produced by *HyperAO* usually have larger number of modules and they exhibit more continuous movement that in some cases resemble tentacles of jellyfish. Other produced organisms are relatively similar for both method. Some organism evolved using *HyperAO* seem to be similar to the organisms evolved by original artificial ontogeny proposed by Bongard and Pfeifer [5], however, they usually exhibit more symmetries.

6.3 Evolving Light Following and Block Pushing

In addition to the locomotion task, two other tasks — light following and block pushing task are implemented to test the proposed control system that is com-

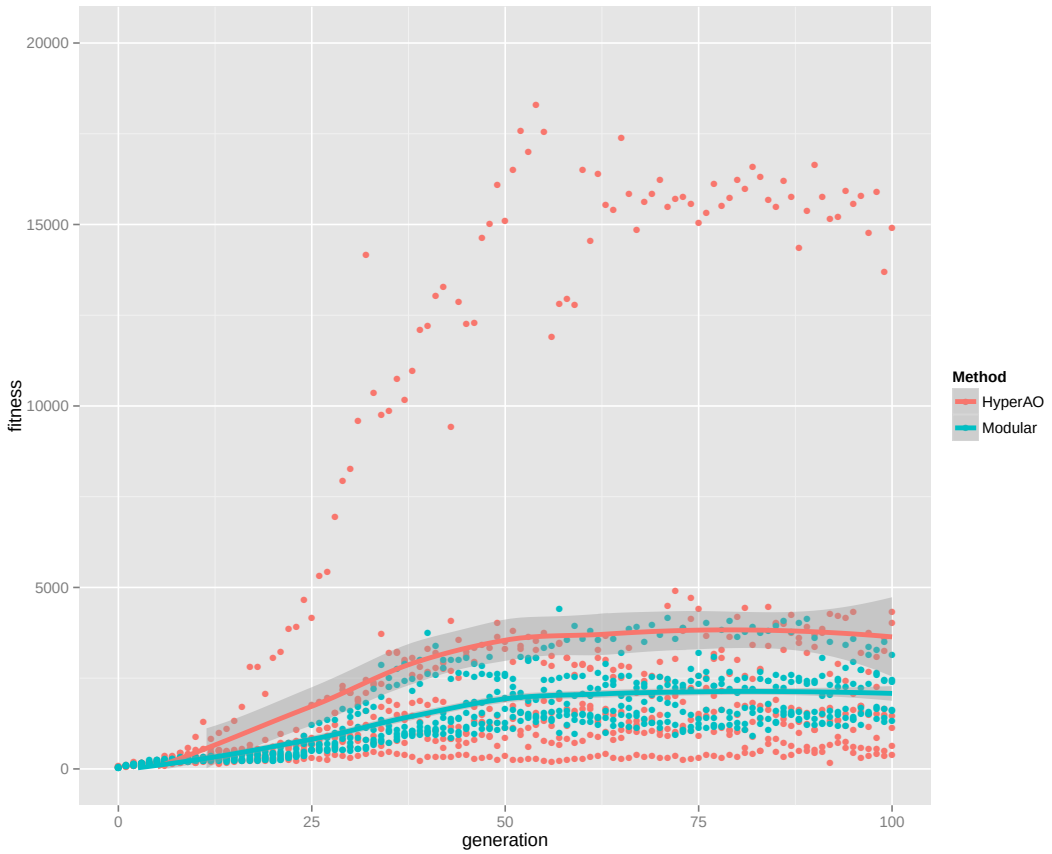


Figure 6.2: **Average fitness in generation.** The figure shows average fitness in generation for each evolution run. The same method as in the previous figure is used to estimate averages from performed evolution runs. Here, in contrast to the previous figure, average fitness of *HyperAO* is significantly higher compared to *Modular representation*. This is most likely caused by an evolution run with high average fitnesses that might indicate low variability in the population.

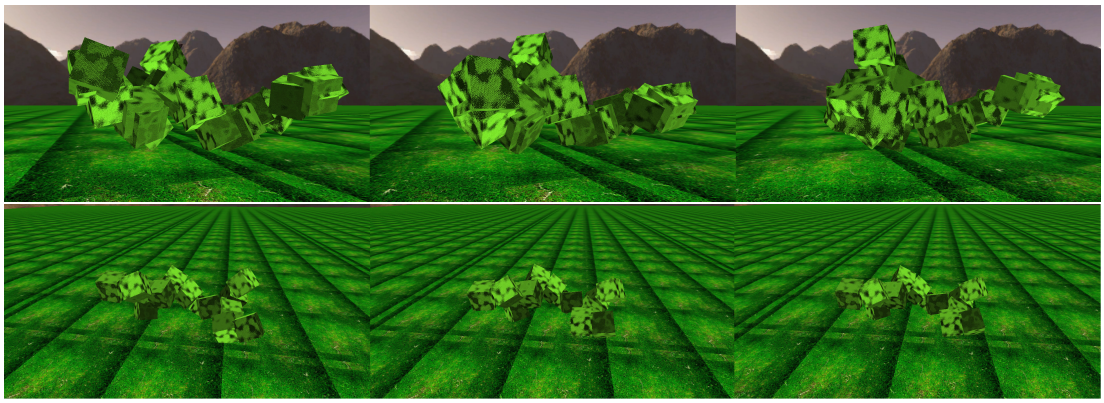


Figure 6.3: **Sample evolved organisms.** The figure shows two organisms evolved for locomotion in the final generation. Top line shows an organism evolved using *HyperAO* and bottom line shows an organism evolved using *Modular representation*.

mon for both methods.

The light following task is designed to evolve organisms that follow an object in the environment (i.e. source of light), where light sensor provides an information about direction to the object position. The fitness is here a weighted sum of the following factors: (1) walked distance from the initial position, (2) walked path as in the locomotion tasks and (3) walked distance to the average distance from the followed object.

The block pushing task is designed to evolve organisms that move a large cube (i.e. block) from its initial position to any direction. The fitness is maximum distance of the block from its initial position.

Preliminary experiments indicate that the proposed control system, which is common for both proposed method is not capable of solving these two tasks. Evolved organisms do not exhibit a locomotion to the followed object (light source or block) in any of preliminary experiment. This may be caused by (1) insufficient influence of light sensors to the organisms locomotion and (2) difficulty of the problem. However, detailed analysis would require further investigation.

6.4 Summary

Based on produced organisms from the locomotion task and compared with organisms evolved for locomotion by Bongard and Pfeifer [5] it seems that evolved organisms using *HyperAO* might performs better on this task compared to the original artificial ontogeny proposed by Bongard and Pfeifer. This could be result of using additional symmetries in the phenotype. On the other hand, unlike artificial ontogeny, *HyperAO* is not capable of solving block pushing task that requires a light sensor to adjust direction of movement to the position of the followed object. This may be caused by a poor influence of the light sensors to the organism‘s movement.

This indicates that based on evolved organisms the following hypotheses might apply to the locomotion in the simulated environment:

Hypothesis 3. *Hyper-cube based artificial ontogeny produces significantly better organisms than the original artificial ontogeny.*

However, more accurate comparison would require further investigation.

Experiments also show that *HyperAO* is outperformed on locomotion task by the first proposed method, *Modular representation*. Based on the experimental data this indicates that the following stated hypothesis does not apply to the locomotion task:

Hypothesis 4. *Hyper-cube based artificial ontogeny is capable of producing comparable or better organisms compared to the first proposed method that evolves morphological structure using directed graphs.*

Note that despite better results produced by the first method both methods are capable of producing organisms capable of locomotion in the simulated environment.

7. Conclusions and Future Work

7.1 Conclusions

This work proposes two methods compared in series of experiments. The experimental results show that the first proposed method, *Modular representation* that is based on bachelor's thesis of the author significantly outperforms the second proposed method, *Hypercube-base artificial ontogeny* (*HyperAO*) on sample tasks, where organisms are evolved for locomotion to the maximum distance from the initial position.

HyperAO represents a proof of concept of a novel approach that combines artificial ontogeny and HyperNEAT generative encoding. It is based on a theoretical comparison of developmental methods with other methods evolving virtual organisms provided in this work. The main innovation to the original artificial ontogeny is realization of genetic regulatory network using a system of artificial neural networks represented using HyperNEAT generative encoding that allows differentiation based on the position in the organism.

HyperAO seems to perform better in evolving organisms for locomotion task compared to original artificial ontogeny proposed by Bongard and Pfeifer [5]. While *Hypercube-base artificial ontogeny* presents concept of a new approach rather that a complete method (i.e. it is a proof of concept), it requires many parameters to be optimized. Thus, even performance on the locomotion task that is on some evolution runs comparable with the first method can be considered as a success. As a result, goal of this thesis have been successfully fulfilled.

7.2 Future Works

The following text provides suggestions for the future extensions of *Hypercube-base artificial ontogeny*. Unlike original artificial ontogeny, where a physical organisms grow during the ontogenetic process, *HyperAO* constructs a physical phenotype after the final grow iteration, when an organism is considered to be adult. Extension of *HyperAO* to construct physical phenotype during the ontogenetic process could allow adaptation to the virtual environment, when the growing phenotype can discover physical constrains or can be adapted to various physical conditions such as growing in water of a lower gravity.

Next suggested extensions are related to HyperNEAT generative encoding. Instead of evolving positions of neurons in GRN substrate, extension ES-HyperNEAT of HyperNEAT could be used. Evolvable-substrate HyperNEAT (ES-HyperNEAT) is a method proposed by Risi, et al. [18] that in addition to HyperNEAT automatically determines positions of hidden neurons based on information stored in evolved CPPN. This could by beneficial to *HyperAO* that uses CPPNs to represent both morphology and control system. Note that modification of ES-HyperNEAT that also evolves position of neurons corresponding to gene products could be even more beneficial.

This leads to further extension of ES-HyperNEAT referred as Adaptive ES-HyperNEAT proposed by Risi, et al. [19] that enhanced ES-HyperNEAT with

ability to learn. ES-HyperNEAT might be used to realize a morphological adaptation during the lifetime of the organism. As a result, further adaptation might increase robustness of evolved organisms as argued by Bongard [14]. Thus, we can conclude that the most significant advantage of *HyperAO* are its possible extensions.

Bibliography

[1] STANLEY, Kenneth O.; MIIKKULAINEN, Risto. *Evolving neural networks through augmenting topologies.* Evolutionary computation, 2002, 10.2: 99–127.

[2] KRČAH Peter. *Towards efficient evolutionary design of autonomous robots.* In: Evolvable Systems: From Biology to Hardware. Springer Berlin Heidelberg, 2008. p. 153–164.

[3] BRAUN, Heinrich; WEISBROD, Joachim. *Evolving neural feedforward networks.* In: Artificial Neural Nets and Genetic Algorithms. Springer Vienna, 1993. p. 25–32.

[4] HyperNEAT for locomotion control in modular robots [[TODO upravit podle normy ISO 690]]

[5] BONGARD, Josh C.; PFEIFER, Rolf. *Repeated structure and dissociation of genotypic and phenotypic complexity in artificial ontogeny.* In: Proceedings of the Genetic and Evolutionary Computation Conference. 2001. p. 829–836.

[6] INDEN, Benjamin. *Stepwise transition from direct encoding to artificial ontogeny in neuroevolution.* In: Advances in Artificial Life. Springer Berlin Heidelberg, 2007. p. 1182–1191.

[7] SIMS, Karl. *Evolving virtual creatures.* In: Proceedings of the 21st annual conference on Computer graphics and interactive techniques. ACM, 1994. p. 15–22.

[8] GAUCI, Jason; STANLEY, Kenneth. *Generating large-scale neural networks through discovering geometric regularities.* In: Proceedings of the 9th annual conference on Genetic and evolutionary computation. ACM, 2007. p. 997–1004.

[9] STANLEY, Kenneth O. *Exploiting regularity without development.* In: Proceedings of the AAAI Fall Symposium on Developmental Systems. Menlo Park, CA: AAAI Press, 2006. p. 37.

[10] ROGGEN, Daniel; FEDERICI, Diego. *Multi-cellular development: is there scalability and robustness to gain?* In: Parallel Problem Solving from Nature-PPSN VIII. Springer Berlin Heidelberg, 2004. p. 391–400.

[11] HARDING, Simon; MILLER, Julian F. *A comparison between developmental and direct encodings.*

[12] LEIBL, Marek. *Evolutionary development of robotic organisms.* Bachelor's thesis, Charles University in Prague, 2012.

[13] GRUAU, Frederic; WHITLEY, Darrell; PYEATT, Larry. *A comparison between cellular encoding and direct encoding for genetic neural networks.* In: Proceedings of the First Annual Conference on Genetic Programming. MIT Press, 1996. p. 81–89.

[14] BONGARD, Josh. *Morphological change in machines accelerates the evolution of robust behavior.* Proceedings of the National Academy of Sciences, 2011, 108.4: 1234–1239.

[15] BONGARD, Josh C.; PFEIFER, Rolf. *Evolving complete agents using artificial ontogeny.* In: Morpho-functional Machines: The New Species. Springer Japan, 2003. p. 237–258.

[16] CHENEY, Nick, et al. *Unshackling evolution: evolving soft robots with multiple materials and a powerful generative encoding.* In: Proceeding of the fifteenth annual conference on Genetic and evolutionary computation conference. ACM, 2013. p. 167–174.

[17] HILLER, Jonathan D.; LIPSON, Hod. *Evolving Amorphous Robots.* In: ALIFE. 2010. p. 717–724.

[18] Risi, Sebastian, Joel Lehman, and Kenneth O. Stanley. *Evolving the placement and density of neurons in the hyperneat substrate.* Proceedings of the 12th annual conference on Genetic and evolutionary computation. ACM, 2010.

[19] Risi, Sebastian, and Kenneth O. Stanley. *A unified approach to evolving plasticity and neural geometry.* Neural Networks (IJCNN), The 2012 International Joint Conference on. IEEE, 2012.