Poznan University of Technology
Faculty of Computing
Institute of Computing Science

Master's thesis

DEVELOPMENT AND COMPARISON OF GENETIC REPRESENTATIONS FOR EVOLVING 3D STRUCTURES

Grzegorz Latosiński

Supervisor
dr hab. inż. Maciej Komosiński

Poznań, 2018

I would like to thank my supervisor, **dr hab. inż. Maciej Komosiński**, for the guidance, patience, and all the help provided during the work on this Master's Thesis.

Contents

Chapter 1

Introduction

Evolutionary design is a field of study that focuses on using evolutionary algorithms in the process of creating, developing and optimizing various kinds of designs, like structures, control systems, artificial life agents or robots – by simulating them in a virtual environment and modifying them with the use of genetic operators, like mutation and crossover.

One of the crucial aspects of developing the evolutionary algorithm is to define a *genetic representation* that will describe the way of mapping *genotypes* to *solutions*, and introduce genetic *operators* that will alter those genotypes during the evolution. The *phenotype space* of the evolutionary design problems is usually infinite, very complex, and it lacks the straightforward definition of a neighbourhood [KRV01]. In most cases, the genetic encoding maps all possible genotypes for the problem to some subset of solutions and defines the neighbourhood between elements of the genotype. Then, genetic operators are exploring the search space by exploiting this defined topology to find the best solution in the available subset.

There can be many genetic encodings that differ in complexity, in the subset of mappable solutions and methods of exploring the fitness landscape. The goal is to design and use such a representation that will lead to the best achievable solution for a given problem.

The aim of this master thesis is to *develop* selected genetic encodings for evolutionary design problems, and *compare* the ability of those encodings to cope with given problems in order to identify and list their advantages and disadvantages.

1.1 Evolutionary algorithms

Evolution is one of the most powerful tools created for the optimization problems. This statement comes from the observation of nature – there is a wide variety of animals and plants that are specialized in living in the surrounding environment. Starting just from the simplest unicellular forms, during the process of modifying genotypes of the individuals and selecting the most adapted organisms the evolution created highly complex species that can be seen today. The concept of evolution may also model the way of people thinking and analysis – the initial idea of single person is modified, compared and possibly mixed with other ideas. Then, the inventions that suit best the given task are selected for further analysis. In this case modifications are the equivalent of *mutations*, and the mix of two ideas may be considered a *crossover*.

Evolutionary algorithms are a subset of metaheuristic optimization algorithms that are inspired by the process of evolution. In this case, the *phenotype* space is the set of possible solutions. They are coded into the *genotypes* that follow defined genetic representation. The *mutation* and the *crossover* operators are designed to modify the genotypes to form new solutions from the previous

ones. During the process of selection, the best solutions have the highest probability of "surviving" and participating in further evolution. Evolutionary algorithms were successfully applied to various optimization problems, like knapsack problem, travelling salesman problem [GGRG85], evolution of classification systems and more [Ron97].

1.2 Evolutionary design

Thanks to the increasing computational power, it is now possible to perform sophisticated and complex computation, especially simulations that are plausible from the physical, chemical or biological point of view. This enables researchers to analyse processes or objects in the virtual environment before implementing them in the reality. With evolutionary algorithms it is possible to *automate* the process of designing entities for a given task. Such use of the evolutionary algorithms that mimic the manual human development of solutions for a problem is called *evolutionary design*.

Evolutionary design tries to solve the problem of developing a structure that maximizes the given *fitness function*. The example applications of the evolutionary design are [BW97]:

- Circuits optimization,
- Evolution of controllers for robots,
- Evolution of neural networks,
- Optimization of mechanical parts, like turbine blades,
- Evolution of art – music and pictures,
- Evolution of models for computer graphics, like trees or creatures.

Evolutionary design is known to provide useful solutions for real-life problems. The process of introducing new and improved solutions by evolutionary design algorithms is faster than by manual optimization, for example the design of a jet engine turbine was improved in only two days by the evolutionary algorithm, while it took about eight weeks for an engineer to introduce a better solution than the original turbine. What is more, the improvement presented by the evolutionary algorithm was three times better than the manual solution [BW97].

Evolutionary algorithms can explore the phenotype space in such a way that the designer would never perform during manual development, because of the slow exploration of the space and the biased way of thinking. Evolutionary design in a short amount of time processes large numbers of possible solutions and usually explores many regions of phenotype space in parallel, depending on the population size. There are various examples of applied solutions that outperformed manually created ones, like evolved antennas [GHLL06], previously mentioned engine turbines or DC motors [BW97].

Another field of study that combines simulations with evolutionary algorithms is *Artificial Life*. It studies fundamental aspects of evolutionary processes, where creatures are represented as agents in the virtual environment defined by the researcher. The environment and models in artificial life are usually less complex than real life. The most popular experiments involve *guided evolution*, where every creature is evaluated, and the probability of creature's surviving in the population depends on its fitness value. Another approach is to mimic *unguided evolution* – in this case, creatures are not evaluated by an externally provided fitness function, and what is studied is usually the behaviour of creatures and the "direction" of evolution. Research in artificial life, when it involves 2D or 3D physical agents, is similar to evolutionary design – virtual agents usually

have bodies and control systems, and they are simulated in some virtual environment. The main difference in both domains of research lays in the aim of the research.

1.3 Genetic representation

As it was mentioned in Section 1.1, an evolutionary algorithm requires the mapping function from each genotype to the corresponding phenotype. The genotype should always create the same phenotype if the mapping is deterministic, but the same phenotype can be sometimes defined by multiple genotypes.

Phenotype spaces for the problems of the evolutionary design are usually very complex – they are infinite, and it is hard to determine the neighbourhood relation between possible solutions. This happens because the evolving structures can be of a variable size and complexity, they can be formed from a different number of used elements and properties [KRV01]. The genetic encoding usually generates an infinite set of genotypes that can be mapped to some subspace of the phenotype space. There can be various types of genetic encodings – they may differ in syntax, available elements describing the final phenotype, the process of creating a solution from the genotype, and more. What is important, the evolutionary representation creates some *topology* on the space of solutions. It describes the set of neighbouring solutions for each solution in the available subset of the phenotype space. This topology [KRV01] is used by mutation and crossover operators to move in some organised manner in the fitness landscape in order to find the best solution for a given problem.

An inadequate representation of phenotypes may lead to poor performance of evolutionary algorithms, but a genetic representation that acts poorly in one optimization problem may give good results for the other problem. This is because fitness landscapes for various problems often differ, and a topology used for one fitness landscape may lead to the very good solutions, while in other fitness landscape the same topology may not provide useful information to the optimization process.

1.4 Scope of the thesis

The goal of this master thesis is the development and comparison of genetic representations for evolving 3D structures along with their neural controlling systems. The development involves three types of genetic encodings – a *developmental encoding*, a *similarity encoding* and a *biological encoding*. The development also covers the implementation of a new *generative encoding* inspired by *Lindenmayer Systems* for creating fractals and virtual plants [PL96]. The second phase of this master thesis covers the empirical analysis of implemented and improved encodings with other encodings currently available in the Framsticks platform [KU18] in order to point out the advantages and disadvantages of each encoding according to a given problem, similarly to previous analyses of this kind [Kom12, KRV01].

1.5 Structure of this thesis

Chapter 2 provides some theoretical background on genetic encodings and describes popular approaches to evolving structures and control systems based on the current literature. The chapter includes a short introduction to Lindenmayer Systems that will be useful later during the description of a new encoding. In Chapter 3, the reader will be provided with a brief description of the encodings that were available in Framsticks before additions described in this thesis. Chapter 4

presents changes that were made during improvements of developmental, biological and similarity encodings. The chapter will also provide the description of the new generative encoding. In Chapter 5, some empirical tests of available and developed encodings will be provided along with visualizations and summaries of identified advantages and disadvantages. Chapter 6 summarizes this thesis.

Chapter 2

Genetic representations for evolving 3D structures

This chapter will focus on basic aspects of the evolutionary design, especially in the field of genetic representations, and on an overview of articles about genetic representations applied for various problems. This chapter will also provide brief overview of the L-Systems.

2.1 Complexity of the evolutionary design

As it was previously mentioned and initially presented in Section 1.3, the problem of the evolutionary design is very complex. First of all, it has an *infinite space of possible solutions* that has *both discrete and continuous character*. The discrete character of the phenotype space comes usually from the use of some predefined elements that form the final phenotype. Continuity may appear if those elements or other aspects of the genotype are parametrized. One of the outcomes of such space is that it is *hard to determine the neighbourhood* [KRV01], and it is hard to define *similarity measures for solutions* [Kom17].

There are no obvious representations for the evolutionary design problems [KRV01]. In opposite to genotypes for problems like *TSP*, *QAP* or *job shop scheduling*, the genotypes for evolutionary design do not usually have constant length, because of the variable complexity of the phenotypes [Kom17]. What is more, the particular elements of the genotypes are usually in *strong relationship with the other elements*, so the small change in the genotype may result in big changes in the phenotype.

The next aspect that heavily increases the complexity of the evolutionary design problems is a *fitness function*. First of all, the fitness landscape usually has *many local optima* that can cause major problem for the evolutionary algorithm. The fitness function may also depend on many factors – they can be aggregated to a single value, or they can together form a multicriteria optimization problem, which will also increase the complexity. The fitness landscape is transformed with the use of genetic encodings, and explored by their operators – it can make the fitness landscape easier (or harder) to explore [KRV01].

Other issues of the evolutionary design problems are the *time-consuming and non-deterministic evaluation* of the solutions [KRV01], and the possible presence of *many constraints*.

As it can be seen from the above analysis, the evolutionary design problems are usually hard, and the selection of a good genetic representation may be of particular importance to the task of evolving valuable solutions [KRV01].

2.2 Exploration and exploitation in evolutionary algorithms

In every optimization algorithms the solutions for a given problem can be obtained with one strategy or mix of two strategies – an *exploration* and an *exploitation* [ES98]. The first one moves dynamically across the fitness landscape and finds the most promising regions for the optimization. The second one moves in some small subset of solutions in order to find the best one within this subset. It is more precise due to small movements in the fitness landscape.

During the optimization an algorithm should use both strategies – the exploration will find the region with the most promising values of the fitness function, while the exploitation will try to find the best solution within this region.

As it is stated in [ES98], it is common to interpret a *selection* as the exploitation factor of the evolutionary algorithms, while *genetic operators* are responsible for the exploration.

2.2.1 Genetic operators

In most cases, for every genetic encoding there are *unary* and *binary* operators [ES98]. The less popular approach is to use *n-ary* operators.

The unary operators are called *mutations*. Those operators are meant to perform some *minimal*, *random*, *unbiased* and *meaningful* change in the genotype from the perspective of the solution. For example, in the genotypes represented as vectors of real numbers, the mutation can be performed by adding some small perturbation randomly with use of Gaussian distribution [ES98], while in tree representations the mutation can add or remove a node, or just modify some properties of existing nodes.

There are two different points of view on the mutation operator. It is commonly assigned to the exploration role, because it presents the new solution without analysing the neighbourhood of the previous phenotype in the fitness landscape and, as a result, it can end up in the point far from previous location in the landscape. On the other hand, the mutation designed to provide only small changes preserves most of the information from the previous solution. As it is stated in [ES98], "(...) *exploiting* a schema could be defined as *exploring* the corresponding hyperplane (...)". While the operator creates new points in the phenotype space, which is understood as exploration, those points refer to the same schema with small disruption. So it can also be interpreted as the exploitation of the schema.

Usually, the binary genetic operator is called a *crossover*. The typical crossover operator takes genotypes from two parents, and distributes the genes[1] between children according to some established strategy. The most popular strategies that can be assigned to the sequences and vectors are *uniform*, *one-point* and *two-point* crossovers. *The uniform crossover* distributes genes of both parents randomly to both children. *The one-point crossover* determines one cut point for both parents and swaps the sequences followed by this point for both children. The two-point crossover selects one substring in both parents and swaps them in children sequences. For tree-like genotypes the crossover is usually performed by swapping subtrees between parents to form children genotypes.

The general idea of the crossover is to preserve some aspects from both parents. One can make an assumption that if two good parents are recombined, then their offspring should also be good. This, of course, depends heavily on the fitness function, and the crossover method – one idea is to design the recombination in a way that child will not be less similar to its parents than one parent is similar to the second.

[1] Genes are interpreted here as some semantically-consistent parts of the genotype.

In different paradigms of the evolutionary algorithms the roles of operators may be considered differently [ES98]. In the *Evolutionary Strategy* the mutation is the main operator, and crossover is used only for adaptation purposes, while in the *Genetic Algorithms* the mutation acts like the background operator, and the crossover is the main operator.

2.2.2 Selection

The *selection* in the genetic algorithms is used to keep the most promising solutions in the gene pool. Selecting only the best phenotypes would lead to a very high selection pressure and low variety of solutions – this creates a risk of staying in a local optima of the fitness landscape. On the other hand, selecting random genotypes leads to low selection pressure, and the low efficiency of the evolutionary algorithm. The common approach of selection involves giving the creatures with high fitness function the higher probability of being selected to the new population.

One of the most obvious selection techniques is a *roulette wheel method*. If the i-th solution has the fitness value equal to f_i, and there are N solutions in the current population, the i-th solution will be selected from the current population with probability:

$$P(i) = \frac{f_i}{\sum_{j=1}^{N} f_j}$$

The selection is repeated until the desired number of genotypes are added to the gene pool. This is sampling with replacement.

The above method of selection is known for premature convergence and problems with selecting the best solutions, when the fitness values are very similar [RG11]. In order to avoid this disadvantage, new methods were proposed. One of the most popular techniques of selection is the k-tournament selection. In this approach k solutions are selected with the roulette method, and then the solution with the highest fitness value among those k solutions is selected for the new population. The process is repeated until all needed solutions are added to the new population. This method also allows repetitions.

Apart from the two above-mentioned popular approaches to selection (roulette wheel and tournament), there are many other variations and accompanying modifications of these methods, as well as more complex selection schemes like *convection* [KM18].

Due to the fact that selection tends to the overall improvement of the population fitness, the selection is interpreted as the main performer of the exploitation in the evolutionary algorithms [ES98].

2.3 Classification of genetic encodings

There are many types of genetic encodings for the evolutionary algorithms. The chosen representation usually depends on the problem that needs to be solved, for example:

- *Knapsack problem* usually uses a binary encoding,
- *TSP* – permutations,
- *Parametrization problems* – vectors of real values,
- *Genetic programming* – the tree encoding,
- *Evolutionary design and artificial life* – biologically-satisfying encodings, developmental encodings, binary encodings, graph encodings, rule-based encodings, recursive encodings and more.

Still, despite the large variety of genetic representations, it is hard to choose a good representation for a particular evolutionary design problem.

For the evolutionary design, all genetic encodings can be divided into *parametrization* and *open-ended representations* [Hor03]. The first one alters the parameters of an existing structure, while the second one changes and evolves all aspects of the structure.

The open-ended representations can be divided into *non-generative* and *generative* representations [Hor03]. The non-generative representations do not reuse genes – in the mapping they are used at most once. This is the popular way to encode the structures due to its simplicity. It is also easy to maintain in terms of genetic operators – it is easy to create a mutation operators that perform small changes to the phenotype. Unfortunately, for the complex problems involving the creation of a massive structure, the non-generative representations have problems with scalability. While the solution grows drastically over time, the changes applied to those solutions are as small as in the beginning. This eventually leads to a slowdown of th evolution. The lack of reusability causes the need to reinvent some required complex elements, like for example table leg, in order to satisfy the fitness function.

The generative encodings can *reuse* genotype fragments during the development of the final phenotype. Such encodings have better *scalability* – with time of the evolution the complexity of used structures for the development increases, as well as the complexity of the phenotype. Unfortunately, because of this increasing complexity and repeatability of elements in the genotype the genetic operators are much harder to implement – the single change in the genotype can cause huge changes in the phenotype, because it will be applied in many places. In this case it is hard to perform "organized" movement in the fitness landscape. The evolution for some generative encodings can be also slowed down by the evolution of special mechanisms, like rules or procedures of development.

The non-generative encodings can be split into *direct* and *indirect* encodings. In the direct encoding genotype reflects directly all elements of a simulated individual. It does not use any higher-level features to make genotype more compact or flexible – this is why such encodings are usually hard for a human to maintain. The direct encodings have one-to-one mapping from each representational element of the genotype to some component or parameter of the phenotype. In the indirect representation, the information about the phenotype is stored implicitly, for example in a form of development suggestions. Indirect genotype requires some translation that will convert it to a direct form, for example to the phenotype itself or other direct representation [Hor03].

The generative encodings can be divided into *explicit* and *implicit* encodings [Hor03]. The explicit encoding can reuse the genotype, but the generation of the phenotype is straightforward. The reusability may come from gene overlapping or such mechanisms as iterations. Implicit encodings usually consist of the set of rules that implicitly describe the phenotype, and the process of development involves deployment of those rules.

The final classification tree is presented in Figure 2.1. This is of course one idea of classifying the genetic representations, presented in [Hor03]. Sometimes, in other representations, the generative representations are assigned to indirect encodings.

Representations can be also classified by their similarity to the programming languages in terms of the following aspects – *combination*, *control-flow* and *abstraction* [HP02]. The first one assumes that representation starts with simple, atomic elements that can be aggregated to more powerful expressions. The control-flow assumes the presence of such mechanisms as conditionals or loops. The abstraction enables labelling of the elements or structures and passing parameters to genotype objects.

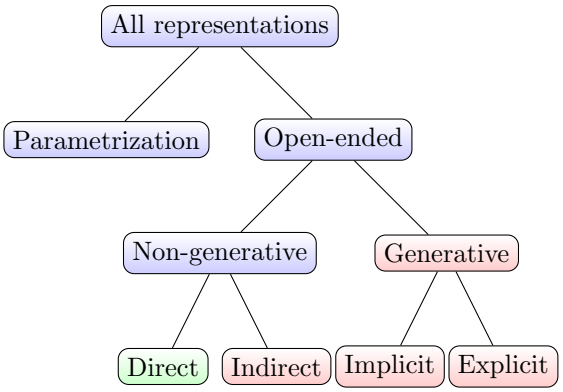


Figure 2.1: Classification of genetic representations based on [Hor03]. Sometimes the direct and indirect representations are presented as the first level of classification – in such cases generative representations are assigned to the indirect representations subtree, as presented by red color (green color represents direct representations).

2.4 Genetic mappings

In evolutionary design genetic representations may form a *hierarchical structure* – some representations need first to be converted to the other ones in a sequence in order to be finally converted to the phenotype. The example of the representation's hierarchy is shown in Figure 3.2 in Chapter 3.

During implementation of the genetic representations or the analysis of the structures generated by the genotype one can ask following questions [KU04]:

- Which genes are responsible for developing a given set of phenes?
- What phenes were generated by a given subset of genes?
- How traced genes are influencing phenes during evolution?

The answer for those and more questions can be answered by the use of *genetic mappings*. Let X be the source sequence of length m, and Y be the destination sequence of length n. The definition of the genetic mapping is following – "the mapping is the relation R on sets $X = 1, 2, ..., m$ and $Y = 1, 2, ..., n$, so it is the subset of the Cartesian product $X \times Y$" and [KU04]:

$$(x, y) \in R \Leftrightarrow \begin{cases} \text{the destination character at position } y \text{ originates from} \\ \text{the source character at position } x \end{cases}$$

The example mappings can be seen in Chapter 4. The mapping can give some useful informations about the properties of the genetic encoding, and about the usability of genes for a given genotype. As stated in [KU04], when some genetic encoding needs to be converted first to other representation instead of phenotype, then it is possible to form a *translation path* mappings that show how the information flows between representations.

For some indirect representations, the mappings allow observing interesting phenomena that occur in the natural DNA genotypes. The first one is a *pleiotropy* – it occurs when a continuous fragment of the genotype influences separate phenes [KU04]. The second one is a *polygeny* – a situation, in which one phene is described by many distant genes.

2.5 Development of a genetic encoding

During the analysis of the current evolutionary algorithms literature, it can be seen that even for slightly modified problems there is a need for a different genetic encoding in order to perform an effective evolution [Ron97]. In case of the evolutionary design or other types of optimization problems there is a need to design a representation that can cope with the various fitness functions. This is mainly because a redesign of the genetic representation for every problem is a time-consuming process. In the following sections two aspects will be raised – the features of the encoding that should be considered during an implementation and common issues of the genetic representations.

2.5.1 Aspects of developing a genetic representation

As it is stated in [Ron97], the process of selecting or developing the genetic representation usually requires the consideration of some important features. First of all, *the building blocks of the genotype should be meaningful for the problem type* [Ron97]. Secondly, *the genetic operators should use those significant blocks* in order to perform progressing evolution – the lack of above properties may result in a slowdown of the evolution of useful solutions, and finally in a stagnation. Thirdly, the genetic encoding that is not direct should have *an appropriate and possibly simple mapping function* to the phenotype or lower-level genetic representation [Ron97]. The final phenotype should be fully derivable from the genotype. The mapping should assign every element of the genotype to the corresponding elements in the phenotype. Every element of the phenotype must have assigned one or more genes from the genotype.

The genetic representation should always *generate the proper phenotype from the given genotype* [Ron97]. This involves such mechanisms as *repairing* of the partially invalid genotype or *reporting* of the incorrect genotypes and penalizing them. The genetic operators should always generate valid genotypes from the parents, otherwise the process of the evolution would be very slow.

What is more, the genetic representation should provide *the adequate level of abstraction* [Ron97]. High level may reduce a set of achievable solutions, but the evolution process may go faster and the implementation of the operators, especially recombination operator, may be easier to perform and more valuable to the evolution. Low level of abstraction, or the direct encoding allows achieving all possible solutions, but the implementation of the operators may be harder, and they can provide too small change that will cause slow evolution.

In the end, the good genetic representation should *suppress isomorphic forms* [Ron97]. That means there should be no genotypes that are mappable to the same phenotype.

Those are the properties of a good encoding. It is a complex task to create a representation for the evolutionary design problems that would satisfy all of the above properties.

2.5.2 Popular encoding issues

As mentioned in [Ron97], there are some issues that often occur during the development of the genetic encoding. Those issues usually come from the conflicting factors that need to be satisfied during the implementation. Solving one issue may cause another one. The idea is to deal with as many issues as possible, so in the end only the least harmful issues for the problem are left.

The list of the popular encoding issues is presented below [Ron97]:

- *Wrong degree of encoding* – the level of detail available in the genotype is too high or low for the problem.

- *Partial cover* – in the indirect representations the phenotype space is sometimes not fully addressed by the available genotypes. In the worst scenario, the global optimum of the fitness landscape may be not available by the genotypes. On the other hand, the use of the direct encoding usually involves complex genetic operators, and it is harder in general to evolve something from such encoding.
- *Too complicated mappings from genotype to phenotype.*
- *Improper genetic operators* – the naively designed mutations and crossover may not be able to add or transfer the valuable information to the offspring in terms of a given problem. What is more, badly defined crossover operator may create children that differ too much from their parents.
- *Isomorphism in the encoding* – the high degree of redundancy in the genetic representation, where many genotypes point to the same solution.

2.6 A brief introduction to Lindenmayer Systems

Lindenmayer Systems are rule-based rewriting systems that were initially used for a mathematical description of appearance and development of plants [PL96]. They were introduced by a biologist Aristid Lindenmayer. L-Systems can describe many types of fractals – they are used for plant analysis, visualization of the plants for computer graphics purposes and creation of the regular patters, like town maps. They are also applied in the evolutionary algorithms, especially in the evolutionary design.

2.6.1 Definitions for L-Systems

L-System is an ordered triplet $G = \langle V, \omega, P \rangle$, where

- V is an alphabet,
- V^* is a set of all words over V,
- V^+ is a set of all non-empty words over V,
- $\omega \in V^+$ is an axiom of the L-System,
- $P \subset V \times V^*$ is a finite set of productions [PL96].

The production rule $a \to \chi$ consists of a predecessor $a \in V$ and a successor $\chi \in V^*$. When there are no production rules for some word $a \in V$, then the default rule for this word $a \to a$ is assumed [PL96].

When there are two words $\mu = a_1...a_n$ and $\nu = \chi_1...\chi_n$, then the word ν is directly derived from μ if and only if for every $i = 1, ..., n$ there is a production rule $a_i \to \chi_i$ [PL96].

In other words, every letter in the current word is replaced synchronously by the successor of the rule with the predecessor that matches this letter during single iteration, for example for such L-System:

$$\omega : b$$
$$p_1 : b \to a$$
$$p_2 : a \to ab$$

The results for next iterations will be following:

0. *b*
1. *a*
2. *ab*
3. *aba*
4. *abaab*
5. *abaababa*
6. *abaababaabaab*
7. ...

2.6.2 The development of structures from L-Systems

L-Systems in [PL96] were used for the development of realistic plants. The process of "growing" is performed with a use of a LOGO turtle.

After developing the final sequence of L-System the turtle "analyses" the given letters as commands of movement. It can move forward, back, left, right, up, and down. In the parametric L-Systems it is possible to determine the angle of rotation or length of the drawn line. Apart from movement and drawing of the structure, the turtle can also store its state in the stack when a branching letter, usually "[", occur. The stacked states are popped with "]" letter. This allows creating complicated branching structures. The state preserves the information about the *position* of the turtle and its *direction* of movement.

Usually available commands for the turtle are move forward, turn left or right, pitch down or up, roll left or right, turn around, and stack or pop its current state.

2.6.3 Classification of L-Systems

Due to different applications and detail requirements various types of L-Systems are defined. The first and the simplest one is *DOL-System*. It consists only of simple letters, without any branching or letter parameters.

The *Bracketed OL-Systems* are designed strictly for the development of more complex structures with the LOGO turtle, and they enable stacking and popping of the turtle's state. This allows creating branches.

Another group of L-Systems are *Stochastic L-Systems*, in which rules are applied with defined probability for each rule [PL96]. In terms of evolutionary design such mechanism is unwanted, because the genotype and the resulting phenotype should be deterministic.

The fourth kind of L-Systems are *Context-sensitive L-Systems*. In comparison to the classic L-Systems the rules additionally check closest meaningful neighbours of the current letter in order to decide if the letter should be replaced by the rule's successor. The information about the desired neighbours is stored in the definition of the rule.

Next group is the *Parametric L-Systems*. This is the most powerful group from all presented. It does have brackets, it may be context-sensitive (*2L-Systems*) or context free (*OL-Systems*). In this group every letter can have a list of real parameters. The parameters can be used for turtle words in order to manipulate the rotation or length of the turtle's drawing step. The parameters also allow adding conditions for rules – in this situation the conditions check the values of predecessor's parameters and determine if the rule should be applied [PL96].

The last group is *Timed DOL-Systems*. This group allows continuous value of the "development time" and introduces the additional parameter, called time, that can be used in letters. Every letter has defined current age and the life length, after which the rule for the letter can be deployed. This allows performing continuous changes to the structure between iterations, for example the length of the drawing line may increase.

More detailed explanation of each version of *L-Systems* is presented in [PL96]. For the rest of the paper, the *letters* in terms of L-Systems will be replaced by *words*, and the *word* in current definition will be replaced by *sequence of words*.

2.7 Genetic encodings for the evolutionary design problems

2.7.1 Biology-based approach (Eggenberger)

One of the encodings presented in this thesis was introduced in [Egg97]. It was designed by Peter Eggenberger to perform simulations of the biological evolution. The idea was to imitate the development of multi-cellular 3D organisms, and to design a genotype that would consist of development rules instead of direct representation. As it is stated in [Egg97], the mechanisms that are agreed to exist in the biological development are:

- Cell division,
- Cell differentiation,
- Programmed cell death, called apoptosis,
- An ability to determine the position of the cell within multi-cellular body by use of morphogenetic gradients.

Usually, with some minor exceptions, all cells in the biological organism have the same genotype. The difference between cells comes from the different subsets of active genes. The activity of a gene is determined by special DNA regions, called *cis-regulators*, and corresponding *trans-regulators*, which are free compounds that exist around DNA with different concentration. When the concentration of the trans-regulator is high enough, the gene followed by the corresponding cis-regulator is activated, and the processing of the gene occurs in the cell [Egg97].

Cells can be differentiated in two possible ways. The first one is called *cell lineage* – it is an autonomous mechanism of the cell that triggers the differentiation process according to the current state of cell. The second one is *cell induction* – in this case cells emit chemical signals to other cells in order to activate the process of differentiation. Inter-cellular differentiation can be triggered by affecting the nearest cells, or by distributing signals that are received by corresponding receptors in other cells.

The genetic encoding proposed in [Egg97] consists of seven digits $\{0, 1, 2, 3, 4, 5, 6\}$. Every gene has fixed length n. The first gene in the genotype is the *regulatory unit*. It is the equivalent of the cis-regulator in the real DNA. The following genes are also regulatory units until the digit 5 appears in the end of some gene. After this digit, all genes are structural genes till the gene that ends with digit 6. The genes followed by this gene are again regulatory and so on. The structural gene is active when the preceding regulatory unit is matched with an activation condition. The regulatory unit is activated, if there are transcription factors in cell for which the calculated affinity to the regulatory unit is positive.

The gene activation may have one of the following consequences:

- Production of a new transcription factor,
- Production of a cell adhesion molecule, CAM – the enough concentration of CAM in two cells connects them,
- Production of receptor for communication with other cells,
- Triggering division or death of the cell.

Mutation changes digits in the genotype, the crossover is one-point.

The results of this representation can be seen in [Egg97]. The encoding was able to form some interesting structures built from spheres, i.e. for problems of developing symmetric creatures.

2.7.2 Autonomous agents with body and nervous system (by Dellaert and Beer)

Another examples of genetic encodings are introduced in [DB96]. They were designed to create 2D autonomous agents with body, brain, basic effectors, and sensors. The creatures are represented in the world as squares built from smaller squares, representing the cells of the creature.

The first genetic representation introduced in [DB96] is biologically defensible, but complex. As it was stated in [DB96], all cells contain the same genotype, and some genes are activated due to the appearance of some proteins. Each protein is represented as an integer number. The genotype is the set of operons – they check with use of Boolean function if some proteins exist or not in the cytoplasm and according to this they may produce another proteins.

Proteins determine cell capabilities, type, and they trigger the process of the cell division. In order to receive different cells from the same parent cell, the *symmetry breaking proteins* and inter-cellular communication, based on passing of proteins between cells, were introduced [DB96].

The process of brain development is performed after the development of the body. The process of neural network growth involves special proteins that link together cells that satisfy some conditions specified in [DB96].

After development and some tests involving the evolution of the *Braitenberg Hate Vehicle*, the author stated that without human involvement the complex model was unable to evolve useful results. This is why new, simplified model was implemented. The simpler representation uses *Random Boolean Network* as a representation of the genome [DB96]. In this case the cell characteristic is determined by the state of RBN – the terminology of the proteins and the cytoplasm is neglected. The division protein is replaced by a single bit value, and breaking of the symmetry is performed by deterministic flips of bits representing RBN state. The neural network is developed in the cells that have set an "axon" bit. The ends of the network are attached to all cells within range that have "target" bit set.

The simplified model was able to evolve much more interesting agents, even for more complicated tasks, like line follower. The results, along with visualizations are available in [DB96].

To sum up, the complex representation was implemented with use of Boolean functions. It was biologically defensible, and it was possible to create agents that would solve the problems. Unfortunately, the evolution was unable to evolve solutions for problems – the search space was hard to explore with this representation. Only manually created solutions could cope with the problems. The simplified encoding was computationally cheaper, easier to analyse, and it was able to create successful agents for the given task that required both morphology and brain.

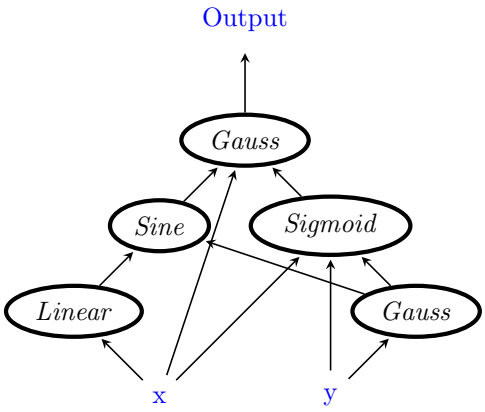


Figure 2.2: An example of CPPN network.

2.7.3 Generative representation based on L-Systems (Hornby)

One of the most popular generative genetic representations was introduced in [Hor03]. As it is stated in [HP02], the generative encodings perform better during complex problems such as evolution of advanced structures due to their ability to reuse some components created during evolution. The proposed representation allows developing both body and brain.

The body development is performed in the same manner as in parametric L-Systems – it has parametrized words for Turtle movement, such as rotations, branching, moving forward and more. Additionally, the representation from [HP02] has the ability to loop several words within braces, so they will be repeated n times.

The brain is a neural network formed as a connected graph. The neural network has separate state stack, the addition of a new neuron is performed by splitting the current neuron. After splitting, the class of the new neuron is inherited from the previous neuron, unless the change of the neuron class is provided in the genotype. There are several words for decreasing and increasing weights, duplicating and moving neural connections. The neural network graph is consistent in this representation.

The mutations for this representation are following [Hor03]: *word replacement, perturbation of word parameters, changing the condition of rule, addition or deletion of words from rules' successors or encapsulation of words from one successor to another rule.* The crossover operator switches some rules between two parents to form children genotypes.

As it was presented in [HP02], the generative encoding outperformed the direct one for the task of evolving a "table"-like structures, while the genotype was no longer than for the direct encoding. The similar representation, available in Framsticks, is presented in Section 3.8.

2.7.4 Compositional pattern producing networks

Till now the presented genetic representations were developmental, and they were based on the grammar and rule deployment, or the imitation of chemical processes that occur in the cells. The other approach that is especially popular with articles focused on the evolution of control systems is based on the idea of *Compositional Pattern Producing Networks (CPPN).*

The CPPN represents the phenotype as a function of Cartesian space, assuming that the complex process of the embryonic development of multi-cellular creatures can be approximated by giving the geometric representation of the individual instead of simulating developmental processes like in previous representations [TM15].

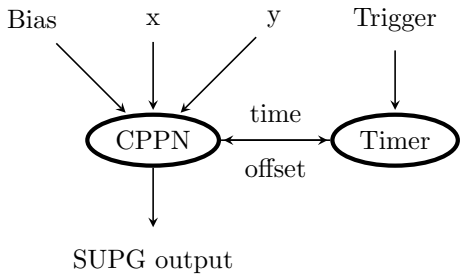


Figure 2.3: The visualization of the SUPG macro-neuron.

The CPPN genotype is represented as a directed graph, with weighted arcs. The nodes represent functions, such as *Sine*, *Gaussian*, *Sigmoid* or *Linear*. The arguments for those functions are the weighted sums of input arcs for nodes. The output of the node is the result of the node function with a given input. Such graph takes (x, y) coordinates of the phenotype space and returns the "gradient" of the point that is used to develop final solution, as presented in Figure 2.2.

The mentioned subset of the node functions enables the CPPN to create various patterns that are visible in real-world solutions, like symmetry or repetition [TM15]. The CPPN networks are used mainly in a generative art[2] and in the evolutionary robotics to evolve control systems.

One of the popular problems of the evolutionary robotics is to evolve the gait for the hexapod robot [TM15]. The robot has eighteen servos, three for each leg. That would give eighteen degrees of freedom, but usually two of the servos for each leg work in antiphase to increase stability, so the robot has twelve degrees of freedom.

The first method for evolving control systems with CPPN networks is called *HyperNEAT*. In this approach the robot is controlled by a neural network that has 14 inputs, a hidden layer, and 12 outputs. Each output is applied to the servo. The inputs are the previous values applied to servos, and additionally sine and cosine functions, to allow cyclic behaviour of movement. The network is fully connected, and the weights of the connections are encoded by the CPPN. The inputs of the CPPN (x, y) correspond to the beginning and the ending of the connection in the network [TM15].

The second popular approach involving CPPN is the use of *SUPG macro-neurons* [TM15]. The schematic of the SUPG neuron is presented in Figure 2.3. When this neuron is triggered by signal, it produces a single cycle of a CPPN encoded activation pattern. The simultaneous triggering of the SUPG results in oscillations that can be used to control the servos of the robot. In this case the outputs of the two SUPGs for each leg specify the desired angles of two servos. Those SUPGs for single leg are triggered when robot touches the ground with this leg.

The current experiments on CPPN-based control systems show the advantage of SUPG-driven systems over other solutions [TM15].

2.7.5 Summary of the presented encodings

The above overview focuses on the various types of genetic representations presented in the articles. First of all, based on [DB96] one can assume that the implementation of the representations that are biologically plausible usually leads to ineffective evolution of solutions. The simpler models can usually achieve more satisfying results with less computational power involved. Secondly, the generative representations for evolutionary design problems are told to perform better than the

[2] `http://picbreeder.org`

typical direct encodings, according to [HP02]. This is because of the ability of the generative encodings to use elements that are becoming more and more complex, as well as the evolved structure itself. In the end, the shown CPPN networks can be considered as the promising idea for the genetic representations.

Chapter 3

Genetic encodings available in Framsticks

3.1 Introduction

The *Framsticks* platform enables users to use various genetic encodings for their artificial life and evolutionary design problems. As it was explained in Chapter 2, the choice of the proper representation may be crucial for the process of finding the best solution for a given optimization function. Every representation may have differently projected fitness landscape, and such projection can speed up or slow down the process of the evolution. Differences in the fitness landscape topologies for the same problem can be the result of genetic representation limitations, differences in body and brain development algorithms, and used genetic operators.

3.1.1 Elements of simulated 3D agents in Framsticks world

In Framsticks platform every agent consists of *parts*, *joints*, *neurons* and neural *connections*. As it is described in [KRV01], the body of the agent is composed of sticks. A single stick is formed from two parts and a joint.

The part in classical *"ball-and-stick"* Framsticks model simulation is interpreted as a material point located in the world. Joints are not physical objects in the simulation – they are physical constraints that force parts to preserve initial distances between them. Every Part has physical properties, like *density* and *friction*, and biological properties, like *assimilation* or *ingestion*. Joints also have physical properties – *stiffness* and *rotational stiffness*. The summary of forces applied

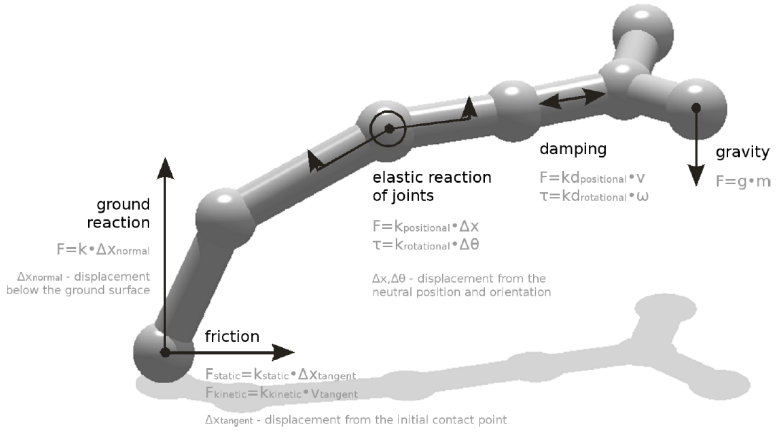


Figure 3.1: The graphical summary of forces that affect each part during simulation [KRV01].

to each Part during simulation is shown in Figure 3.1.

It is also possible in Framsticks to use ellipsoid, cuboid, or cylinder solids as a part [KU]. In such case, those solid shapes are interpreted as physical elements that are connected with solid constraints. This allows creating even more custom and realistic structures. In such case, additional properties of Parts are a scale, and a shape type.

Apart from the agent's structure, the genotype also defines a control system, represented in a form of recursive neural networks. There can be many classes of neurons. They can be attached to parts, joints, or they can also be unattached, depending on their application. Each neuron type has an internal state s, which is updated with use of some user-defined functions. The argument of those functions is usually the weighted sum of input connections [KRV01]. Each neuron type can use from zero to multiple inputs and outputs. What is more, each neuron class can have properties that determine the behaviour of the neuron. For example, in classical *sigmoid neuron* (N) there are properties called $force$, *inertia* and *sigmoid*. The current state of the neuron is determined with following formula [KRV01]:

$$velocity_t = velocity_{t-1} \cdot inertia + force \cdot (weightedsum_t - s_{t-1})$$

$$s_t = s_{t-1} + velocity_t$$

Where $t-1$ represents variable values from previous iteration. After computing the current state of the neuron, the final output is computed as:

$$out_t = \frac{2}{1 + e^{-s_t \cdot sigmoid}} - 1$$

Some neuron classes enable interactions with the environment – they are divided into *sensors* and *effectors* [KU]. Sensors have no inputs, they gain information straight from the environment. The sensor state can depend on applied forces (like *Gyroscope* (G) neuron), closeness to other objects (like ground for *touch* (T) receptor), or concentration of some attractors (like light, smell, or other range signals). There can be also various types of user-defined stimuli.

The available effectors are *bending* and *rotating* muscles. Those types of neurons are always attached to joints. The first type rotates the second part of the joint along the Z-axis of the first part, and the second type rotates the second part along the X-axis of the first part.

The neurons can be connected with use of connections, with respect for constraints in number of allowed inputs and outputs for connected neurons. Every connection has a weight.

In the end, for every Model in Framsticks there are several constraints that have to be satisfied in order to create the valid creature:

- Two parts can be connected by at most one joint,
- A joint cannot connect a part with itself,
- All body elements must form a single entity – every part needs to be directly or indirectly connected to the rest of Model parts,
- For each joint, the distance between two parts must not exceed joint's maximal possible length value.

3.1.2 Hierarchy of genetic encodings in Framsticks

Section 3.1.1 was focused on the phenotype of a creature. In the following sections various kinds of the genetic encodings will be presented. All Framsticks genetic encodings need to be converted

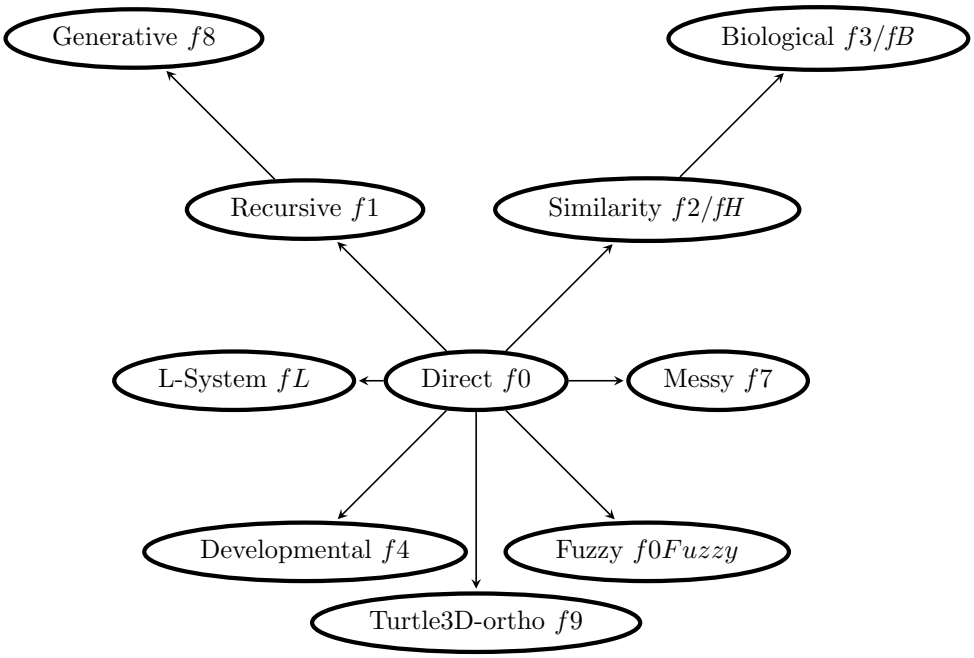


Figure 3.2: Hierarchy of genetic encodings in Framsticks.

to the *direct encoding*, called *f0*. Encodings can be formed on top of other encodings, so that multiple conversions will be required to convert it to the phenotype [KU04, Kom12]. The current hierarchy of genetic representations in Framsticks platform is presented in Figure 3.2 [Kom12]. The following sections will describe genetic representations that were available in Framsticks before the development made for this thesis.

3.2 Direct encoding (f0)

3.2.1 Syntax

As the name states, the *f0* encoding directly describes every element and parameter of the parts, joints, neurons, and neuron connections. This is the phenotype represented in a text form.

Each line in *f0* genotype describes one object of the final model. Based on the *Framsticks Website*, the syntax for each line in the genotype looks like following:

```
CLASSID:PROPERTY1=VAL1, PROPERTY2=VAL2, ...
```

where *CLASSID* is an alphanumeric identifier of the object's type, and *PROPERTYN* is the name of a single property of the object. The *CLASSID* values are "p" for parts, "j" for joints, "n" for neurons, and "c" for neural connections [KU18].

Each class if characterized by a different set of properties:

- *Part* (p:) properties are following:
  - *Absolute position in 3D space* (x, y, z) – determines the position of the part in the model, default values are 0,
  - *Physical properties* – density (dn), friction (fr) and size (s),
  - *Biological properties* – assimilation (as) and ingestion (ing).
- *Joint* (j:) is characterized by:

  - *Absolute indexes* of parts connected by the joint (p1 and p2) – those properties are required arguments of the joint,
  - *Physical properties* – stiffness (stif) and rotation stiffness (rotstif),
  - *Stamina* (stam) – biological property determining the strength of creature's joint.

- *Neuron* (n:) has the *index* of part (p) or joint (j), to which it is attached, and *neuron description* (d). The description contains *neuron class name*, and optionally *properties of the neuron* derived from the neuron class, like sigmoid or force for sigmoid neuron "N".

- Connection (c:) has the *absolute indexes* of the parent neuron (n) and source neuron (i). The last property is a *weight* of the connection (w).

Every part can be connected to another, which allows *f0* genotypes to *form loops*. Every attachment – of joints to parts, of neurons to joints or parts, and of neurons to neurons requires an absolute identifier to the element or elements. This may lead to problems with a manual edition of a genotype – small change, for example removal of a single neuron shifts the identifiers followed by this neuron and requires redefinition of the all neural connections. This also makes the encoding hard to perform crossover on the genotype level.

The example genotype for f0 encoding is following:

```
//0
p:y=-2
p:
p:1, -1
p:y=-1, 1
p:-1, -1
p:y=-1, -1
j:0, 1
j:1, 2
j:1, 3
j:1, 4
j:1, 5
n:j=1, d=|
n:j=2, d=@:p=1
n:j=3, d=|
n:j=4, d=@
n:d="N:in=0.89236, fo=0.808508, si=2, s=0"
n:d="Sin:f0=0.061, t=0.117"
n:d=*
c:0, 4, -0.25
c:1, 4, 0.25
c:2, 4, 0.25
c:3, 4, -0.25
c:4, 5, 5.667
c:4, 6, 2.124
c:4, 4, 1.746
```

Visualization of body and brain for this creature is shown in Figure 3.3. The creature was taken from the list of example genotypes, designed for the problem of swimming in water environment, and is called a *jellyfish*. For joints and connections the first two properties of both elements are absolute identifiers for parts and neurons, respectively.

The simplest genotype for this representation is single Part:

```
//0
p:
```

The phenotype space is fully addressed by this representation – the *f0* genotype can describe any possible phenotype.

3.2.2 Genetic operators

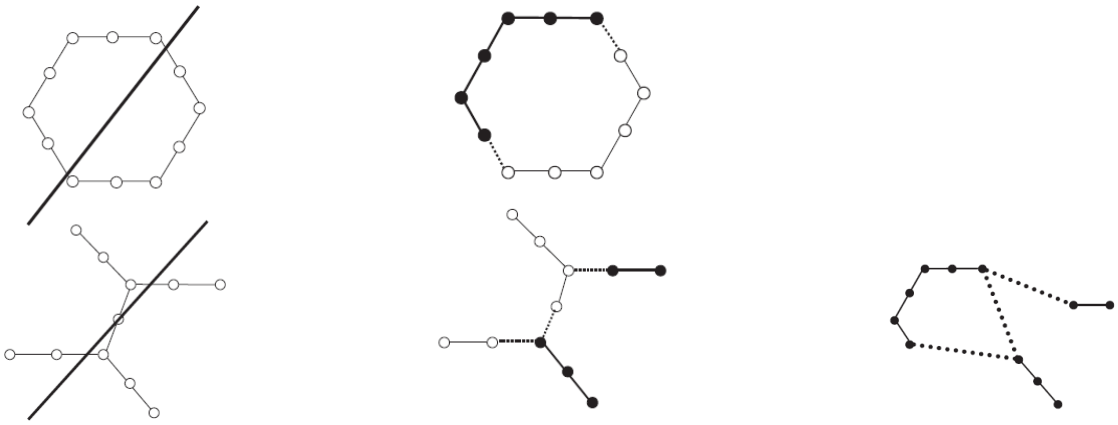
Mutations

Available mutations for this representation are [KU]:

(a) Body of the creature

(b) Brain of the creature

Figure 3.3: Body and brain developed from example f0 genotype, called jellyfish



(a) Random selection of cutting planes

(b) Selection of the parts from both parents

(c) Example offspring

Figure 3.4: The process of crossover in f0 encoding – firstly both parents are subdivided into two parts each by a random plane, and then parts from each of parent are connected into two children [KRV01].

- *Addition* of Parts, Joints, Neurons or Neural connections,
- *Deletion* of one of previously mentioned element types,
- *Alteration* of properties of one of Model elements,
- *Swapping* of Parts.

Each of the above methods provides relatively small change to the current genotype, so those genetic operators fulfil the idea of performing possibly small changes.

Crossover

As it is stated in [KRV01], the direct encoding does not offer a straightforward method for the crossover. It bases directly on the geometry of parents' phenotypes – the selection of parent gene is performed by creating a random plane in the space where the parent is placed. After this, the plane cuts this parent into two parts, like in Figure 3.4a. The same procedure is performed

for the second parent. In the end, both parts of the first parent are grafted randomly with both parts of the second parent. An example of such crossover is presented in Figure 3.4. The reason for performing crossover on existing models comes from the nature of the direct encoding – the genotype holds absolute indexes for every connection in body and brain, and the exchange of definitions of elements in both parents would lead to invalid genotypes.

3.3 Fuzzy encoding (f0Fuzzy)

Maciej Hapke and Maciej Komosinski proposed an extension for the *f0* encoding, which would enable evolving fuzzy controllers for creatures [HK08]. This approach, called *f0Fuzzy* was motivated by successful implementations of control systems with use of fuzzy rules, listed in [HK08].

This encoding enhances classic *f0* with mutation and crossover operators that are focused on the evolution of fuzzy neuron.

3.3.1 Syntax

The syntax for the *f0Fuzzy* encoding is the same as in *f0*. The class of fuzzy neurons is called "Fuzzy" and the definition contains fuzzy sets and rules.

The example definition of fuzzy neuron, presented in [HK08], looks like following:

```
...
n:d="Fuzzy:ns=4, nr=2,
fs=-0.1647;-0.1526;-0.0087;0.0631;
-1;-0.8774;-0.7725;-0.6767;
0.0087;0.2308;0.3585;0.4806;
0.011;0.1664;0.2362;0.2718;,
fr=0;3;1;0;2;0:0;2;3;1;2;1;1;3/
2;0;0;2;1;2:3;1;2;0;1;2;0;0/"
...
```

A "ns" field determines number of fuzzy sets, and a "nr" field defines number of rules in neuron. A list of numbers within range $[-1, 1]$ in a "fs" field represents fuzzy sets. Every fuzzy set is represented in a trapezoid form, described by four numbers each. The definition of the rules in a "fr" field consists of integer numbers. They represent the number of inputs in the premise part, the fuzzy set used for the fuzzification of the input, the number of outputs in the conclusion part and the fuzzy set used for the defuzzification of the output [HKW03].

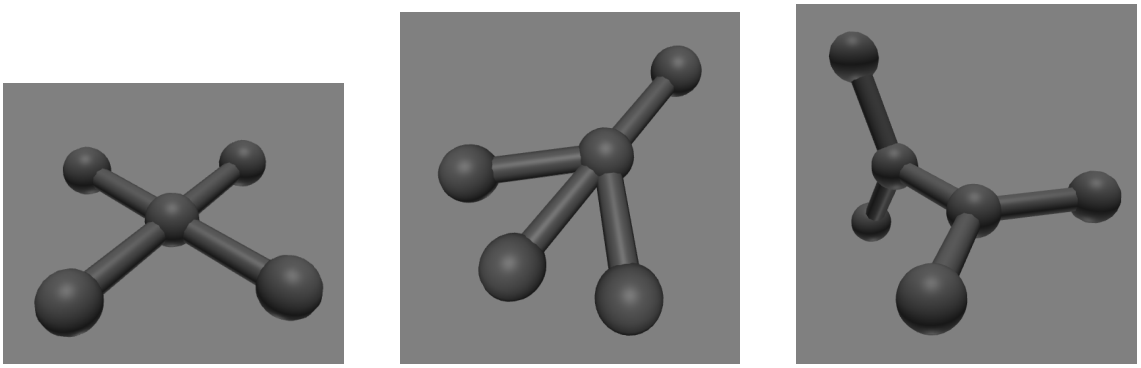
3.3.2 Genetic operators

The classic *f0* operators are ignored during *f0Fuzzy* evolution – the body is fixed during the process of the evolution. The aim of evolving the *f0Fuzzy* solution is to develop best control system for the particular agent.

Mutations

For the development of fuzzy neurons new operators were introduced in the *f0Fuzzy* encoding [HK08]:

- *Addition of a new fuzzy set* – adds new set with random values for trapezoid ranges, and a new fuzzy rule using the newly created set in order to avoid unused fuzzy sets
- *Deletion of a random fuzzy set*
- *Addition of a new fuzzy rule* – creates a rule with random number of inputs and outputs, with random fuzzy sets assigned to them



(a) `X(X,X,X)` (b) `X(,,X,X,X,,)` (c) `X(X,RRX(X,X))`

Figure 3.5: Examples for branching and rotations in *f1* encoding.

- *Addition of a new input to rule*
- *Deletion of an input of random rule*

Crossover

Firstly, all fuzzy sets from both parents are passed to the descendants. If there are identical sets in both parents, then they are passed only once to the offspring. The number of fuzzy rules added to the descendant is selected randomly from range

$$[\min(numrules_1, numrules_2), \max(numrules_1, numrules_2)]$$

Where $numrules_i$ represents number of rules in i-th parent. The similar procedure is performed for defining number of inputs and outputs of fuzzy rules.

The more detailed description of the crossing over in the *f0Fuzzy* encoding is provided in [HKW03].

3.4 Recursive encoding (f1)

The recursive encoding is the first higher-level encoding implemented in the Framsticks system. It is an easy-to-use representation, in which the linkage between body parts is performed implicitly, without the need to specify absolute indexes of parts or neurons. In contrast to the *f0* encoding, this encoding is user-friendly, and it is easy to edit it by hand.

3.4.1 Syntax and development

The body is represented as a tree structure, where "X" is single stick, and "(...)" is the branch. Within brackets the nodes of the tree are separated by commas [KU18]. During branching, the commas determine to how many angles the full angle placed on the current rotation plane should be divided [KRV01]. Then, the defined sticks grow in the direction according to their placement between commas. The examples of such branching are presented in Figure 3.5.

The properties of the sticks are altered by several modifiers. Most of them propagate their effects along sticks followed by modifier character in the genotype. The propagation is performed with different strengths, adjusted experimentally for every modifier. The available modifiers are listed in Table 3.1. Uppercase letters increase values of the properties, while the lowercase letters

Modifier	Altered property	Additional information
	Body development properties	
R, r	Rotates the branching plane by 45 degrees	Does not propagate
Q, q	Skew of the branching plane	
C, C	Curvedness of the sticks	
	Physical properties of the sticks	
W, w	Weight of the stick	
F, f	Friction of the stick	
L, l	Length of the stick	
	Biological properties of the sticks (all values are normalized to 1)	
M, m	Muscle strength (muscle speed)	The stronger the muscle, the bigger force, speed, stress resistance and use of energy
A, a	Assimilation	Ability to photosynthesise
S, s	Stamina	The durability of the stick
I, i	Ingestion	The ability to gain energy from the food

Table 3.1: The list of available modifiers in *f1* and *f4* encoding. The biological properties of sticks are normalized for each stick after propagation, so they all sum up to 1.

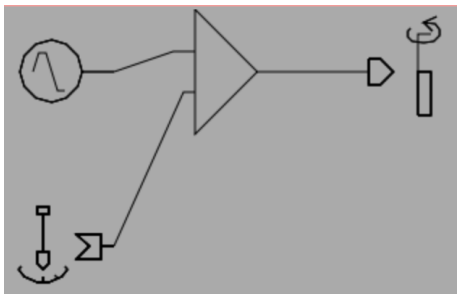


Figure 3.6: Example of neural network definition in *f1* encoding. The genotype for this network is following: `X[Sin][N,-1:1,1:2][G][@,-1:1]`

decrease them. Biological properties are normalized for every stick after propagation – the values of every biological property within part or joint sum up to 1.

The neuron definition has the following syntax [KU18]:

```
[CLASSNAME, PROPERTYANDINPUTLIST]
```

"*PROPERTYANDINPULIST*" is the list of neuron properties and input definitions separated by comma. The input connection is represented as $RELID : WEIGHT$, where the first argument is a relative position of input neuron. Zero value represents this neuron, positive values represent neurons following the current one, and negative values represent neurons preceding the current one. The example of neural network connection is presented in Figure 3.6. The neuron definition can be placed anywhere after the first occurrence of the "X" letter. If the neuron requires an attachment to the part or joint, then it is attached to the required element of the directly preceding stick in the genotype.

The simplest available genotype for *f1* is a single stick: X.

3.4.2 Genetic operators

Mutations

The mutations for the morphology of the creature base on *addition* or *deletion* of sticks (X), branches, commas and modifiers. As for the neural network, the available mutations are the *addition* or *deletion* of the neuron, or *modification* of the neuron definition. The modification involves changing the numerical values of neuron weights or properties, and addition or deletion of a neural connection.

Crossover

The crossover operator for *f1* encoding selects two cutting points for each parent in such a way that the syntax of the encoding is not violated, and swaps sequences delimited by those cutting points in both parents to form two children.

3.5 Developmental encoding (f4)

The developmental encoding, called *f4*, is inspired by the development of multicellular organisms. At first, the cells are *totipotent*. This means that they do not have any particular function, but they preserve the ability to divide. During chemical reactions that act as signals for development, those cells are specialized to perform particular functions in the body. Usually those cells cannot divide later after specialization, but there are some exceptions to this rule.

In the *f4* encoding the genotype describes the development process instead of the explicit coding of the body elements, like in *f1* encoding [KRV01]. In the beginning, the creature consists of one totipotent cell. The genotype specifies whether cell should divide or specialize. The division is represented by "<" character. After this, the current cell is divided into two new cells. The cell can be specialized to stick (X) or neuron (N). After specialization to stick, the cell cannot be divided anymore. The division of neurons is possible, but the children of neuron cell will be always neurons. The end of current cell development is delimited by ">" .

The genotype can be represented in a form of a tree, where every meaningful token is represented as a single node. Every branching in the tree is initialized by the "<" nodes, and every branch ends with the ">" leaf. The example of such tree is in Figure 3.7. As it is mentioned in [KRV01], the processing of each cell is done in parallel – during division the first child inherits the thread number from the parent, and the second child gets the first available thread number. The list of modifiers and sticks are analysed from division node to the end-of-development node, not in inverted order. If there is more than one division in a row, the branching occurs. The branching is performed in a similar way as in the *f1* encoding – the full angle on the branching plane is divided as many times as there are sticks that share the same part. The commas needed for branching are implicit, but the number of divisions of the full angle may be altered with additional commas in the genotype. The development process of the phenotype presented in the tree form in Figure 3.7 is shown in the Figure 3.8. The sticks are added in the order resulting from the thread number.

The modifiers for the *f4* encoding are the same as for the *f1* encoding, and they were listed in Table 3.1. Due to those similarities simple *f4* genotypes can be approximately converted to the *f1* genotype. For example, the presented earlier genotype:

$$<<<<X><ccX>X>X>X><<RR,X>X>X$$

has an equivalent *f1* genotype:

$$X(ccXX,X,X,RRX(X,,X))$$

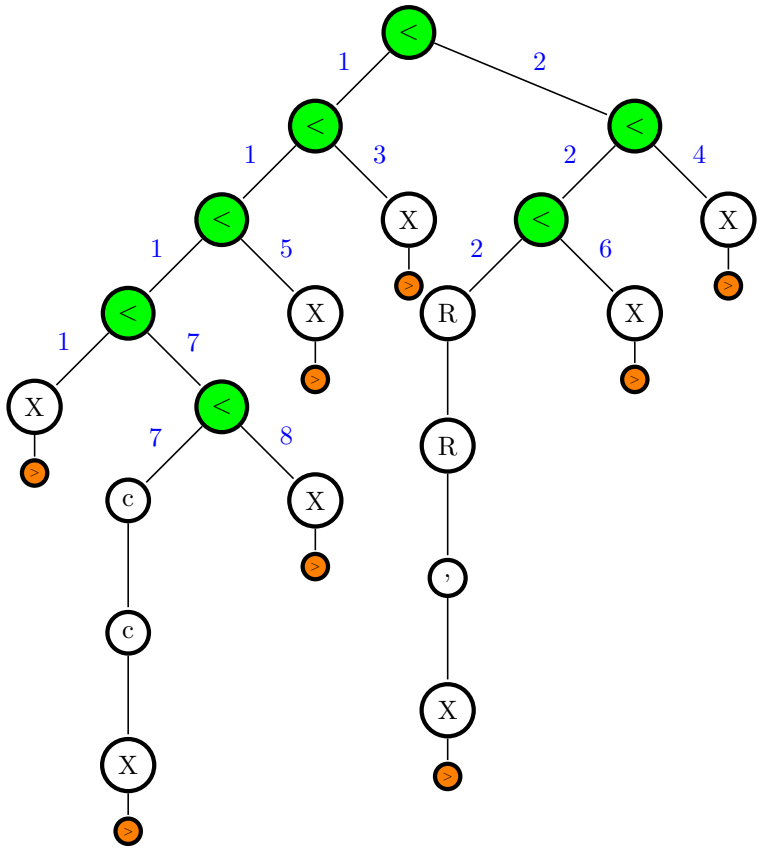


Figure 3.7: Tree representation of `<<<<X><ccX>X>X>X><<RR,X>X>X`. Green nodes are representing the division nodes, the orange ones represent the termination of the cell development. Blue numbers for edges are representing thread numbers of the cell development. All threads are processed in parallel. The final elements are added to the body in the order represented by edge numbers, as it can be seen in Figure 3.8.

.

The possible neuron definitions in the *f4* encoding are following:

```
N
N@
N|
@
|
```

As opposed to the *f1* genotype, the neuron definition always starts with "N". The first neuron definition is the default one, in which the sigmoid neuron is created. The next two definitions are the old semantics, in which sigmoid neuron is created along with the muscle. In this case the inputs for such pair are defined for the sigmoid neuron. The last two definitions of the muscle create single muscles without neurons.

After the definition of the neuron class, the inputs are defined like following [KU18, KRV01]:

```
[SENSOR:WEIGHT]
[RELID:WEIGHT]
```

All neural connections, unlike in the *f1* genotype, are defined in separate square brackets. In the first definition, new sensor is created and attached only to the parent neuron. In the second connection type the representation considers the neuron development order instead of the order of definitions in the genotype. That means that negative values of position refer to the neurons that were developed earlier (they had a lower thread index), and the positive values refer to the neurons that will be created later. It means that relative index does not refer exactly to the position of

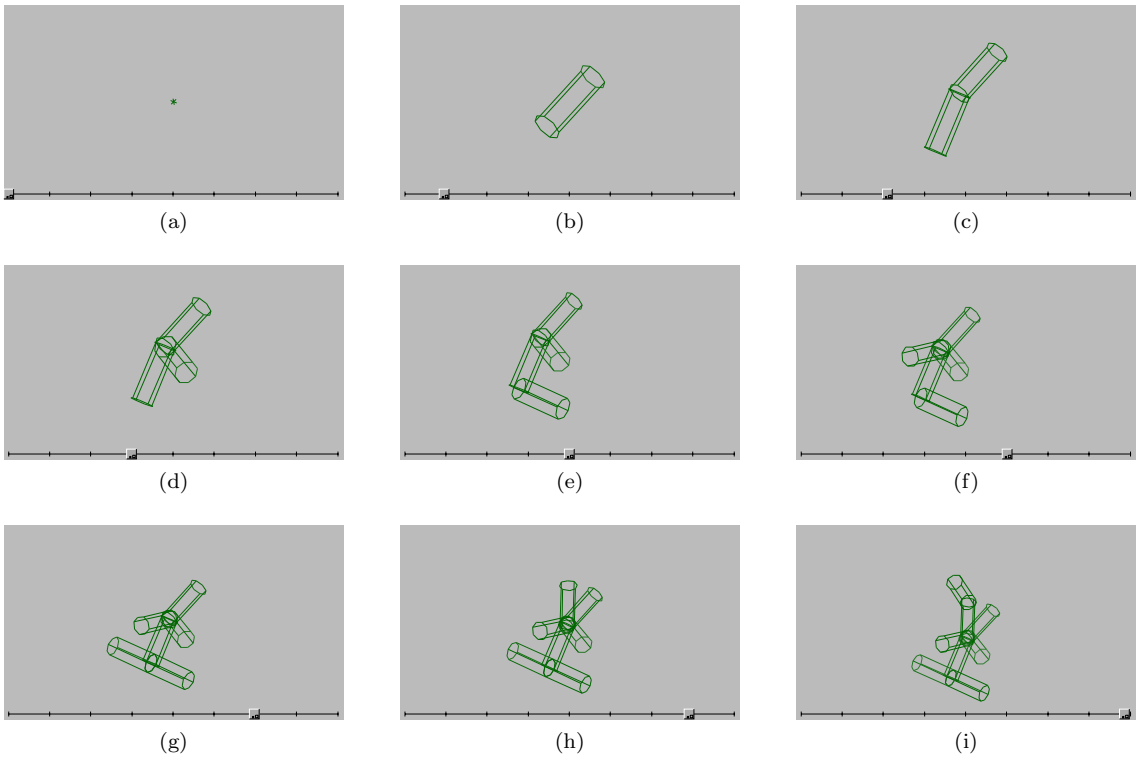


Figure 3.8: Development of the creature presented in the Figure 3.7.

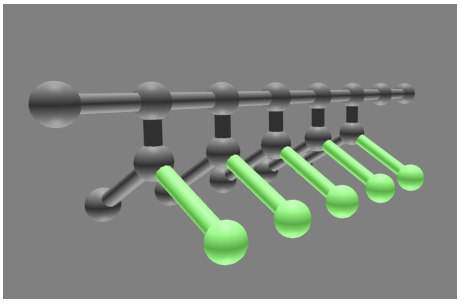


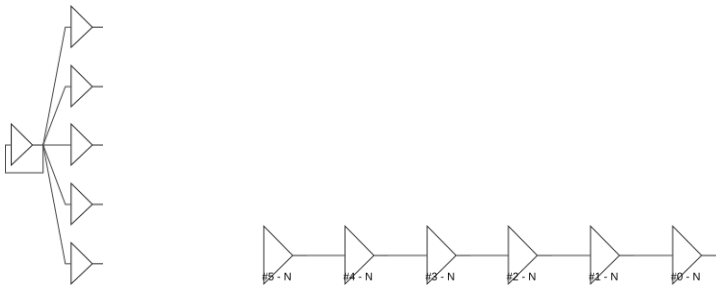
Figure 3.9: Example of node repetition in the *f4* genotype. The green color shows the phenotypic result of the underlined fragment of following genotype: `rr<X>#5<,<X>RR<<llX>LX>LX>>X`

neuron in the genotype, but to the position of the neuron in the tree. For example phenotypes for `<<X>N>N[1:1]` and `<X><N>N[-1:1]` are the same – in both cases one neuron is connected to another.

The neurons, unlike sticks, can be divided after differentiation, for example `<X>N:Sin<><>` creates three sine wave neurons. The connections for parent neuron are duplicated for its children. Every child can have its own neuron connection within <>.

The simplest genotype in *f4* is the same as in the *f1* representation: X.

The main difference between *f1* and *f4* encoding is that the second one enables iterations in the genotype. The iteration $\#NUM$ repeats followed node NUM times. Repetition can be applied both to body and neural networks, which enables the evolution of large organisms from smaller cloned parts. The example for body repetition is shown in Figure 3.9, and examples for neuron repetitions with connections are shown in Figure 3.10. The repetition allows reusing genotype elements multiple times, which helps with evolving very interesting structures and control systems. The disadvantage of such solution from the evolutionary point of view is that repetition



(a) (b)

Figure 3.10: Examples of neuron repetitions with connections. The first one is `<X>N[0:1]#5<>`, and it sets the first neuron as input for the others. The second one is `<X>N#5<[1:1]>`, and it forms a sequence.

may provide dramatic changes in the final phenotype.

3.5.1 Genetic operators

Mutations

Mutations and crossover are performed on the tree formed from *f4* genotype. The available mutations are:

- *Addition of cell division* with simple child node, neural connection, sigmoid neuron parameter, repetition or simple modifier,
- *Deletion of random node*,
- *Modification of random node*.

Crossover

Crossover in the *f4* encoding is performed by swapping subtrees from both parents, like in the genetic programming [KRV01].

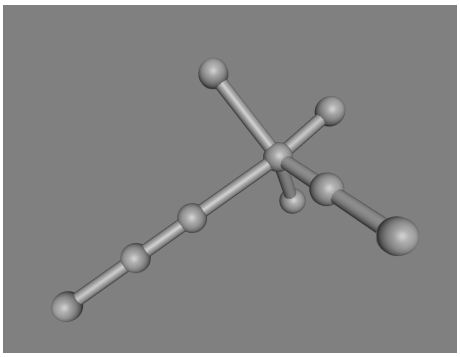
3.6 Similarity encoding (f2/fH)

This encoding may resemble developmental processes occurring on a chemical level – complex compounds connect with each other according to some dependencies, usually caused by matching connection points. The similarity encoding imitates those complex dependencies by n-dimensional vectors, called *handles*. In the beginning the genotype only describes the elements that will be used during the development. It can be interpreted as generation of compounds from the genotype sequence. Secondly, those elements form the final creature by matching them according to the similarity of their handles. More detailed explanation of body and brain development in the *fH* encoding will be presented in Section 4.2, along with comparison with the old approach, called *f2*.

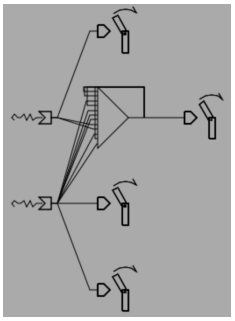
3.6.1 Syntax

The syntax for similarity encoding looks like for *f0* with some slight changes. First of all, the elements available in this representation are neurons, neural connections and sticks.

The first line of a genotype is an integer number representing the dimensionality of handle vectors n. Every element in the *fH* encoding has two vectors of length n. Those handles are



(a) Body of the creature



(b) Brain of the creature

Figure 3.11: Body and brain developed from example *fH* genotype, evolved during evolution of the fastest creature. The "leg" built from 2 sticks moves the creature by performing small jumps.

used during the development process to determine how all body and brain elements should be connected.

Following lines represent definitions for body and brain elements. Every line should begin with the "*CLASSID*:", where *CLASSID* can be "j" for sticks, "n" for neurons, and "c" for connections. The following $2n$ properties represent two vectors of an element. The default value for each of handle's element is equal to 0. After that, the remaining properties depend on the element's type, for example stick has length "l", density "dn", friction "fr" and other of the selected properties of parts and joints.

The simplest genotype for *fH* encoding is the definition of a single stick:

```
//H
3
j:
```

This is the equivalent to the following genotype:

```
//H
3
j:0,0,0,0,0,0
```

More complex example of a *fH* genotype is shown below. This is the result of maximization of the velocity of the agent. The visualization of the agent and its brain is shown in Figure 3.11. This agent evolved jumping movement.

```
//H
3
j:x2=-0.141, dn=0.372, fr=0.066, as=0.463, stif=0.834, rotstif=0.893, stam=0.171
j:-0.081, -0.166, y2=-0.038, dn=1.05, fr=0.947, as=0.073, rotstif=0.84, stam=0.256, l=1.497
j:-0.502, -0.812, 0.090, -0.087, -0.302, 0.851, fr=0.343, ing=0.277, as=0.047, stif=0.987, rotstif
    ↪ =0.912, stam=0.202, l=1.174
j:x1=-0.018, y2=0.375, fr=0.116, as=0.416, stif=0.99, rotstif=0.931, stam=0.329, l=1.082
j:-0.081, -0.166, y0=-0.009, y2=-0.038, fr=0.567, ing=0.149, stam=0.368
j:-0.502, -0.811, 0.332, -0.253, -0.302, 0.521, ing=0.313, as=0.01, stif=0.987, rotstif=0.574, stam
    ↪ =0.187, l=0.98
j:-0.412, -0.664, 0.152, 0.039, -0.347, 0.851, ing=0.442, l=1.51
j:-0.863, 0.978, -0.121, 0.602, -0.784, 0.143, dn=1.586, fr=0.018, ing=0.183, stif=0.939, rotstif
    ↪ =0.81, l=1.089
n:-0.52, 0.273, 0.135, 0.784, -0.861, -0.383, d="|: p=0.324"
n:-0.961, -0.638, 0.142, 0.62, 0.792, -0.566, d="T: r=0.965"
n:0.196, 0.0580, 0.8, 0.26, 0.761, -0.263, d="|: p=0.101"
n:0.818, -0.006, -0.976, 0.333, 0.746, 0.855, d="N: s=-0.208"
n:-0.815, 0.938, 0.304, 0.794, 0.454, 0.609, d="T: r=0.904"
n:0.335, 0.372, -0.384, 0.230, -0.723, 0.008, d="|: p=0.344"
n:0.460, -0.159, -0.531, -0.801, -0.089, -0.82, d="|: r=0.864"
c:-0.101, -0.027, -0.317, -0.543, -0.777, 0.238, -2.112
c:-0.411, -0.630, 0.735, -0.886, -0.058, 0.588, 4.888
c:0.959, -0.114, -0.824, 0.382, -0.780, 0.566, 4.692
c:0.538, -0.025, 0.946, 0.436, 0.899, 0.386, 0.71
c:-0.278, 0.08, 0.845, -0.282, 0.848, -0.395, 12.367
c:-0.996, -0.561, 0.366, 0.812, 0.879, 0.576, 1.001
c:0.126, -0.686, -0.716, -0.803, 0.820, 0.302, -0.439
c:-0.342, 0.516, 0.170, -0.423, 0.132, -0.337, 0.052
c:0.363, 0.123, -0.964, -0.740, 0.884, -0.291, 6.481
```

```
c:-0.315, 0.123, -0.151, -0.648, -0.824, 0.827, -0.963
c:-0.723, -0.059, -0.375, -0.637, 0.199, -0.425, -1.273
c:0.822, 0.84, 0.811, -0.171, -0.499, -0.056, 5.816
c:0.619, -0.183, 0.103, 0.414, 0.776, -0.562, -1.547
c:-0.601, -0.407, -0.204, -0.706, 0.366, 0.631, -0.672
c:-0.557, 0.134, 0.442, -0.080, -0.404, -0.062, 1.999
c:-0.418, 0.753, -0.149, 0.711, -0.344, -0.126, -0.016
```

3.6.2 Genetic operators

Mutations

Mutations for similarity encoding consist of addition, deletion, and modification of handle's vectors or element property. *The addition* in fH creates a new random element with uniformly distributed vector values for both handles. If the generated element defines neuron, the random neuron class will be selected for this element. Other properties of every element are initialized with default values during addition.

The deletion removes the randomly selected element defined in the genotype. If the deleted element was neuron, the definitions of its connections will be left. This will cause the rearrangement of neural connections, according to the vectors of handles. If there are no neurons to connect to, the connections will be left as "junk code". Thanks to the indirect definition of neural connection's beginning and end, those connection definitions may become useful when new neurons will appear.

Both mutation of handle values and element properties in the *fH* encoding are designed to provide minimal numerical change. This is performed by selecting random value with Gaussian distribution centred at the previous value. The difference of mutating handle values for change and addition comes from an idea, in which mutation of value should not largely affect the final phenotype, and using uniform distribution for simple change of value could move the current element, like stick, to completely another place in the phenotype, while Gaussian distribution gives slower, but more "controllable" changes of the phenotype.

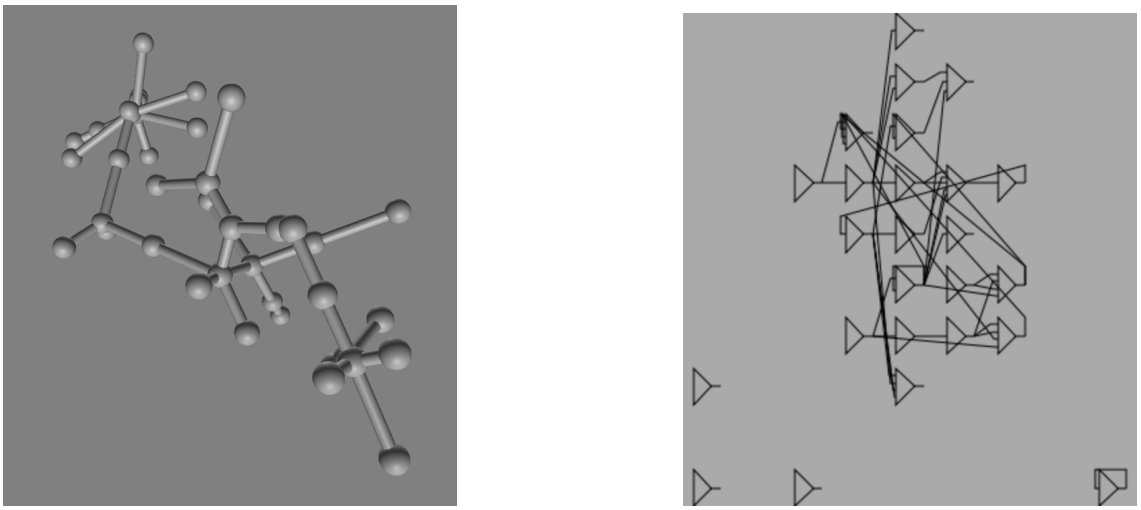
The mutation operator is selected with use of the roulette wheel method. If the genotype consists of only one stick, the deletion is skipped during method selection.

Crossover

Crossover in the similarity encoding takes two parents and evenly distributes their elements to two children. If one child after the distribution of sticks does not have any of them, the remaining elements from both parents will be passed to the second child. The crossover returns at least one child.

3.7 Biological encoding (f3/fB)

The biological encoding is implemented on top of the similarity encoding presented in Section 3.6. It is inspired by DNA genotypes that represent every creature as a sequence of letters from an alphabet $\{A, C, G, T\}$. Those letters stand for *adenine*, *cytosine*, *guanine* and *thymine*, which are names of the nucleobases of nucleotides forming the final creature. During the development and lifetime of the cell the information in DNA is used to form new chemical compounds that are needed during the cell forming and signal processing. The portions of data determining single protein are delimited by *START* and *STOP* codons. This portion of the information, called *a gene*, is taken from the original sequence and processed sequentially by ribosomes in order to form a protein. Those proteins are later involved in the message passing, cell construction, transportation and other tasks. This concept of information storing is presented in biological encoding in Framsticks.



(a) Body of the creature

(b) Brain of the creature

Figure 3.12: Body and brain developed from example *fB* genotype, created from "Lorem ipsum dolor sit amet..." text.

The *f3* encoding is represented by a single integer value in the beginning, and a sequence of lower-case letters from "a" to "z". The *f3* encoding cannot be directly converted to the phenotype – it needs first to be converted to the similarity encoding. The integer in the *f3* genotype represents the number of dimensions of element's handles. The following letters are used to form final *f2* genotype. The sequence can be subdivided into genes. Each gene always starts with an "aa" codon and ends with a "zz" stopping codon or with the end of the sequence, when there was no STOP codon.

The first letter in a gene after "aa" codon is used to determine what kind of *f2* element does the gene represent. The letters from "a" to "i" represent the stick, from "j" to "p" the neuron, and the remaining letters represent neural connections. Letters after the type letter are mapped to values of handles and properties. The mapping from characters to real values of handles, properties, and methods considered for coding neuron classes in this encoding will be presented in Section 4.3.

Just like in the DNA genotype, every letter in *f3* encoding can be used in different contexts, because this encoding allows the gene overlapping. Such a situation happens when there are multiple starting codons sharing same stopping codons.

3.7.1 Syntax

The genotype in *f3* encoding consists of the lower-case letters. The simplest creature that can be created in this encoding has following genotype:

```
//B
3
aabzz
```

This genotype creates one stick with default values of properties. The letter "b" guarantees that only one stick will be created. The genotype "aaazz" has two starting codons – for the first gene the "a" letter determines that gene represents a stick. The second gene does not have determining letter – in such case stick with default values is created.

The "toy" example of creature built from "Lorem ipsum dolor sit amet" for the *fB* encoding (reimplemented version of the *f3* encoding) is shown in Figure 3.12. This example was presented to show that any sequence of lower-case letters can be used to form some more or less complicated

phenotype, and there are no syntax restrictions, except for number of dimensions in the beginning of the text and a starting codon.

3.7.2 Genetic operators

Mutations

The available mutations for biological encoding are similar to one present in biology. The list of possible changes to the *f3* genotype are following:

- *Insertion* – places randomly one random letter, the distribution of placement and the character selection is uniform,
- *Deletion* – removes random character from the genotype,
- *Substitution* – changes random letter to another randomly selected letter,
- *Duplication* – creates a copy of a random gene sequence and appends it to the beginning of the genotype,
- *Translocation* – swaps two random genes sequences in the genotype.

Crossover

There are two possible types of exchange of genes – a *crossover* and a *horizontal gene transfer*. The crossover operator distributes randomly genes of parents between two children. As in the similarity encoding, at least one offspring is produced during this operation. If there were overlapping genes, the overlap is also randomly assigned to one of the children. This means that the same gene can be added multiple times to the same child.

The horizontal gene transfer in biology can be seen especially during the evolution of bacteria, in which an antibiotic resistance information is passed from one organism to another with use of mechanisms, such as passing plasmids during conjugation. In the biological encoding in Framsticks, the horizontal gene transfer means that the first child gets full genotype of the first parent and a random gene of the second parent appended to the beginning of the child's genotype. The second child gets full genotype of the second parent with a random gene from the first parent appended to the beginning of the final genotype.

3.8 Generative encoding (f8)

The next encoding available in Framsticks is the generative encoding, called *f8*. It was presented in [Waj09], and was inspired by the work of Hornby and Pollack on generative encodings presented in [HP01, HP02]. The main idea of the generative encoding is based on the modification of the parametric L-Systems. The *f8* genotype is converted to the *f1* genotype.

3.8.1 Syntax

The syntax of the *f8* encoding is following [Waj09]:

```
NUMOFITERATIONS
INITIAL_PARAMETERS
---
AXIOM
PRODUCTIONS
```

"NUMOFITERATIONS" is the number of iterations to perform for the L-System. The second line of the genotype, called "INITIAL_PARAMETERS", is the list representing initial values of axiom word parameters. Every parameter is defined in a separated line as $nI = V$, where I is an integer representing index of the parameter, and V is the value. The "AXIOM" of the genotype is the beginning word, from which the L-System is developed. "PRODUCTIONS" are a list of production rules defined in a form [Waj09]:

```
PREDNAME(PARAMETERS): CONDITIONS1 | SUCCESSORS1 : CONDITIONS2 | SUCCESSORS2 : ...
```

"PREDNAME" is the name of the word that should be replaced by a rule. Only words that satisfy "CONDITIONS" are replaced by "SUCCESSORS". "CONDITIONS |" are optional – there can be rules that are deployed for words regardless to parameters. Conditions are comma-separated comparisons of the parameter values – this is a conjunctive form. There can be more than one rule for one word – in this case other conditions with their successors are separated by a comma and listed in one line.

Most of the body and brain syntax is derived from the *f1* encoding – this includes X, modifiers listed in Table 3.1 and neuron definitions. The differences and additional elements for *f8* encoding in comparison with *f1* encoding are following:

- [is the equivalent of (,
-] is the equivalent of),
- ∧ is the equivalent of a comma,
- \[is the equivalent of [,
- \] is the equivalent of].

The example genotype, taken from [Waj09] is following:

```
3
n0=1.0
n1=2.0
---
P0
P0(n0,n1):n0<5|X:P1(n0+n1):n1<4|cX
P1(n0):P0(n0+1,n0+2)
```

In this example there are two words – P0 with two arguments, and P1 with one argument. P0 has two possible successors that are deployed with respect to the conditions. P1 word has one non-conditional rule that will always be deployed. The final *f1* sequence for this genotype will be the following:

```
0. P0(1, 2)
1. X P1(3) cX
2. X P0(4,5) cX
3. X X P1(9) cX
Final f1: XXcX
```

All rules for word that are satisfied are deployed in order presented in the genotype. That is why in the first iteration all possible rules are deployed [Waj09]. The final result neglects L-System words leaving only *f1* genotype.

It is possible to define the neurons in the same way as in the *f1* encoding (with respect to change of writing squared brackets mentioned above).

3.8.2 Genetic operators

Mutations

Available mutations for the *f8* encoding are following [Waj09]:

- *Change of number of the iterations,*
- *Change of an axiom,*
- *Change of a parameter of word in the successor of rule,*
- *Addition of a word (command),*
- *Deletion of a word,*
- *Encapsulation* – extracts some fragment of the L-System and replaces it with word, for which the created a production will insert the extracted fragment (and possibly enable the L-System to reuse this fragment in other places),
- *Addition of a condition,*
- *Deletion of a condition,*
- *Modification of a condition sign or operator,*
- *Addition of a parameter to word,*
- *Deletion of a parameter of word,*
- *Addition of a successor with condition.*

While *f8* encoding enables defining own L-Systems with neurons and connections, the mutation operators do not add or modify neurons in this encoding [Waj09]. This means that this encoding is unable to evolve control systems.

Crossover

The crossover for *f8* encoding creates initially the first child with the full genotype of the first parent. After this, random rules from the second parent are selected. If the word name for the rule from the second parent matches the word name of some rule in the first parent, then successors are swapped for those rules. The selection and successor's swap for the first child are repeated until all rules from the second parent are used, or the number of selected rules is going to exceed the one-quarter of count of the first parent rules (selection is performed at least once) [Waj09]. The complementary procedure goes with the second child.

During such process some conflicts with words and parameters may occur. When there are conditions that are using non-existing parameter, then this parameter is replaced by a random available parameter. The same procedure goes when there are not available parameters in rule's successor. If the selected rule for one of the children has the call to the word that does not exist for this child, then such word will be removed.

3.9 Messy encoding (f7)

This encoding, called also *f7* encoding represents the idea of a "hashing function" [Kom12]. The genotype consists only of upper-case letters. It can be divided into four sections, representing parts, joints, neurons, and connections. Each section contains labels and parameters for elements.

3.9.1 Syntax

The first letter in the section is called *section tag*. The second appearance of this letter ends the section. There are up to 4 sections – the letters after the end of the last section are treated as "junk code". The sections describe parts, joints, neurons and their connections, respectively.

Within the part section, the genotype is divided into groups of five letters. In the minimal approach, four groups form single part definition. The first group represents *part label*, which is used later for connecting parts to joints. The remaining groups are the values of properties coded in base-26 numeral system, and they represent coordinates. The section tag letters are only considered within label definition of a part, otherwise they are ignored.

In the minimal approach the joint section consists only of labels of parts defined in the previous section. The procedure of assigning the joints uses Breadth First Search graph traversing algorithm. The part represented by the current label (*start*) is the starting point of all joints until its label reappears (*stop*). The parts with labels between *start* and *stop* act as ending points for joints. After reaching the *stop* label, the part with label after *start* label becomes the new starting point for joints.

The extended approach involves coding of other properties of parts and joints, like density or stiffness.

Neurons in the *f7* encoding are described by three tokens – a label, a neuron type and an attachment type. The neuron type is converted into a decimal number and mapped to the index of one of the available neurons. If the neuron type token matches label token, the default sigmoid neuron is created. Attachment type token points to the part, joint or one of them according to the neuron attachment specification. If there is no requirement of attaching neuron to part or joint, then the token is ignored.

Connections are described by three tokens, representing a parent neuron, an input neuron and a connection weight. The first token after conversion to a number points to an element in list of neurons with free outputs, and the second token points to an element in list of neurons with free inputs.

The simplest genotype for "f7" encoding, representing a single stick, is the following:

```
ZAAAAAAAAAAAAAAAAAAAABBBBBAABMMAAAAAAAAAAZZAAAAAAAAAABAAAAAAAAABZ
```

3.9.2 Genetic operators

The mutations available in the messy encoding operate on simple character replacements:

- *Addition of a character*,
- *Substitution of a character*,
- *Deletion of a character*,
- *Addition of a gene* – random addition of five characters in sequence,
- *Change of a gene* – changes all five consecutive characters from random character to new characters,
- *Deletion of a gene* – removes five consecutive characters from genotype.

Those mutations are *context-free* – they do not analyze the placement of sections nor gene alignment.

The crossover is performed as standard two-point crossover.

3.10 Other encodings

3.10.1 Turtle3D-ortho encoding (f9)

The simplest encoding for body development is the *f9* encoding, in which the 3D structure is drawn by a movement of a 3D "turtle" (as in Logo [PL96]) based on a simple list of commands written in the genotype [PL96, Kom12].

This simple encoding consists of six letters – *up U, down D, left L, right R, forth F and back B.* The letter says in which direction the turtle forming the phenotype should move. During the movement, it creates sticks with fixed length. Genotype has no modifiers, it does not implement neural network development.

The *mutation* enables addition, deletion, or substitution of single letter, the *crossover* is two-point [Kom12].

3.10.2 Numerical encoding (fn)

This encoding stores only a vector of real numbers. The mutation operator changes a random element of the vector to another value selected with *Gaussian distribution*, where the current value of the element acts as the mean of Gaussian distribution, and standard deviation is given as a parameter of the experiment, along with value limits. In another available mutation mode, all values are mutated at once [KU].

The *crossover* for this representation is calculated as a linear mix of parents' vectors, where 50% inheritance percentage returns vector that is the mean of two input vectors, 80% means that i-th child has 80% of the i-th parent values. In another available crossover mode, the amount of mixing is randomly selected on each crossover operation.

This representation is used for *numerical optimization* [KU18, KM18].

3.10.3 Foraminifera 10-parameter encoding (fF)

This encoding is designed for the evolution of *foraminifera* – simple unicellular organisms that usually form tests – small external shells. One of the interesting aspects of studying foraminifera is their diversity in the morphology. Secondly, the analysis of foraminifera's fossils helps in gaining knowledge about such aspects as climate changes on the Earth.

Foraminifera creatures are described by 10 parameters [KMTT16]:

- N – number of the chambers,
- R_x, R_y, R_z – radius of the initial chamber,
- K_x – scaling factor on OX axis,
- K_y – scaling factor on OY axis,
- K_z – scaling factor on OZ axis,
- TF – translation factor,
- $\Delta\phi$ – deflection angle,
- $\Delta\beta$ – rotation angle.

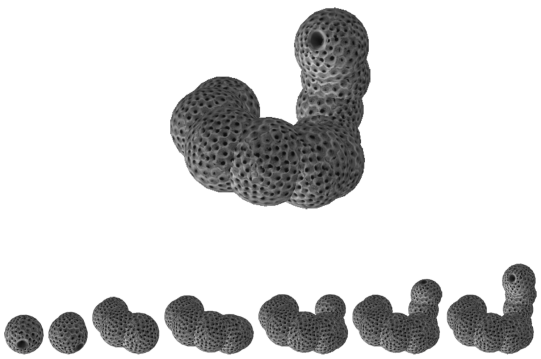


Figure 3.13: Example of fF encoding creature with its development [KMTT16]. The dark spot is the aperture.

The example creature with genotype:

$$N = 10, K_x = 1, K_y = 1, K_z = 1, TF = -0.02, \Delta\phi = 0.64, \Delta\beta = 0.72$$

along with its development is shown in Figure 3.13. *Mutation* and *crossover* operators are similar to the ones used in the *fn* encoding.

Chapter 4

Updating selected genetic encodings

This chapter will focus on presenting the aspects that were improved or reimplemented from scratch for a few of the existing encodings. A description of a new genetic representation called the *L-System encoding, fL* will be also included, along with its syntax and genetic operators.

4.1 Development of the f4 encoding

4.1.1 Improvements in the f4 encoding

The disadvantage of the previous version of the *f4* encoding was *the lack of support for custom neuron classes*. The only supported neuron classes were sigmoid neuron (N), bending muscle (|) and rotating muscle (@).

First of all, apart from the presented neuron class definitions in Section 3.5, a new definition of neuron class is introduced:

```
N:CLASSNAME
```

CLASSNAME can be the name of any available neuron class. This creates a single neuron with the desired class. It is worth noticing that the results of the old muscle semantics (*N@* and *N|*) are different from the results of using the muscles with the new definition (*N:@* and *N:|*). The old semantics creates a sigmoid neuron that is connected to the input of a given muscle. The new semantics creates a new muscle only.

The second change in syntax was made in the definition of the neural connection with embedded neuron class name (*[SENSOR, WEIGHT]*). In the previous version of this encoding only four classes were allowed: gyroscope (*G*), touch sensor (*T*), smell sensor (*S*) and a constant neuron (***). The developed version allows using any kind of sensor that is available for the current experiment.

In the end, the genetic operators in *f4* encoding were modified to support the new syntax and use all available neuron classes, instead of using only previously mentioned neuron classes during evolution.

4.1.2 Properties of the developmental encoding

One of the main advantages of the *f4* encoding is *the ability to reuse developmental patterns*. This allows the encoding to create complex structures with more ease than any other previously mentioned representations (except for *f8*). Figure 4.1 shows how a short *f4* genotype can create a complex phenotype, visualized in Figure 4.2. The highlighted elements are the example of *pleiotropy* in the *f4* encoding – separate elements in the phenotype are coded by a continuous substring in the genotype.

```
                                                  rr<X>#2<,<X>RR<<,X>X>X>
                                      p: p:1.0  : rr<X>..................
                                         p:2.0  : rr<..#.<,<X>...........
                                         p:3.0  : rr<..#.<,<X>..........>
p:1.9999963267949, 0.000196326793634088, -0.9 : rr<..#.<,<..RR<<,X>....
                                  99999980721149  : rr<..#.<,<..RR<<,X>....
                                         p:4.0  : rr<..#.<,<X>..........>
p:2.9999963267949, 0.000196326793634088, -0.9 : rr<..#.<,<..RR<<,X>...>
                                  99999980721149  : rr<..#.<,<..RR<<,X>...>
p:2.00019265360202, -0.999803635376417, -1.00 : rr<..#.<,<..RR<......X>
                                    019263502712  : rr<..#.<,<..RR<......X>
p:4.49999787927255, 0.000170024231114163, -0. : rr<..#.<,<..RR<<,X>...>
                                  866026611493387  : rr<..#.<,<..RR<<,X>...>
p:1.99980000001476, 1.00019628752139, -0.9997 : rr<..#.<,<..RR<<...X>..
                                    99980005117  : rr<..#.<,<..RR<<...X>..
p:3.00019265360202, -0.999803635376417, -1.00 : rr<..#.<,<..RR<......X>
019263502712 p:4.5001660669089, -0.9998299571 : rr<..#.<,<..RR<......X>
                     14487, -0.866121594211299  : rr<..#.<,<..RR<......X>
p:2.99980000001476, 1.00019628752139, -0.9997 : rr<..#.<,<..RR<<...X>.>
99980005117 p:4.49982601844667, 1.00017000432 : rr<..#.<,<..RR<<...X>.>
                      765, -0.865925266588737  : rr<..#.<,<..RR<<...X>.>
        j:0, 1, rx=-1.5706, dx=1.0, 0.0, 0.0  : rr<X>..................
                     j:1, 2, dx=1.0, 0.0, 0.0  : rr<..#.<,<X>...........
                     j:2, 3, dx=1.0, 0.0, 0.0  : rr<..#.<,<X>..........>
j:2, 4, rx=1.5706, rz=1.5708, dx=1.0, 0.0, 0. : rr<..#.<,<..RR<<,X>....
                                           0  : rr<..#.<,<..RR<<,X>....
                     j:3, 5, dx=1.0, 0.0, 0.0  : rr<..#.<,<X>..........>
j:3, 6, rx=1.5706, rz=1.5708, dx=1.0, 0.0, 0. : rr<..#.<,<..RR<<,X>...>
                                           0  : rr<..#.<,<..RR<<,X>...>
        j:4, 7, rz=-1.5708, dx=1.0, 0.0, 0.0  : rr<..#.<,<..RR<......X>
j:5, 8, rx=1.5706, rz=1.0472, dx=1.0, 0.0, 0. : rr<..#.<,<..RR<<,X>...>
                                           0  : rr<..#.<,<..RR<<,X>...>
         j:4, 9, rz=1.5708, dx=1.0, 0.0, 0.0  : rr<..#.<,<..RR<<...X>..
j:6, 10, rz=-1.5708, dx=1.0, 0.0, 0.0 j:8, 11 : rr<..#.<,<..RR<......X>
              , rz=-1.5708, dx=1.0, 0.0, 0.0  : rr<..#.<,<..RR<......X>
j:6, 12, rz=1.5708, dx=1.0, 0.0, 0.0 j:8, 13, : rr<..#.<,<..RR<<...X>.>
                rz=1.5708, dx=1.0, 0.0, 0.0  : rr<..#.<,<..RR<<...X>.>
```

Figure 4.1: The mapping for the example *f4* genotype. The rows are representing the lines of the *f0* genotype, and the colums represent the letters of the *f4* genotype. If the letter was not significant for the particular element of the *f0* genotype, then it is replaced by a dot.

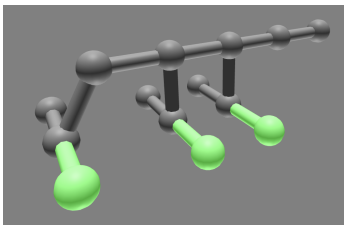


Figure 4.2: The example of the pleiotropy in the *f4* encoding for the example genotype `rr<X>#2<,<X>RR<<,X>X>X>`.

```
                                                  rr<<<X>rr<<X><<rrX>X>X>X>X
                                      p: p:1.0  : rr<<<X>..................
p:0.999996326794897, -0.000196326793634088, 0 : rr<..X>..................
                            .999999980721149  : rr<..X>..................
                                      p:2.0  : rr<<.....................X
p:0.999996326794897, 0.000196326793634088, -0 : rr<<<..rr<<X>.............
                            .999999980721149  : rr<<<..rr<<X>.............
p:1.00016451443125, 0.866321084577449, -1.499 : rr<<<..rr<.............X>.
                              82782675101  : rr<<<..rr<.............X>.
p:0.999824465969022, -0.865732105029258, -1.5 : rr<<<..rr<<..<<rrX>.......
                              0016787395761  : rr<<<..rr<<..<<rrX>.......
p:0.133711957272388, -1.29843942028253, -1.75 : rr<<<..rr<<..<.......X>...
                              039477420122  : rr<<<..rr<<..<.......X>...
p:1.86576511456872, -1.29894954880248, -1.750 : rr<<<..rr<<..<<....X>.....
                               1067455109  : rr<<<..rr<<..<<....X>.....
        j:0, 1, rx=-1.5706, dx=1.0, 0.0, 0.0  : rr<<<X>..................
        j:1, 2, rz=-1.5708, dx=1.0, 0.0, 0.0  : rr<..X>..................
                     j:1, 3, dx=1.0, 0.0, 0.0  : rr<<.....................X
j:1, 4, rx=-1.5706, rz=1.5708, dx=1.0, 0.0, 0 : rr<<<..rr<<X>.............
                                         .0  : rr<<<..rr<<X>.............
        j:4, 5, rz=-1.0472, dx=1.0, 0.0, 0.0  : rr<<<..rr<.............X>.
j:4, 6, rx=-1.5706, rz=1.0472, dx=1.0, 0.0, 0 : rr<<<..rr<<..<<rrX>.......
                                       .0  : rr<<<..rr<<..<<rrX>.......
        j:6, 7, rz=-1.0472, dx=1.0, 0.0, 0.0  : rr<<<..rr<<..<.......X>...
         j:6, 8, rz=1.0472, dx=1.0, 0.0, 0.0  : rr<<<..rr<<..<<....X>.....
```

Figure 4.3: The example of mapping of the genotype with marked polygeny example (green color).

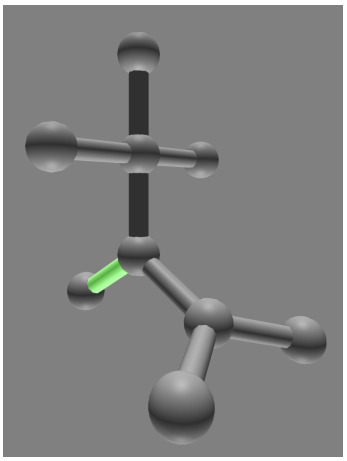


Figure 4.4: The example of polygeny – the underlined fragments of the genotype are responsible for the highlighted element in the phenotype: `/*4*/rr<<<X>rr<<X><<rrX>X>X>X>X`

The polygeny can be also encountered in this representation – the example of this phenomena is shown in Figure 4.4. The mapping representing the polygeny is in Figure 4.3.

This representation has the similar abilities as the *f1* encoding. The additional element is the ability to form loops. The representation in the beginning is less intuitive than the *f1* encoding, and requires more sophisticated operators. The operators, due to the possibly high re-usability of the code can cause bigger changes than they suppose to. This can result in less accurate exploration of the fitness landscape.

4.2 Reimplementation of the similarity encoding – the fH encoding

The second developed representation was the *similarity encoding*. Due to full reimplementation it is called now *fH* encoding. The syntax and operators of the *fH* representation were introduced earlier in Section 3.6. It has improved body construction and mutation operators.

4.2.1 The development of a creature

As it was stated in Section 3.6, the n-dimensional genotype of the *fH* encoding consists of sticks, neurons, and connections. Apart from the element-specific properties, every element has $2n$ real numbers within range $[-1, 1]$. Those are two n-dimensional vectors, describing the element's affinity with other elements.

Connection rules for elements

Every genotype can be represented as a set of handles H, which is formed from three disjoint subsets describing sticks S, neurons N, and connections C.

The main idea of the representation is to connect handles that are most similar to each other. That means that the distance between corresponding vectors of the two handles should be minimal. The distance used for *fH* encoding is the Euclidean distance for the n-dimensional vectors l and r:

$$dist(l, r) = \sqrt{\sum_{i=1}^{n} (l_i - r_i)^2}$$

The development of the creature starts with creating a list of pairs $P \subset S \times S$ of the elements from S that describe how all the sticks are connected to each other. The B set will contain all

sticks used in pairs to form the body. Both sets are initially empty. In the beginning the algorithm finds two sticks that have the minimal distance between the second vector of the first stick and the first vector of the second stick:

$$p = \underset{\substack{(x,y)\in S\times S \\ x\neq y}}{\mathbf{argmin}}\ dist(x_2, y_1)$$

where 2 denotes the second vector of the handle, and the 1 the first vector of the handle. The pair p is added as the beginning of the body to P, and chosen handles x, y are removed from the S set and added to the B set. The sticks remaining in the set S are added to the body in the order determined by the distance of their first vector to one of the second vectors available in the body. The next pair that is added to P is formed with a following formula:

$$p = \underset{(b,n)\in B\times S}{\mathbf{argmin}}\ dist(b_2, n_1)$$

The elements are selected from S and added to B until the first set is empty. The list P will be used later during body development.

The next aspect is the brain development. In this case, the neurons, sensors, and effectors need first to be attached to the body. When neuron needs to be attached to the joint, then it is attached to the joint that has minimal distance of its average vector of both vectors to the neuron's average of both vectors. In other words, the neuron n is attached to such the joint of the stick s that:

$$s = \underset{i\in B}{\mathbf{argmin}}\, dist\left(\frac{i_1 + i_2}{2}, \frac{n_1 + n_2}{2}\right)$$

When the neuron needs to be attached to the part, the method selects the stick for which one of the vectors is most similar to the average of two vectors of the neuron. If the first vector of the stick was the closest one, the first part of this stick will be selected. Otherwise, the second part will be selected.

In the end, the connections are the only thing left to determine. The neuron handle n^1 is the beginning of the connection c if its second vector is the closest one to the first vector of the connection, and the neuron handle n^2 is the end of the connection if its first vector is the closest one to the second vector of the connection.

$$n^1 = \underset{i\in N}{\mathbf{argmin}}\, dist(i_2, c_1)$$

$$n^2 = \underset{i\in N}{\mathbf{argmin}}\, dist(c_2, i_1)$$

In the end, all handles of the creature are connected to each other, and the connection between them is consistent.

Construction of a body

The *fH* representation describes the properties and vectors used for connecting all the body and brain elements. On the other hand, the body structure is created implicitly according to the pairs stored in P. It does not depend on vectors of handles.

The development of the creature starts at point $(0, 0, 0)$. Consecutive sticks are appended to the body according to the order in the list P. For each pair (x, y) the x stick is already added to the body, and the y stick starts with the second part of the x stick, and creates its second part at a distance of l (given in the property of stick) along the current direction. When there is only one stick coming from the parent, then it has the same direction as the parent.

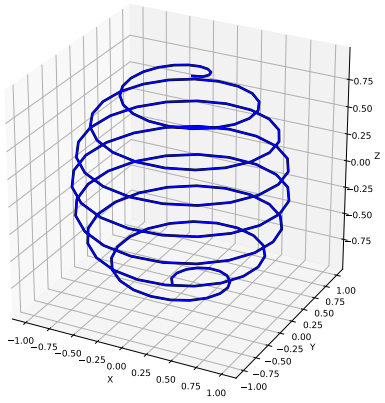


Figure 4.5: Example of a spiral created along sphere's surface.

The question is – how to determine the direction of a new stick, when multiple sticks come from the same part? The aim is to arrange the parts in the 3D space to preserve distances between them, according to length (l) property of the stick. In the previous version of this representation, called *f2*, the direction for branching sticks was determined with the use of the heuristic based on trigonometric formulas. The heuristic was meant to evenly distribute joints from a single part – up to some number of branches it managed with the problem, but later it created not evenly distributed directions. The other disadvantage of this heuristic method was the bias towards creating flat structures – for the low number of branches it performed branching mostly in 2D.

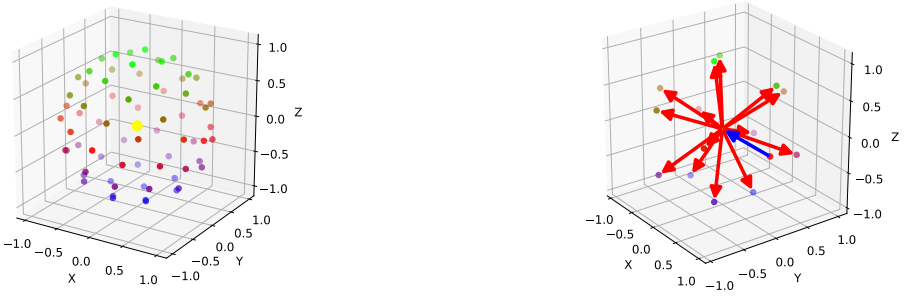
The problem in this case is to perform branching in such a way that all three dimensions are used, and the joints are evenly distributed. The problem can be generalized to *the problem of evenly distributing n points on a sphere.*

The *truly* even distribution of points on a sphere is possible only for some special cases, like $n = 2$, or when points can be represented as the vertices for the platonic solids inscribed in the sphere. It happens when n is equal to 4, 6, 8, 12, or 20. For the other values of n it is impossible to evenly distribute points on a sphere, but there can be some approximations that look like even distribution.

There are various algorithms designed for the problem of even distribution of points on the sphere [SK97]. The most popular algorithms for this problem are *iterative*. In general, each point is treated as a *charged particle* that is initially placed randomly on a sphere. Secondly, those particles "repel each other according to Coulomb's law" [SK97]. The algorithm, during each repetition, tends to maximize the distances between every point on the sphere. In the end, the iterative algorithm reach a state in which every point is distant from other points as far as possible. The disadvantage of this method is the *time-consuming* process of getting even distribution. Iterative methods are too slow for the task of branching sticks in the *fH* encoding.

The second method involves *geodesic subdivision* [Art15] – the algorithm starts with an octahedron, or icosahedron, and subdivides every side of the solid shape into 4 triangular faces. The generated points during subdivision are aligned to the sphere. This creates the results that are less satisfying than previously mentioned method.

The third and chosen method for the body development was introduced by Saff, Rakhmanov,



(a) Points generated by the method

(b) Directions of stick growth

Figure 4.6: Visualizations of even distribution of n points on a sphere.The yellow point in Figure 4.6a represents the center of the sphere. During branching, one of the points is used to align the rest of directions to the current direction of development (blue vector). It is represented as the beginning of blue vector.

and Zhou [Art15]. In this method, n points are evenly distributed on *the spiral created along the sphere's surface*, like in Figure 4.5. The good parametrization of the spiral allows getting sufficiently good "even" distribution of the points on a sphere [SK97]. The final formulas used in implementation were presented in [Art15]. Let n be the number of points to distribute, and let i be the index of the currently distributed point. This method works partially on spherical coordinates (r and θ) and Cartesian coordinates (z). The method calculates (x, y, z) coordinates with following equations:

$$z = 1 - (i + 0.5) \cdot \frac{2}{n}$$

$$r = \sqrt{1 - z^2}$$

$$\theta = \Phi \cdot i$$

$$x = r \cdot \cos(\theta)$$

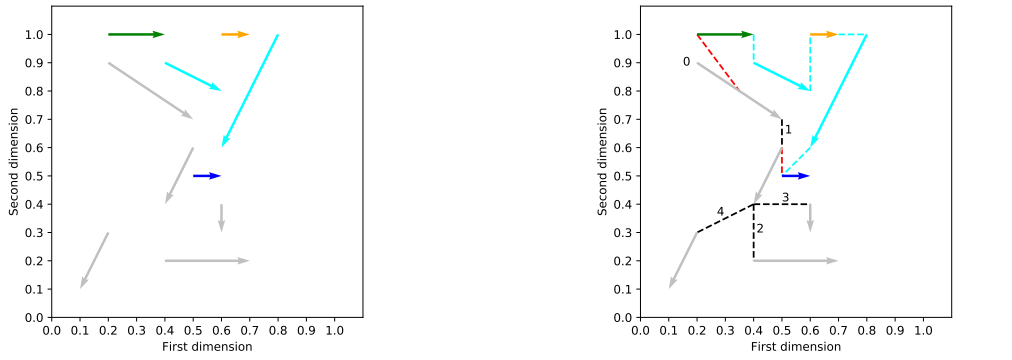
$$y = r \cdot \sin(\theta)$$

The advantages of this method in terms of even distribution of sticks during branching are the constant time of calculation single direction of the stick and satisfying three-dimensional results for any number of branches. The example visualization of distribution for 70 points is shown in Figure 4.6a.

For the n branches coming from a single stick $n + 1$ evenly distributed points are generated. Then, the first generated point, with an index 0, is used to calculate initial direction that is "aligned" to the direction of the parent stick with use of "aligning rotation matrix" R. The initial direction is represented as the blue vector in Figure 4.6b. The direction for the i-th branch is calculated as unit vector between center of the sphere and the $(i + 1)$-th point on the sphere's surface, multiplied by R matrix.

The values of properties of the part are the average of part-related properties in all definitions of sticks sharing this part. When there are no values provided for element properties, then the default values for those properties are assigned.

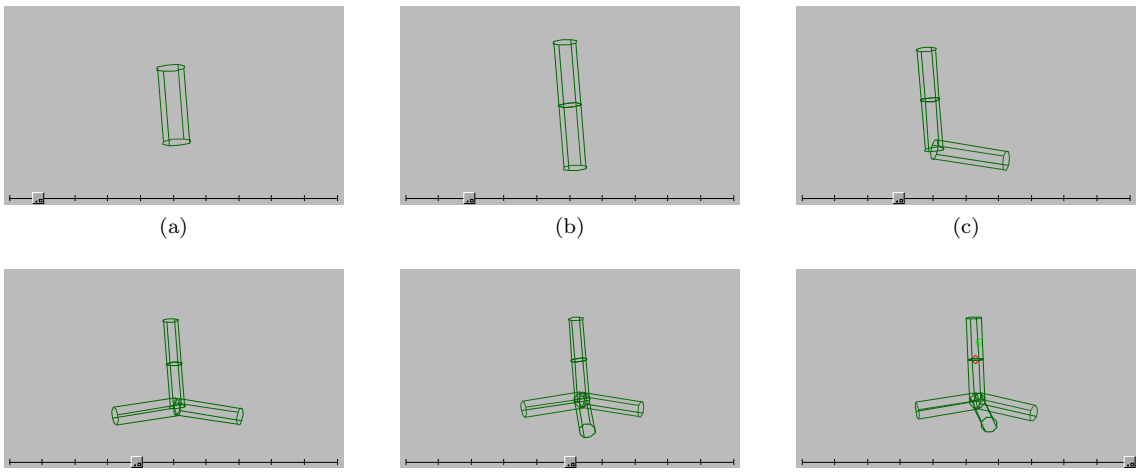
For example, consider the following genotype:



(a) Genes represented by vectors

(b) Connections created during development

Figure 4.7: The development process of the example *fH* genotype. Every element of the genotype is represented as a vector, where the beginning coordinates are the values of the first vector in the element, and the ending coordinates are the values of the second vector. "Silver" vectors represent sticks, a green vector represents gyroscope, a blue vector represents bending muscle, an orange vector represents sigmoid neuron, and cyan vectors represent neural connections. The similarity connections between elements are represented as dashed lines – black for sticks, red for attachments of neurons to sticks, and cyan for neural connections.



(a) (b) (c)

(d) (e) (f)

Figure 4.8: The development process of the creature with *fH* genotype.

```
//H
2
j:0.2,0.9,0.5,0.7
j:0.5,0.6,0.4,0.4
j:0.4,0.2,0.7,0.2,l=1.5
j:0.2,0.3,0.1,0.1,l=1.5
j:0.6,0.4,0.6,0.3,l=1.5
n:0.5,0.5,0.6,0.5,d="|"
n:0.2,1.0,0.4,1.0,d="G"
n:0.6,1.0,0.7,1.0,d="N"
c:0.4,0.9,0.6,0.8,w=1.0
c:0.8,1.0,0.6,0.6,w=1.0
```

The visualization of the "gene space" is presented in Figure 4.7a. In the beginning, all sticks are connected by similarity in the order represented by numbers in Figure 4.7b. This is the order of pairs in the P set. The development of the body, according to generated pairs, is presented in Figure 4.8. Secondly, neurons that require attachment to the body are assigned to joints or parts. In the end, the neurons are connected with given connections.

4.2.2 Changes in operators

The set of the available mutations did not change for the *fH* encoding in comparison to *f2*, but some aspects were improved. First of all, the previous version during modification was changing every property and values of vectors with uniform distribution. The modification in the current version changes single value with use of Gaussian distribution with mean value equal to the previous value of property or vector's element. What is more, the previous version of the encoding was mutating neuron classes and assigning random values to neuron's properties. The current version mutates one of previous values with use of Gaussian distribution.

4.2.3 Properties of the similarity encoding

The similarity encoding is mostly similar in syntax to the *f0* encoding. The main difference is the implicit connections of body and brain elements. When the syntax is valid, then the phenotype will be always valid. The major disadvantage of this encoding is a highly reduced subset of the phenotype space that can be explored with use of this representation – this representation does not allow alterations of the growth directions, it has strict branching rules.

Because this representation belongs to the direct encodings, the pleiotropy is not observable in the mappings. What can be observed is the polygeny – multiple definitions of sticks that are distant in the genotype can share the same part, and alter its properties.

4.3 Reimplementation of the biological encoding – the fB encoding

The biological encoding was reimplemented, as well as the similarity encoding. During reimplementation several aspects were reconsidered and changed – genetic operators, conversion of letters to values of neuron connection weights, and the coding of neuron classes.

4.3.1 Numerical conversion from biological to similarity encoding

As it was mentioned in Section 3.7, the conversion from biological to similarity encoding starts with determining gene sequences and their type. After this, the remaining letters in the gene are used for describing the values of elements of vectors and properties of the encoded element. The conversion from letters to values can be performed in several ways. The representation can use multiple letters or one letter. The approach with multiple letters could use them as a mantissa, and an exponent in 26-base system, or use them just as fixed-point representation of numbers. Such solutions would allow the representation to code values of properties with high precision. The problem with using such representations is that the mutations would have different strength for different letters, due to the position of the mutated letter in 26-base numeric system. This would make the representation even more unpredictable than it currently is, because of the insertions and deletions. In this case one-letter representations of values are the safest choice. The cost of such solution is the limited precision of values, but the mutation process is more stable.

In vectors' elements and properties of joints and parts the conversion should allow achieving all valid values with uniform probability. The letter c is converted to the property value within range $[p_{min}, p_{max}]$ with following formula:

$$v = (p_{max} - p_{min}) \cdot \frac{c}{25} + p_{min}$$

The different approach is performed with weights of connections. The range of available values is in general very large, but during evolution only values within small range, usually $[-5, 5]$, are

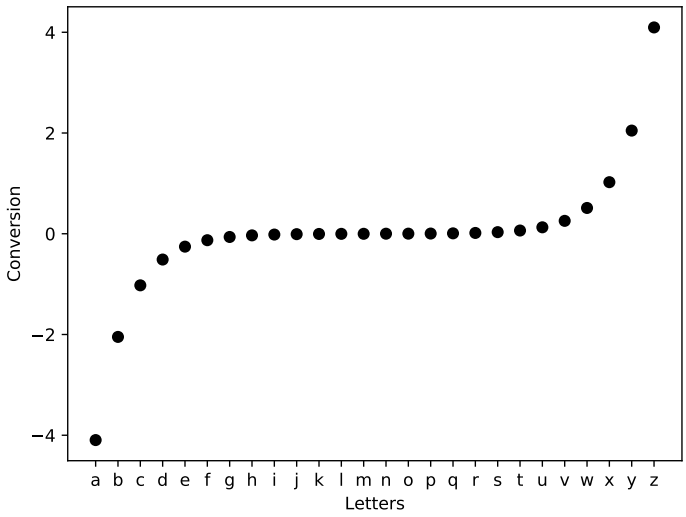


Figure 4.9: The result of mapping letters to weight values in *fB* encoding.

Features	C1(old)	C2(pad)	C3(hash)	C4(TF-IDF)	C5(C3 w. list)	C6(quotes)
F1 (explicit)	no	yes	no	no	yes	yes
F2 (robustness)	low	low	low/medium	high	low/medium	high
F3 (constant length)	yes	yes	yes	yes/no[a]	yes	no
F4 (correlation)	no	yes	yes[b]	yes	yes	yes
F5 (manageable)	yes	yes	no	yes/no[c]	no	yes
F6 (no conflicts)	no	yes	yes	yes/no[d]	yes	yes
F7 (returns classes)	low	medium	low	high	high	high
F8 ("N" probability)	no	yes	yes	yes	yes	yes
F9 (lower-case only)	yes	yes	yes	yes	yes	no

[a]TF-IDF method can use some predefined number of letters to perform "voting", or it can use all letters in variable-length gene.

[b]The correlation is implicit due to the use of hash function – the method is fully deterministic, but the nature of hashing is not known to the user.

[c]When there is a lot of neuron classes, then the list can be too large – this decreases the manageability.

[d]The ability to cope with similar neuron classes depends on available voting letters and the coding of upper-cases in genotype.

Table 4.1: Summary of different methods for neuron class definitions in the biological encoding.

used. The conversion from letter to value for weights is shown below (letters in quotes are ASCII letters):

$$v = \begin{cases} -0.001 \cdot 2^{\mathbf{m}-c} & : c \leq \mathbf{m} \\ 0.001 \cdot 2^{c-\mathbf{n}} & : c > \mathbf{m} \end{cases}$$

The bolded m and n are ASCII letters. This method limits the weight values to the range $[-4.096, 4.096]$. The values for each letter are presented in Figure 4.9.

4.3.2 Coding of neuron classes in the biological encoding

The previous concept of the biological encoding neglected the idea of using other symbols than lower-case letters. This led to the need of special coding of the neuron classes. In the original *f3* encoding the class name that began with the given letters was chosen. If there was no such class, the letter was mapped to a number k, and the k-th neuron class in the list of neurons was chosen. When none of above methods worked, the default sigmoid neuron was taken.

The disadvantage of this approach is that the genotype can have different interpretations, depending on the available neuron classes, and their order in the list.

Another difficulty in defining neuron classes in fB encoding is its chaotic nature of mutations. Insertion and deletion can completely change the interpretation of gene due to their obliviousness. This aspect of the biological encoding makes it hard to create a good coding of neuron class with use of only lower-case letters.

First of all, the coding should unambiguously determine the class. This means that regardless to the Framsticks platform and available neuron classes the genotype should always point to the same class. Secondly, mutations like insertion, deletion, or substitution should not invalidate immediately neuron class name, so it is impossible to assign it to the other class. Those two rules may conflict with each other – the direct interpretation of the neuron class name, for example when the insertion will cause change in the name from "sin" to "sdin", will be impossible. On the other hand, the use of the "fuzzy" definition of the neuron class may cause more conflicts between available names and genotype portability issues, because the same insertion may not affect phenotype in one Framsticks platform, but can change class definition in the other platform due to the differences in defined or active neuron classes. What is more, the neuron class names may differ only in letter capitalization.

The next feature that should be covered by the good class coding method is the full correlation between the genotype and phenotype. This means that the coded information must reflect the resulting creature and avoid mechanisms that have no explanation coming from the genotype. The next significant aspect is an easy management of the genotype. It should be as compact as possible, but some additional information, like list of used neuron classes, could be useful. The disadvantage of such solution is that some platform can have large database of neuron classes, which will result in a long list.

Last but not least, the probability of choosing each neuron class should be uniform, or at least specified in the experiment definition. There should be no situation, where one neuron class is favoured over other classes uncontrollably.

To sum up, the full list of desired features is listed below:

F1. *Explicitness* – decoding of a sequence should always return the same result, regardless to the context, like number of activated neurons, or available neuron class names.

F2. *Robustness* – the coding should not change its meaning after performing a single mutation, like substitution or insertion.

F3. *Constant length* – the coding should return the fixed-length output sequence from variable-length input sequence.

F4. *Full genotype-phenotype correlation* – the phenotypic result of the neuron class development should be easily derivable from the genotype.

F5. *Manageability of the genotype* – the coding should be compact and easy-to-use.

F6. *Conflict robustness* – the coding should cope with similar input sequences.

F7. *Ability of returning neuron classes* – the coding should guarantee high probability of selecting available neuron classes. High miss rate is unwanted for the development of creature.

F8. *Controllable probability of selecting classic "N" neuron* – the neuron class determining method should enable the control over selection rate of the classic neuron.

F9. *Preserving lower-case letter notation.*

Methods considered for the problem of coding a variable-length sequence representing neuron classes are listed below:

C1. *Original method* – first a name, which begins with given letter is chosen. If there is no such a class, the letter is converted to the number of class. When the selected neuron class is inactive, the default neuron "N" is chosen.

C2. *Padding method* – the neuron class definition has fixed number of letters, large enough to cover all lengths of names of the neuron classes. The name of class is written explicitly, and the remaining space of definition is filled with non-conflicting random letters or fixed letter. When neuron class with a given name does not exist, the default neuron class "N" is chosen.

C3. *Hashing method* – a fixed-length or variable-length strings from the genotype are mapped with a hashing function into neuron class id.

C4. *TF-IDF method* – the neuron class definition consists of remaining letters in the gene. For all letters in neuron class names the IDF measure is calculated. The count of each letter in a given gene is used to calculate TF-IDF measure for each neuron class. All neuron classes used for the genotype by its author need to be listed as the second line of the genotype – the first line would be the number of dimensions, the second line would be list of used neurons, and the last line would be the current genotype.

C5. *Hashing method with the list of neuron classes in the genotype* – the method uses concept presented in the hashing method, but it stores information about neuron classes that were considered during genotype creation.

C6. *Representing neuron classes within quotes* – this method neglects the idea of using only lowercase letters, and introduces random placement of neuron class definition with its properties within quotes in the genotype.

In Table 4.1 all above methods are analysed in order to check if they satisfy features that were mentioned earlier. The approach that was finally chosen for the representation is the last of the presented ones, where classical neuron class definitions, known from *f0* and *fH* encoding are inserted in the random places of the genotype.

In this approach, if the gene is defining the neuron class, then the conversion algorithm looks for neuron class definition within gene. If it finds the definition, it uses that definition. When there are multiple neurons within one gene, then for the i-th neuron the $min(i, num_{nclassdefs})$ neuron class definition is chosen, where $num_{nclassdefs}$ is the number of neuron class definitions within the current gene.

4.3.3 Changes in operators

In comparison to the previous biological encoding several changes were made in the genetic operators. The nature of most mutations, especially insertion and deletion, is destructive by definition. They can completely change the meaning of the gene, they can change starting codon, so the gene followed by this codon will disappear, and they can drastically change and shift the values assigned to the properties.

While there is no rescue from insertions and deletions, the substitution operator was slightly improved. It can only change the current letter to its nearest neighbours in the alphabet, for example *"c"* can be changed only to *"b"* or *"d"*. When the mutated letters are *"a"* or *"z"*, they

can be changed only to *"b"* or *"y"*, respectively – like in the reflect approach. This change in the operator makes it harder to change the meaning of the gene, when the first letter of the gene is mutated, and the values of properties do not change drastically.

The second change is the result of the new neuron class definition. First of all, the neuron class definition is treated as a single letter – when there are n letters in the genotype (excluding letters within neuron class definitions and quotes) and m neuron class definitions, where the length of the neuron class definition may vary, then algorithm selects random value i within range $[0, m+n)$. If the i-th element of the genotype is the neuron class definition, then for substitution the random property of the neuron is changed with use of Gaussian distribution, and for deletion the neuron definition is removed from the genotype. There is also a new operator added – *insertion of a neuron class definition*. It adds neuron class definition in random place of the genotype.

The last change in the *fB* operators was made for the crossover operator. The original crossover operator was taking genes from both parents and distributing them randomly across two children. The problem with this solution is following – when there are multiple overlapping genes, they are replicated many times in the children. That causes an extraordinary growth of the genotypes. For example for two parents:

```
1. aaabnwefaazzaagbaabha
2. aabcdfbaanfzzaakczzz
```

There are 5 genes for the first parent, and 3 for the second parent. One of the possible results of an old crossover is following:

```
1. aaabnwefaazzaabnwefaazzaazzaagbaabhaaabcdfbaanfzzaakczz
2. aabhaaanfzz
```

The nested genes are replicated – during the process of the evolution the length of the genotypes drastically increases. Such bloating slows down the evolution and makes the crossover unpredictable. To prevent this, the new crossover is introduced. It also distributes genes of the parents between two children, *but the genes nested in other genes are not repeated*. For the above parents there are only two not nested genes for the first parent, and also two for the second. Example of the children genotypes is following:

```
1. aaabnwefaazzaakczz
2. aabcdfbaanfzzaagbaabha
```

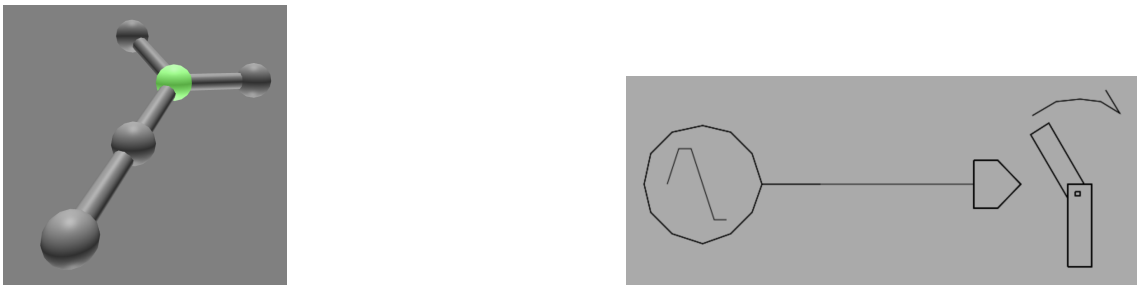
This is also, if not more, biologically plausible than the previous solution. The new crossover limits most of the growth of the genotype. Still, the duplication mutation and horizontal gene transfer can cause some growth of the genotype over time. They are more useful than the old crossover, and they do not cause uncontrollable bloating, but due to their nature it is still recommended to assign very low probabilities to those operators.

4.3.4 Properties of the biological encoding

This representation inherits all limitations of the similarity encoding – it explores very limited subset of possible solutions. Still, the observations of the biological encoding, especially its mappings to the phenotype, can be interesting. The same letters of this genotype can participate in describing different elements of the body, just like in *f4* encoding. What differs the *fB* encoding from the *f4* encoding is the fact that fragments of the *f4* genotype are always interpreted in the same manner, even during pleiotropy, while the interpretation of letters in the *fB* encoding is usually completely different for each gene. The gene overlapping allows compressing of the information.

The example genotype of the *fB* encoding is following:

```
//B
2
aabaacdefazzaamonohp"Sin"b"|"dldyzzzzaaqfgbcbzzaababcdzz
```



(a) Body of the crawler

(b) Brain of the crawler

Figure 4.10: The phenotype of the *fB* crawler. The highlighted part has the bending muscle.

```
                                                2 aabaacdefazzaamonohp"Sin"b"|"dldyzzzzaaqfgbcbzzaababcdzz
                                    p:dn=0.776  : ..................................................aababcdzz
p:1.0, dn=0.584, fr=0.48, ing=0.2333333333333 : ..aabaacdefazz.................................aababcdzz
                      33, as=0.166666666666667  : ..aabaacdefazz.................................aababcdzz
p:1.54960231195863, 0.675490294261524, -0.491 : .....aacdefazz.................................bzz.........
                           579252052213, dn=0.6  : .....aacdefazz.................................bzz.........
p:1.39587459737947, -0.742502054863492, 0.540 : ..aabaacdefazz.............................................
346186877577, dn=0.776, fr=0.64, ing=0.2, as= : ..aabaacdefazz.............................................
                                          0.0  : ..aabaacdefazz.............................................
p:2.09920462391725, 1.35098058852305, -0.9831 : ..................................................bzz.........
                                    58504104426  : ..................................................bzz.........
                                         j:0, 1  : ..................................................aababcdzz
                                         j:1, 2  : .....aacdefazz.....................................................
                                         j:1, 3  : ..aabaacdefazz.....................................................
                                         j:2, 4  : ..................................................bzz.........
                                       n:d=Sin   : ..............aamonohp"Sin"b"|"dldyzz.....................
                        n:j=1, d=| c:1, 0, -1.024  : ......................p"Sin"b"|"dldyzz..aaqfgbcbzz.........
```

Figure 4.11: The mapping of the crawler genotype. The polygeny is seen for several parts that were shared by sticks defined by two distant genes. The pleiotropy on the other hand is visible in two last lines of the mapping, were the same sequence was used for defining two neurons.

It shows multiple aspects and new syntax elements of the *fB* encoding. It creates a simple crawler with bending muscle connected to the sine generator, shown in Figure 4.10. The mapping for this genotype is shown in Figure 4.11. The two last lines of mapping show the definition of neurons. Due to the gene overlapping, the first gene takes the "*Sin*" definition, while the second one takes "|" definition. As it can be seen from the mapping, the polygeny occurs when there are separate genes that define sticks with the same part. The pleiotropy occurs here during gene overlapping.

This representation imitates to some simple degree the nature of the DNA representation – it is only expressed with letters, it has most of the DNA properties, and the implementation of DNA-like genetic operators is simple. Yet this approach leads to several major disadvantages – the destructive nature of genetic operators, hardly readable form, the unpredictable behaviour and slow evolution.

4.4 Implementation of the fL encoding

The *fL* encoding is a new proposed encoding for Framsticks platform. It is based on the concept of the *Timed DOL-Systems*, explained shortly in Section 2.6.3. The following section will introduce the syntax, operators and considered aspects of the generative encoding.

4.4.1 Syntax

The syntax of the *fL* encoding can be subdivided into three parts – the definitions of custom words, an "information" line with axiom and number of iterations, and the production rules. This order of writing definitions in this representation is required.

The custom words are defined one by one, with the following syntax:

```
w: name=WORDNAME, npar=NUMPARAMETERS
w: WORDNAME, NUMPARAMETERS
```

The *WORDNAME* is any alphanumeric string, while *NUMPARAMETERS* is the integer number representing number of available parameters for the current word. Every custom word can have up to 3 parameters. The custom words must not have duplicated names. They cannot also have the names of built-in words. The list of available built-in words will be presented later. The names of the properties are omittable.

After lines defining the custom words, the main "information" line needs to be written. The syntax of the "information" line is shown below:

```
i: axiom="WORDSEQ", time=NUMIT, numckp=NUMCHECKPOINTS, maxwords=MAXWORDS
```

This line can only occur only once in the genotype. It describes the starting conditions and basic development instructions for the L-Systems. The *NUMIT* is the real number describing the time of creature's development. In terms of classic L-Systems, when the real value describing time is not used, the integer values are only considered. The value *NUMCHECKPOINTS* describes how many checkpoints of the creature development should be captured per single iteration. This means that when $numckp = 5$, then apart from integer steps of development, the $(0.2, 0.4, 0.6, 0.8)$ fractions of time within the integer step will be captured during development.

The *WORDSEQ* requires more attention. This is a sequence of words written in the following way:

```
w1name(PARAMETERSLIST1)w2name(PARAMETERSLIST2)w3name(PARAMETERSLIST3)...
```

The `w1name, w2name, w3name...` are the names of available custom or built-in words. The *PARAMETERSLIST* in the axiom is the comma-separated list of mathematical expressions that are the parameters for the word. Round brackets are required for every word in sequence, even if there are no parameters for such word – the definition in such a situation is just "*wordname()*". If there are more parameters available for the word than given in brackets, the default value 0 is used for missing values. The more detailed explanation for parameters in *fL* encoding will be given in Section 4.4.2. The words in axiom appear in the initial state of the L-System. The axiom must have at least one word defining sticks – *S()*. Otherwise, the genotype will be invalidated.

The last property, called *maxwords* is given because of the bloating nature of the L-Systems. It is easy to achieve exponentially growing sequences for L-System rules, especially in the evolved genotypes. The *maxwords* property is the additional stopping condition, aside from number of iterations, that stops development when number of words in the current sequence of the iteration exceeds the value stored in *maxwords* parameter. If the value is equal to -1, this stopping condition is not used.

The remaining lines of the genotype describe production rules for L-Systems. They have a following syntax:

```
r:pred="PREDECESSORWORD", cond="CONDITION", succ="SUCCESSORSEQUENCE"
```

The *PREDECESSORWORD* is just the word name, without parameters. It accepts only the custom words, built-in words are not available for the predecessor. The *CONDITION* field is the logical mathematical evaluation that can be used to determine if the parameters of predecessor satisfy some conditions. This field can be empty – in this case this rule will be deployed for matching predecessor.

In the end, the *SUCCESSORSEQUENCE* has the same syntax as the *WORDSEQ* in the axiom, and represents the successors that will replace the predecessor upon deployment of production rule.

4.4.2 Mathematical expressions in L-System encoding

The *fL* encoding allows simple parametrization with mathematical evaluations. The available operators are $+, -, *, >, <, >=, <=, =, <>, \&, |$. The last three operators are *not equal*, *logical AND*, and *logical OR*. The round brackets are also allowed in this expressions. The example of simple mathematical formula is following:

```
3*(2+4) - 2
```

Such sequence written in the infix notation is tokenized and stored as a list of objects in a form of a *Reverse Polish Notation*. The detailed process of conversion from the infix notation to the RPN notation is presented in [Dij61]. For the above example, the RPN form will be following:

```
3 2 4 + * 2 -
```

The advantage of such solution is the ease of computing the formula result from such representation, and the possibility to reuse this list of objects efficiently for different values of variables.

The variables for mathematical evaluations are a time (t) and word parameters. Word parameters are represented in the infix sequence as $\$i$, where i is the 0-based index of word parameters. The time variable is represented as $\$t$. When a word is created in the current iteration, the time variable changes continuously from 0 to 1, according to the current time of the iteration. After this, the time variable for further iterations for the word has the value of 1. This means that the "continuous" development of the word is limited to the single iteration. Still, if one want to create the illusion of a longer-developing word, it can be done in the following way:

```
...
w:a(1)
...
r:pred="a", cond="$0 < 3", succ="a($0 + $t)"
```

In the above example, when the word can no longer be developed in the iteration, it is replaced by the same word with parameter equal to its previous value. In this case the condition is responsible for stopping further increase of the value in the word.

The number of available variables different from time depends on context – for the axiom there are no variables, for rules there are as many variables as parameters in the predecessor.

Such mathematical formulas are hard to control in terms of the returned values. When it is not a problem for words to some degree, the properties of body and brain require obeying some predefined limits. This leads to the problem of parametrization of built-in words that will be presented in Section 4.4.4.

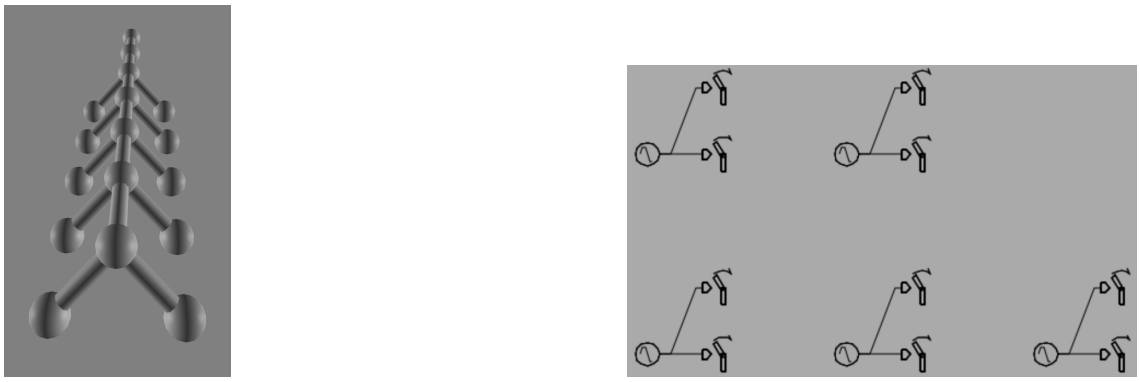
4.4.3 Built-in words in L-System encoding

In order to develop body and brain, several built-in words were introduced. They are responsible for managing the *turtle's* state during phenotype development on the level of the body and the brain. The turtle's state stores information about the current *position*, *direction*, and last defined *neuron* with inputs during its analysis of the genotype.

The first built-in word represents stick – *S*. It has eight properties:

- *For parts* – density (dn), friction (fr), ingestion (ing), assimilation (as),
- *For joints* – stiffness (stif), rotation stiffness (rotstif), stamina (stam),
- And *length of the stick* (l).

Every property can be represented by a mathematical formula with variables coming from predecessors. The words responsible for changing the direction of the turtle are:



(a) Body of the creature

(b) Brain of the creature

Figure 4.12: The example of phenotype with connections and attractors.

- *rotX(par)* – rotates the direction of the *turtle* movement along OX axis,
- *rotY(par)* – rotates the direction along OY axis,
- *rotZ(par)* – rotates the direction along OZ axis,

The next built-in word "N" represents neuron. It takes one text argument – neuron class definition with its parameters. The parameters for neurons cannot have mathematical formulas – only explicit values are possible. The example of the neuron definition is following:

```
...
i:axiom="S(l=0.5)N(\"N: fo=0.1, in=0.8\")"
...
```

The word "C" is used to define connection between neurons. It takes two arguments – *a weight* of the connection, and *an attractor*. The first one can be represented as mathematical formula. The second one is a single custom word with parameters that can be written as mathematical formula.

The aim of the attractor is to connect distant neurons by "labels" instead of a relative or an absolute indexes. The end of the connection is always the current neuron remembered in the state of the *turtle*. If there were no neurons before in the genotype, the connection cannot be established. If the attractor is not given, then the nearest neuron to the ending (parent) neuron of the connection is used as the beginning of the connection. The searching goes in both directions along the current genotype sequence.

When the attractor is given, the neuron that is the closest to the attractor is chosen. When attractor has given parameters, the corresponding word closest with its parameters is chosen. The closeness of parameters is calculated as Euclidean distance between the points created from values of parameters of the pattern attractor and matching words. The example of the genotype that uses attractor with parameters is shown below:

```
//L
w:a, 1
w:b, 1
i:axiom="S()b(0)", 5
r:pred=b, , succ="S()[rotY(2)rotZ(1.57)S()N(|)C(0.5,a($0+0.2))][rotY(-2)rotZ(1.57)S()N(|)C(-0.5,a($0
    ↪ +0.2))]a($0)N(Sin)b($0+1)"
```

It has one unconditional production rule that uses b word as a predecessor. This word has one parameter. In all connections the attractor word is the word a. The single iteration of this L-System creates one segment with two "legs". Each leg has the bending muscle attached. The connection for muscle has the attractor word $a(\$0 + 0.2)$. That means that for $b(1)$ this attractor

word will have the form of $a(1.2)$. Just before the sine generator for a single segment the word $a(\$0)$ appears. Again, for the $b(1)$ predecessor the production rule will generate $a(1)$ word. During connection creation the word closest to the current attractor will be always the word in the current segment. In this case only $a(1)$ is the most similar word in the sequence to the attractor $a(1.2)$. The resulting body and brain and be seen in Figure 4.12.

4.4.4 Parametrization of the built-in words for L-System

L-Systems allow altering numerical properties of the words, including built-in words for sticks and connections. The problem with such parametrization is that it is unaware of the ranges of elements' properties. It should use information about minimal and maximal values of each property to produce the fully valid creature and to perform mutations on these properties that will bring changes to the final creatures.

There are several ways of dealing with the problem of exceeding the range $[mn, mx]$ of some property. The considered solutions are listed below:

S1. Clamp the formula result to $[mn, mx]$ range, *absorb method* – this method allows only changes within a given range. When property value x is below minimal value the final property value will be set to mn, and if value is above maximal, then value will be equal to mx.

$$f(x) = \min(\max(x, mn), mx)$$

S2. *Use of the sigmoid function* – this solution takes property value x from the genotype as an argument for the sigmoid function scaled to $[mn, mx]$ range.

$$f(x) = mn + (mx - mn) \cdot \frac{1}{1 + e^{-x}}$$

S3. *Finding maximal and minimal value for the property in all its occurrences in the genotype and scaling current value of property with found values* – this solution finds all usages of the property in the current genotype iteration t and determines mn_t and mx_t values that would be used to scale the value of the property. If there was only one occurrence of the property, it would be set to its default value.

$$f(x) = \begin{cases} \frac{x - mn_t}{mx_t - mn_t} & : propcount \geq 2 \\ propdef & : propcount = 1 \end{cases}$$

S4. *Use of the "reflect" method* – if value x exceeds one of extremes $[0, 1]$ by a distance d, then the result is the value within range at a distance d from the extreme exceeded earlier. If the distance d exceeds given range, then the reflection must be repeated unless the resulting value is located within this range. The final value would be calculated by scaling reflect result to $[mn, mx]$ range.

$$reflect(x) = \begin{cases} |x| & : x < 0 \\ 1 - |x - 1| & : x > 1 \\ x & : 0 <= x <= 1 \end{cases}$$

$$f(x) = \frac{reflect(x) - mn}{mx - mn}$$

S5. *Use of the sine wave* – the final parameter value is calculated with following formula:

$$f(x) = mn + (mx - mn) \cdot \frac{1}{2}(\sin(\pi x - \frac{\pi}{2}) + 1)$$

Properties	S1 (absorb)	S2 (sigmoid)	S3 (global)	S4 (reflect)	S5 (sine)
P1 (all values)	yes	no[a]	yes/no	yes	yes
P2 (uniform probability)	no	no	no	yes	yes
P3 (noticeable changes)	no	yes	yes	yes	yes
P4 ("stable")	yes	yes	no	yes	yes
P5 (monotonic)	yes	yes	yes	no	no
P6 (context-free)	yes	yes	no	yes	yes
P7 ("time-friendly")	yes	yes	no	no	no

[a]Asymptotes prevent reaching extremes

Table 4.2: Summary of different methods for parameter values scaling to the desired range.

This method, unlike reflect approach, is smoother around extremes and does not require repeating. The formula ensures that if $x \in [0, 1]$, the values of property will increase smoothly from mn to mx.

There are other methods that could perform such scaling. This is important to use a method that will cope and adapt to evolving environment. The method chosen for "parameter scaling" should:

P1. be able to easily achieve all values within given range,

P2. have uniform probability of selecting all values within range,

P3. allow performing mutations that will provide noticeable changes,

P4. not lead to the increasing gap between extremes and other values,

P5. be monotonic,

P6. return the same scaled value for a given input regardless to context,

P7. allow normal growth with use of the time variable.

The summary showing how presented methods cover those properties is introduced in Table 4.2. After analysis of the properties, and due to the use of *Timed L-Systems*, the finally chosen approach is the use of sigmoid function. This decision was made mostly because the sigmoid function always provide some change, even very small, that can guide the evolution process, while the absorb method does not provide this information. The use of reflect and sine wave methods could give such information, but the direction of the evolution could be misguided due to cyclic nature of both approaches. What is more, the use of the sine wave or reflect method would give unnatural results for the *Timed L-Systems*.

4.4.5 Operators for the generative encoding

Due to the complex nature of the *L-Systems* it is possible to implement various types of mutations and crossovers. The main idea of following implementations was to always introduce possibly small change to the genotype. Such small change will likely make small change in the phenotype. For the illustrative examples the following genotype will be used:

```
//L
w:a, 1
w:b, 1
i:axiom="S()b(0)", 5
r:pred=b, , succ="S()[rotY(2)S()a(0)]b($0+1)"
```

The available mutation operators for *fL* encoding are:

- *Addition of a word to one of sequences* – this operator adds random available word to the axiom or one of the successors. It is possible to control the probability of the addition of word to the axiom or one of successors, and the probability of selecting built-in words, branches and defined words.

```
//L
w:a, 1
w:b, 1
i:axiom="S()b(0)", 5
r:pred=b, , succ="S()[rotY(2)S()a(0)]S()b($0+1)"
```

- *Addition of a new word definition* – adds new word definition with random number of parameters. The name of word is wi, where i is the first available integer number. If the number of defined words reached maximum value, the other operator is used.

```
//L
w:a, 1
w:b, 1
w:w0,2
i:axiom="S()b(0)", 5
r:pred=b, , succ="S()[rotY(2)S()a(0)]S()b($0+1)"
```

- *Addition of a new rule* – adds a new rule to the genotype. If there are no defined words except for the built-in words, then the *addition of a new word definition* is performed. Otherwise, the operator finds all defined words that does not have a production rule. If there are words with no definition, the random word is selected and the random rule with no condition and single word in the successor is created. If all words have defined rules, then the new rule is created with condition. If the existing rule for the selected word had condition, then the new rule takes the opposite condition. Otherwise, the random condition is created. All conditions created by evolution are represented in a conjunctive form.

```
//L
w:a, 1
w:b, 1
i:axiom="S()b(0)", 5
r:pred=b, , succ="S()[rotY(2)S()a(0)]S()b($0+1)"
r:pred=a, , succ="S()"
```

- *Mutation of a rule condition* – the mutation can create a condition for a rule without condition. It can also add or remove one of the parameter comparisons in the conjunctive form. This mutation is only applicable for production rules, where predecessors have at least one parameter.

```
//L
w:a, 1
w:b, 1
i:axiom="S()b(0)", 5
r:pred=b, cond="$0<0.55", succ="S()[rotY(2)S()a(0)]S()b($0+1)"
```

- *Modification of a random word* – the modification can change the formula of one of word's parameters, or it can change the name of the word to another defined word with same number of parameters.

```
//L
w:a, 1
w:b, 1
i:axiom="S()b(0)", 5
r:pred=b, , succ="S()[rotY(2)S()a(0)]S()b($0*$0+1)"
```

- *Modification of the L-System iteration* – slightly increases or decreases the development time of the *L-System* creature.

```
//L
w:a, 1
w:b, 1
i:axiom="S()b(0)", 6
r:pred=b, , succ="S()[rotY(2)S()a(0)]b($0+1)"
```

- *Deletion of a word* – removes a random word from the axiom or one of successors. If the axiom consists of only one word representing stick, and there are no successors, the addition of word is performed instead. If the word deleted the only word in the successor, then all rule is deleted. If the selected word is branch bracket, then this and its corresponding closing or opening bracket is removed, while the words within brackets are left.

```
//L
w:a, 1
w:b, 1
i:axiom="S()b(0)", 5
r:pred=b, , succ="S()[rotY(2)S()]b($0+1)"
```

The crossover operator for the *fL* representation copies the genotype of parents into children, and then adds a few random rules from the opposite parent to the genotype of the child. For example for parents:

```
//L
w:w0, 1
i:axiom="S()w0(0)", 7
r:pred=w0, cond="$0<3", succ="S()w0($0+1)"
r:pred=w0, , succ="[rotY(1.57)rotZ(1.57)S()w0(0)][rotY(-1.57)rotZ(1.57)S()w0(0)]S()w0(0)"

//L
w:w0, 2
w:w1, 1
i:axiom="S()w0(1.57,0)w1(0)", 7
r: pred=w0, cond="$1<1", succ="S()w0($0,$1+1)"
r:pred=w0, , succ="rotY($0)S()w0(-$0,0)"
```

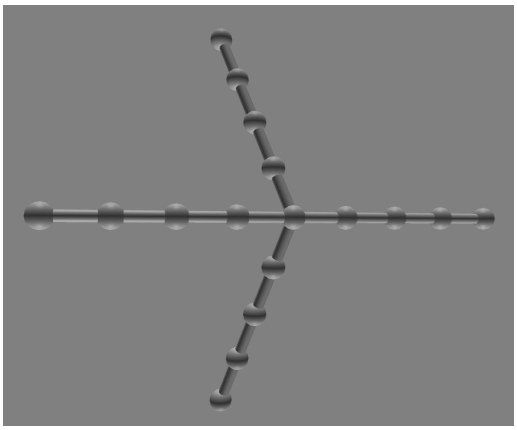
The first child is initialized with the genotype of the first parent. After this, the random rule of the second parent is selected to be added to this genotype. The rule used is marked with red color. As it can be seen in the current child genotype, it does not have any word with two parameters that could be used for this rule. In such cases new word definitions are introduced with needed number of the parameters. The final genotype is following:

```
//L
w:w0, 1
w:w1, 2
i:axiom="S()w0(0)", 7
r:pred=w0, cond="$0<3", succ="S()w0($0+1)"
r:pred=w0, , succ="[rotY(1.57)rotZ(1.57)S()w0(0)][rotY(-1.57)rotZ(1.57)S()w0(0)]S()w0(0)"
r:pred=w1, , succ="rotY($0)S()w1(-$0,0)"
```

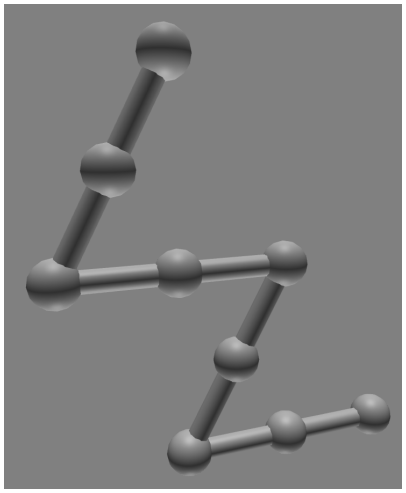
As it can be seen, the result of the crossover will not be visible in the beginning, but the rule may be used in further development. For the words in the successor sequence of the new rule the defined words are translated for the genotype in a following way:

1. If in the current genotype there is the defined word with the same name and number of parameters, then it is used.
2. If there is a word with the same name, but different number of parameters, the algorithm looks for the word with the same number of parameter.
3. If such word is found, then all appearances of the original word in the successor are renamed to the name of found word.
4. If there are no words with the same number of parameters, a new word definition is introduced for child genotype.

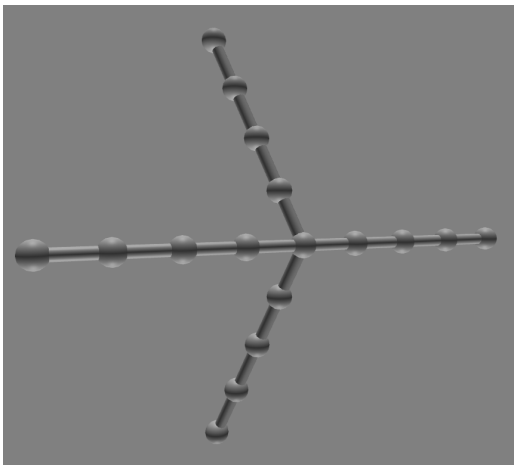
The second child inherits all the genotype from the second parent, and the random rule from the first parent is selected. In this case the word definition with the same name *w0* as in the incoming successor has different number of parameters. But there is a word *w1* in the set of word definitions that has the same number of parameters as the *w0* word in incoming rule. In this case all appearances of *w0* in the new rule are replaced with *w1*. The final genotype is shown below.



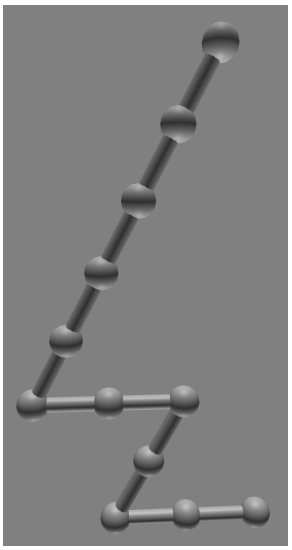
(a) The first parent



(b) The second parent



(c) The first child



(d) The second child

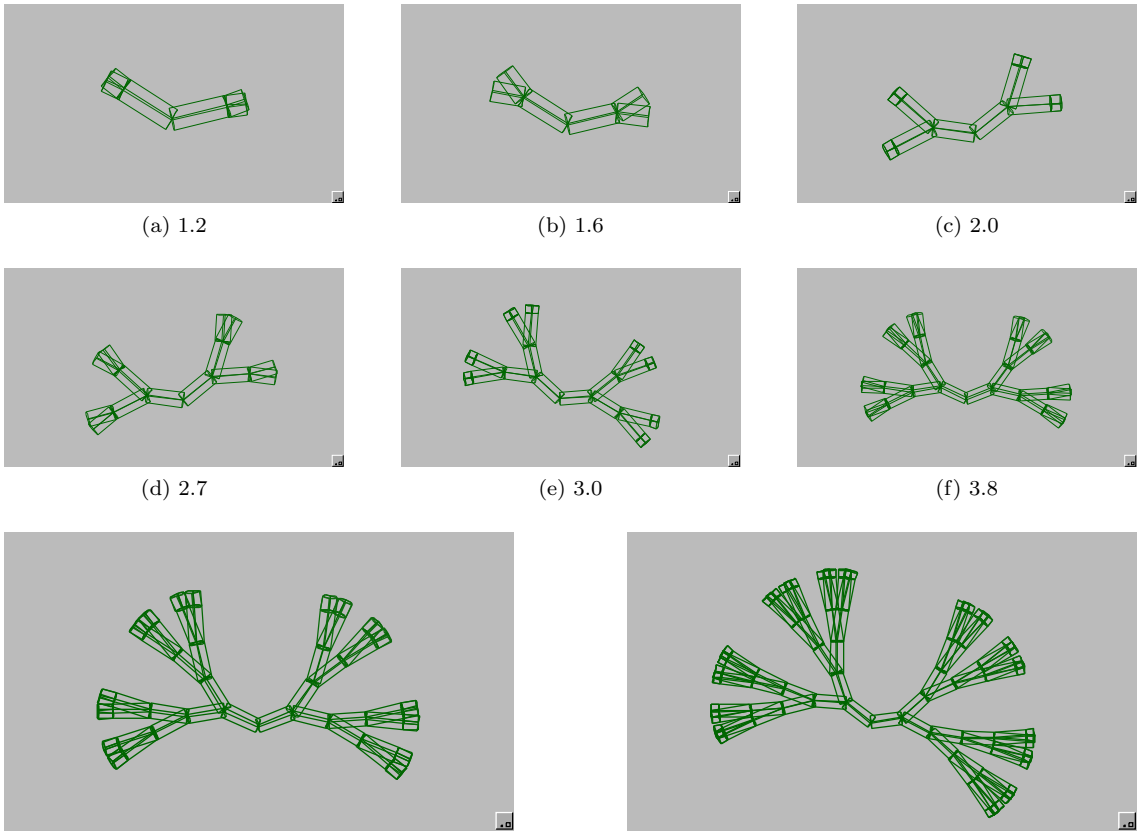
Figure 4.13: The phenotypes of parents and children for example crossover.

```
//L
w:w0, 2
w:w1, 1
i:axiom="S()w0(1.57,0)w1(0)", 7
r: pred=w0, cond="$1<1", succ="S()w0($0,$1+1)"
r:pred=w0, , succ="rotY($0)S()w0(-$0,0)"
r:pred=w1, cond="$0<3", succ="S()w1($0+1)"
```

In this case the change in the phenotype is seen instantly. The phenotypes of parents and their children are shown in Figure 4.13. This operator is unfortunately not applicable for genotypes without rules.

4.4.6 Properties of the L-System encoding

The mapping for *fL* encoding has simple mapping that provides little information – it only tells from which rules the given element was developed. This genetic representation is highly parametrized – it allows performing operations on the very low, numeric level. This enables more continuous exploration of the fitness landscape. The genotype represents the development process. It is possible to design such genotype that will provide smooth development process, for example the



(a) 1.2 (b) 1.6 (c) 2.0

(d) 2.7 (e) 3.0 (f) 3.8

(g) 4.3 (h) 5.0

Figure 4.14: The example of creature development with use of *Timed DOL-Systems*.

development of the below genotype is shown in Figure 4.14.

```
//L
w:w0,1
i:"[rotY(0.78*$t)S(l=$t*2-2)w0(0.5)][rotY(-0.78*$t)S(l=$t*2-2)w0(0.5)]",5.0
r:pred="w0",, succ="[rotY(0.78*$t*$0)S(l=$t*2-2)w0($0*0.5)][rotY(-0.78*$t*$0)S(l=$t*2-2)w0($0*0.5)]"
```

The advantage of this encoding is the possibility of reusing the genotype elements – this leads to the scaling of the used elements for the development in time. The second advantage of this representation is that due to its parametrization it addresses the large subspace of the phenotypes. It is also able to explore more precisely the neighbourhood of the current solution.

The main disadvantage is the overhead that happens during development of "words" meaning and instructions. The representation can waste a lot of time on developing more or less useful rules for the given task. What is more, this representation has a high risk of uncontrollable bloating – a single word can explicitly or implicitly double itself over time, which can lead to the exponential growth of the genotype. Because of this, the additional stopping condition was added. It checks if during current development step the number of produced words exceeded the given threshold. If so, then the current genotype is "repaired" by reducing the number of iterations to a safe value, which will not exceed this limit.

4.5 Summary of genetic encodings in Framsticks

As it can be seen, there are various representations available in the Framsticks platform. The list of selected aspects that differ in representations for the evolutionary design problem are listed below:

	f0	f1	fH	fB	f4	f7	f8	fL	f9
P1 classify	NGD	NGI	NGI	NGI/GE	GE	NGI	GI	GI	NGI
P2 pheno	ALL	MOD,NOL	LIM,CON,NOL	LIM,CON,NOL	MOD,NOL	ALL	MOD,NOL	CON,NOL	LIM
P3 loops	no	no	no	no	yes	no	yes	implicitly	no
P4 cond	no	no	no	no	no	no	yes	yes	no
P5 devel	no	no	no	no	yes	no	yes	yes	no
P6 changes	VSM	SM	VSM	CH	SM/CH	CH	SM/CH	VSM/CH	SM
P7 complex	NO	LOW	MED	HIGH	MED	HIGH	HIGH	HIGH	LOW
P8 globals	no	no	yes	yes	no	no	yes	yes	no

Table 4.3: The summary of genetic representations in Framsticks, based on implemented capabilities.

P1. *The classification of encoding* – non-generative direct (NGD), non-generative indirect (NGI), generative implicit (GI), generative explicit (GE),

P2. *Available phenotype subspace* – the representation addresses all possible phenotypes (ALL), allows continuous exploration of the subspace (CON), is not available to form loops (NOL), allows semi-discrete exploration with use of special modifiers (MOD), has very limited addressable subspace of phenotypes (LIM),

P3. *Allows loops*,

P4. *Allows conditions*,

P5. *Allows creating simple development procedures*,

P6. *Changes made by operators* – very small (VSM), small (SM), chaotic (CH),

P7. *Complexity of the mapping* – none (NO), low (LOW), medium (MED) or high (HIGH),

P8. *Presence of global parameters of encoding.*

Above aspects were considered for genetic encodings available for Framsticks and summarized in Table 4.3. Other aspects like speed and evolution efficiency of selected encodings will be empirically analysed and presented in Chapter 5.

Chapter 5

Comparing the performance of genetic encodings

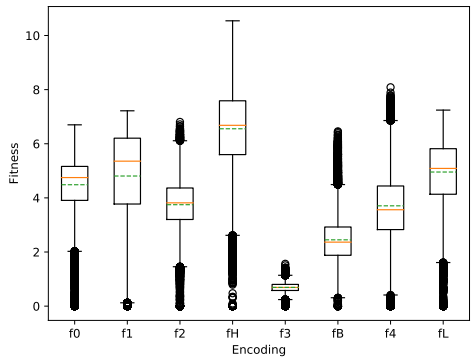
This chapter focuses on the empirical analysis of the *f0*, *f1*, *f2*, *f3*, *f4*, *fH*, *fB* and *fL* encodings. All parameters of the encodings in the experiments were adjusted to maximize the similarity of probabilities for analogous mutation types, like addition, modification, deletion. The full similarity in the probabilities of mutations between every encoding is nearly impossible, due to the different levels of abstraction of these representations. The experiment parameters that were the same for all the experiments were following:

- 90% probability of performing the mutation and 10% probability of performing the crossover on the solutions,
- Random selection of creatures during a negative selection,
- Number of evaluations as a stopping condition (the number of evaluations used differed between experiments),
- Non-rigid-body Mechastick physic as the simulation engine [KU18],
- Beginning from the simplest genotypes,
- Limiting the number of parts, joints, neurons and neural connections to 30 by returning 0 fitness values for creatures exceeding those constraints (the only experiment that does not have this constraints was the experiment described in Section 5.4).

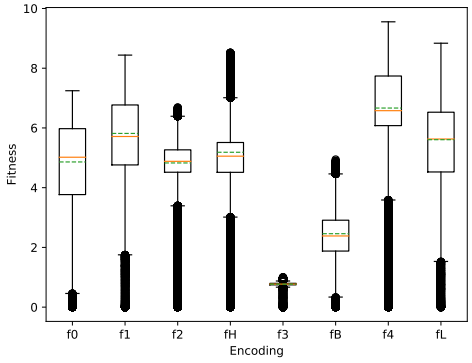
During the tasks involving neural networks the available classes were the *sigmoid neuron (N), gyroscope (G), touch sensor (T), constant signal (*), bending muscle (|) and rotating muscle (@).* Every experiment had been repeated ten times for every representation.

5.1 Maximization of the vertical position

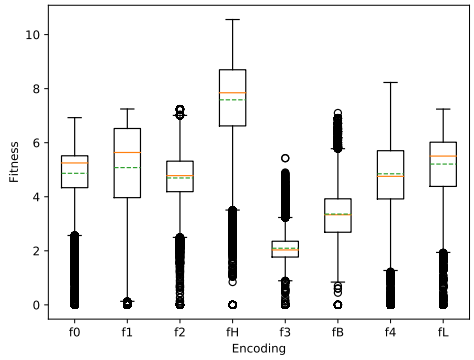
These experiments were focused on creating possibly high structures. There were two fitness functions considered – the maximization of *the vertical position of the highest part*, and the maximization of *the center of mass for the creature*. The maximization can be performed in an *active* or *passive* setup. The active maximization allows for the evolution of control systems that help the solution in improving its performance via a movement, while the second one disables the neural



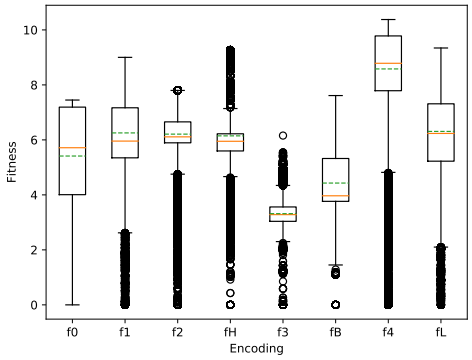
(a) Average values, $POPSIZE = 20$



(b) Average values, $POPSIZE = 500$



(c) Maximal values, $POPSIZE = 20$



(d) Maximal values, $POPSIZE = 500$

Figure 5.1: A distribution of the fitness values for the experiment of the active maximization of the height of the highest part. The orange lines represent the median of the populations, while the green dashed lines represent the mean value for the population.

network simulation, and prevents the use of neurons and neural connections[1]. The third parameter was *the number of genotypes in a population* – the considered population sizes were 20 and 500.

Every of those eight experiments was run 10 times with 750 000 evaluations (i.e., new genotypes). The results for those experiments are in below sections.

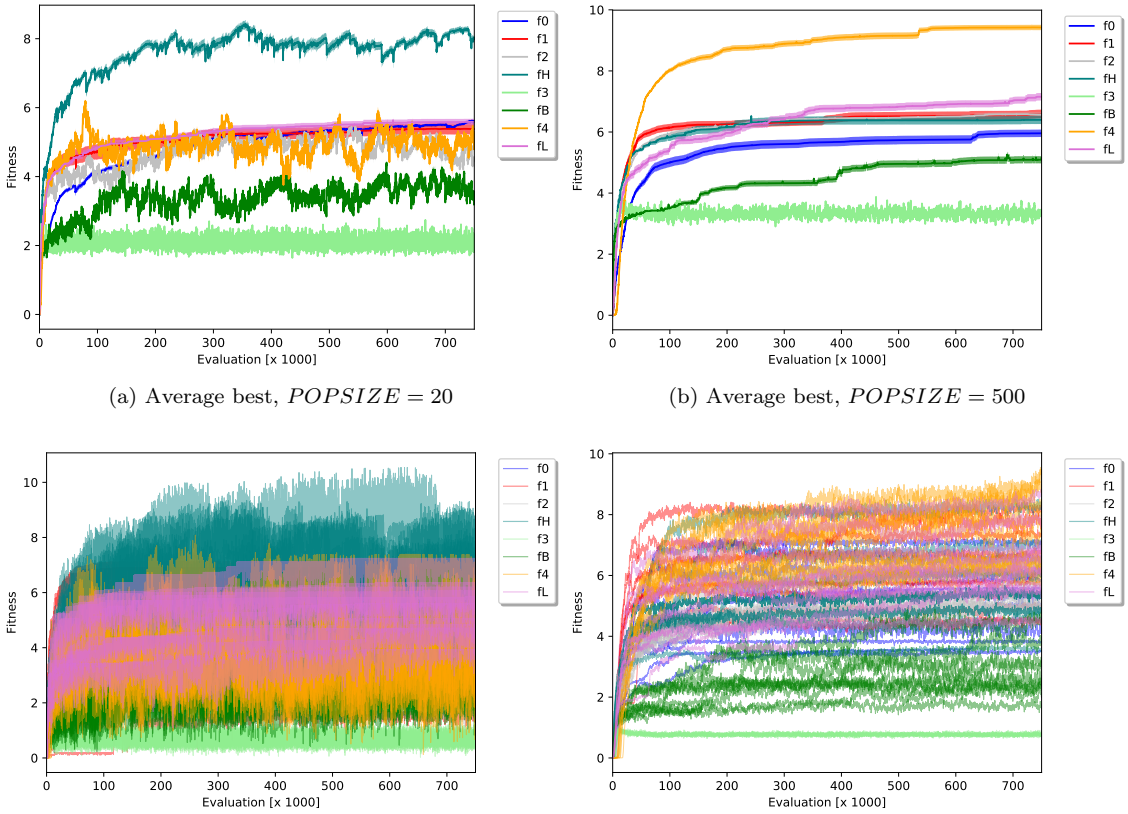
The *fL* encoding is able to evolve perfectly vertical structures, and it has used this ability to create high fractals or L-shaped structures that were fully two-dimensional. Because the physics engine allows perfect equilibrium, such constructs were staying still. Due to this fact the solutions created with this representation during experiments had very small perturbations added to every part to make the two-dimensional solutions fall. Other representations were unable to evolve such solutions.

5.1.1 The maximization of the vertical position of the highest part

Active maximization

As it can be seen in Figure 5.1, for the smaller population size the best solutions are encountered for the *fH* encoding. This representation was usually evolving large and heavy bases for the high towers that were kept straight by muscles. The representation was able to develop quickly the

[1]These settings are not applicable only for the *f3* and the *fB* encodings, because these encodings are unaware of the results of the mutations



(a) Average best, $POPSIZE = 20$

(b) Average best, $POPSIZE = 500$

(c) All runs, $POPSIZE = 20$

(d) All runs, $POPSIZE = 500$

Figure 5.2: The top two plots represent the average best fitness values for the evolutionary runs for the active maximization of the height of the highest part. The semi-transparent backgrounds for the every line in the first two plots represent the standard deviations for every evaluation multiplied by 0.1. The two bottom plots represent average fitness values for all the runs of the experiment for every representation.

solutions that were outperforming the rest of the representations, as it can be seen in Figure 5.2b. The second encoding that had the best average result for the population size of 20 was the *fL* encoding. It was mostly evolving the *L-shaped* structures that had additional stabilization in a form of small branches near the bending.

The similar behaviour of the *fL* encoding can be observed with the population size equal to 500. In comparison to the other encodings this representation had quite good results. The representation that was the best for the population size of 500 was the *f4* encoding. It was able to evolve high structures that consisted of *T-shaped* legs that elevated the rest of the creature. It is worth mentioning that for the smaller population size the difference of the means for the *f4* and *f0* encodings for the best fitness values was this small that the Tukey HSD test[2] [AW18] with *FWER* equal to 0.01 was unable to reject the null hypotheses that those two means are not equal. This may suggest that the performance of the *f4* encoding was similar to the performance of the *f0* encoding.

The another interesting thing in this experiment was observed in the *f2* encoding. Some of the best solutions were starting with the laying tower, and after stabilization they slowly elevated their very high towers to a vertical position, in order not to fall. The last significant observation that will be visible for all of the experiments for height maximization is the forming of loops in

[2]This test is usually performed when ANOVA test rejects the null hypothesis that the means of populations are equal. The Tukey HSD test performs pairwise comparison of all populations to verify, which populations differ.

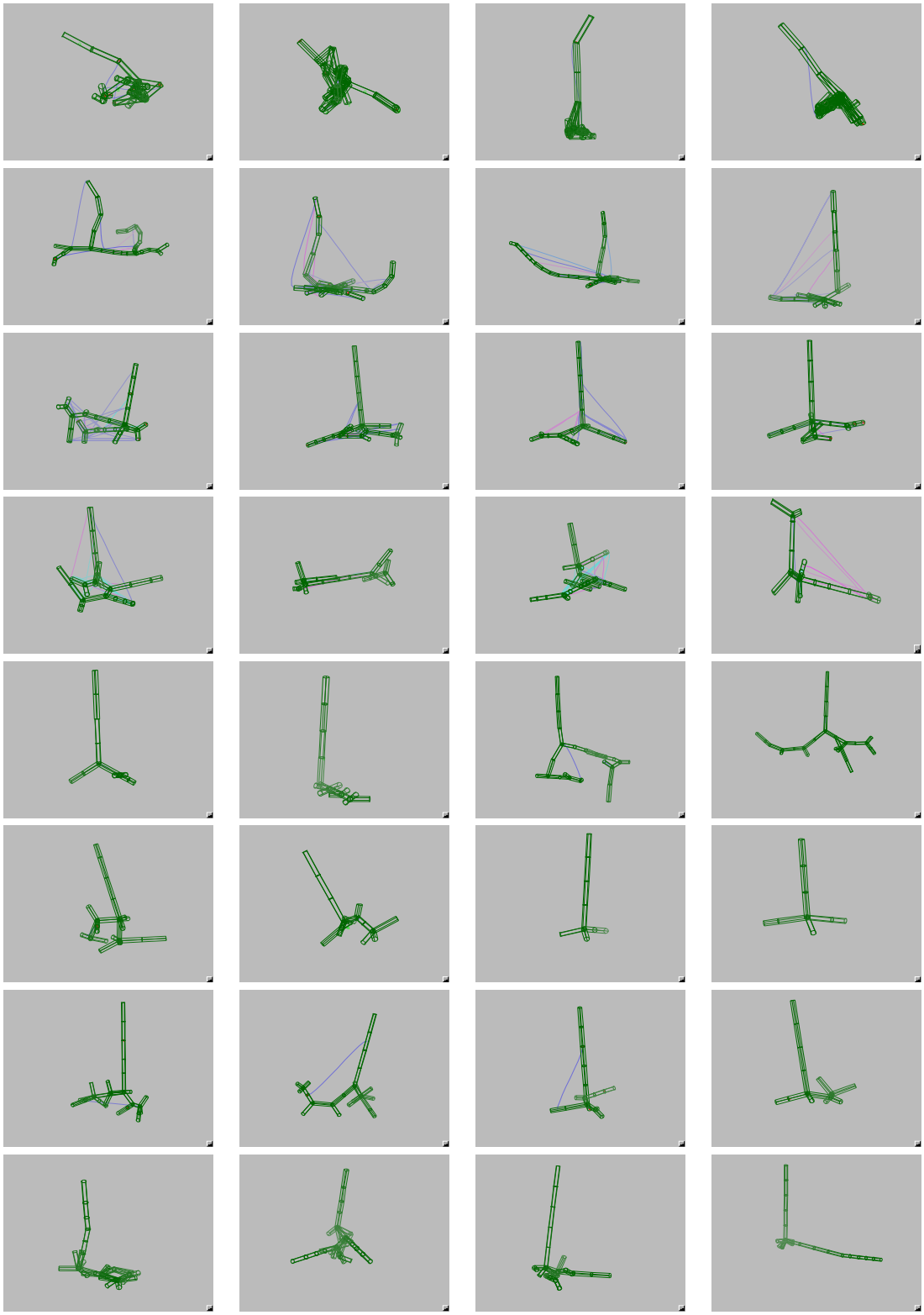
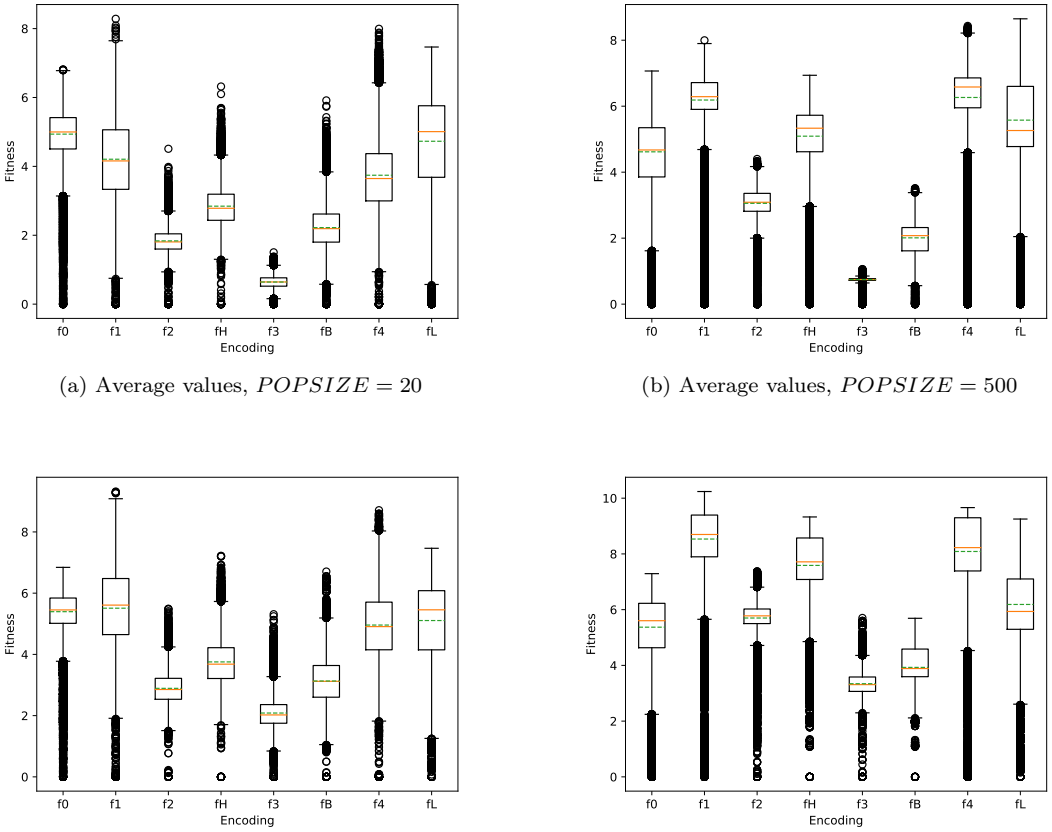


Figure 5.3: The results of the encodings for the task of active maximization of the height of the highest part of the creature. Figures in each row represent *f0, f1, f2, f3, f4, fB, fH, fL* encodings, respectively. First two images in each row represent the best solutions for the population size equal to 20, and the next two images represent the best solutions for the population size equal to 500.



(a) Average values, $POPSIZE = 20$

(b) Average values, $POPSIZE = 500$

(c) Maximal values, $POPSIZE = 20$

(d) Maximal values, $POPSIZE = 500$

Figure 5.4: A distribution of the fitness values for the experiment of the passive maximization of the height of the highest part. The orange lines represent the median of the populations, while green dashed line represents the mean value.

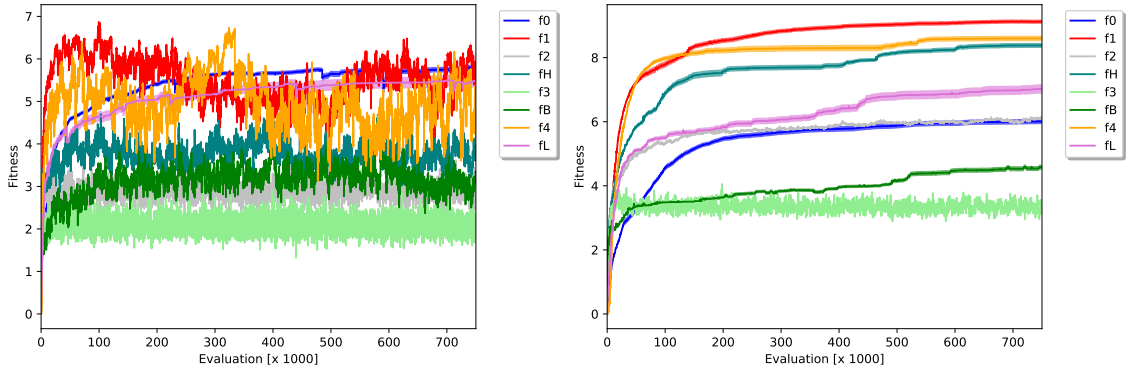
the *f0* encoding. As it was mentioned in Section 3.2, the *f0* representation is the only one among tested that has an ability to form loops. It uses this advantage to create small and stable legs for the towers.

The creatures that were evolved for this experiment are presented in Figure 5.3. Despite the fact that the evolution was allowed to evolve neural networks, most of the solutions for every encoding created "brainless" structures that were as high as possible.

Passive maximization

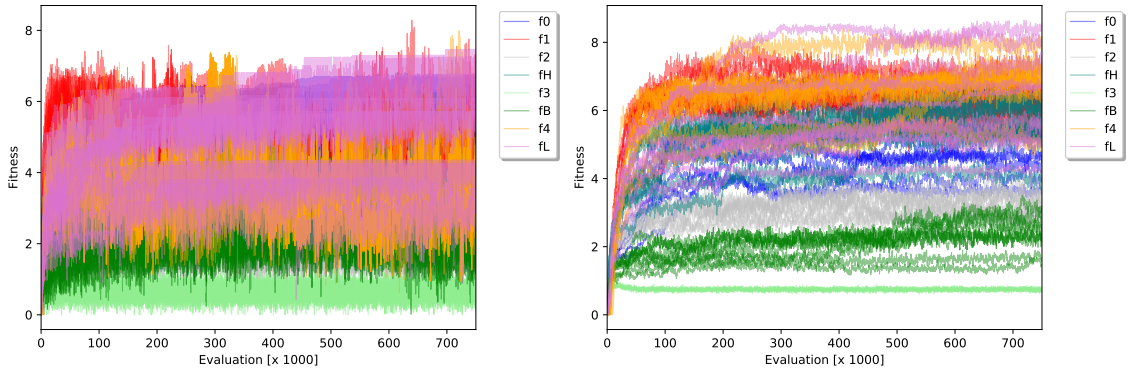
In this task the evolution of the control system was disabled. The results of this experiment show the *f0* as the leader when it comes to the average performance for the population size of 20. The models for the *f0* encoding, presented in Figure 5.6, may not look promising – for example the last phenotype in the row looks like it is laying on the ground. But this solution managed to use the heavy leg to "rise" the structure to form a tall tower, as shown in Figure 5.7. The other examples of *f0* encodings also used physics to actively elevate the tower after falling on the ground.

The best solutions for the population size equal to 20 were evolved for the *f1* encoding – visually it looks like the *f0* and *f1* encodings have nearly the same mean fitness for the best solutions, as seen in the Figure 5.4c, but the Tukey HSD post-hoc test rejected the hypothesis that those two representations have the same mean with the FWER equal to 0.01. For the populations of size 500,



(a) Average best, $POPSIZE = 20$

(b) Average best, $POPSIZE = 500$



(c) All runs, $POPSIZE = 20$

(d) All runs, $POPSIZE = 500$

Figure 5.5: The top two plots represent the average best fitness values for the evolutionary run for the passive maximization of the height of the highest part. The semi-transparent backgrounds for every line in the first two plots represent the standard deviations for every evaluation multiplied by 0.1. The bottom two plots represent average fitness values for all the runs of the experiment for every representation.

the *f4* encoding again had very good results.

The *fH* and *fB* encodings sometimes had the problem with evolving "legs" with more than 3 stabilizing sticks, because for larger number of legs one of them was growing towards the ground, making the whole structure unstable. The *fB* encoding during this experiments was able to overcome this problem by the reduction of the problematic leg to the minimal length.

5.1.2 The maximization of the vertical position of the center of mass

This experiment had the same setup as the previous one, except for the optimized function – the aim was to maximize the height of the center of mass of the creature. In this case the usual method is to create possibly small "leg" that holds the heavy and tall structures.

Active maximization

The examples of the creatures for this experiment are shown in Figure 5.10. In many cases the evolved structures look similarly to the examples from the experiments in Section 5.1.1, especially for the *f2, f3, fH* and *fB* encodings, but there were some exceptions. First of all, the *fH* encoding was able to evolve a high creature that was jumping immediately after touching the ground, so it was spending a lot of time in the air. The *f1* encoding created bush-like structures that were

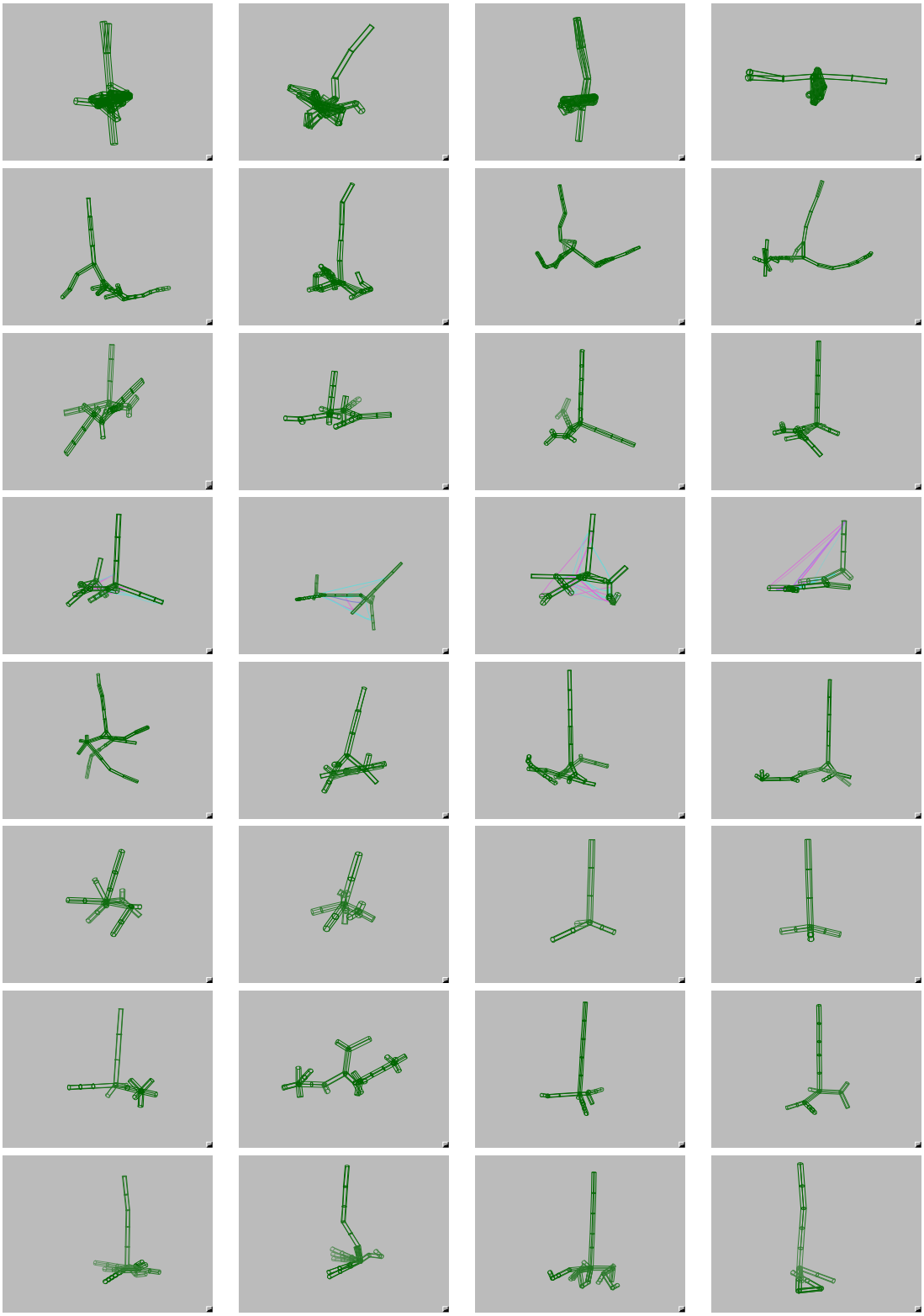


Figure 5.6: The results of the encodings for the task of passive maximization of the height of the highest part of the creature. Figures in each row represent *f0, f1, f2, f3, f4, fB, fH, fL* encodings, respectively. First two images in each row represent the best solutions for the population size 20, and the next two images represent the best solutions for the population size 500.

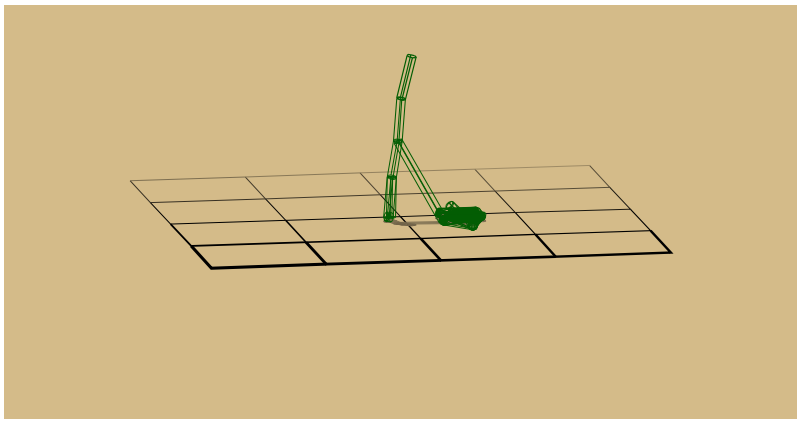
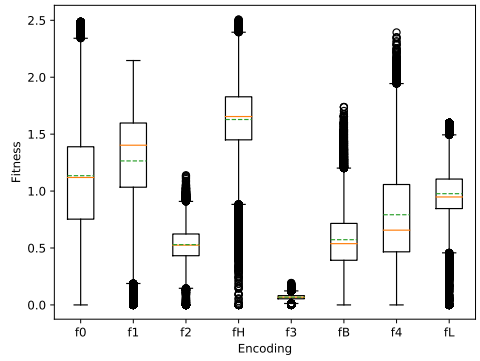
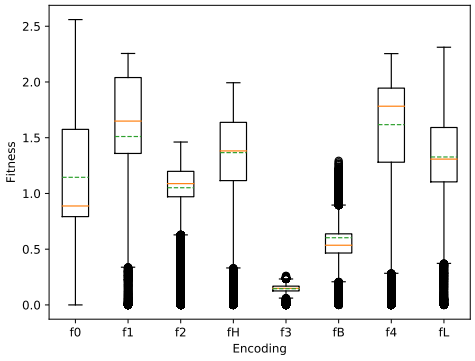


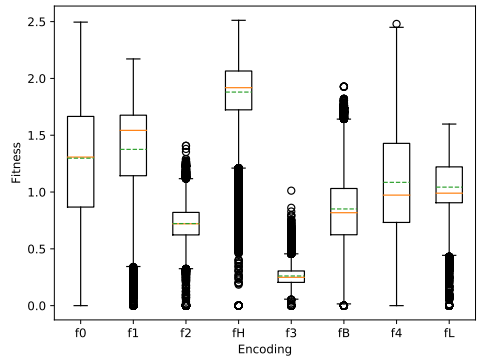
Figure 5.7: The example of rising the tower in the *f0* genotypes for maximizing the vertical height of the highest part of the creature.



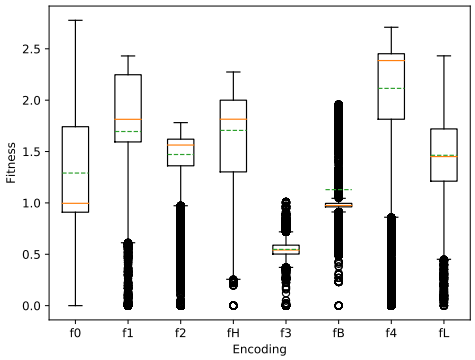
(a) Average values, $POPSIZE = 20$



(b) Average values, $POPSIZE = 500$

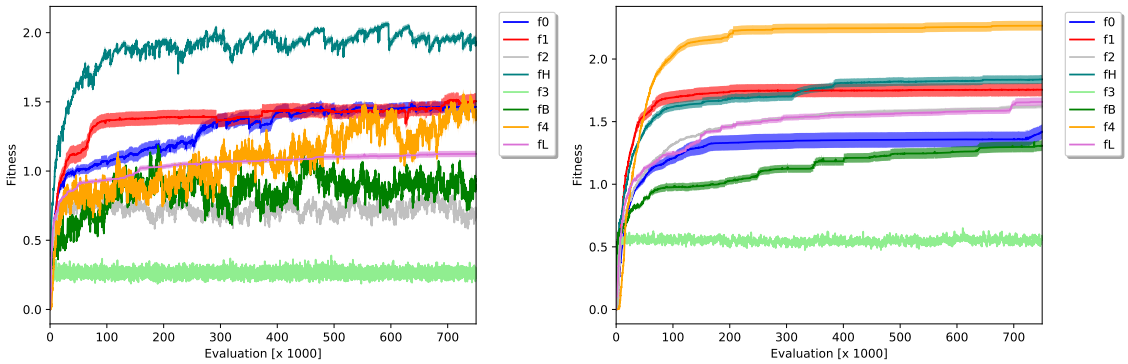


(c) Maximal values, $POPSIZE = 20$



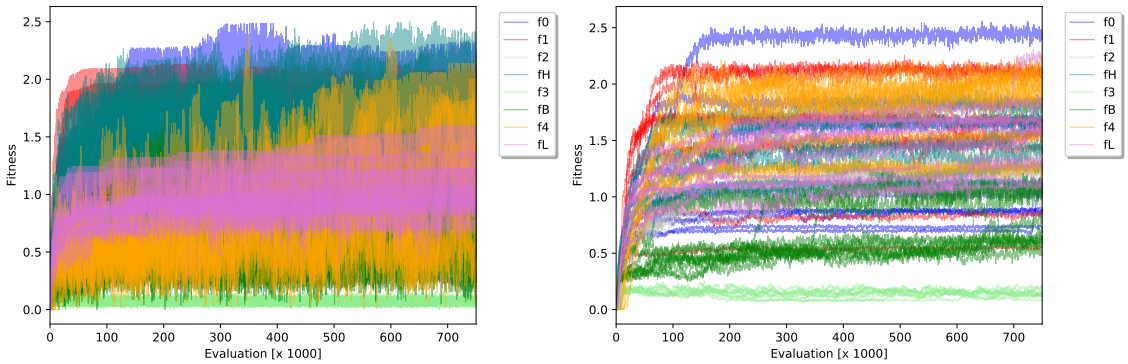
(d) Maximal values, $POPSIZE = 500$

Figure 5.8: A distribution of the fitness values for the experiment of an active maximization of the height of the center of mass. The orange lines represent the median of the populations, while the green dashed line represents the mean value.



(a) Average best, $POPSIZE = 20$

(b) Average best, $POPSIZE = 500$



(c) All runs, $POPSIZE = 20$

(d) All runs, $POPSIZE = 500$

Figure 5.9: The top two plots represent the average best fitness values for the evolutionary run for the active maximization of the height of the center of mass. The semi-transparent backgrounds for every line in the first two plots represent the standard deviations for every evaluation multiplied by 0.1. The bottom two plots represent average fitness values for all the runs of the experiment for every representation.

straightening their branches. Other interesting solution could be observed in the *f4* encoding. It had a long "spine" that was located above the ground. This "spine" was on the sticks that acted like table legs. The *fL* encoding evolved structures similar to the solutions of the *f0* encoding, but it tended to create very small branches on the top of the towers in order to increase the height of the center of the mass.

As it can be seen in Figure 5.8a, the *fH* again performed very well in the active version of the problem for small population size, in this case for the maximization of the center of mass. For the larger population size, the *f1* and *f4* encodings performed better than the *fH* encoding. The analogous situation happened in Figure 5.1b.

Passive maximization

The last task for maximizing the vertical position was the passive maximization of the vertical position of the center of mass. As it can be seen in Figure 5.13, there are some interesting structures that have not been seen before. First of all, the *f0* encoding for small population size had very good results, according to Figure 5.11. It evolved structures with several nearly parallel sticks on the top of the tower and a triangular base – such structures were stable and they had high fitness value. Other interesting structures were in the *f2* encoding – it created structures that had umbrella-like towers. Most of the umbrella bended due to the gravity, but the rest of the sticks

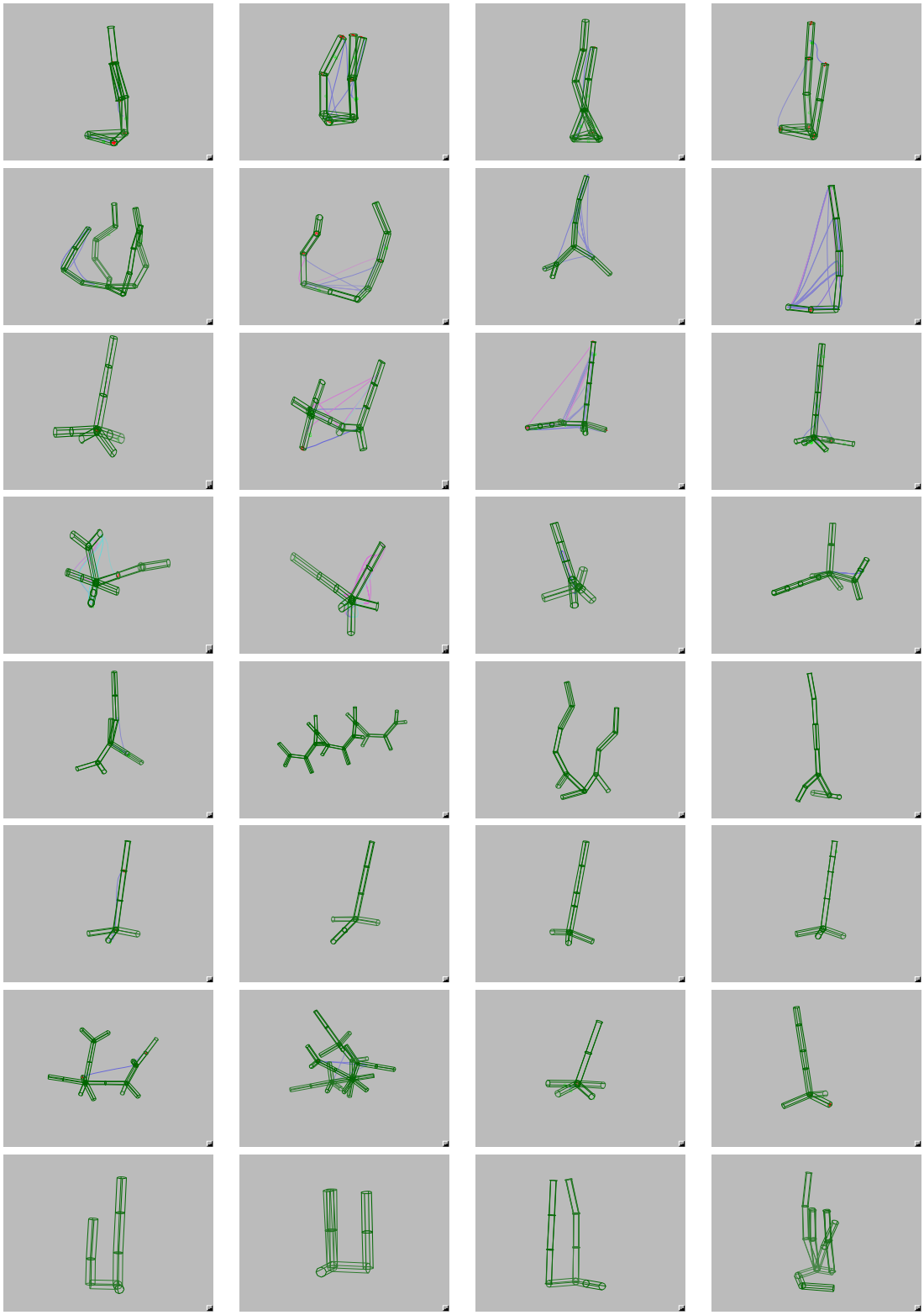
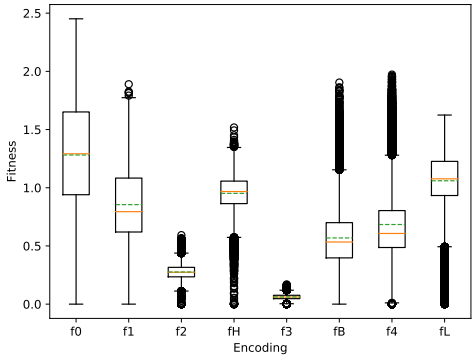
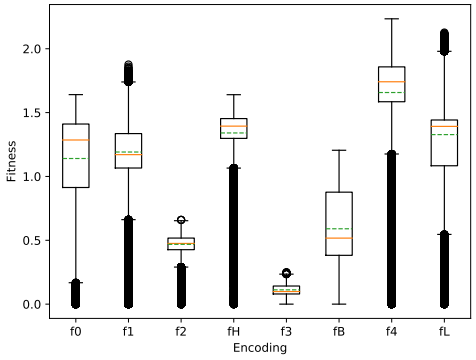


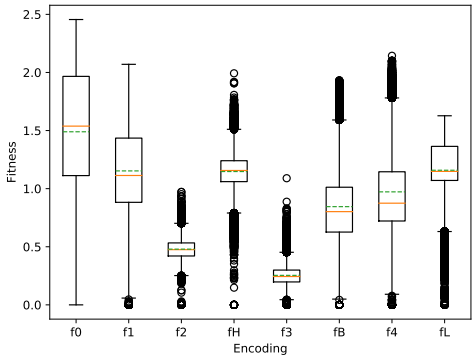
Figure 5.10: The results of the encodings for the task of the active maximization of the height of the center of mass for the creature. Figures in each row represent *f0, f1, f2, f3, f4, fB, fH, fL* encodings, respectively. First two images in each row represent the best solutions for the population size equal to 20, and the next two images represent the best solutions for the population size equal to 500.



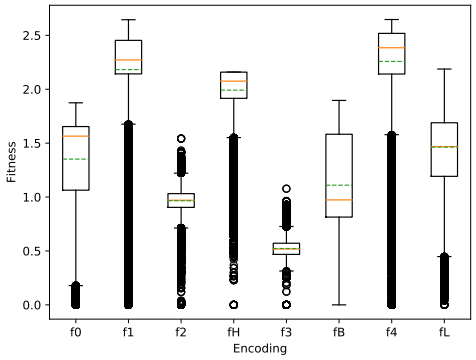
(a) Average values, $POPSIZE = 20$



(b) Average values, $POPSIZE = 500$



(c) Maximal values, $POPSIZE = 20$



(d) Maximal values, $POPSIZE = 500$

Figure 5.11: A distribution of the fitness values for the experiment of a passive maximization of the height of the center of mass. The orange lines represent the median of the populations, while a green dashed line represents the mean value.

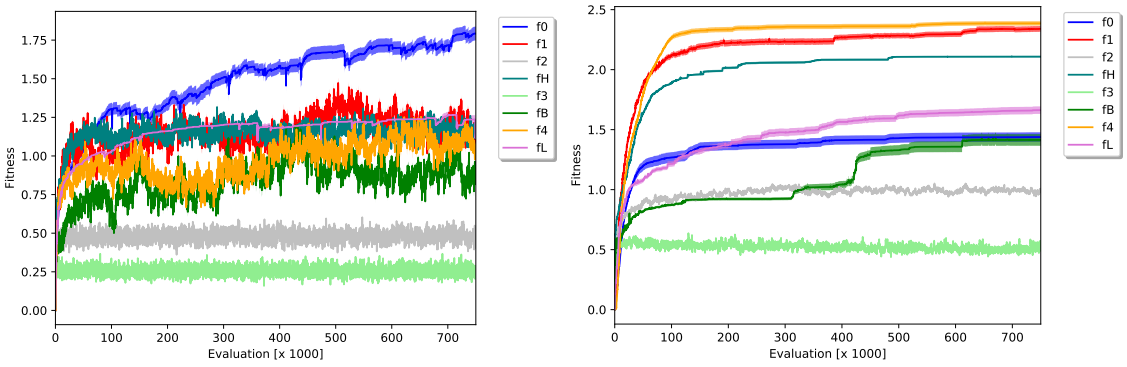
EXP	*PSIZE*	f0	f1	f2	fH	f3	fB	f4	fL
HPa	20	4	3	6	1	8	7	4	2
	500	6	3	4	5	8	7	1	2
HPp	20	2	1	7	5	8	6	4	3
	500	6	1	5	3	8	7	2	4
COMa	20	3	2	7	1	8	6	4	5
	500	6	3	4	2	8	7	1	4
COMp	20	1	3	7	4	8	6	5	2
	500	5	2	7	3	8	6	1	4
Borda voting ranks		5	1	6	3	8	7	2	4

Table 5.1: The table of ranks based on means of the best fitness for each representation. The Borda voting was performed in such a way that *ex aequo* ranks were resolved with the use of the real mean values.

for the tower increased the score of the creature. Similar, but more stable solution was proposed by the *fB* encoding – it created the towers shaped as "Y" letter. In the *f4* encoding, apart from previously known elevated spine, the other interesting structure was a spiral forming a cylinder. The similar structure could be observed in the *fL* encoding.

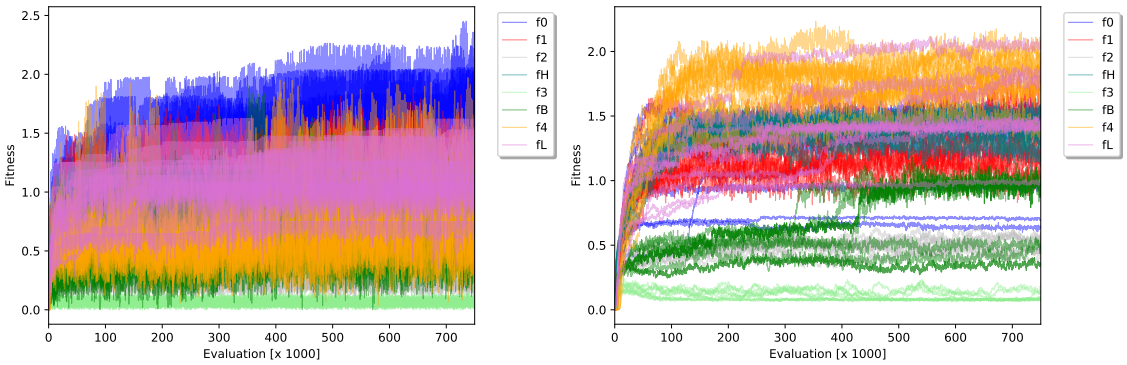
5.1.3 Summary of the results for the maximization of vertical position

Table 5.1 and Table 5.2 represent the ranks for every representation for the problems of the height maximization. The *HP* tasks represent the maximization of the height of the highest part, and the



(a) Average best, $POPSIZE = 20$

(b) Average best, $POPSIZE = 500$



(c) All runs, $POPSIZE = 20$

(d) All runs, $POPSIZE = 500$

Figure 5.12: The top two plots represent the average best fitness values for the evolutionary run for the passive maximization of the height of the center of mass. The semi-transparent backgrounds for every line in the first two plots represent the standard deviations for every evaluation multiplied by 0.1.

EXP	$PSIZE$	f0	f1	f2	fH	f3	fB	f4	fL
HPa	20	4	3	5	1	8	7	6	2
	500	5	2	6	4	8	7	1	3
HPp	20	1	3	7	5	8	6	4	2
	500	5	2	6	4	8	7	1	3
COMa	20	3	2	7	1	8	6	5	4
	500	5	2	6	3	8	7	1	4
COMp	20	1	4	7	3	8	6	5	2
	500	5	4	7	2	8	6	1	3
Borda voting ranks		4	1	5	2	7	6	3	2

Table 5.2: The table of ranks based on means of the average fitness for each representation.

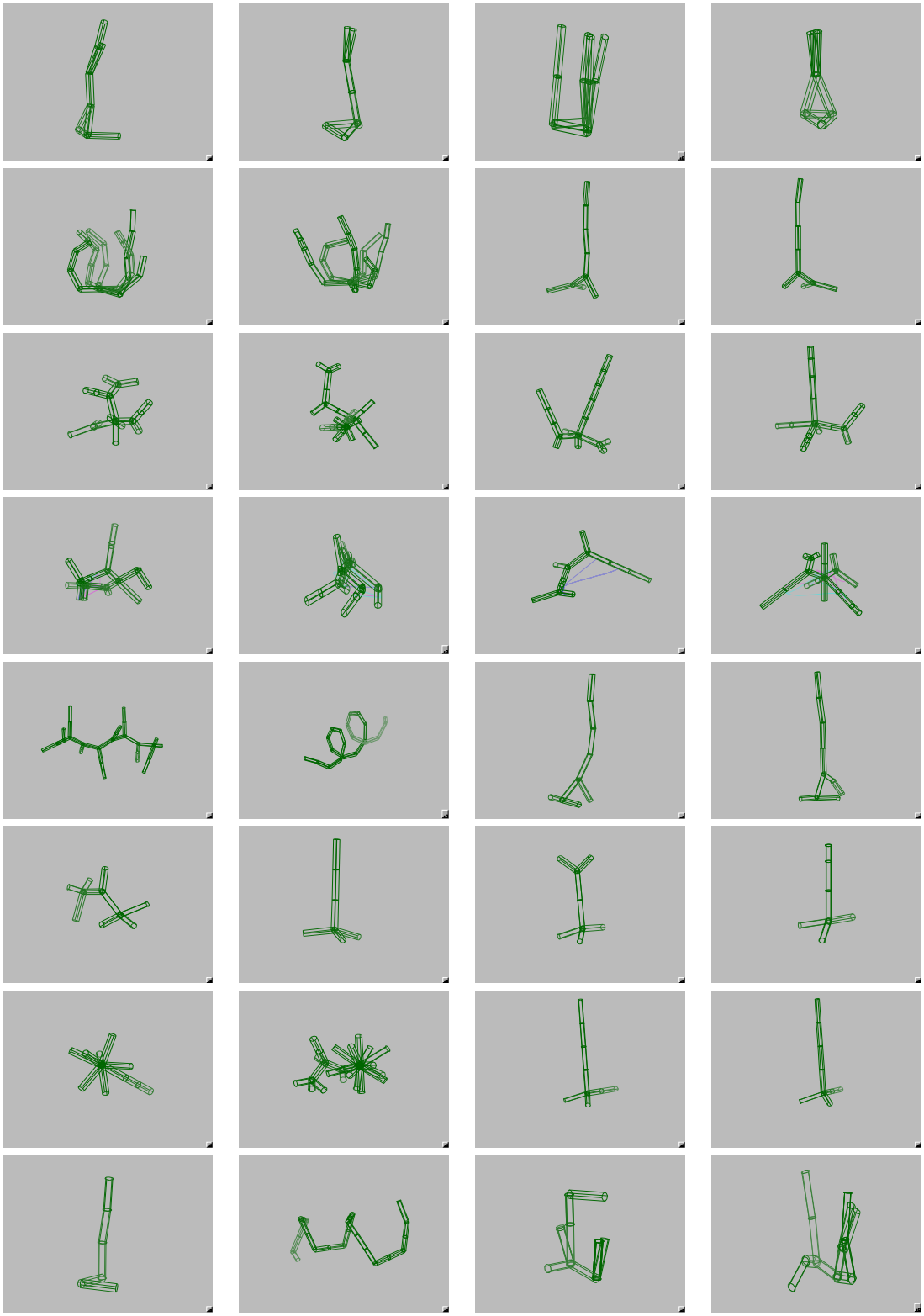


Figure 5.13: The results of the encodings for the task of passive maximization of the height of the center of mass for the creature. Figures in each row represent *f0, f1, f2, f3, f4, fB, fH, fL* encodings, respectively. First two images in each row represent the best solutions for the population size 20, and the next two images represent the best solutions for the population size of 500.

COM stands for the maximization of the height of the center of mass. Small letters after this codes denote if the experiment was active (a) or passive (p). $PSIZE$ stands for the population size. The last line of each table has the aggregated ranks computed with use of the Borda voting rule [Lip13], where the experiments were the voters. In this voting system points for each *candidate* (encoding) are assigned based on the ranking generated by each *voter* (experiment) – for eight representations the best representation receives 7 points, the second receives 6 points, and so on. The higher the overall score, the higher the final rank of the candidate. Borda Count is known for violating the majority criterion [Lip13] – it chooses a more broadly acceptable option over the one with majority support. In other words, the Borda Voting finds the candidate that most of the voters can agree on, according to their ranking. In case of the results of experiments, the Borda Voting is used to tell which representation performs well on most of the representations.

When the two representations share the same rank, it means that the null hypothesis saying that the mean values for those representations are the same could not be rejected.

The first observation coming from the presented plots and tables is the better results of the *fH* and *fB* encodings in compare of their previous versions – *f2* and *f3* encodings. The *fH* encoding was outperformed only once by *f2* encoding for the task of active maximization of the height of the highest part, as it can be seen in Table 5.1. But, after looking at Figure 5.2b it is observable that the *f2* encoding does not significantly outperform the *fH* encoding – they have nearly same results over time.

The *f1* encoding had quite good results for all the tasks. It was always able to evolve good structures, regardless to the use of neural networks and the size of the population. Maybe it was not outperforming the other representations as the *fH* encoding for some cases, but it could cope without larger problems with all tasks. The result of this stability is the first place in the Borda voting, presented in Table 5.1 and Table 5.2. It created both interesting structures and functional neural networks. On the other hand, as it can be seen in plots showing the change of the fitness values over time, this representation was usually evolving slower for active tasks than the *fH* encoding.

The *fH* encoding was usually evolving good solutions in a short amount of time for the active tasks. It can be observed in Figure 5.2 and Figure 5.9. It is especially visible for small sizes of populations. Also, the discovered solutions are much better than for the other encodings. There are two possible reasons for this phenomena. The first one is following – the *fH* encoding makes the "proper" use of the neural network. That means that it can create such control systems that help in achieving better performance. It was able to create jumping structures and straightening mechanisms for towers. The second reason is the fact that the three-dimensional structure is created implicitly during the process of adding sticks, while the other representations require separate mutations for changing the directions of sticks to form three-dimensional structures. What is more, the *fH* encoding, thanks to its development algorithm, can easily evolve small towers that look like "bird claws". For the larger populations this encoding was usually outperformed by the generative *f4* encoding. The evolution for this encoding was also slightly slower for the large populations. According to Borda ranks from Tables 5.1 and 5.2, this representation was performing very well for the tasks of vertical maximization. It only performed worse for the task of the passive maximization of the height of the highest part.

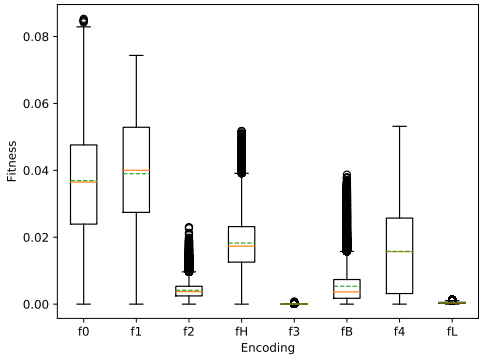
The *f4* encoding was evolving very interesting structures for all presented experiments – it was involving the reusability to create regular structures, like spiral pipes and curvy spines with "thorns". Those thorns were used as legs for the maximization of the height of the center of mass in order to minimize the number of parts near the ground. This representation usually required time before outperforming the other solutions, as it can be especially seen in Figure 5.2d, Figure 5.9a,

and Figure 5.12d. This representation was performing especially good, when the population size was large. It can be seen in the plots showing the change of the fitness function over time that the operators in this representation were introducing big changes in the final phenotype – for the small population sizes the fitness value oscillated and was unstable. This shows that when the population size is small, there is a possibility of loosing information about the good solutions, especially for the random negative selection. For the large population size the *f4* encoding had usually very good results, especially for the active maximization.

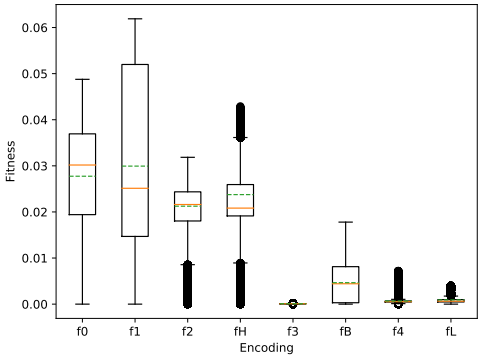
The second generative encoding, called *fL*, was usually evolving good solutions, but not outstanding ones. Just like the *f1* encoding, it was usually in the top three representations for every task, according to the Borda voting. As it can be seen in all plots of changes of fitness values over time, the *fL* encoding provides stable and slow, but progressing evolution. This shows that the genetic operators used for this representation indeed provide minimal changes to the genotype, and the phenotype. This allows more precise analysis of the fitness landscape, but results in slow exploration. The second thing that slows down the *fL* encoding is the evolution of the development process system. While *f4* and other representations have fixed elements, and their operators almost always provide a significant change to the phenotype, the *fL* representation can spend a lot of mutations on improving the rules that are not immediately applicable, nor useful. The results for this representation were sometimes similar to the results for the *f0* encoding, and sometimes to the resutls of the *f4* encoding. This is mainly because the evolution in *fL* is able to modify the phenotype on the very low level, inaccessible for most of the representations, except for *f0*. When the evolution spends most of the time on such modifications, then the evolved structures are similar to one in the *f0* encoding. On the other hand, during the process of evolution one or more valuable producing rules can be evolved, so the evolution will "focus" on exploring the aspects of reusability, which will make the phenotypes look more like *f4* encoding.

The *f0* encoding also had slow and stable increase of the fitness value over time, and it was performing especially good for the small population sizes. The direct encoding had usually very wide range of the fitness values, especially in Figure 5.8 and Figure 5.11. It managed to outperform the other encodings three times – in passive tasks with small population size. The *f0* and *fL* encodings have usually similar fitness values to some degree. The *fL* encoding was in most cases evolving faster the better solutions for height maximization tasks, but it was sometimes outrun by the direct encoding, especially for the passive tasks.

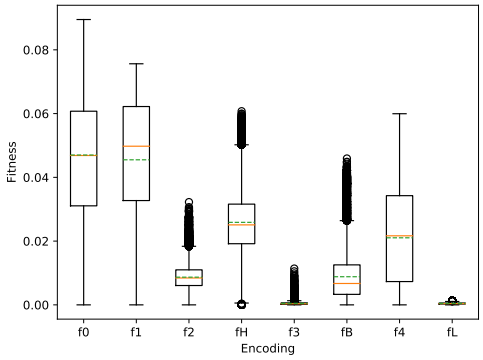
The *fB* usually does not have good results. It has oscillating fitness values for small population sizes – this is because of the chaotic genetic operators that can completely change the phenotype. On the other hand, for the large population sizes the *fB* encoding showed an interesting property. Most of the representations reach some level of the fitness value, from which it is hard for them to find any better solution – they reach the stagnation point. It takes a lot of time for those representations to leave the local optima, and sometimes they do not leave it at all. The *fB* encoding evolves slowly for the large population sizes, but it often manages to leave the local optima and jump to some new place in the fitness landscape, resulting in sharp increase of the fitness value and continuation of the evolution. This phenomenon can be observed in Figure 5.2b, Figure 5.9b, and Figure 5.12b. It is again the result of the chaotic character of the operators – while the other encodings have operators that usually preserve most of the information of the available solutions in the population, the *fB* encoding is able to perform such chain of mutations that will move unpredictably to some other subspace of the phenotype space. This shows that although mutations in the *fB* encoding can be considered harmful and unpredictable, this chaotic nature can be handy for leaving local optima, especially with large populations and long-lasting experiments.



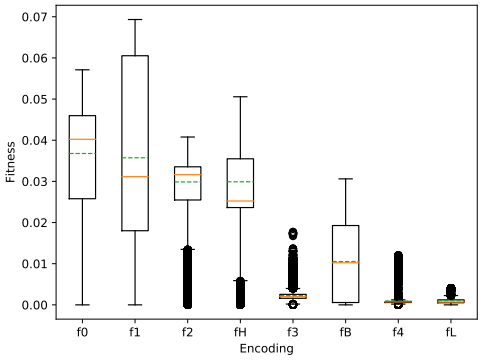
(a) Average values, $POPSIZE = 20$



(b) Average values, $POPSIZE = 500$



(c) Maximal values, $POPSIZE = 20$



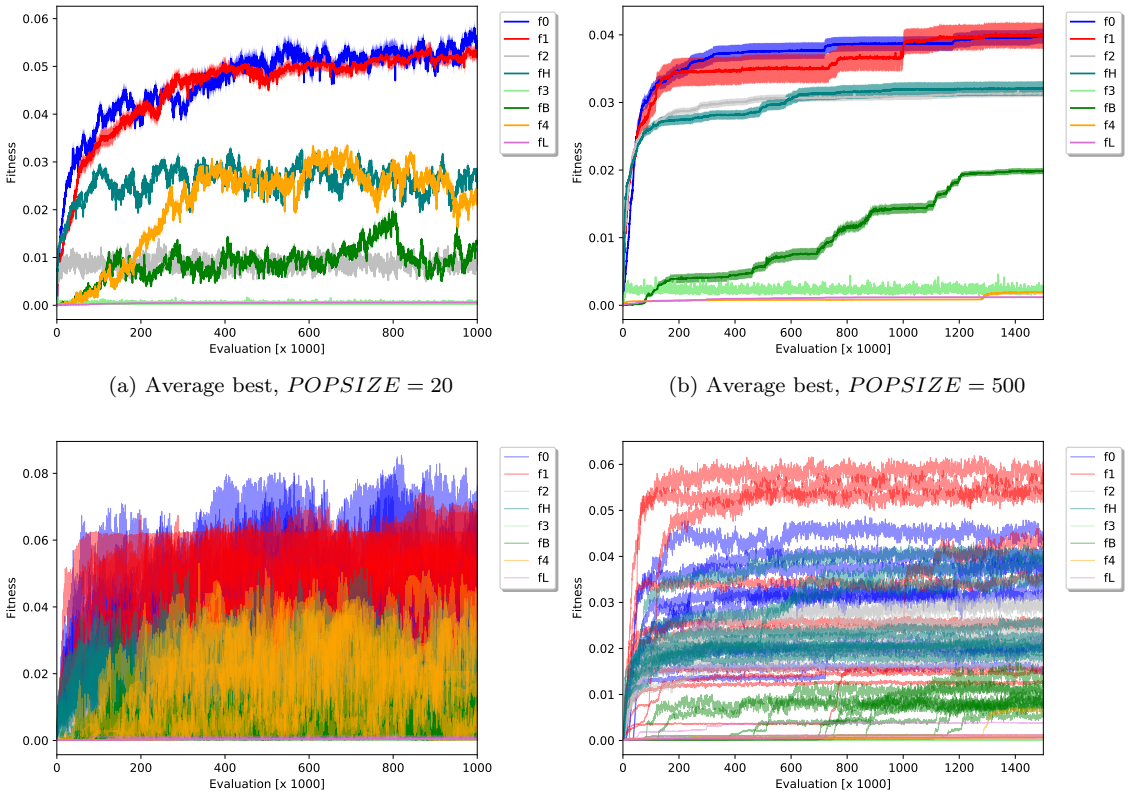
(d) Maximal values, $POPSIZE = 500$

Figure 5.14: Distribution of fitness values for the experiment of evolving the fastest creature.

RESULTS	$PSIZE$	f0	f1	f2	fH	f3	fB	f4	fL
AVG	20	2	1	6	3	8	5	4	7
	500	2	1	4	3	8	5	7	6
MAX	20	1	2	6	3	7	5	4	8
	500	1	2	3	4	6	5	8	7
Borda voting ranks		1	1	3	2	7	4	5	6

Table 5.3: The table of ranks based on the average fitness for each representation for velocity tasks. Ranks for the means of the best solutions are presented as MAX, the ranks for the means of the average solutions are presented as AVG.

The *f3* encoding can be considered as the worst encoding according to this set of experiments. As it was explained in Section 4.3, the main differences between the *f3* and *fB* encodings are the notation of the neuron classes, the modification of the substitution, the crossover operators, and the underlying body development performed by the *fH* encoding. The *f3* encoding, according to the results and changes of fitness values over time, was not introducing any useful changes to the genotype, and the modifications of current solutions were in general random. The *fB* encoding was providing significant improvement over time in comparison to the *f3* encoding, which was oscillating around small fitness value.



(a) Average best, $POPSIZE = 20$

(b) Average best, $POPSIZE = 500$

(c) All runs, $POPSIZE = 20$

(d) All runs, $POPSIZE = 500$

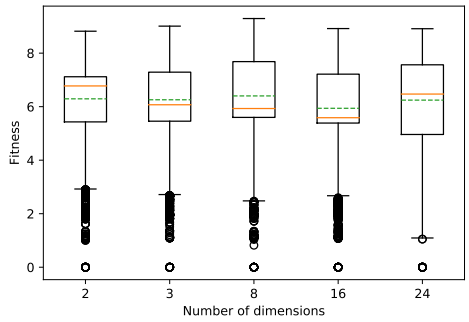
Figure 5.15: The best fitness values for the evolutionary run for the experiment of evolving the fastest creature.

5.2 The maximization of the velocity

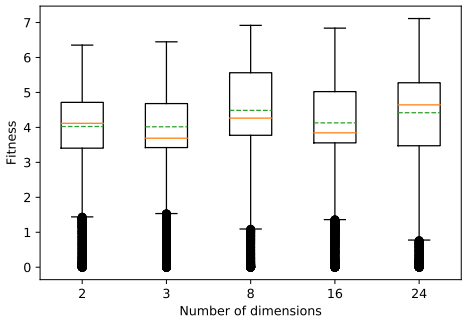
The next experiment was focused on an evolution of the creatures that could move as fast as possible. This experiment was performed with population sizes 20 (1 000 000 evaluations) and 500 (1 500 000 evaluations). The distributions of the fitness values are shown in Figure 5.14, while the changes of the fitness values over time are shown in Figure 5.15.

In this experiment the two leading representations were the *f0* and *f1* encodings, as it can be seen in Table 5.3. The *f1* encoding was the best in terms of the distribution of the average fitness, while the *f0* encoding created the best-performing solutions. The third representation that had interesting results was the *fH* encoding. It managed to evolve some simple jumping creatures. The generative representations acted poorly during this experiments – from both of them only *f4* managed to evolve some sufficiently moving creatures, usually basic crawlers. The *fL* encoding created very slow solutions – the reason of this behaviour may be the waste of the time spent on evolving some development rules, and not efficient evolution of the control systems. The *fB* encoding, just like in the height maximization experiments, evolved slowly and was leaving the local optima.

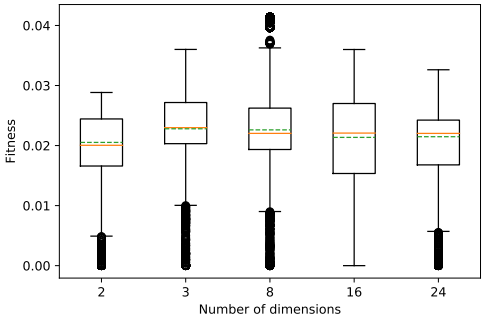
At first the results may be surprising – the *f0* representation was able to outperform the other representations in such a complex problem as evolving movement. On the other hand, the solutions for this problem are usually small structures with simple, but adjusted neural networks. In this case, the direct encodings can easily create such structures and adjust them during the rest of the evolution. The development of the production rules, like in *fL* encoding, is needless and can only



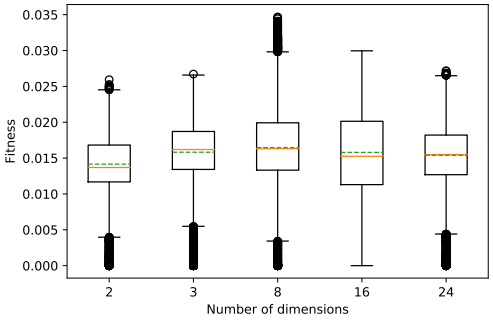
(a) Highest part height, the best fitnesses.



(b) Highest part height, the average fitnesses.



(c) Velocity, the best fitnesses.



(d) Velocity, the average fitnesses.

Figure 5.16: Distributions of fitness values for two experiments performed for the *fH* encoding.

slow down the evolution process.

5.3 The test of the parametrization of the similarity representation

The next presented experiments were focused on the parametrized representation available in Framsticks – the *fH* encoding. This representation has one parameter – the number of the dimensions for vectors of the handles. The question stated in this experiment is following – is there any dependency between the fitness values for a given problem and the number of dimensions for the *fH* encoding.

There were two experiments performed to analyse this aspect:

- Passive maximization of the height of the highest part, with population size equal to 500,
- The maximization of the velocity of the creature, with population size equal to 100.

Both experiments were ran ten times for 200 000 evaluations. The compared numbers of dimensions were *2, 3, 8, 16,* and *24*. Because the results did not differ much according to the observations of plots in Figure 5.16, for the every population of the number of dimensions the ANOVA test[3] [Kim14] was performed. The null hypothesis for this statistical test states that the mean values for n populations are equal. The p-value returned by this test for all experiments was

[3]The ANOVA test is a statistical test that verifies for multiple populations, if their distributions are significantly different. The null hypothesis of this test states that mean values for all populations are equal. The rejection of this hypothesis proofs that one or more populations have different distributions. Due to the fact that ANOVA test does not provide information which populations differ, the post-hoc tests, like Tukey HSD test [AW18], need to be used.

Populations		Height, best		Height, average		Velocity, best		Velocity, average	
G1	G2	meandiff	reject	meandiff	reject	meandiff	reject	meandiff	reject
2	3	−0.0349	False	−0.0074	False	0.0023	True	0.0017	True
2	8	0.1087	True	0.4615	True	0.0021	True	0.0023	True
2	16	−0.3541	True	0.1049	True	0.0008	True	0.0016	True
2	24	−0.0488	True	0.3949	True	0.0010	True	0.0012	True
3	8	0.1437	True	0.4689	True	−0.0002	False	0.0006	True
3	16	−0.3191	True	0.1123	True	−0.0014	True	0.0	False
3	24	−0.0138	False	0.4023	True	−0.0013	True	−0.0004	True
8	16	−0.4628	True	−0.3566	True	−0.0013	True	−0.0007	True
8	24	−0.1575	True	−0.0666	True	−0.0011	True	−0.0011	True
16	24	0.3053	True	0.29	True	0.0001	False	−0.0004	True

Table 5.4: The results for Tukey HSD test. The True for rejection means that the difference between two groups is significant. The 'meandiff' columns present the difference $\mu_{G2} - \mu_{G1}$.

Problem	2 dimensions	3 dimensions	8 dimensions	16 dimensions	24 dimensions
Height, best fitness values	2	3	1	5	4
Height, average fitness values	4	5	1	3	2
Velocity, best fitness values	5	1	2	4	3
Velocity, average fitness values	5	2	1	3	4

Table 5.5: The ranks for every number of dimensions for the *fH* encoding.

much smaller than the threshold 0.001, which leaded to the rejection of the null hypothesis – this meant that the one or more populations differed significantly.

This results proved that there were some differences in distributions of fitness values for different numbers of dimensions for every experiment in both best and average cases. Because of that, the next thing that required checking was the verification how do those encodings differ in comparison to each other. For this task the Tukey HSD test was used. The results of running this test are presented in Table 5.4. Except for the situation, in which the hypothesis that the results for two- and three-dimensional vectors are similar could not be rejected, there were no visible trends in the statistics.

In the end, the ranks for a given number of dimensions were computed for this experiments, based on their mean values. It can be seen that the 8-dimensional version of the *fH* encoding performed the best for most of the cases. This may suggest that this number of dimensions should be chosen. On the other hand, the dominance was not this big in comparison to other numbers of dimensions – from the evolutionary and computationally point of view all of the *fH* versions used were performing well, and the solutions for three-dimensional *fH* can be slightly faster developed, which enables the evolution to perform more evaluations in the same amount of time.

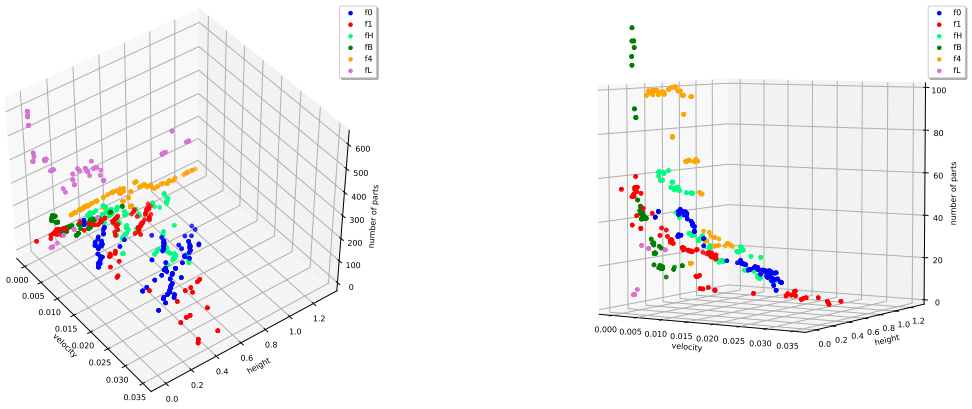
5.4 Three-criteria optimization

The last experiment performed for the representations in this thesis involved three-criteria optimization with use of a *NSGA-II* algorithm. The criteria were following:

- Maximization of number of parts,
- Maximization of the velocity of the creature,
- Maximization of the height of the creature.

The experiment was performed with population size equal to 100, with allowed neural network development, and 35 000 evaluations for the single evolutionary run.

The results for a single run for the every encoding are presented in Figure 5.17. The *fL* encoding managed to evolve very big creatures – while the structures evolved with other encodings had about



(a) Number of parts in full range.

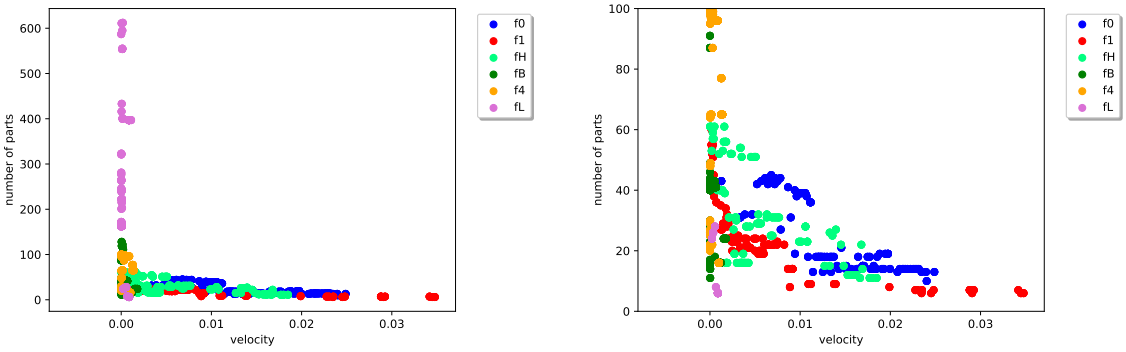
(b) Limited to up to 100 parts.

Figure 5.17: The three-dimensional plots representing the best solutions for every representation for a single line for the multi-criteria problem.

40-70 parts in average, depending on the representation, the *fL* encoding was able to easily evolve structures consisting of 300-600 parts. This is why there are two versions of plots for this problem – one without limit and one with limited number of parts up to 100 parts. The same procedure was performed with all plots in Figure 5.18, where the "number of parts" criterion was involved. As it can be seen in Figure 5.18a, the *fL* encoding was able to evolve large structures, but it acted poorly for the task of maximizing the velocity comparing to other representations. The *fL* encoding was also able to evolve quite high structures. The *f0* representation and *f1* representation were evolving the fastest creatures, as it is shown in Figure 5.18e. The most "universal" encoding was the *fH* encoding – it was evolving structures for all criteria at the same time. It was not as good as the *f0* encoding for the velocity criterion or the *fL* encoding for the number of parts criterion, but it was performing well in all criteria.

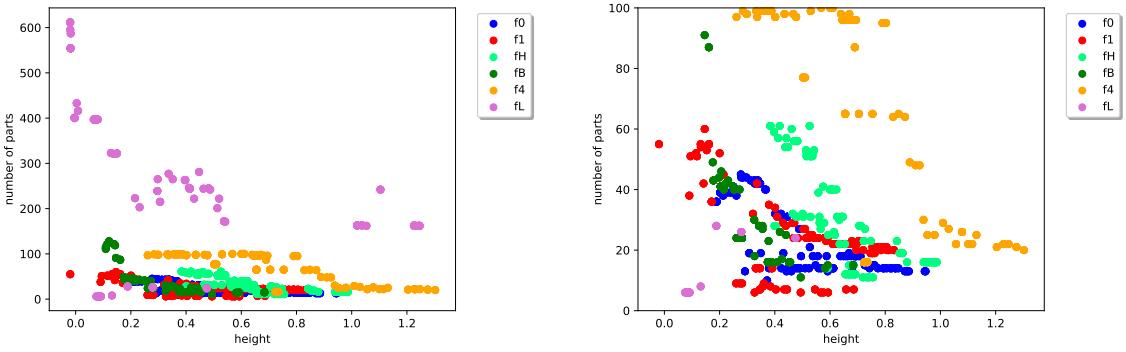
The final visualization for the NSGA-II experiment is shown in Figure 5.19. This plot represents how many times the solutions of the representation in the row dominated the solutions from the representation in the column. The darker the (i, j) cell, the more times the i-th encoding dominated j-th encoding. As it can be seen, the mostly dominated representation for this task was the *fB* representation – the evolution process was probably too short for this representation to evolve more "defensible" solutions. The *fH* encoding, on the other hand, had the most dominating solutions – this shows that despite the highly limited subspace of the available phenotypes the representation can create structures that are very good and universal for most of the cases. There were no other significant observations for this experiment.

The visualizations of some examples of solutions evolved for this task are shown in Figure 5.20. The *f0* representation mostly focused on creating jumping creatures with heavy arms that were looking like "shovels". The *f1* encoding was creating usually simple crawlers in order to maximize velocity. Those crawlers were later evolving to structures, where the "crawling arm" was positioned vertically, and it acted like an active tower on a massive star structure.The *f4* and *fB* encodings were creating creatures looking like tufts of grass, with several straight vertical sticks. Both of them were poorly evolving moving creatures, as well as the *fL* encoding. The *fL* encoding was creating the biggest structures – they were usually a very long cylinders with wide diameter. The



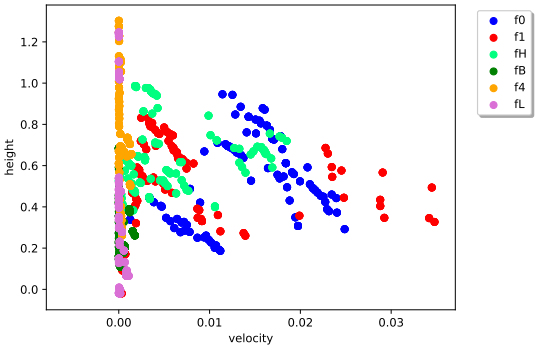
(a) Velocity vs number of parts.

(b) Velocity vs number of parts, limited to up to 100 parts.



(c) Height vs number of parts.

(d) Height vs number of parts, limited to up to 100 parts.



(e) Velocity vs height.

Figure 5.18: The projected results for every pair of criteria for the original three-criteria problem.

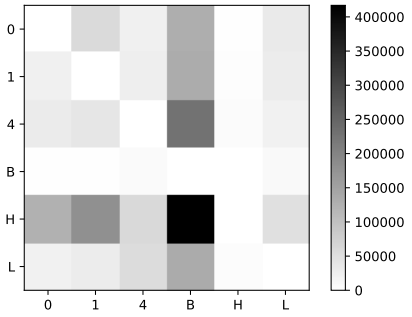


Figure 5.19: The pairwise comparison of the solutions between every encoding for the three-criteria experiment. Each representation had approximately 1 000 solutions, from 10 evolutionary runs with population size of 100. The (i, j) cell represents how many times the solutions of the i-th representation dominated the solutions of the j-th representation.

structure also had usually flat "floor", which was making such structure very stable. The rarer cases were the tufts of grass or long sticks with many branches.

The widest variety of creatures could be observed in the *fH* encoding, as could be deduced from the above results. It usually combined the height criterion with velocity criterion or the number of parts criterion. There were no visible examples of combining velocity with number of parts.

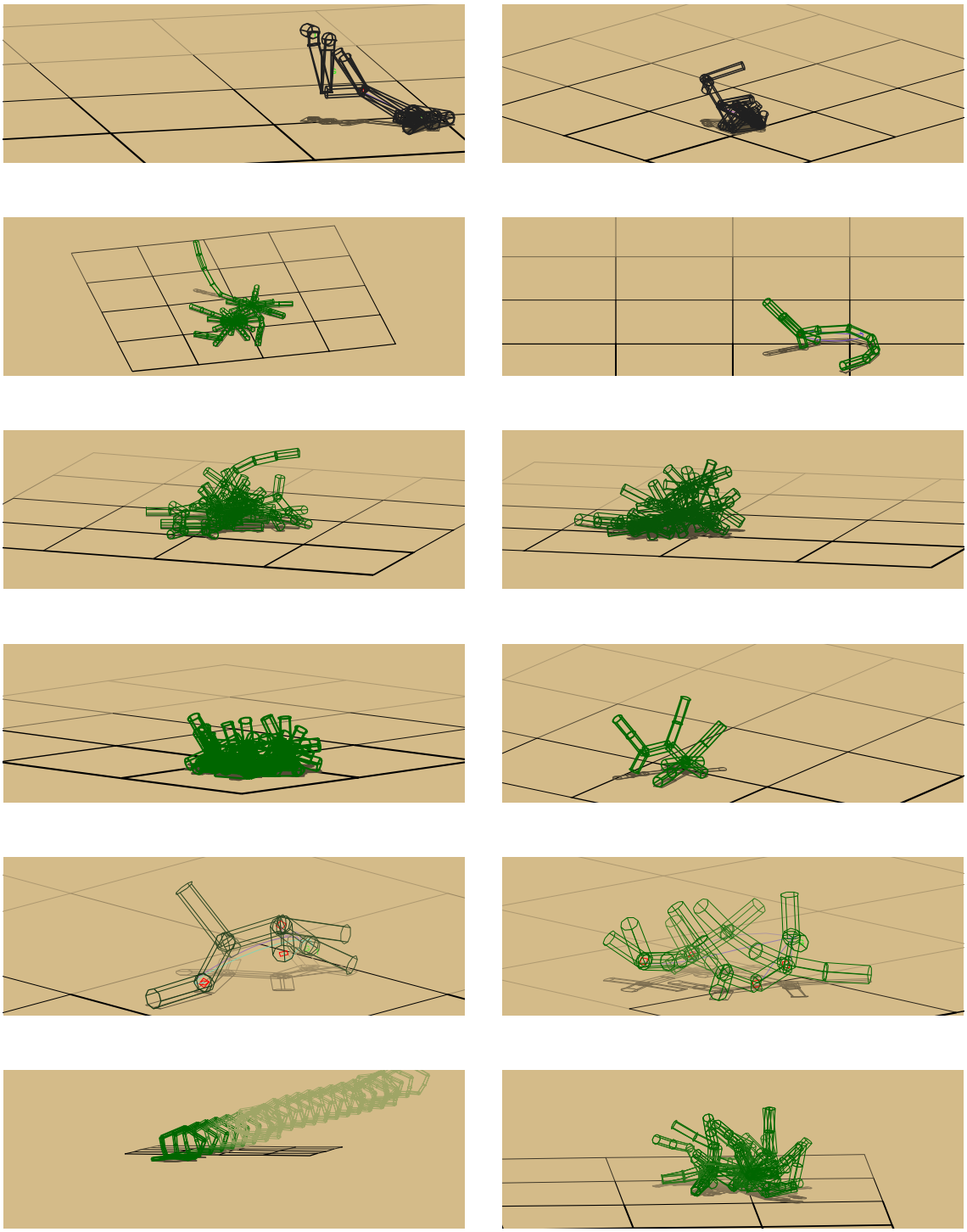


Figure 5.20: The example solutions for the three-criteria problem for every representation. Pictures in each row represent *f0, f1, f4, fB, fH, fL* encodings, respectively.

Chapter 6

Conclusions

The main goals of this master thesis were to develop and improve some of the existing genetic representations in Framsticks, to introduce and implement a new genetic representation, and to perform experiments with all these genetic representations for various optimization problems.

6.1 Summary of the development of genetic encodings

In the beginning, the *f4* encoding was improved to support all neuron classes both in syntax and in genetic operators, in order to make this representation applicable for various active optimization problems. Secondly, the *similarity encoding* (*fH*) was reimplemented. The reimplementation covered the changes in developing the body structure to make it more three-dimensional, and the changes in genetic mutations in order to smooth the numerical modifications. On top of the implemented *fH* encoding the successor of the biological (*f3*) encoding was developed. The *fB* encoding is different from the *f3* encoding in a few aspects – the conversion of letters to neural connection weights, the embedding of the neuron classes within the genotype, and the substitution and crossover operators.

In the end, the new generative encoding, called *fL*, was introduced. It is based mostly on the idea of the *Timed DOL-Systems* – it allows making continuous changes to the phenotype from the genotype level. The operators for this representation were designed to provide possibly minimal changes to the phenotype in order to make the evolution process more stable. In comparison to the existing generative encoding in Framsticks (*f8*), this encoding supports neuron classes and enables changing the properties of creature's components on the numerical level, while the *f8* encoding is limited by the *f1* grammar.

From the technical point of view, the development of those encodings was involving the implementation, backward-compatibility verification (for the *f4* encoding), testing, memory-leak checking, and documentation of the code.

During the experimentation process some modifications were also applied from the observations of the initial results of evolutionary experiments.

6.2 Summary of the experiments

As it can be seen from the experiments described earlier, there are no definite winning representations that outperform all other representations in all experiments. This shows that the task of designing and using the genetic representation for a given problem is complex and requires careful analysis of the nature of the problem.

The experiment of evolving the fastest agent showed that the direct encoding can outperform the other encodings, and there is no need to use more sophisticated representations, especially generative ones. The ranges of achieved fitness values for the *f0* encoding were usually wide in comparison to other encodings, but the average and median results were in general worse for the direct encoding, especially for the evolution of the complex structures. This encoding, especially for the task of evolving tall agents, was making use of its ability to evolve loops. Those loops allowed creating very stable basis. Other representations used in the experiments were forced to find another way of creating stable high structures.

The *fH* and *fB* representations proved in the above experiments that they are able to evolve better solutions than the *f2* and *f3* encodings. This supports the idea that the development process was successful. The *fH* encoding, as it was mentioned earlier, has one great disadvantage – it has highly reduced subspace of the mappable solutions, due to the limited implicit development process. On the other hand, the experiments showed that this limitation is not problematic for this representation. What is more, the *fH* encoding was able to outperform the other solutions for some problems. It was especially visible in the active tasks, involving neural networks. The solutions generated with this representation were dominating large number of solutions introduced by other representations. This representation was evolving especially good solutions in comparison to other encodings for the small population size. The same observation could be performed for the *f0* encoding.

The *fB* encoding obtained much worse results in comparison to its underlying representation – the evolution process was slow. This was the consequence of the mutation and crossover changes in this representation. The advantage of this representation is its robustness to stagnation. When the other representations were staying in the local optima for the rest of the evolutionary process during experiments, the *fB* encoding was able after some time to leave such local optima and proceed with improvement of its solutions.

The *f4* encoding performed poorly for the small population sizes – it was evolving slowly and inefficiently. For the big population size it was able to outperform other representations for some of the experiments. The main advantage of this representation is the re-usability of the elements of the genotype's tree. This allows the *f4* encoding to create complicated structures with repeatable components. This representation had often similar results to the *f1* encoding, possibly because both representations are sharing the same set of modifiers. This representation was acting poorly for the problem of evolving fast agents, when the population size was big, and performed much better for the small population size. This is the opposite of the situation visible in the evolution of the highest agent. In the plots, where all the fitnesses for all of evolutionary runs are presented, the *f4* encoding shows the greatest instability of the fitness value – this shows the destructive and unpredictable effects of operators in this encoding.

The newly introduced *fL* encoding yielded quite good results for the evolution of the highest agents. It was able to evolve interesting solutions, like L-shaped structures, pipes, and high towers on stable legs. It was able to use its main generative tool, that means rules, to evolve very complex solutions for the three-criteria experiment. The other advantage of this representation was the ability of performing small, but progressive changes during the process of the evolution. This showed that the designed operators were providing small changes to the phenotype. The first disadvantage of this representation was very poor results for the active task of evolving the fastest agent. The reasons for such results could be following: the loss of time caused by developing useless production rules that were not applicable for the task, and the problem of this representation to evolve usable neural networks. The second thing that can be considered as the disadvantage of this representation is the crossover operator that does not provide immediate changes in the phenotype,

and sometimes causes the growth of the genotype, where most of the rules in the genotype are not used.

To sum up, the most general observations from the performed experiments are:

- Every genetic representation has the problem of existing local optima, where the evolution can slow down, or even stop,
- There is no representation that would outperform other representations for some defined group of problems,
- The highly reduced phenotype space for the representation can be beneficial for some tasks,
- For some problems, like evolving the fastest creatures, simple structures are sufficient, and the direct encodings can perform better than the other, more complicated representations,
- The generative encodings can be useful, but they are not the leaders among all the representations,
- Various genetic representations can yield (relatively) different results depending on the population size,
- The use of the different number of dimensions for the *fH* encoding may affect the results,
- Every encoding can evolve interesting structures with different features thanks to the unique properties of the representation.

Some of above conclusions were already identified and reported in [KRV01].

6.3 Further work

This thesis demonstrated some advantages and disadvantages of genetic representations implemented in the Framsticks software. An interesting idea would be to introduce the ability to form loops in such encodings as the *fH* encoding. This encoding proved to perform very well despite the highest reduction of possible solutions among the tested experiments. It would be interesting to introduce some commands that could slightly affect the process of building the structures in such a way that the *fH* encoding could still perform as good as it did in this thesis.

The other representation that could benefit from an improvement is the *fL* encoding. The experiments proved that the evolution of the control systems for the *fL* encoding needs improvement in order to make this representation better for such problems as evolving the fastest agent. The second thing that can be considered for the *fL* encoding are additional genetic operators, or a slightly different crossover operator.

In the experimenation field, there are still a lot of tests and comparisons that can be performed to analyse the evolvability and the quality of solutions for different evolutionary design problems.

Appendix A

The content of the enclosed CD

The CD enclosed with this Master's thesis contains:

- A digital version of this document in PDF format,
- A digital version of this document as LaTeX files,
- Figures used for the thesis,
- Framsticks SDK with full source code,
- Files with simulation parameters used for experiments (*.sim* files that can be used in Framsticks software to set the experiment parameters).

The *frams-genotypes* directory contains the latest version of the Framsticks-SDK source files that were used to build the Framsticks API for experiments. The files that were modified or created for the purpose of this master thesis are:

- All scripts within *frams-genotypes/bash-tests* directory – they were created to test backward compatibility of the *f4* encoding, and the proper work of the developed encodings (the testing covered also valgrind verification for memory leaks and other problems with the execution of the program) [**?**],
- All files within *frams-genotypes/cpp/frams/genetics/f4* directory – they were modified in order to support features mentioned in Section 4.1.1,
- All files within *frams-genotypes/cpp/frams/genetics/fB* – they were created for the *fB* encoding,
- All files within *frams-genotypes/cpp/frams/genetics/fH* – they were created for the *fH* encoding,
- All files within *frams-genotypes/cpp/frams/genetics/fL* – they were created for the *fL* encoding,
- Some of the files in the *frams-genotypes/cpp/frams/genetics* were modified in order to support *fH, fB, and fL* encodings.

For the simulation files, the Framsticks API is required[1]. In order to run this simulation files for the *fL* encoding, one needs to set (in the "Parameters" window in GUI) the *Body disturbance* parameter in the *Experiment/Creature/Imperfection* to the value of 0.0001.

[1] Built with revision r809 or above of the *Framsticks SDK* (2018-06-29)

The bash scripts for testing the new encodings require test programs that can be built with Makefile. Running the *all_tests.bash* will execute all tests. In order to run them, *Valgrind* is required.

Bibliography

[Art15] Mary K Arthur. Point picking and distributing on the disc and sphere. Technical report, Army Research Laboratory, Aberdeen Proving Ground, 2015. URL: http://www.dtic.mil/dtic/tr/fulltext/u2/a626479.pdf.

[AW18] Herve Abdi and Lynne Williams. Tukey's honestly signiflcant difierence (hsd) test. 06 2018.

[BW97] Peter Bentley and Jonathan Wakefield. Generic evolutionary design. 01 1997.

[DB96] Frank Dellaert and Randall D. Beer. A developmental model for the evolution of complete autonomous agents. 1996.

[Dij61] Edsger W. Dijkstra. Algol 60 translation : An Algol 60 translator for the x1 and Making a translator for Algol 60. Technical Report 35, Mathematisch Centrum, Amsterdam, 1961. URL: http://www.cs.utexas.edu/users/EWD/MCReps/MR35.PDF.

[Egg97] Peter Eggenberger. Evolving morphologies of simulated 3d organisms based on differential gene expression. In *PROCEEDINGS OF THE FOURTH EUROPEAN CONFERENCE ON ARTIFICIAL LIFE, ECAL97*, pages 205–213. MIT Press, 1997.

[ES98] A. E. Eiben and C. A. Schippers. On evolutionary exploration and exploitation. *Fundam. Inf.*, 35(1-4):35–50, August 1998. URL: http://dl.acm.org/citation.cfm?id=297119.297124.

[GGRG85] John J. Grefenstette, Rajeev Gopal, Brian J. Rosmaita, and Dirk Van Gucht. Genetic algorithms for the traveling salesman problem. In *Proceedings of the 1st International Conference on Genetic Algorithms*, pages 160–168, Hillsdale, NJ, USA, 1985. L. Erlbaum Associates Inc. URL: http://dl.acm.org/citation.cfm?id=645511.657094.

[GHLL06] Al Globus, Greg Hornby, Derek Linden, and Jason Lohn. Automated antenna design with evolutionary algorithms, 2006.

[HK08] Maciej Hapke and Maciej Komosinski. Evolutionary design of interpretable fuzzy controllers. *Foundations of Computing and Decision Sciences*, 33(4):351–367, 2008. URL: http://www.framsticks.com/files/common/Komosinski_EvolveInterpretableFuzzy.pdf.

[HKW03] Maciej Hapke, Maciej Komosinski, and Dawid Waclawski. Application of evolutionarily optimized fuzzy controllers for virtual robots. In *Proceedings of the 7th Joint Conference on Information Sciences*, pages 1605–1608, North Carolina, USA, September 2003. Association for Intelligent Machinery. URL: http://www.framsticks.com/files/common/Komosinski_FuzzyControl_CINC2003.pdf.

[Hor03] Gregory Scott Hornby. *Generative Representations for Evolutionary Design Automation.* PhD thesis, Brandeis University, Waltham, MA, USA, 2003. AAI3073875.

[HP01] Gregory S. Hornby and Jordan B. Pollack. The advantages of generative grammatical encodings for physical design. In *Proceedings of the 2001 Congress on Evolutionary Computation CEC2001*, pages 600–607, COEX, World Trade Center, 159 Samseong-dong, Gangnam-gu, Seoul, Korea, 27-30 May 2001. IEEE Press. URL: http://www.demo.cs.brandeis.edu/papers/hornby_cec01.ps.

[HP02] Gregory S. Hornby and Jordan B. Pollack. Creating high-level components with a generative representation for body-brain evolution. *Artif. Life*, 8(3):223–246, August 2002. URL: http://dx.doi.org/10.1162/106454602320991837, doi:10.1162/106454602320991837.

[Kim14] Hae-Young Kim. Analysis of variance (anova) comparing means of more than two groups. 2 2014. URL: http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3916511/.

[KM18] Maciej Komosinski and Konrad Miazga. Tournament-based convection selection in evolutionary algorithms. *PPAM 2017 proceedings, Lecture Notes in Computer Science*, 10778:466–475, 2018. doi:10.1007/978-3-319-78054-2_44.

[KMTT16] Maciej Komosinski, Agnieszka Mensfelt, Paweł Topa, and Jarosław Tyszka. Application of a morphological similarity measure to the analysis of shell morphogenesis in Foraminifera. In Aleksandra Gruca, Agnieszka Brachman, Stanisław Kozielski, and Tadeusz Czachórski, editors, *Man–Machine Interactions 4*, volume 391 of *Advances in Intelligent Systems and Computing*, pages 215–224. Springer, 2016. URL: http://www.framsticks.com/files/common/ForaminiferaGenotypePhenotypeMapping.pdf, doi:10.1007/978-3-319-23437-3_18.

[Kom12] Maciej Komosinski. Evolutionary design of tall structures. Research report RA–06/12, Poznan University of Technology, Institute of Computing Science, 2012.

[Kom17] Maciej Komosinski. Applications of a similarity measure in the analysis of populations of 3D agents. *Journal of Computational Science*, 21:407–418, 2017. URL: http://dx.doi.org/10.1016/j.jocs.2016.10.004, doi:10.1016/j.jocs.2016.10.004.

[KRV01] Maciej Komosinski and Adam Rotaru-Varga. Comparison of different genotype encodings for simulated 3D agents. *Artificial Life Journal*, 7(4):395–418, Fall 2001. URL: http://dx.doi.org/10.1162/106454601317297022, doi:10.1162/106454601317297022.

[KU] Maciej Komosinski and Szymon Ulatowski. Framsticks SDK (Software Development Kit). URL: http://www.framsticks.com/sdk.

[KU04] Maciej Komosinski and Szymon Ulatowski. Genetic mappings in artificial genomes. *Theory in Biosciences*, 123(2):125–137, September 2004. URL: http://dx.doi.org/10.1016/j.thbio.2004.04.002, doi:10.1016/j.thbio.2004.04.002.

[KU18] Maciej Komosinski and Szymon Ulatowski. Framsticks web site, 2018. URL: http://www.framsticks.com.

[Lip13] David Lippman. *Math in Society*. 12 2013.

[PL96] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The Algorithmic Beauty of Plants*. Springer-Verlag, Berlin, Heidelberg, 1996.

[RG11] Noraini Mohd Razali and John Geraghty. Genetic algorithm performance with different selection strategies in solving tsp. 2011. URL: http://www.iaeng.org/publication/WCE2011/WCE2011_pp1134-1139.pdf.

[Ron97] Simon Ronald. Robust encodings in genetic algorithms: A survey of encoding issues. *Proceedings of 1997 IEEE International Conference on Evolutionary Computation (ICEC '97)*, pages 43 – 48, 05 1997.

[SK97] E. B. Saff and A. B. J. Kuijlaars. Distributing many points on a sphere. *The Mathematical Intelligencer*, 19(1):5–11, Dec 1997. URL: https://doi.org/10.1007/BF03024331, doi:10.1007/BF03024331.

[TM15] Danesh Tarapore and Jean-Baptiste Mouret. Evolvability signatures of generative encodings: Beyond standard performance benchmarks. *Information Sciences*, 313:43 – 61, 2015. URL: http://www.sciencedirect.com/science/article/pii/S002002551500211X, doi:https://doi.org/10.1016/j.ins.2015.03.046.

[Waj09] Maciej Wajcht. Reprezentacja generatywna w ewolucji konstrukcji. Master's thesis, Poznan University of Technology, Poznań, Poland, 2009.

Poznan University of Technology
Faculty of Computing
Institute of Computing Science

Typeset using LaTeX in Computer Modern.

BibTeX:

```
@mastersthesis{ key,
    author = "Grzegorz Latosiński",
    title = "{Development and comparison of genetic representations for evolving 3D structures}",
    school = "Poznan University of Technology",
    address = "Pozna{\'n}, Poland",
    year = "2018",
}
```