Institute of Parallel and Distributed Systems
University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3189

A hierarchical framework for classical and evolutionary robot control

Jochen Haag

Course of Study:	Informatik
Examiner:	Prof. Dr. rer. nat. habil. Paul Levi
Supervisor:	Dipl. Inf. Florian Schlachter
Commenced:	18.01.2011
Completed:	20.07.2011
CR-Classification:	C.2.2

In nature, every living creature evolves. Humans are an example for the power of evolution. But evolution takes a long time for complex Problems, 200000 Generations of thousands of individuals, from the first 'homo' to today. For complex tasks in evolutionary robotics this is much to long. In this diplom thesis I will use a hierarchical approach, breaking complex tasks down to planning and basic tasks. Then I will first evolve these basic tasks, making them available as parts for complex tasks.

Contents

List of Figures

List of Tables

Acknownledgements

I want to thank Prof. Levi for allowing me to work in his department. Special thanks for my supervisor Florian Schlachter, who always helped when problems occured. I also thank my family for enduring frustrated self-talking when I encountered another problem. Last but not least, I thank Michael Mattes, Andreas Barth, Patrick Alschbach and Benjamin Girault for good cooperation within the Symbrion project.

1 Introduction

1.1 Motivation

Evolution and learning are the principles that formed all living beings. The complexity of these principles is vast- there are many unknown facets, so that we are not able to describe the process analytically. We know that learning uses sensoric inputs to achieve optimisation of solutions for certain situations. Eating when hungry- in the morning when we are still tired, certain learned-by-hard solutions are running quite automatic. Starting the day, make and eat breakfast, became routine through learning. It is like a series of predefined actions- gather a plate, put a slice of bread on it, get some cheese and butter, ready the bread, eat. If it fails, e.g. the bread falls to the floor, we try to compensate the failed action and retry it. If all prerequisites are met, (e.g. bread on plate, butter and cheese on bread), we sense it and go to the next step of the plan. Of course some complex steps consist of substeps, there may be alternatives, (e.g. there ist no cheese, use salami,) and after too many failures we tend to give up. The abstract planning(, get bread,) and the execution(, tension of the different muscles,) are different, only sparse related things. This motivates the following approach: Instead of using one monolithic block that is able to do all the work, It may be better to have a hierarchical framework with a planning level and a execution level. The big andvantage is, that one will be able to optimize and reuse 'primitive' behaviour to get 'building blocks' for the solution of complex problems.

1.2 Structure of this Document

First, I will motivate, why my approach promises better performance and additional value, compared to the straight- forward approach. Then there will be some background information about the human brain and central nerve system, from which I got many ideas. Next Chapter is about controller approaches, namely Decision Trees, Finite State Machines and Neural Nets. I'll point out the difference of supervised, unsupervised and reinforced Learning. This is followed by a chapter about evolutionary Algorithms, expecially about EANT, which is the Basis for the Evolution in the Framework. At some of these chapters there will be small subsections with related approaches and work, where I will show some papers and other scientific work in comparation with mine, exspecially with focus on how it effected my own work.

The next block is about the Framework I designed and implemented, containing the resulting, real architecture, an overview over the classes and excerpts of important code segments. I will show how (and if) the abstraction of Sensors and and Actuators work and what problems the hierarchical evolution approach has. Thereafter the grand experiment is introduced, showing how the hierarchical modell may help putting together a complex behavior by training the parts. Apart from that, I will make experiments to find the best parameter values for the EANT algorithm. The thesis is concluded by the evaluation of the work done, the results and the failures, and some additional approaches for later works. For that, I focus on the key concepts: Platform independent sensors and actors, the hierarchical approach and the overall usability and performance of the framework.

2 The biological Model

The human brain is a very complex neural net. By the building artificial neural nets, we assume, that every human thought and conclusion can be duplicated by that mean.
So let's take a look on this biological prototype.

2.1 The neuron

One of the best books about the nervous system of humans, showing the key concepts, is 'Heinrich Reichelt, Neurobiologie, Thieme Stuttgart, 2000',[Rei00], which I will use in this and further chapters.

First of all, a neuron is a cell of its own rights: ribosomes, golgi apparatus, nucleus, etc. - it has every cell parts, that others have too. But of course, it is more. The structure of a neuron determines its functionality; There is a great variarity of structures. The Dendrites, branches of the cell membrane, get the input of the neuron. For that, they use so called neurotransmitter, molecules, that are able to cross the gap between nerve cells. Most neurotransmitter are quite simple, as acetylcholin and the infamous glutamat. There are inhibitory and exhibitory neurotransmitter, making the electrical potential smaller (inhibitory) or bigger(exhibitory) The cell membrane is also different to other cells- it can produce and carry electrical signals. The potential of this membrane is important- when it reaches a certain value, the neuron **fires**
For that, the axon carrys the electrical signal along it's path. Axons can be quite long- more than one meter is possible, so that the output of a neurons input summation can be used for anything else in the body.

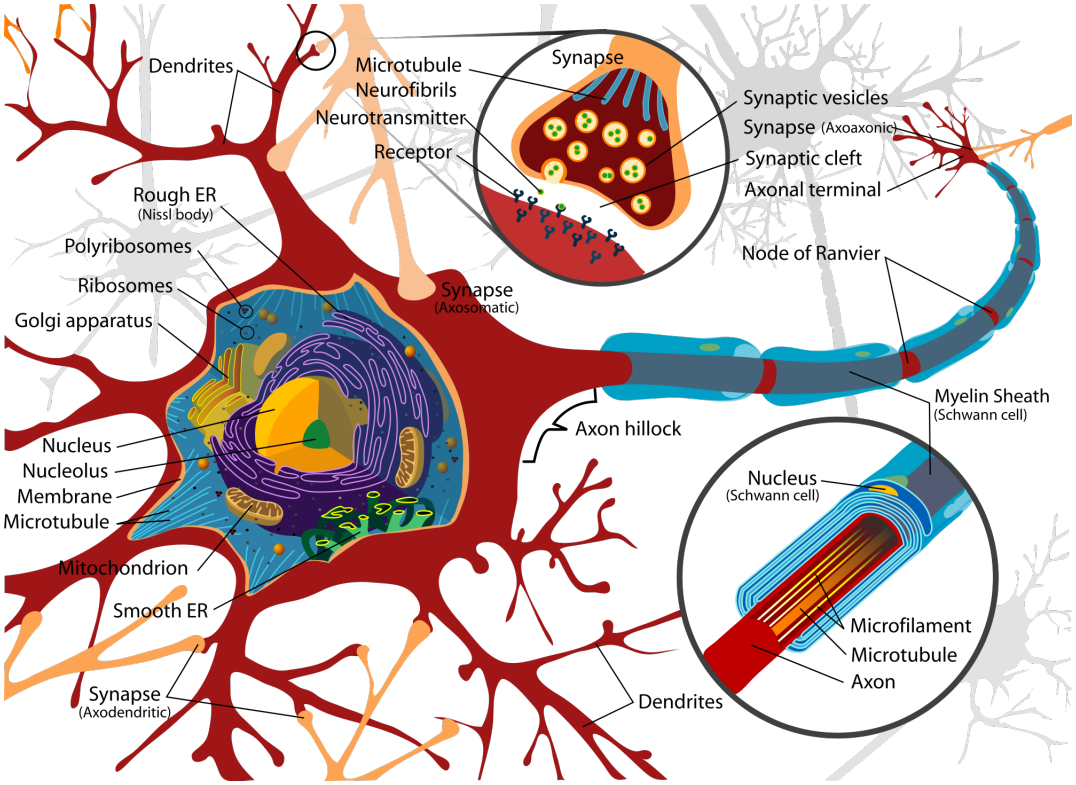


Figure 2.1: The schema of a human neuron cell; source: 'wiki commons'.

2.2 Cerebral planning and motor cortex

The parts of the human brain that are necessary for motorical calculations are organized hierarchically. Many motor reactions consist of quite complex units of action. Somehow the brain is able to coordinate running, swimming or hand motions, where dozends of muscles have to work in perfect harmony. There are two main theorys about the control of these complex behaviours: The hypothesis of peripheral controll, (there is a chain of reflexes, one leading to the next, combining a complex motor program,) and the therory of central control (, where a complex and rythmic activity are controlled by neural oscilators, aka. Central Pattern generators. For more information about this, read the diplom thesis of Andreas Barth: Evolutionary Design of Central Pattern Generators for Multi-Robot Organisms, 2010)

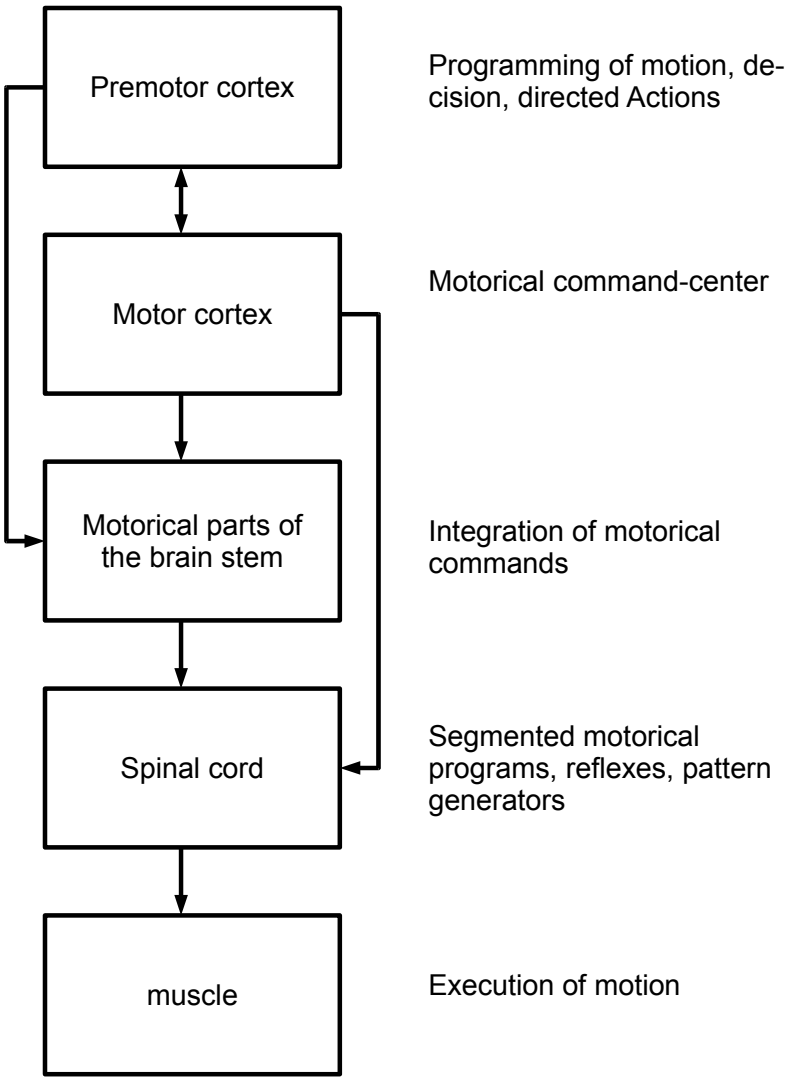


Figure 2.2: The hierarchies of motor planning. Translation from [Rei00].

The figure above shows the hierarchies in the cerebral planning. The 'will' of

a human is located in the premotor cortex- there, decisions are made and motion plans are selected. Making a motion and thinking of a motion activates the same neurons, but the progress of the motion is inhibited by a part of the brain stem. Of course the motion processing of a real human is not strictly hierarchical. 'Shortcuts', rethinking and much more is researched by neurobilologists.

2.3 Human learning

There are 4 main ways of learning or improving for humans:

- Habituation/Sensitization: Getting used to something or becoming more sensitive about it. This corresponds to unsupervized learning.
- Associative learning/Conditioning: The carrot and stick - principle. Reward for something 'good', punishment for something 'bad'. This corresponds to supervized learning.
- Observational learning: Learning by immitation of others behaviour or by playing (, experimenting,) and observing the results. This corresponds to reinforced learning.
- Evolution: Sucessful beings have children and they tend to be more sucessful than others. Evolution can improve Genes and therefore the basics of life. (They impose a limitation of possible fitness.)

2.4 Evolution as learning

Microorganisms or very primitive animals don't have a learning method for their own existence- behavior is innate and doesn't change. The only method for these beings to improve is evolution. Life on earth came from these primitive origins, so it is right to say, that evolution is the most basic learning algorithm.
It's quite likely that any other learning behaviour was created by evolution, providing in-time adaption, because the generation cycle got too long, and the cost for a generation became so high, that in-generation-adaption became a reasonable investment.
W.l.o.g. we can conclude, that all higher learning methods will eventually result from evolution, when the overhead becomes reasonable. So evolution is the most basic learning method and this is it's greatest advantage, even if its not as fast (and often much more dangerous) as the other methods.

3 Controller types and machine learning

When we want robots to perform a given task, there is the need to get information and decide between alternative actions. There are several ways to accomplish this. I'll give a overviev about a selection of some often used means to achieve this.

3.1 Decision Trees

Decision trees[Mat10] are an intuitive and easy to understand way for decision making. Implementation of a decision tree algorithm is also easy. Input is a given *situation*, where all *attibutes* of the decision tree have values. We start at the root of the tree, answer the questions of the node, follow the edge that stands for the result and continue with the associated subtree. If we reached a leaf, it contains the decision of the tree.

3.1.1 Decision tree example

This example is about deciding, what meal to cook for a couple of friends: Steak with salad, fried potatoes or lasagna. Attributes are:
•Vegetarians? (all/few/none)
•Allergy against lactose? (yes/no)
•How much time until they arrive? (<30 minutes/>30 minutes)

This leads to the following decision tree:

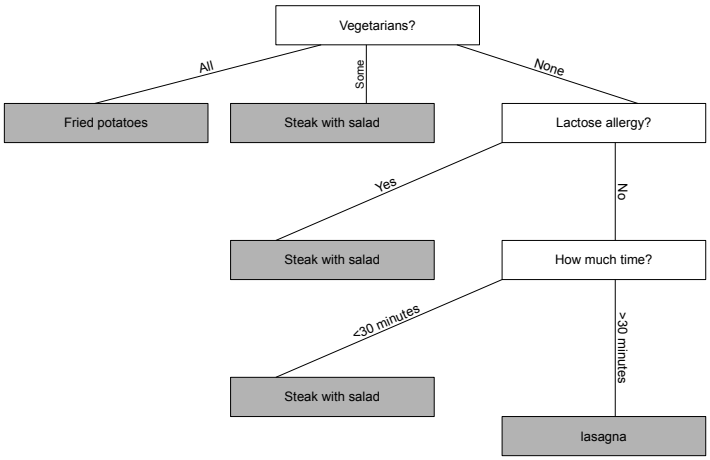


Figure 3.1: Example for an decision tree.

3.2 Finite State Machine

A Finite State Machine is a common mean for artificial intelligence. It is also used in compilers for syntax-checks and is a basic model for many other derived models. There are two main groups of State Machines: Acceptors (,used for detecting regular languages) and *Moore* or *Mealy machines*, used for controlling robots or agents, because they have output functions.
The formal definition of Moore and Mealy machines is both a sixtuple with the difference, that the output of Moore machines is created by the states- the output of Mealy machines by the transitions.

Definition: A state machine M is defined by the tuple

$$M = (S, s_0, \Sigma, \Delta, \delta, \lambda)$$

where
S: finite, non-empty set of states
s_0: start state ($s_0 \epsilon S$)
Σ : finite input alphabet
Δ : finite output alphabet
δ : transition function (δ : S x $\Sigma \rightarrow S$)
λ : output function (Moore: S $\rightarrow \Delta$, Mealy: S x $\Sigma \rightarrow \Delta$)

Finite State Machines are often designed by hand, but there are also methods for reinforced learning and evolutionary algorithms.[Kön07] The Finite State Machine eighter uses polling on existing world data actualized by the sensors or is event-driven by the arrival of a data set.

3.2.1 Example

[Mat10] A swarm of mini robots explores a turbine avoiding obstacles (walls and other robots) searching for blades and then inspecting found blades. A robot is only allowed to leave a blade at its tip. The picture below 3.2 shows the environment of the robots, simulated as well as real. Optionally, robots can serve as a beacon for other robots by remaining at the tip of a blade for some time. Another robot seeing a robot at the tip of a blade knows that blade was already inspected and can avoid it, thus saving time.

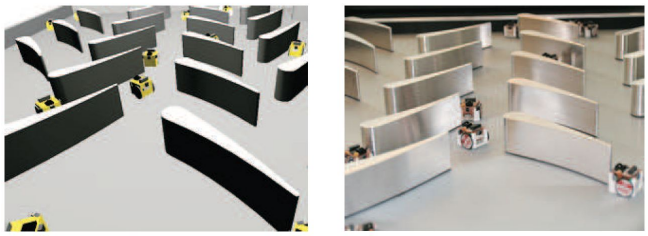


Figure 3.2: Left: Overviev of the setup in simulation, Right: Overview over setup with real robots.

The fintie state machine for the high level controll looks like the following [CM06]:

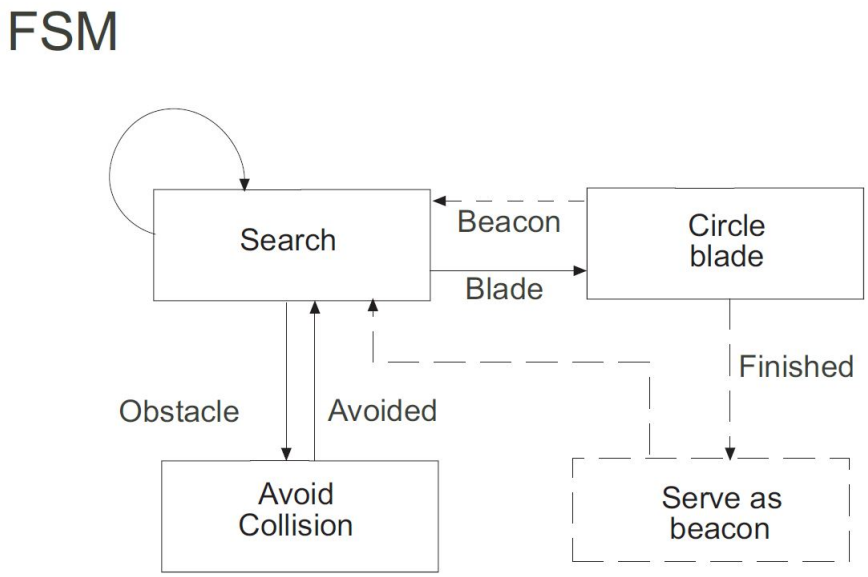


Figure 3.3: The high-level behavioral flowchart of the robot controller as a Finite State Machine.

3.3 Petri Nets

Petri nets are directed bipartite graphs, in which one of the node groups represent transitions, the other places. They can be used for many applications, varying from the modelling of state machines to the simulation of workflow applications.

3.3.1 Formal Definition

[Pet81]
A Petri net graph is a 3-tuple (S,T,W), where

- S is a finite set of places
- T is a finite set of transitions
- S and T are disjoint, i.e. no object can be both a place and a transition
- W: $(S \times T) \cup (T \times S) \rightarrow \mathbb{N}$ is a multiset of arcs, i.e. it defines arcs and assigns to each arc a non-negative integer arc multiplicity; note that because of bipartite no arc may connect two places or two transitions.

The flow relation is the set of arcs: $F = \{(x, y) \mid W(x, y) > 0\}$. In many textbooks, arcs can only have multiplicity 1, and they often define Petri nets using F instead of W. A Petri net graph is a bipartite multidigraph (S ∪ T, F) with node partitions S and T. The preset of a transition t is the set of its input places: ${}^\bullet t = \{s \in S \mid W(s, t) > 0\}$; its postset is the set of its output places: $t^\bullet = \{s \in S \mid W(t, s) > 0\}$. Definitions of pre- and postsets of places are analogous. A marking of a Petri net (graph) is a multiset of its places, i.e., a mapping M: S→ ℕ. We say the marking assigns to each place a number of tokens. A Petri net (called marked Petri net by some, see above) is a 4-tuple (S,T,W,M_0), where

- (S,T,W) is a Petri net graph
- M_0 is the initial marking, a marking of the Petri net graph.

3.3.2 State machines and Petri nets (PN)

[ea09] Petri nets can be used to model State Machines. Because of that, it is a mighty tool for the moddeling of behaviour, when we extend the model in a way that transitions have certain conditions in whitch they are fireing. This will be later used as the model for the high level part in the implementation of the hierarchical framework.

3.4 Statistical analysis

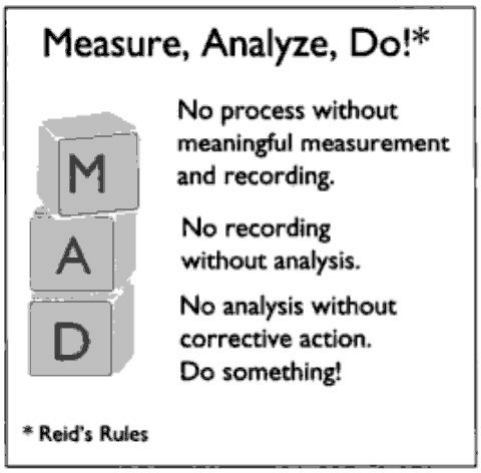


Figure 3.4: [HJH98] Measure, Analyse, Do: The three steps for statistical analysis.

In Statistical data analysis, there are three main steps: Measure, Analyse, Do. The mesurement contains data collection, the analysis the allocation of semantic to that. This means: Measurement -> Data -> Information. [HJH98] The use of information together with expertise generates knowledge. Knowledge provides the basis for action. It answers the Question of how something happens.
The steps from Data to Information and from Information to Knowledge, are highly non-trivial. The methods of statistical analysis are beyond the scope of this thesis- just note, they exist.

3.5 Artificial Neural Nets

Artificial Neural Networks (ANNs) are an bio- inspired mean for decision making. The ANN is modeled as a graph, where *Neurons* are modeled as *nodes*, and *dentrides* and *axons* are modeled as *vertices*. The vertices carry *weights*, the nodes have *functions* assigned to. The weights and function parameters are the most important parameters for any ANN.

3.5.1 Definition

A ANN is a 8-Tupel (N,V,I,O,P,F,f where:

- N: set of Neurons
- V: set of Vertices
- I: ϵ N, Input Nodes, they are set from outside
- O: ϵ N, I and O are disjunct. Output Nodes, they contain the results
- P: ϵ V x $\mathbb{R}$ Weight- parameter, associated with Vertices
- F: ϵ N x $\mathbb{R}^k$ Parameter for functions associated with Nodes
- f: set of activation functions associated with Nodes

3.5.2 Activation functions and their parameters

The figure below shows the most common activation functions for neurons. The one that is nearest to the biological archetype is the logistic function. The most simple one is the Identity- for the EANT- implementation I'll use this function. Even if most other implementations use other activation functions, I'll use this because in the papers about EANT, this is also the choise of it's creator.

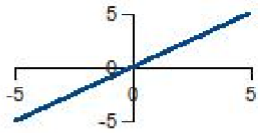
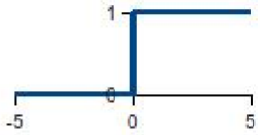
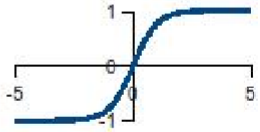
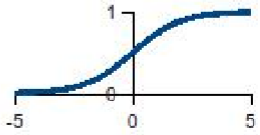
Identity	$y = net$	
Binary Function	$y = \begin{cases} 1, \text{ if } net > 0 \\ 0, \text{ else} \end{cases}$	
Tangens hyperbolicus	$y = \tanh(net)$	
Logistic function	$\frac{1}{1+e^{-k*net}}, k = 1$	

Figure 3.5: [Mat10] Common activation functions for ANN's.

The Output of an neuron is: Output = W_F * F(Σ $I_0, I_1, ...I_n$)
where:
W_F is the associated weight
$I_0, I_1, ...I_n$ are the input values

3.5.3 The simplest ANN's: Perceptrons

Presented by Mc- Culloch and Pits in 1943 [McC43], they constist only of a single output neuron and its inputs. The parameters are: the weights P of the output neurons inputs, the function and its parameters.
When we add a new layer to the perceptron, consisting of a number of neurons that take the original input and whose outputs are the inputs of the original layer: This method changes the depth of the perceptron, making it multi layer. The layers between input layer and output layer are called hidden layers.

3.5.4 Artifical Neural Nets as Robot Controllers

[Ant90, The use of neural networks in control systems can be seen as a natural step in the evolution of control methodology to meet new challenges. Looking back. the evolution in the control area has been fueled by three major needs:

- The need to deal with increasingly complex systems,
- The need to accomplish increasingly demanding design requirements,
- And the need to attain these requirements with less precise advanced knowledge of the plant and environment -that is, the need to control under increased uncertainty.

Today, the need to control, in a better way, increasingly complex dynamical systems under significant uncertainty has led to a reevaluation of the conventional control methods, and it has made the need for new methods quite apparent. It has also led to a more general concept of control, one that includes higher-level decision making, planning, and learning, which are capabilities necessary when higher degrees of system autonomy are desirable.
]

As Antsaklis [Ant90] wrote, there are challenges in controll systems, which demand a highly flexibility. Artificial Neural Nets can provide this flexibility, but also have problems, other solutions don't have. It still has to be seen, if ANN's become widely accepted as robot controllers. The hierarchical Framework makes a try to combine the two main approaches: Artificial Neural Nets and State Machines.

3.6 Supervised Learning

'Supervised lerarning' means the neural net is trained by a trainer with a set of training data, minimizing the error for each of this taining sets by the external trainer. To get the right weight parameters there are several possibilities. The most prominent of these is the Delta Rule and others which are derived from it.

[Mat10] The *Delta Rule* is a learning method for one-layered networks. It minimizes equation **??**. First the output of neuron j for input vector x of pattern μ,

$$y_i^\mu = \Phi\left(\sum_{j=0}^{n-1} w_{ij} x_j\right) \tag{3.1}$$

is computed. After obtaining the output of neuron i the *delta error* for this neuron can be determined for pattern μ:

$$\delta_i^\mu = \dot{\Phi}\left(\sum_{j=0}^{n-1} w_{ij}x_j\right)\left(t_i^\mu - y_i^\mu\right). \tag{3.2}$$

$\dot{\Phi}$ is the derivate of the neuron's activation function Φ. Afer determining the delta error

$$\Delta w_{ij}^\mu = \delta_i^\mu x_j^\mu \tag{3.3}$$

is added to all connection weights.
The Delta Rule algorithm can be extended for multi-layered ANN's (, the method is called *Backpropagation of error*) and for recurrent neural nets (Backpropagation trough time).

3.7 Unsupervised Learning

Unsupervised learning methods like Hebb's Rule don't need an error function. They rely on input patterns only, connection weights are updated self-organized. The underlying principle is the use of similarity. In Hebb's Rule connection weights are updated if both connected neurons are active simultaneously. The basic formula is [Mat10]

$$\Delta w_{ij} = \eta x_i y_j \tag{3.4}$$

where x_i are y_i are the activation values of the connection's source i and its target j and η is the learning rate. Statistical learning methods like most clustering algorithms also use unsupervised learning.

3.8 Reinforcement Learning

This learning method, inspired by behaviorist psychology, uses feedback from the surrounding area to learn. Mostly used, when it is not possible to get good training data, it rewards the **results** that are wanted, punishing unwanted results. (This is the same as in evolutionary algorithms.) In reinforced learning there is always a tradeoff: Using old, reliable tactics to get a average result or trying something new, risking total failure. This tradeoff (, again like evolutionary algorithms,) must be balanced carefully for an agent to succeed in his task.

The learning model consists of: The reinforcement learning model consists of:

- a environment E;

- the observed environment $E_0 \epsilon$ E
- a set of possible actions;
- a set of rules to calculate reward/punishment;

4 Evolutionary algorithms

The following sources were used for this chapter; citations are mentioned in addition:

- Prof. Claus, University of Stuttgart, lecture: 'Evolutionäre Algorithmen' [Cla07]
- Schwefel, H. P. and Kursawe, F.: Künstliche Evolution als Modell für natürliche Intelligenz. [SK]
- http://www.geatbx.com/docu/algindex.html, 2.5.2011. [Poh11]

When we use evolutionary algorithms, we adapt one of the strongest principles of nature. Every living beeing we know originates from evolution processes. The adaption of this principles should be able to solve all our problems, enough calculation power provided. That is not true though- there are optimation problems, e.g. the famous 'Traveling Salesman Problem' where evolutionary algorithms provide very good solutions, but at the bulk of problems, there's still no 'evolutionary strategy' that is good enough. There are three main problems, that need to be solved, before a evolutionary algorithm can work:

- The **encoding** of the problem- a bidirectional function between the data of the problem and the data used for, and fitting for, evolution.
- **The Fitness - Function.** A function, that evaluates the solutions, created by the evolutionary algorithm.
- **The bootstrap problem.** For a working evolution, it is necessary to have a number of 'somewhat working solutions', that means we need a good collection of 'bad solutions' that solve the problem- then the evolution can optimize them.

Because at most cases it is not possible to solve all these problems, artificial evolution doesn't work then.

4.1 The evolution cycle

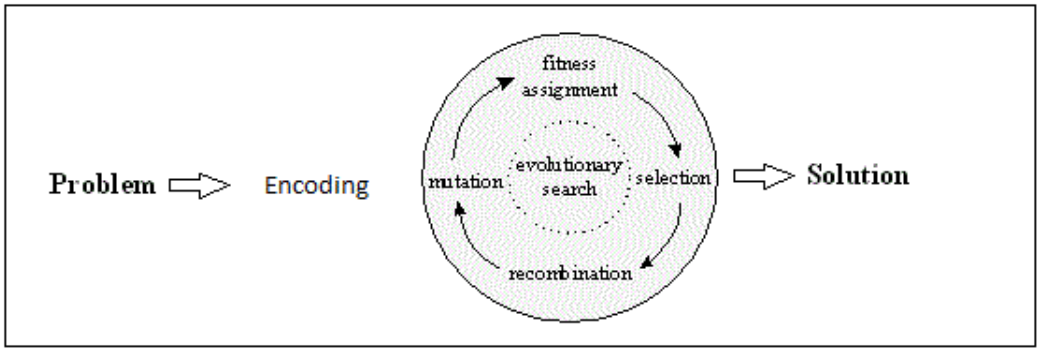


Figure 4.1: [Poh11] The evolutionary cycle.

The core of any evolutionary algorithm is deduced from the evolutionary cycle. A Problem is **encoded**, a **population** of solutions is generated, (see bootstrap problem above,) the **fitness-function** is created, then the evolutionary circle starts working:

4.1.1 Mutation

By Mutation an EA is able to find new solutions. Different algorithms use different mutation operators, roughly divided into structural or parametric operators. Parametric mutations changes the weights and parameters of the given structure. Structural operators on the other hand modify the topology, the structure of a solution adding or deleting some parts. Not all algorithms use all of those operators. Mutation is the power source of evolution- every innovation originates from mutation.

4.1.2 Crossover

The combination of two 'genomes' is called crossover. When applying a crossover, we hope that the best of the parents is decended to the child. Crossover is the key to transfer good innovations to other genomes. Technically, in crossover parts of the string encoding of two solutions are swapped to create a new solution. Depending on the number of breaking points (n), it is called a n-Point crossover.

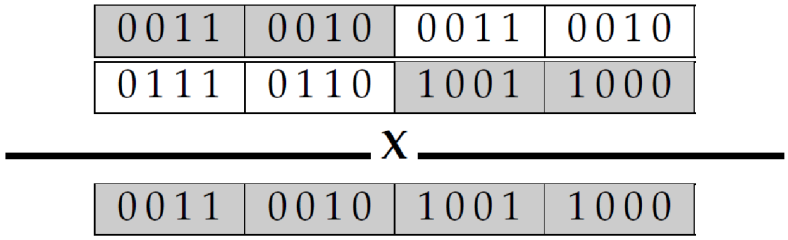


Figure 4.2: [Mat10] An example for a crossover operation between two genetic encodings.

4.1.3 Fitness Function and Selection

The fitness function is a non-trivial, important part of evolutionary processes. They reward a solution for accomplishing the set tasks and thereby provide a selection criteria. There are several problems in creating a fitness function. First common problem is that at complex tasks all solutions may have zero fitness, because no starting genome is able to solve even a bit of it. This leads to the bootstrap problem mentioned above. The bootstrap problem prevents EAs from selecting promising solutions and the search for solutions lacks direction. Technically the problem is covered by the fact that the fitness function is seldom strict monotone. The second common problem of fitness functions is, that they are often quite 'noisy'. The infamous butterfly-effect can occur, so that the same genome produces two totally different fitness results, even when only minor changes were made. Re-evaluation is the key for preventing this effect- but this greatly diminishes the speed of evolution, as re-evaluation takes time.

4.1.4 Schwefel and Kursawe: "On Natural Life's Tricks to Survive and Evolve", and others

In many papers related to evolutionary algorithms Schwefel contributed many algorithms and their analysis. In the paper mentioned in the head, he answers the question about the fundamental principles of life: evolution is a substancial part of it. For my work, I adopted this view: evolution as the driving part of life, as the mechanism to achieve the best possible results, given a specific challenge.

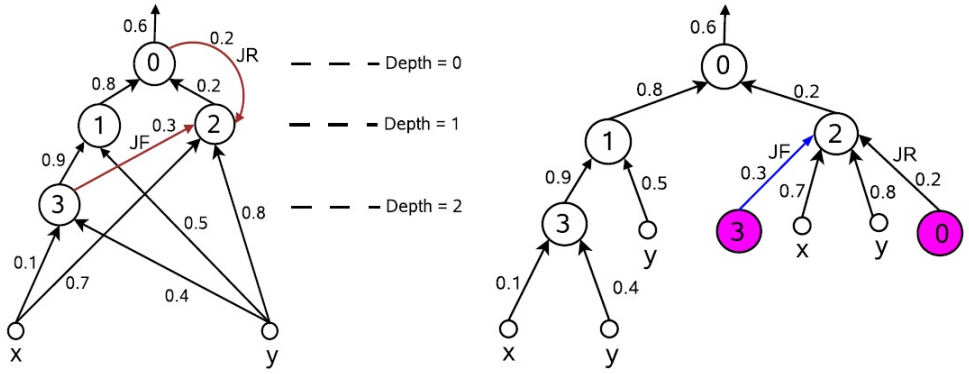
4.2 EANT

Evolutionary Acquisition of Neural Topologies (EANT) [Kir09] is a evolutionary algorithm used to evolve the structures and weights of recurrent neural networks.

It will be the core of the framework, using the Compact Genetic Encoding (CGE) to provide a very fast implementation which evaluates in linear time.

4.2.1 Compact Genetic Encoding (CGE)

The common genetic encoding allows evaluation of a neural net in linear time. The figure below shows how a net is encoded. In the linear genome, N stands for a neuron, I for an input to the neural network, JF for a forward jumper connection, and JR for a recurrent jumper connection. The numbers beside N represent the global identification numbers of the neurons, showing how many inputs the neuron has, and x or y represent the labels of the inputs encoded by the input genes.



Allele + ID	N 0	N 1	N 3	I x	I y	I y	N 2	JF 3	I x	I y	JR 0
Weight	0.6	0.8	0.9	0.1	0.4	0.5	0.2	0.3	0.7	0.8	0.2
Number of inputs	2	2	2	–	–	–	4	–	–	–	–

(c)

Figure 4.3: [Kir09] An example of encoding a neural network using a CGE. (a) The neural network to be encoded, with one forward and one recurrent jumper connection. (b) The neural network interpreted as a tree structure, where the jumper connections are considered as terminals. (c) The linear genome encoding the neural network shown in (a).

4.2.2 Evolutionary Operators

Three operators are defined on CGE encoded neural networks:

- Parametric mutation: it chances the weights of the network.

- Structural mutation: it changes the network structure by adding or removing genes.
- Structural crossover: it takes two genomes and merges them to a new genome.

Parametric Mutation

Parametric Mutation of EANT [Mat10] changes the weight of each gene in the genome. To get all weight updates, the parameters τ and τ' have to be calulated first:

$$\tau' = \frac{1}{\sqrt{2n}} \tag{4.1}$$

and

$$\tau = \frac{1}{\sqrt{2\sqrt{n}}} \, . \tag{4.2}$$

With that the adjusted learning rate σ_i' can be calculated:

$$\sigma_i' = \sigma_i e^{\tau' N(0,1) + \tau N(0,1)} \, . \tag{4.3}$$

If this adjusted learning rate is lower than a threshold e_0 it is set to the threshold value:

$$\sigma_i' < \epsilon_0 \Rightarrow \sigma_i' = \epsilon_0 \, . \tag{4.4}$$

Finally the weight update can be caluclated:

$$w_i' = w_i + \sigma_i' N_i(0,1) \qquad 0 \leq i < n \, . \tag{4.5}$$

$N(0,1)$ is a gaussian distribution of zero mean and unity standard devation and n is the number of genes in the genome.

Structural Mutation

There are three types of structural mutations in EANT:

- adding a recurrent or forward jumper gene,
- removing a recurrent or forward jumper gene
- adding a subnetwork.

Each neuron gene can have one of these changes. Which one is actually performed is determined by randomizing with equal possibility for each alternative.

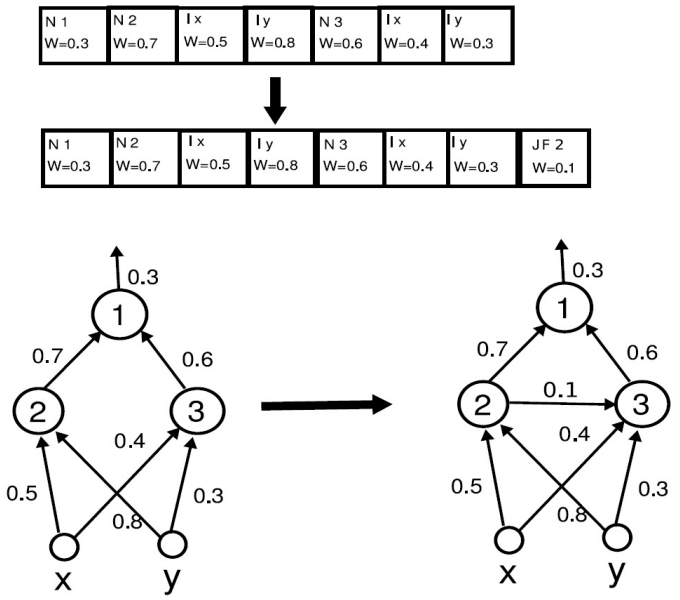


Figure 4.4: [Kir09]An example of structural mutation. A forward jumper gene connecting the node N2 with the node N3 is added and the number of inputs of N3 is increased by one.

Structural Crossover

The EANT-Structural Crossover merges two genomes to a new genome. This new genome is the unification of the two parents; it contains every innovation which at least one of the parents have. When both parents have a gene in common, the weight of the child genome is the average weight of the parent genomes. The premise for the structural crossover to work is, that both parents have a common ancestor.

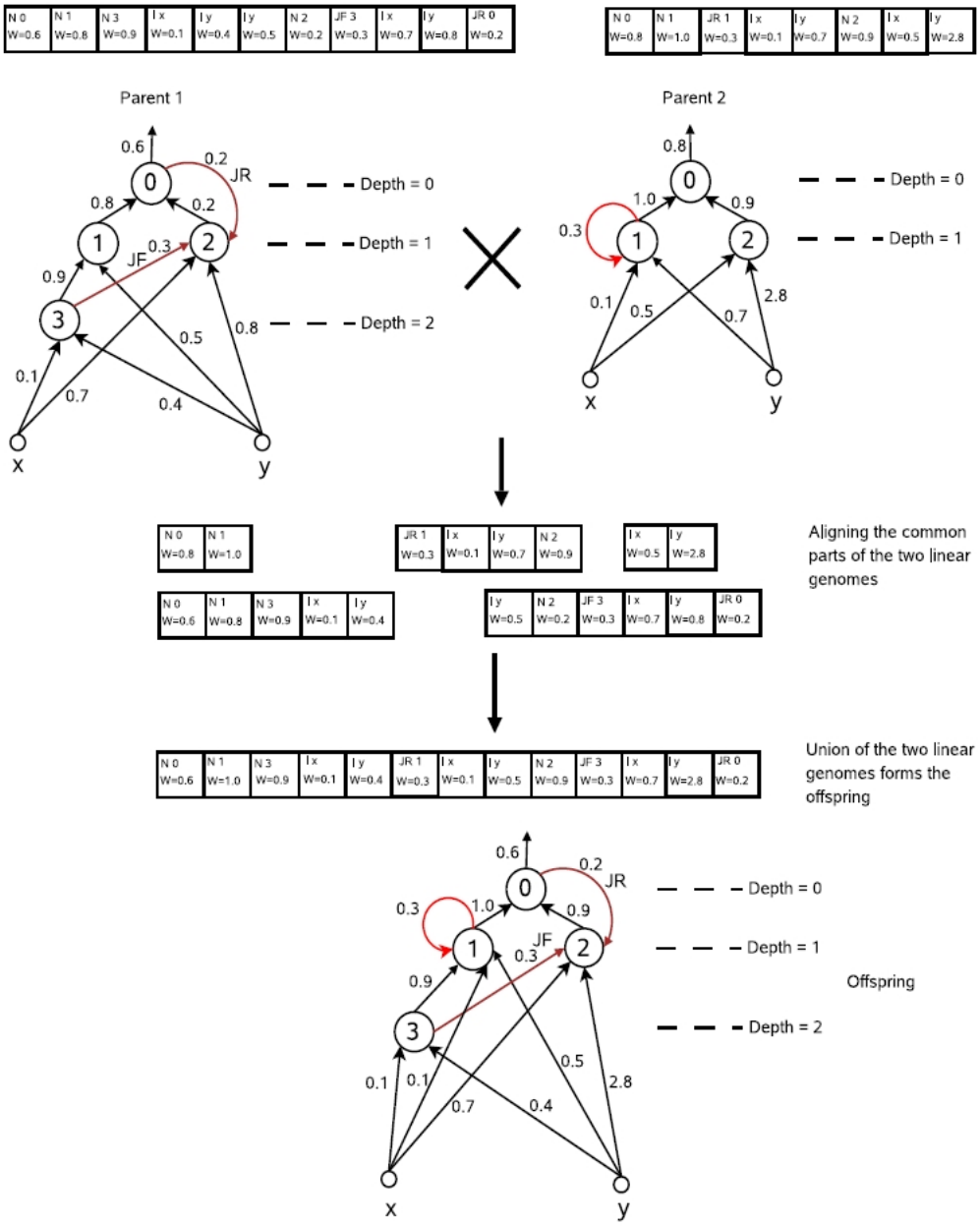


Figure 4.5: [Kir09] Performing crossover between two linear genomes. The genetic encoding is closed under this type of crossover operator since the resulting linear genome maps to a valid phenotype network. The weights of the nodes of the resulting linear genomes are inherited randomly from both parents.

4.3 Evolutionary Robotics

[BS08][Evolutionary robotics is a method for the automatic creation of autonomous robots. It is inspired by the Darwinian principle of selective reproduction of the fittest, captured by evolutionary algorithms. In evolutionary robotics, robots are considered as autonomous artificial organisms that develop their own control system and body configuration in close interaction with the environment without human intervention.]

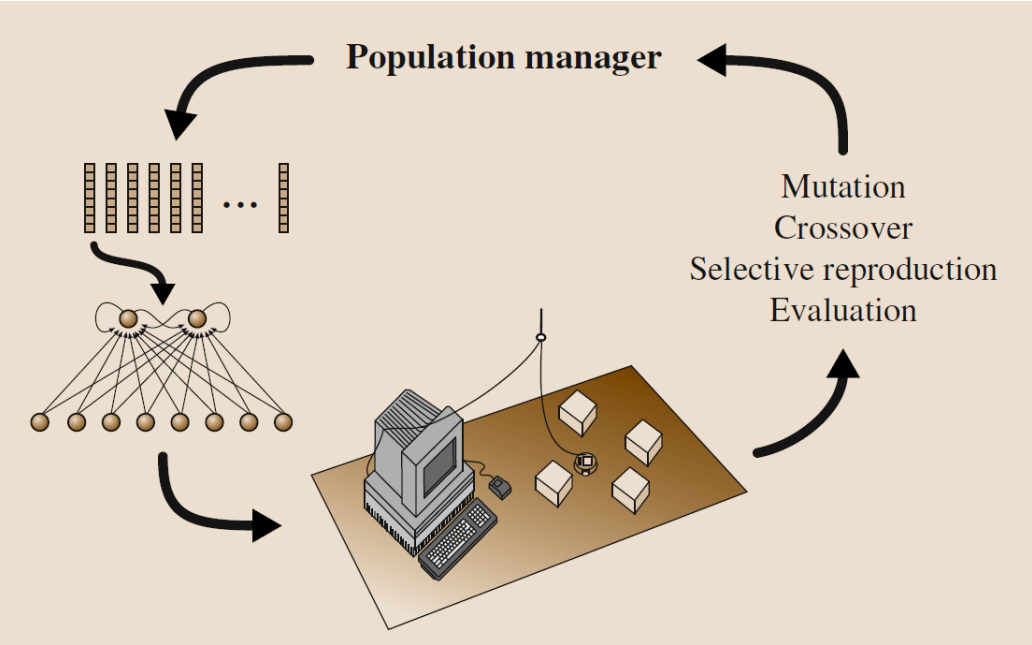


Figure 4.6: [BS08] Evolutionary experiments on a single robot. Each individual of the population is decoded into a corresponding neurocontroller which reads sensory information and sends motor commands to the robot every 300 ms while its fitness is automatically evaluated and stored away for reproductive selection

The application of evolution algorithms to the neural nets that drive robots has a long history: Valentino Braitenberg developed the famous 'Braitenberg vehicle' in the 80s. Since the 90s the simulation of robots was used to test evolution based on genetic encodings of the neural net controlling sensory motor systems, because the hardware was too complicated and expensive. With the progress in power storage technology and of course in computer technology, real robots became more and more common. The gap between simulation and a real robot is quite broad, because the noise and artifacts of the sensors are difficult to simulate. Even today, evolution

cannot entirely be on-line, because of the long time it takes to get useable results from evolution, because it is quite impossible to guarantee the same starting and experiment conditions in reality, because many good fitness-functions need global knowledge. This list can be further enlarged, so simultion stays an important part of evolutionary robotics.
One of the main goals of this thesis is, to make the simulation- reality- gap smaller, so one can use pre-evaluated nets on the robots. Theroretically the online- evolutions only goal is the overcome of the gap.

4.3.1 'D. Floreano: Bio-inspired Artificial Intelligence' and other work from him

Floreanos Work on evolutionary robotics is very extensive; many experiments on different platforms and much analytic work on this research theme. For my work his competing prey-hunter scenario was most important. I designed my own test for the framework as adaption from his idea.

5 The Framework

In this, the core of the thesis, the framework, is introduced. Containing over 3000 lines of code, only small, important parts are shown, the rest is only summarized.
When implementing the framework, I encountered a problem: To make structural mutations in the top level of the hierarchical Framework, it is necessary to develop controllers dynamically. For this, there is the need for a sensor- input neuron mapping, a output neuron- actuator mapping, and the generation of a fitness function to get low-level evolution working.
When a top-level mutation generates a new controller it is not possible to guess what purpose this should have. So it is also not possible to guess a corresponding fitness-function.
Because of this problem, I reduced the originally planned two level evolution to a simpler solution: The hierarchical approach is continued, but the top level is now exclusivly human driven. A human has to develop the petri-net, combining pre-existent controllers, (or writing new ones,) to achieve his goal. The controllers can be everything from simple hard-coded behaviour to a highly versatile evolutionary algorithm. A instruction, how to write new controllers, is enclosed with the framework.

5.0.2 M. Mattes, Design and Implementation of a Framework for Online Evolution of Robotic Behaviour, 2010

His diplom thesis was commenced to early to have a chance to run on the symbrion robots. My work can be seen as a succession of his framework. As I worked with his code on my previous project, many procedures and the design of my code is inspired by this. Some flaws of the implementation, that I experienced, led to different design desicions.

5.1 The Architecture

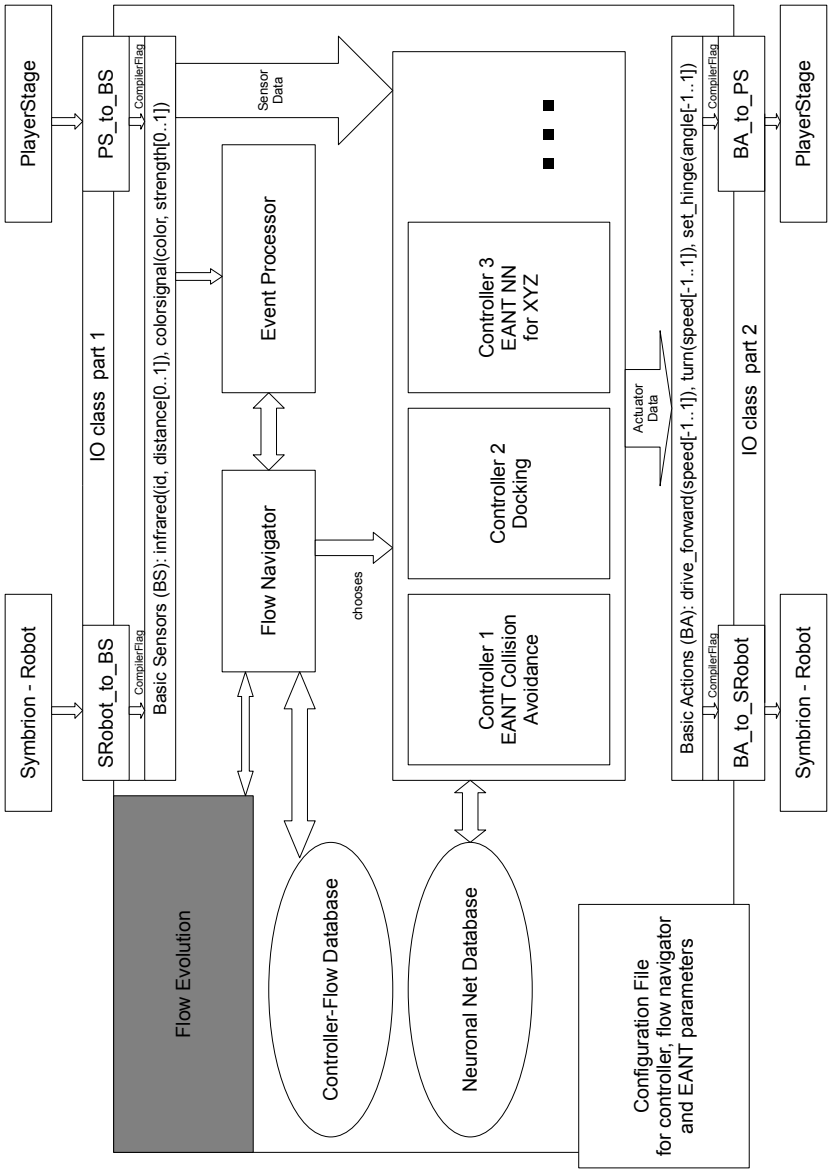


Figure 5.1: The architecture of the framework as it was realized. The grey Flow-Evolution-Part was not implemented.

This is the architecture of the framework. It was written in native c++, with no external dependencies for better useability and a minimum of possible error sources. The individual parts of it are:

- The IO-class. (Top and bottom of the architecture.) This provides the interface to the external world, be it a real robot (Symbrion - Robot) or a simulation (PlayerStage). To get the framework working, the only preconditions are that there exists a toolchain from native c++ to the platform and that there is a qualitative mapping between the IO-values and the platform.
- The FlowNavigator-class. This is the central part of the implementation. The FlowNavigator-class manages which controller(s) is(/are) initialized, loaded, when its triggered and when the IO-communication takes place. It is supported by:
- The EventProcessor-class. With this, the conditions of the underlying Petrinet are checked, so that the controllers can be coordinated.
- The Controllers. There are many posiblilities for controllers; EANT controllers and hard-coded controllers are supported directly, others can be implemented and used by inheriting from the abstractController-class
- ControllerFlowDatabase. This consists of a parser class for loading and saving controller flows, and of at least one controller flow file. The name of the file is written in the:
- ConfigurationFile. It contains every variable which is needed by the implementation and can be expanded. Also there is a parser included in the framework to load the configuration into a std::map.
- Neuronal Net Database. For every controller with a EANT population there exists a folder in which the nets are saved as text files, and from which they can be loaded again.

5.2 The EANT-Implementation

The framework features a fully functional, complete EANT - implementation in native c++. Considerable efford was made to make this implementation fast, exspecially at the evaluation of a Neural Network.

5.2.1 Loading Saving of EANT-Nets

A CGE- EANT net consists of a genome which can be evaluated directly; there is no need to turn the genotype into a phenotype, so an efficient representation is the key to make the implementation run fast. I have chosen the following structure:

```
enum Genetype {N, I, JF, JR};

struct EANTID {
  Genetype GT;
  string InputLabel;
  int ID;
};

struct EANTItem {
  EANTID ID;
  float Weight;
  int Number_of_Inputs;
};

EANTItem *Genome;
```

An EANT genome is a variable size array of EANTItems. Each EANTItem consists of an EANTID, a weight- parameter which is used as factor, threshold value, etc. depending on the activation function and an integer with the number of inputs, which shows the number of children for Neuron genes; 0 (or '-' as synonyme) for the others. The EANTID consists of the Genetype, (N for Neuron, I for Input, JF for JumperForward, JR for JumperRecursive) and an Input Label for identifying Inputs or an ID number. This ID number is a unique identifier for neurons and shows the source neuron for the jumper genes.

N 0	JR 0	I b	I c	I a	I b
1.567	0.073	-1.987	1.294	1.023	-0.721
5	-	-	-	-	-

Table 5.1: An example for a EANT -genome in CGE(Compact Genetic Enconding).

The corresponding file looks like this:

"N 0" "JR 0" "I b" "I c" "I a" "I b"
"1.567" "0.073" "-1.987" "1.294" "1.023" "-0.721"
"5" "-" "-" "-" "-" "-"

5.2.2 Parametric Mutation

The implementation of the parametric mutation was quite straightforward. NormalDist() is a function that returns a random number between -3 and +3, gaussian dis-

tribution of zero mean and unity standard devation.

```
void EANT::ParametricMutation (float LearningRate){
  unsigned int n = Genomelenght;
  // parameter tau for calculating adjusted learning rate
  double t1 = 1 / sqrt(2*sqrt(n));
  // parameter tau' for calculating adjusted learning rate
  double t2 = 1 / sqrt(2*n);
  for (unsigned int i = 0; i < n; i++) {
    // adjusted learning rate for gene i
    double adjLR = LearningRate * pow(M_E, t1* NormalDist() + t2*
        NormalDist());
    Genome[i].Weight=Genome[i].Weight +(adjLR * NormalDist());
  }
}
```

5.2.3 EANT Structural Mutations

At the structural mutation I loop over node genes to check for possible structural mutation. A neuron (N) gene has a chance of EANTMutationProb (see section "The configuration file") to get mutated. Only ONE is chosen! After this, the function returns. When a neuron is chosen, it gets mutated by one of the following measures with equal probability:

- Add a jumper gene to this neuron.
- remove a jumper gene (if existing) from this neuron
- add a new subnet to the neuron

Adding a jumper gene

For this function, all possible sources of a jumper are put in a std::vector, then one is chosen randomly. Now we have to find out, if the new jumper is recursive or forward. A jumper is recursive if its origin neuron gene is nearer at the root of the tree as itself. It is also necessary to check if this jumper already exists. If this is the case, no mutation takes place.

Removing a jumper gene

In this, the first jumper gene, which is imput for the neuron is deleted. For that, the function checks every input of the given neuron. When no jumper is found, the entire structural mutation does nothing!

Adding a new subnet

This function adds a new subnet to the neuron. The subnet consists of a new neuron gene and at least one input gene. All possible inputs have a chance of <EANTNewSubNetInputProbability> to become inputs for the new subnet. Because adding new subnets makes the genome very large, there is a maximum lenght, where no new subnets are allowed to be added.

5.2.4 EANT Crossover

The EANT crossover merges the genome of the class with a second genome, taken as parameter:

void EANT::StructuralCrossover (EANTItem *Parent, int Parentsize)

First we go through the Parent. We check if the number of trees and the name type of the inputs are equal. If this is true, the merging can begin. It is an recursion on (sub)trees. In a given tree:

- if a gene, identified by EANTID, exists in both trees, the resulting gene's Weight is the average of the two origin Weights.
- if a gene, identified by EANTID, exists in only one tree, the resulting gene is this one gene.
- if the gene is a neuron gene, which identified by EANTID, exists in both trees, the recursion is called on these two subtrees.
- if the gene is a neuron gene, which identified by EANTID, exists in only one tree, the whole subtree is added to the resulting genome.

5.2.5 EANT Drive

For driving a given Eant -net a very fast implementation is availlable through a filo - stack with the following rules: 1. Start from the first node 2. Traverse the linear genome from right to left in computing the output of the neural network by: 2.a If the current node is an input node, push its current value and the weight associated with it onto the stack. 2.b If the current node is a neuron node with n inputs, pop n values with their associated weights from the stack and push the result of the computation onto the stack. I use the weighted summation, but other functions are possible. 2.c If the current node is a recurrent jumper node, get the output of the node in the last evaluation of the neural network of the corresponding neuron node, saved in an array. Then push the value multiplied with the weight associated with the jumper node onto the stack. 2.d If the current node is a forward jumper node, first evaluate a copy the corresponding subgenome. Then push the value multiplied with the weight associated with the jumper node onto the stack. 3.

After traversing the genome from right to left completely, pop the resulting values from the stack. They are the outputs of the genome left to right in reversed order.

5.2.6 Example

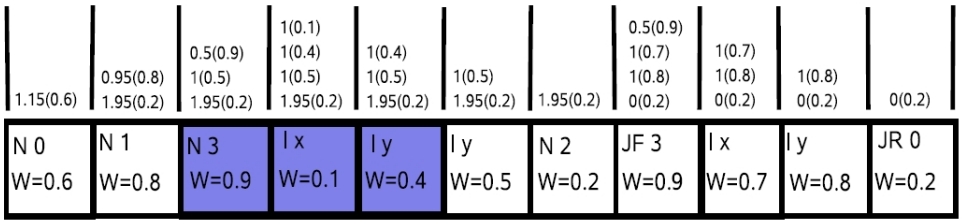


Figure 5.2: [Kir09] An example for the EANTDrive function.

5.3 The Controllers

Many different controllers can be used in the framework. The abstractController-class is the abstract class from that every controller must derive.

5.3.1 The abstract Controller

This is the abstract class, every controller has to inherit. The virtual functions have to be implemented by this controller.

```
   class AbstractController {
public:
      AbstractController();
      virtual  AbstractController();
      virtual void Drive()= 0;
      virtual void EndAndReset(string Foldername);
      virtual void initialize(string Foldername, map<string, string> ParaMap);
      virtual IO GetActuatorOutput(IO AO);
      virtual void SetSensorInput(IO SI);
};
```

5.3.2 Writing new controllers

To add a new controller:

- write controller that provides the following funktions:
  - void Drive();
  - void EndAndReset();
  - void initialize(string Foldername, map<string, string> ParaMap);
  - ActuatorOutput GetActuatorOutput();
  - void SetSensorInput(SensorInput SI);
- include .h in FlowNavigator.h
- include it in ControllerNames

All Controllers are derived from AbstractController
To make a new EANT-n-Population-Standard-Controller:

- Copy CollAvoid1;
- Change Nameprefixes of Funktions
- Change Inputsize and Outputsize static Variables
- Change ActuatorOutput GetActuatorOutput();, float Fitness(); and void SetSensorInput(SensorInput SI); to get the right sensors and actuators
- Make new folder in /ControllerLevel/ with name = name of the new class + "Nets"
- Add a .status -file to folder
- Change 'sstr«Foldername«"CollAvoid1.status"' to the new status file

5.3.3 The EANT evolution Controller

The EANT evolution controller is a vital part of the concept of cooperation between classical and evolutionary controllers. EANT evolution controllers like CollAvoid1 or Hunter are capable of conserving any made progresses of their evolution. They even tolerate crashs, as every net is saved as a text file and the controller automatically loads the best of them when it is restarted. A net which is saved by this mechanism, codes it's fitness and ID in it's filename:
filename = fitness«" _ "«ID«.gen
As case-by-case handling is very bulky and the saving and loading mechanisms are complex, the best method for new controllers of this type is, to copy one of them and change the names.

5.3.4 The EANT net driver

The EANT net driver only drives the best EANT net which is in it's folder without any evolution. WallFollow1 is an example of this in the framework.

5.3.5 The hardcoded Controller

Of course it is possible to make a hardcoded controller. An example for a hardcoded controller running in the framework is Explore1, that makes exploring and collision avoidance with less than 50 lines of code.

5.4 Datatypes

Several datatypes are involved in the framework, but two are very important:

- The Datatype for the high-level petri net named **ControllerFlow**
- The parameters of the class for the sensor and actuator abstraction. This is covered by other sections.
- The EANT net datatype above

5.4.1 ControllerFlow

These are the structs for the petri net that is navigated by the FlowNavigator-class.

```
  // PetriNetStructs
struct Place {
  bool Token;
  string Controller;
  string ID;
};

  struct Transition {
  string Eventname;
  float Parameter;
  string ID;
};

  struct Link {
  Transition* T;
  Place* P;
};
```

```
  struct ControllerFlow{
  vector<Place> ControllerNode;
  vector<Transition> EventTransition;
  vector<Link> Tr2Pl;
  vector<Link> Pl2Tr;
} ;
```

The Flow File

Below an example for a flow file:

```
  Places:
// ID;Controller;Token
1;CollAvoid1;0;
2;WallFollow1;0;
3;Explore1;1;
4;END;0;

  Transitions:
// ID;Eventname;Parameter
01;UnderDistance;0.2;
02;OverDistance;0.5;
03;TickTimer;200;
04;AbsTicks;10000;

  Links:
// Transition;Place or Place;Transition. Transitions start their ID with a 0.
3;01;
01;1;
1;02;
02;2;
2;03;
03;3;
3;04;
04;4;
```

This is loaded into the controller flow struct above.

5.5 Sensors and Actuators

The Robot which is the model of the sensors and actuators looks like this:

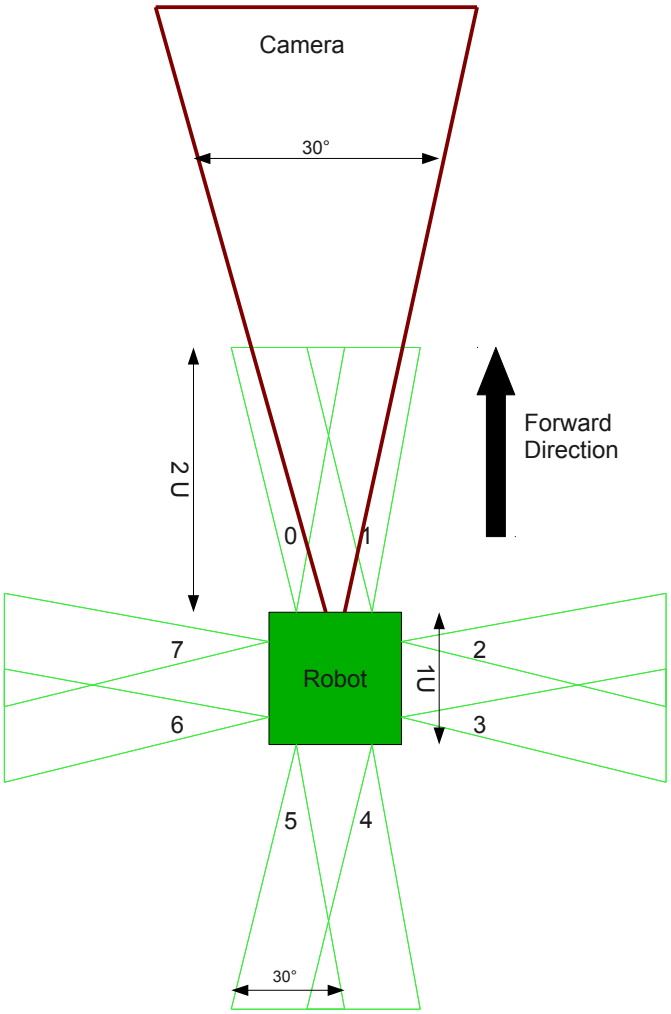


Figure 5.3: The Robot model which is the basis of the IO model.

Of course it is possible to extend this model, so that new sensors can be added. The abstract sensors are part of the IO class which abstracts the heterogenous environment to close the simulation-reality-gap.

```
struct Infrared{
   int ID;
   float Distance; //[0..1]
};
```

Infrared IR[8];

The 8 infrared sensors of the model are numbered from 0 to 7. The mapping should turn the extern values to floats between 0 and 1. At the 'Karlsruhe Robot' this is achieved by: IR[x].Distance = 10/(<SensorValue> + 10);

enum Color {red, blue, green, white};

```
struct Colorsignal{
Color C;
float intensity; //[0..1]
};
```

Color detection is very complicated in both, simulation and reality. They are detected by the camera; the intensity a qualitative measurement for a colors occurence.

The abstract actuators are also part of the IO class:

```
 float drive_forward; //speed[-1..+1]
float turn; //direction and speed[-1..+1]
float set_hinge; //angle [-1..+1]
```

5.6 Event-driven Flow Navigation

The very core of the framework is the event-driven flow navigation. In this class, every controller is instanciated and used if necessary. It is possible to run more than one controller simultaniously; in this case a round-robin procedure is used. So it is possible to run a collision avoidance and a exploration controller simultaniously. The datatype for the controllerflow is the Petri Net (s.a.), which is conditional. The conditions are modeled as events with their own class. Events must be registered and unregistered, so only relevant events have to be checked. Any Event, which can be implemented, is possible. Predefined and already implemented are the following events:

- OverDistance(Parameter): true if the minimal mesurable distance to an obstacle is lower than the parameter
- UnderDistance(Parameter): true if the minimal mesurable distance to an obstacle is higher than the parameter
- TickTimer(Parameter): true if more than <parameter> ticks have passed since the registration.

- AbsTick(Parameter): true if more than <parameter> ticks have passed since the start of the program.
- Always: always true

5.7 The configuration file

At the framework configuration file allows many changes; the parameter EANTMutationProb defines the probability for each gene to get mutated. EANTLearningRate defines how strong the changes of weight are, when the parametric mutation is triggered, etc. Below there is an commented example for an configuration file:

```
  # <-this is an comment; after the first '#' the rest of the line is ignored
# "Parameter" ; "Value" is the Format of the Configuration file
 "ControllerFlowName"                ; "Hunter.fl"
# The name of the PetriNet
 "EANTinitialPop_m"                  ; "3"
# number of initial mutations when initializing a new population
 "EANTLearningRate"                  ; "0.3"
# measure for the strenght of parametric mutation
 "EANTMutationProb"                  ; "0.4"
# 1>x>0. Probablility for a structural mutation to occur at every neuron gene.
 "EANTNewSubNetInputProbability"     ; "0.4"
# Probablility for every possible input gene to become input for a new subnet
 "EANTBorderForSubnetting"           ; "100"
# Border for the size of the nets, because of the chance for exponential growth.

  "CONTROLLER_numberofnets"            ; "14"
# The number of nets in a population, should be at least 1
 "CONTROLLER_exploitationlenght"     ; "5"
# How often the parametric mutation is used between two structural mutation and
crossover steps.
 "CONTROLLER_TicksPerStep"           ; "200"
# How many loops in the main loop for each net.
 "CONTROLLER_deleteoldnets"          ; "1"
# Delete nets from database that are thrown out of the population 0 = no, 1 = yes
 "CONTROLLER_crossoverchance"        ; "0.5"
# Probablility for crossover
 "CONTROLLER_sleep_msec"             ; "100"
# Pause between two loops in the main loop
```

6 Application of the Framework

For the purpose of testing and evaluating the framework, I'll first define a Challenge, then systematically solve the steps for accomplishing it: The goal of the first step is to find the best parameters for the frameworks EANT - evolution. Then I will use the hierarchical approach to train the two main behavior parts seperatly. At last, I will try to create the scenario in simulation and the real world and compare the simulated robot and the real robot. They (should) have the same behavior at the beginning, but ongoing evolution will soon create differences.

6.1 The Challenge

The Challenge for my framework will be a hunter-prey scenario (aka. play tag) with alternate roles: Robot 1 will be the first hunter, Robot 2 the first prey. They will start in direct contact, executing an exploration step for 20 seconds. Then the hunt startes- Robot 1 will be rewarded if he 'catches' Robot 2 fast, Robot 2 will be rewarded if he is NOT catched. When the Robot is catched, Roles are changed and the exploration phase begins again.

6.2 Step 1: Finding good parameters for Robot-EANT-Evolution

For this step I'll use a simple problem: Collision Avoidance. The figure below shows the used labyrinth.

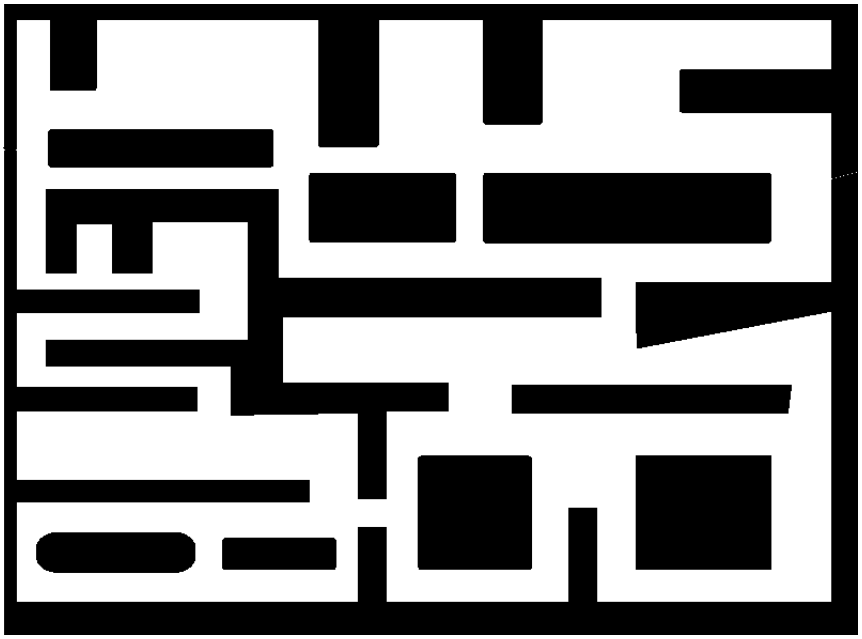


Figure 6.1: The arena used for the evolution of collision avoidance.

The following parameters are tested:

- global learning rate `(EANTLearningRate)`
- number of structural mutations for the starting population `(EANTinitialPop_m)`
- Probability for mutating a EANT gene `(EANTMutationProb)`
- Probability for each input to become part of a new subnet `(EANTNewSubNetInputProbability)`
- number of nets in EANT population `(CONTROLLER_numberofnets)`
- number of EANT-exploitation-steps between exploration-steps `(CONTROLLER_exploitationlenght)`
- Probability for crossover of a newly mutated net with the champion `(CONTROLLER_crossoverchance)`

Each configuration is tested for 20000 steps, the medium and maximum fitness is noted. Each experiment is made twice, to lessen the effect of coincidence.

6.2.1 The Robot

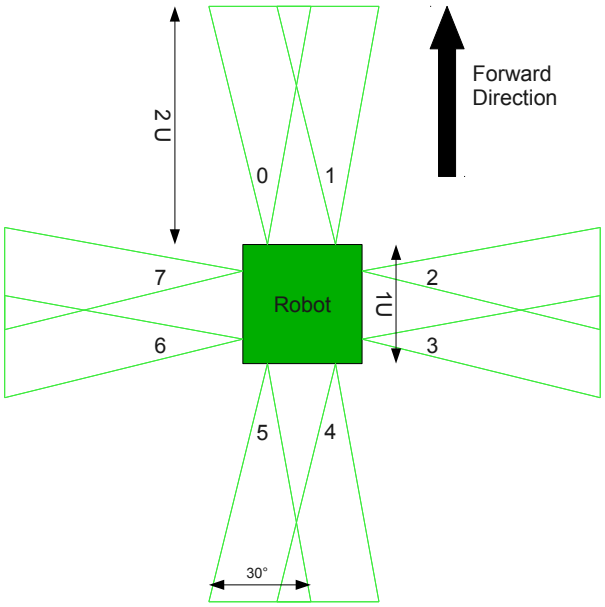


Figure 6.2: The robot used for the experiments.

Each sensor input is standardized to the range of [0.025,1].

6.2.2 Improvement of global learning rate

The global learning rate influences how strong the changes of paramtertic mutations are.

Configuration:

- `(EANTinitialPop_m)` "3"

- (EANTMutationProb) "0.5"
- (EANTNewSubNetInputProbability) "0.6"
- (CONTROLLER_numberofnets) "10"
- (CONTROLLER_exploitationlenght) "4"
- (CONTROLLER_crossoverchance) "0.5"

EANTLearningRate	max. fitness 1	median fitness 1	max. fitness 2	median fitness 2
0.1	0.50	0.49	0.56	0.46
0.2	0.55	0.33	0.37	0.27
0.3	0.50	0.37	0.34	0.21
0.4	0.81	0.31	0.41	0.28
0.5	0.46	0.19	0.50	0.30

Table 6.1: Improvement of global learning rate.

The robots position

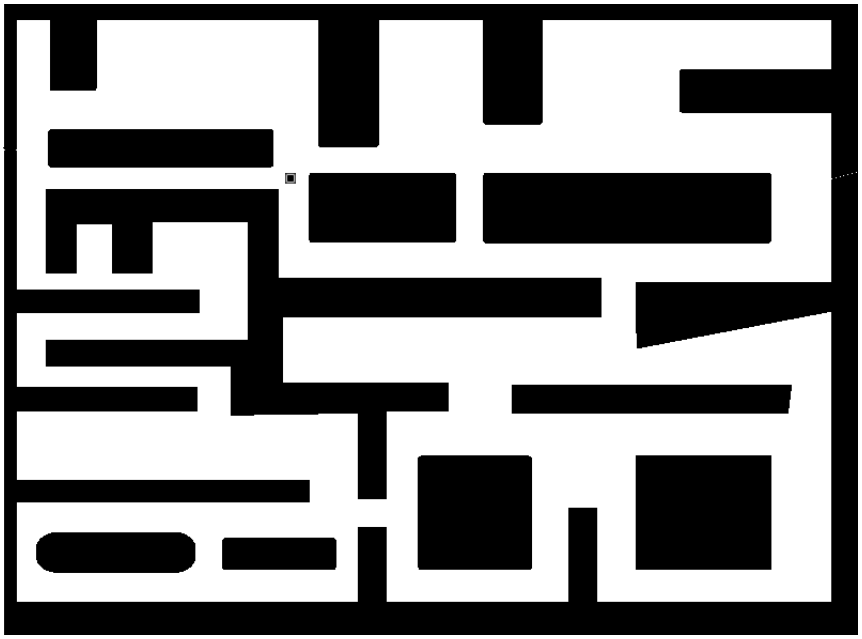


Figure 6.3: Position of the Robot for optimation of global learning rate.

The Inputs

The neural nets have the following two inputs:
I_0 : (Dist(7) + Dist(0))/2
I_1 : (Dist(1) + Dist(2))/2

Fitness calculation

The fitness is calculated by taking the **minimum value** of all sensors. This guarantees, that a high distance to any obstacle is rewarded.

Interpretation

In the analysis of of these results there are some limitations to be attended: Fitness values of 0.3 are possible when the robot doesn't move at all, up to 0.6 small adjustments are enough. Because of that, only fitness values larger than 0.7 show a real solution to the problem- if the fitness F is: 0.3 < F < 0.7, it can be interpreted as a working but suboptimal collision avoidance. Values smaller than 0.3 are nonfunctional nets. **I will take 0.4 as the Value for EANTLearningRate**.

6.2.3 Improvement of initial mutation steps

Configuration:

- `(EANTLearningRate)` "0.4"
- `(EANTMutationProb)` "0.5"
- `(EANTNewSubNetInputProbability)` "0.6"
- `(CONTROLLER_numberofnets)` "10"
- `(CONTROLLER_exploitationlenght)` "4"
- `(CONTROLLER_crossoverchance)` "0.5"

EANTinitialPop_m	max. fitness 1	median fitness 1	max. fitness 2	median fitness 2
1	0.39	0.28	0.33	0.29
2	0.27	0.20	0.36	0.25
3	0.29	0.23	0.79	0.35
4	0.49	0.26	0.38	0.23
5	0.29	0.23	0.26	0.13
10	0.42	0.03	0.33	0.25

Table 6.2: Improvement of initial mutation steps.

The robots position

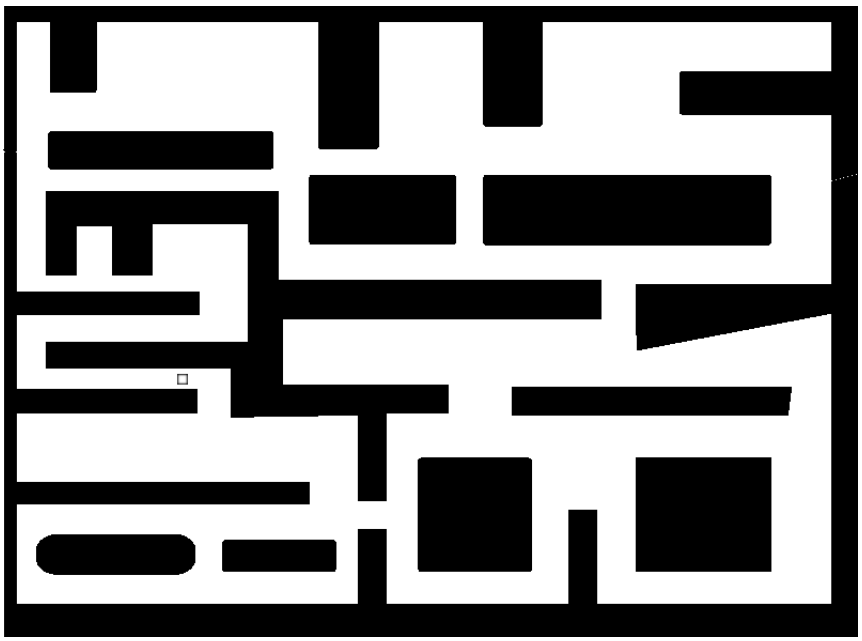


Figure 6.4: Position of the Robot for optimation of initial mutation steps.

The Inputs

The neural nets have the following three inputs:
I_0 : (Dist(7) + Dist(0))/2
I_1 : (Dist(1) + Dist(2))/2
I_2 : (Dist(0) + Dist(1))/2

Fitness calculation

The fitness is calculated by taking the minimum value of all sensors. This guarantees, that a high distance to any obstacle is rewarded.

Interpretation

Values < 0.32 can be achieved by only small adjustions. Values between 0.33 and 0.50 can be achieved by turning of right at the bend and staying in the middle; Values >0.50 are only achieved when the robot reaches the open area. **I will use 4 as default value**

6.2.4 Improvement of the probability for mutating a EANT gene

Configuration:

- (EANTLearningRate) "0.4"
- (EANTinitialPop_m) "4"
- (EANTNewSubNetInputProbability) "0.6"
- (CONTROLLER_numberofnets) "10"
- (CONTROLLER_exploitationlenght) "4"
- (CONTROLLER_crossoverchance) "0.5"

EANTMutationProb	max. fitness 1	median fitness 1	max. fitness 2	median fitness 2
0.1	0.35	0.18	0.43	0.21
0.2	0.56	0.31	0.31	0.06
0.3	0.72	0.31	0.38	0.23
0.4	1.00	0.21	0.32	0.12
0.5	0.34	0.24	0.70	0.28
0.6	0.37	0.21	0.83	0.21
0.7	0.32	0.21	0.28	0.17

Table 6.3: Improvement of probability for mutating a EANT gene

The robots position, inputs, fitness calculation and the interpretation of results are equal to the specification at the Improvement of initial mutation steps. **I will use 0.4 as default value**

6.2.5 Improvement of the probability for adding a given input to a newly created subnet

No new subnet can have no input. If this is the case, there will be 10 retrys- if there is still no input, the adding of the subnet is canceled and a a warning is posted.

Configuration:

- (EANTLearningRate) "0.4"
- (EANTinitialPop_m) "4"
- (EANTMutationProb) "0.4"
- (CONTROLLER_numberofnets) "10"

- `(CONTROLLER_exploitationlenght)` "4"
- `(CONTROLLER_crossoverchance)` "0.5"

Variable	max. fitness 1	median fitness 1	max. fitness 2	median fitness 2
0.1	0.33	0.17	0.38	0.15
0.2	0.55	0.21	0.31	0.25
0.3	0.38	0.17	0.37	0.23
0.4	1.00	0.33	0.25	0.09
0.5	0.25	0.05	0.56	0.41
0.6	0.41	0.21	0.37	0.25
0.7	0.39	0.06	0.33	0.26

Table 6.4: Improvement of the probability for adding a given input to a newly created subnet.

The robots position, inputs, fitness calculation and the interpretation of results are equal to the specification at the Optimation of initial mutation steps. **I will use 0.4 as default value**

6.2.6 Improvement of the number of nets in Population

No new subnet can have no input. If this is the case, there will be 10 retrys- if there is still no input, the adding of the subnet is canceled and a a warning is posted.

Configuration:

- `(EANTLearningRate)` "0.4"
- `(EANTinitialPop_m)` "4"
- `(EANTMutationProb)` "0.4"
- `(EANTNewSubNetInputProbability)` "0.4"
- `(CONTROLLER_exploitationlenght)` "4"
- `(CONTROLLER_crossoverchance)` "0.5"

Number of nets	max. fitness 1	median fitness 1	max. fitness 2	median fitness 2
2	0.26	0.26	0.31	0.31
4	0.37	0.26	0.45	0.41
6	1.00	0.98	0.35	0.30
8	0.40	0.31	0.81	0.59
10	0.41	0.28	0.28	0.20
12	0.35	0.25	0.29	0.15
14	0.33	0.24	0.35	0.23

Table 6.5: Improvement of the number of nets in Population.

The robots position, inputs, fitness calculation and the interpretation of results are equal to the specification at the Optimation of initial mutation steps. **I will use 8 as default value**

6.2.7 Improvement of the lenght of the EANT exploitation phase

Configuration:

- `(EANTLearningRate)` "0.4"
- `(EANTinitialPop_m)` "4"
- `(EANTMutationProb)` "0.4"
- `(EANTNewSubNetInputProbability)` "0.4"
- `(CONTROLLER_numberofnets)` "8"
- `(CONTROLLER_crossoverchance)` "0.5"

Exploitationlenght	max. fitness 1	median fitness 1	max. fitness 2	median fitness 2
1	0.40	0.40	0.36	0.25
2	0.40	0.33	0.36	0.25
3	0.28	0.23	1.00	0.67
4	0.41	0.40	0.34	0.31
5	0.31	0.18	0.37	0.32
6	0.40	0.05	0.42	0.30

Table 6.6: Improvement of the lenght of the EANT exploitation phase.

The robots position, inputs, fitness calculation and the interpretation of results are equal to the specification at the Optimation of initial mutation steps. **I will use 3 as default value**

6.2.8 Improvement of the chance for crossover with the champion

Configuration:

- `(EANTLearningRate)` "0.4"
- `(EANTinitialPop_m)` "4"
- `(EANTMutationProb)` "0.4"
- `(EANTNewSubNetInputProbability)` "0.4"
- `(CONTROLLER_numberofnets)` "8"
- `(CONTROLLER_exploitationlenght)` "3"

Crossoverchance	max. fitness 1	median fitness 1	max. fitness 2	median fitness 2
0	0.42	0.25	1.00	0.30
0.5	0.30	0.24	0.36	0.27
1	0.40	0.36	1.00	0.59

Table 6.7: Optimation of the chance for crossover with the champion.

The robots position, inputs, fitness calculation and the interpretation of results are equal to the specification at the Optimation of initial mutation steps. **I will use 0.5 as default value**

6.3 Step 2: Train EANT-populations for Hunter

For this step, I had to enhance the robots capabilities: It must have a sort of RGB sensor, to locate the Prey. The Prey Robot is green, the hunter robot is red. To avoid the bootstrap-problem, the challenge is enhanced step by step, starting with a simple scenario:

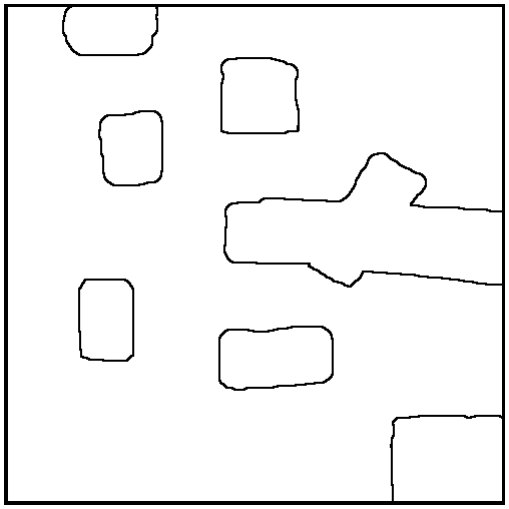


Figure 6.5: The map for the initial step.

The green robot won't move, the red is evolved. It's fitness is the part of green pixels in relation to the camera's all-up. Camera's resolution is 100x100Pixel.

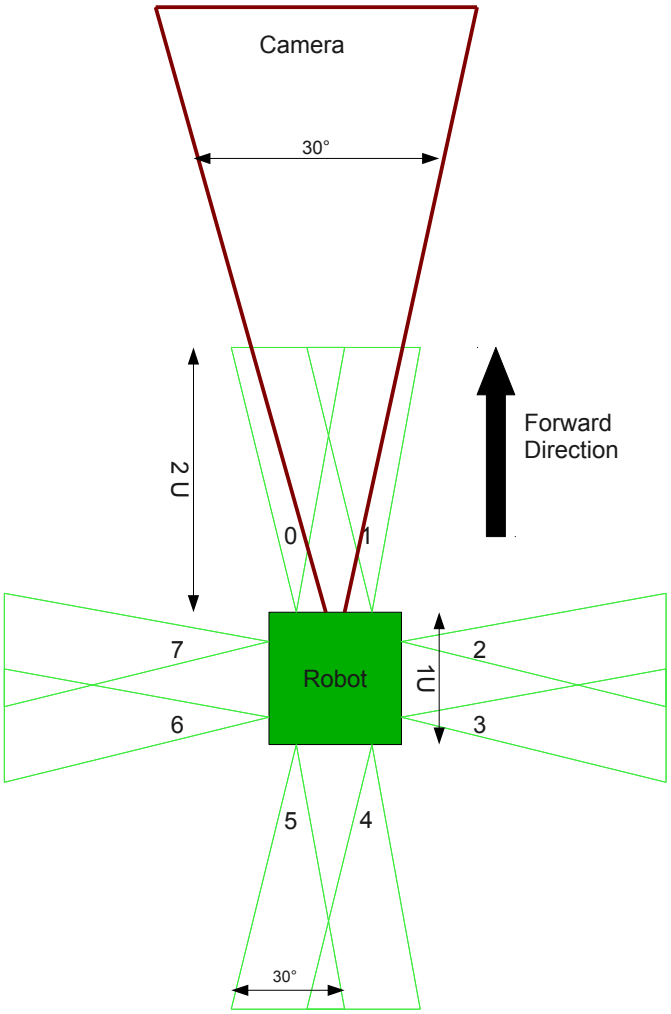


Figure 6.6: New robot layout with a camera.

There will be 4 runs, I'll take the best net from each experiment as population for further.
Configuration is:

- `(EANTLearningRate)` "0.4"
- `(EANTinitialPop_m)` "4"
- `(EANTMutationProb)` "0.4"

- `(EANTNewSubNetInputProbability)` "0.4"
- `(CONTROLLER_numberofnets)` "10" (more variability)
- `(CONTROLLER_exploitationlenght)` "3"
- `(CONTROLLER_crossoverchance)` "0.5"

6.3.1 First experiment

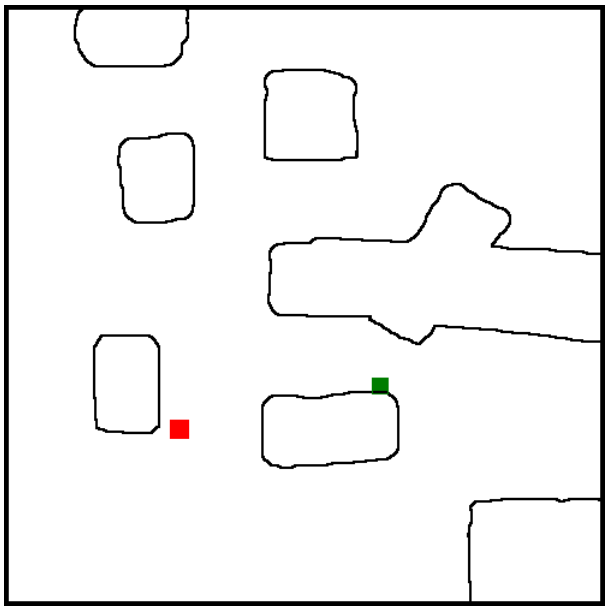


Figure 6.7: The position of the Robots for the initial step.

The first trial is a simple one: explore the forward area and find the green blob. The fitness F is calculated by: F=(min((Euclidian Distance between start and end)/10, distance to obstacle) + (1/(2*Distance to nearest blob)))

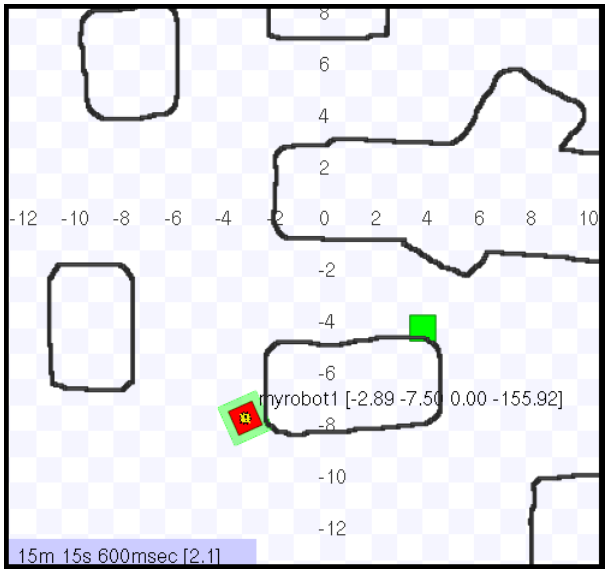


Figure 6.8: A Screenshot of a running, and not very successful net.

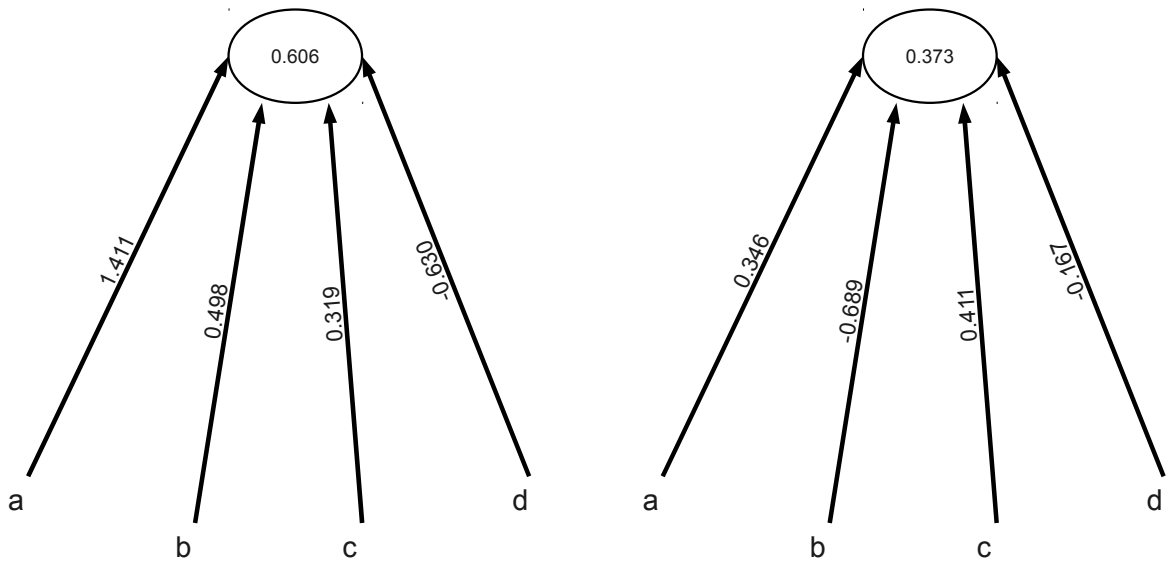


Figure 6.9: The best net with fitness 0.45, that proved to be the best one for the first run.

It seems, that the problem is too difficult to solve or I had not enough luck- only one net found the blob. The Komplexity of the sucessful net is quite low, so I'll change the parameters accordingly for the next experiment; The population size will be increased further, for a broader basis.

6.3.2 Second experiment

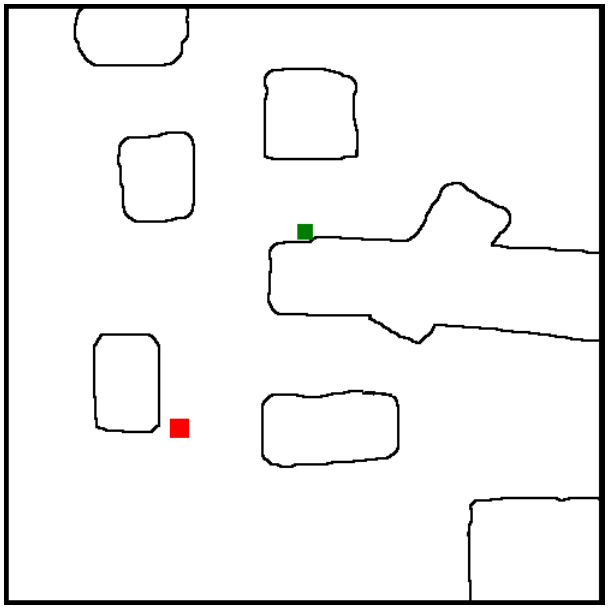


Figure 6.10: Position of the Robots for the second experiment.

New Configuration will be:

- `(EANTLearningRate)` "0.3"
- `(EANTinitialPop_m)` "3"
- `(EANTMutationProb)` "0.3"
- `(EANTNewSubNetInputProbability)` "0.4"
- `(CONTROLLER_numberofnets)` "14"
- `(CONTROLLER_exploitationlenght)` "6"
- `(CONTROLLER_crossoverchance)` "0.5"

The results of the experiment are not satisfying: None of the robots has reached the green one. Some reached the upper left corner of the area, but didn't see the green robot. In the following experiments, I'll use a smaller arena for further experiments, so that a exploring robot is more likely to meet the green one in acceptable time.

6.3.3 Third experiment

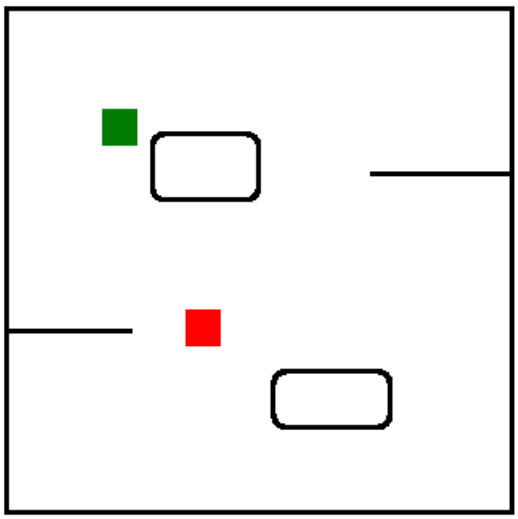


Figure 6.11: A smaller arena for the robots; Experiment one.

The corresponding configuration will be:

- `(EANTLearningRate)` "0.3"
- `(EANTinitialPop_m)` "3"
- `(EANTMutationProb)` "0.4"
- `(EANTNewSubNetInputProbability)` "0.4"
- `(CONTROLLER_numberofnets)` "14"
- `(CONTROLLER_exploitationlenght)` "5"
- `(CONTROLLER_crossoverchance)` "0.5"

Even with the smaller labyrinth, no net reached the green blob. It seems, the bootstrap problem is too much for the robots. I'll enhance the maximum range of the camera from 4 to 10 for the fourth experiment, so the target can be seen from a greater distance. (The arena's size is 20x20)

6.3.4 Fourth experiment

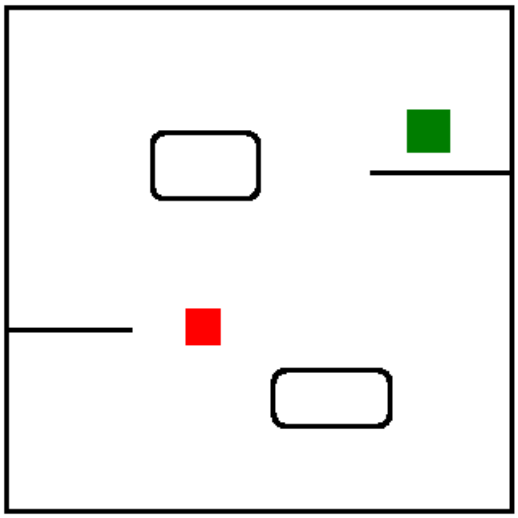


Figure 6.12: A smaller arena for the robots; Experiment two.

The enhancement of the camera range was quit sucessful- two nets made it to the green blob.

6.3.5 Running away

The last step of the training: the population, i got in the previous experiments has to hunt down a moving target. This target has a simple, hand-written exploration-controller in it, and drives away from the hunter. There will be two long-lasting experiments to hopefully get some really good neural nets.

The prey-controller is:

```
  robot2->Read();
double front = (robotSonar->GetScan(0)+robotSonar->GetScan(1))/4;
double left  = (robotSonar->GetScan(7)+robotSonar->GetScan(0))/4;
double right = (robotSonar->GetScan(1)+robotSonar->GetScan(2))/4;
robotPos->SetSpeed((front-0.2)*0.6,left-right+((drand48()-0.5)/5));
```

It drives forward when there is no obstacle and turns, depending on the difference of right and left. Aditionally a random value is added to make it more flexible and to make sure that it can drive alternate paths.

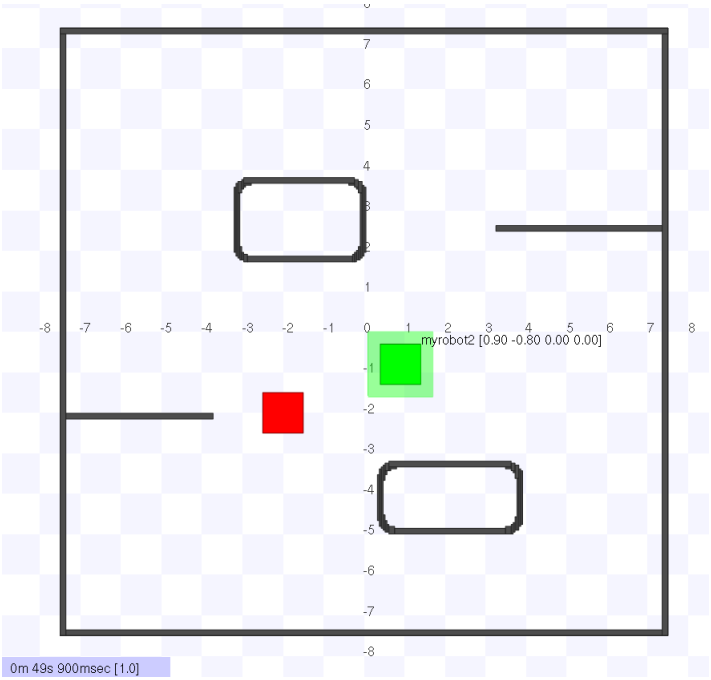


Figure 6.13: Position of the Robots for the hunting evaluation.

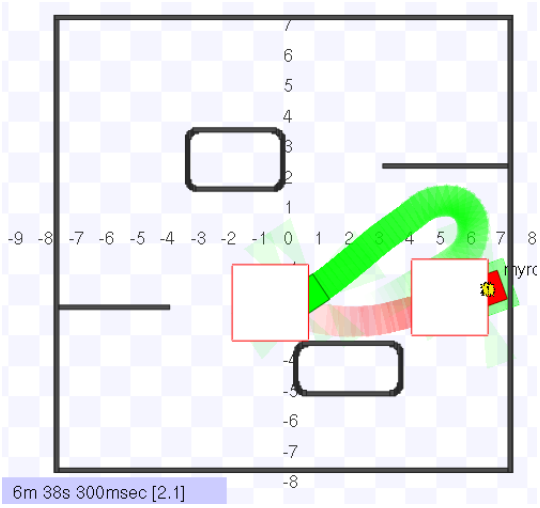


Figure 6.14: Example of a failed catch, the hunter(red) followed the prey(green) but was not able to catch up, when the green robot made a sharp turn.

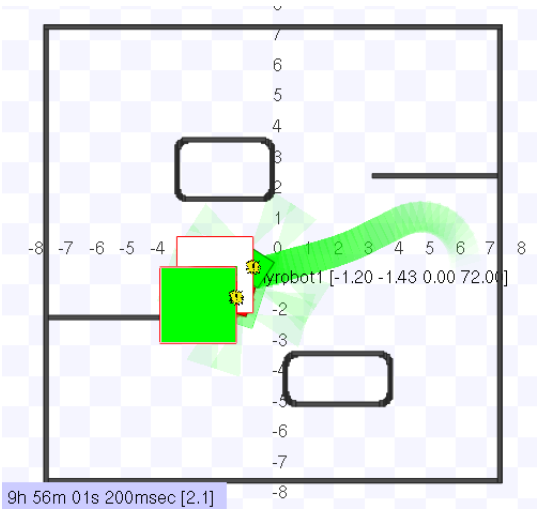


Figure 6.15: Example of a strategy that evolved: the hunter(red) waited until the prey(green) was near; then it made a small but fast approach into its flank. The green robot's evasion(it tried to turn and get away) was to late.

The two best results of each of the two experiments are represented by the following nets:

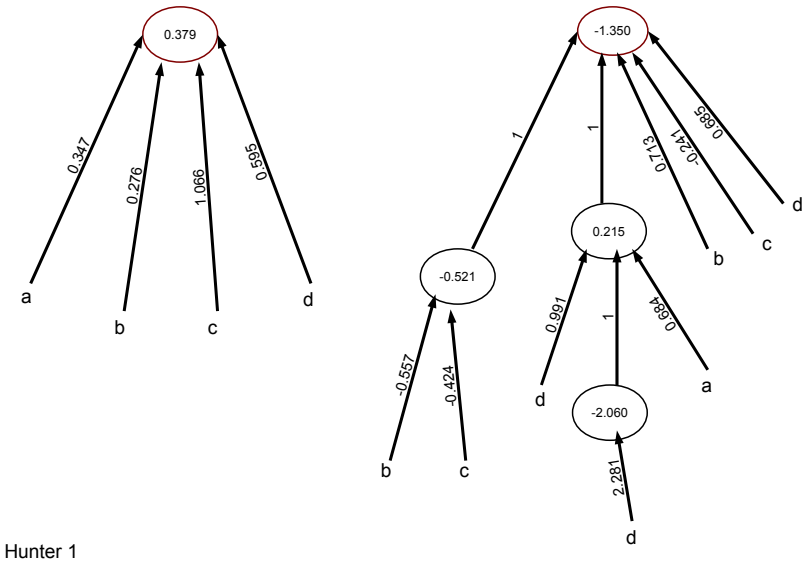


Figure 6.16: The net of one of the victorious genomes in the hunting contest.

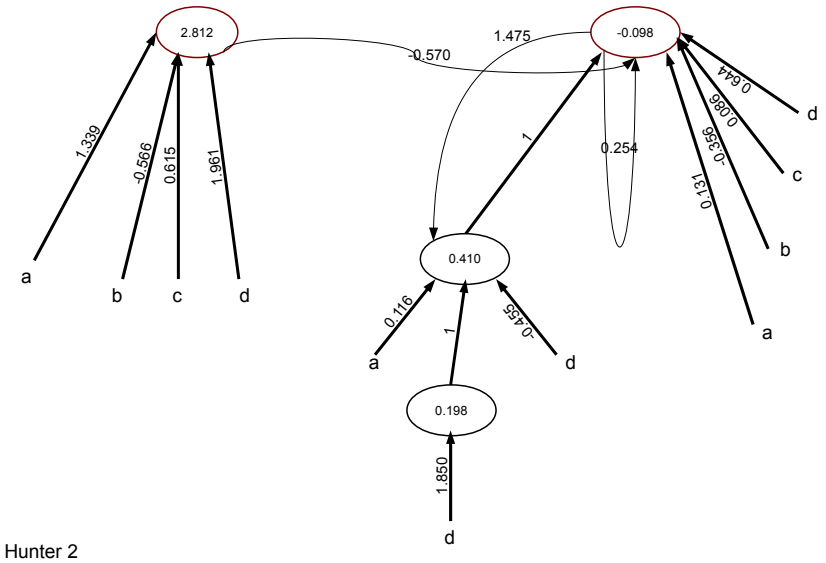
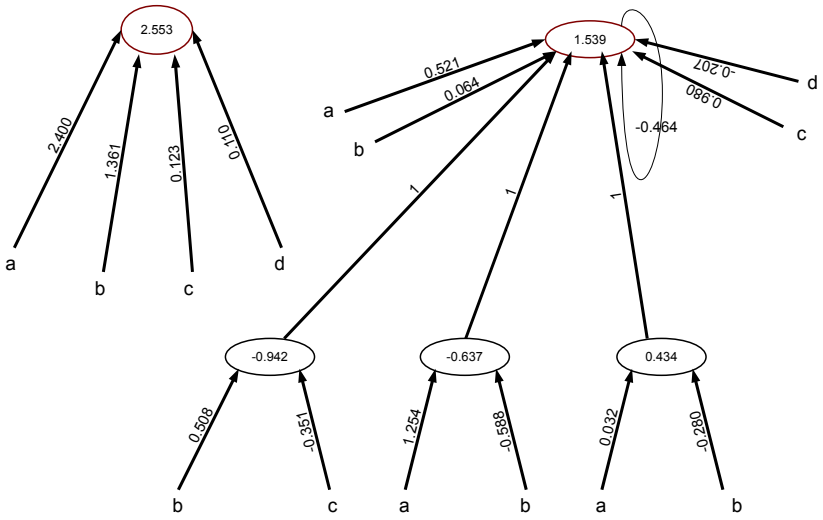
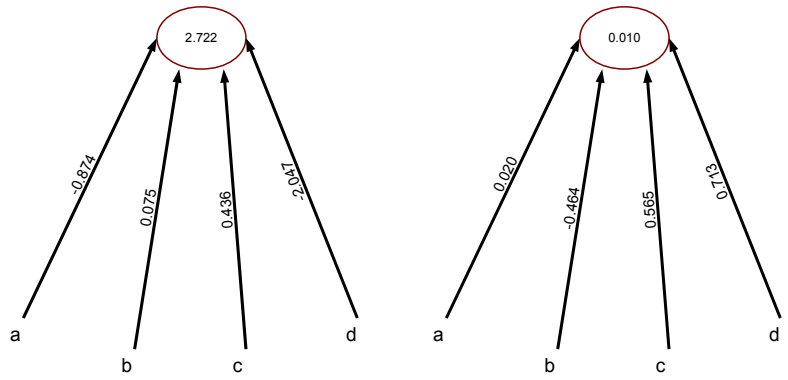


Figure 6.17: The net of one of the victorious genomes in the hunting contest.



Hunter 3

Figure 6.18: The net of one of the victorious genomes in the hunting contest.



Hunter 4

Figure 6.19: The net of one of the victorious genomes in the hunting contest.

6.4 Step 3: Train EANT-populations for Prey

Here I search for a couple of good nets to run away from a chaser.

6.4.1 First Experiment

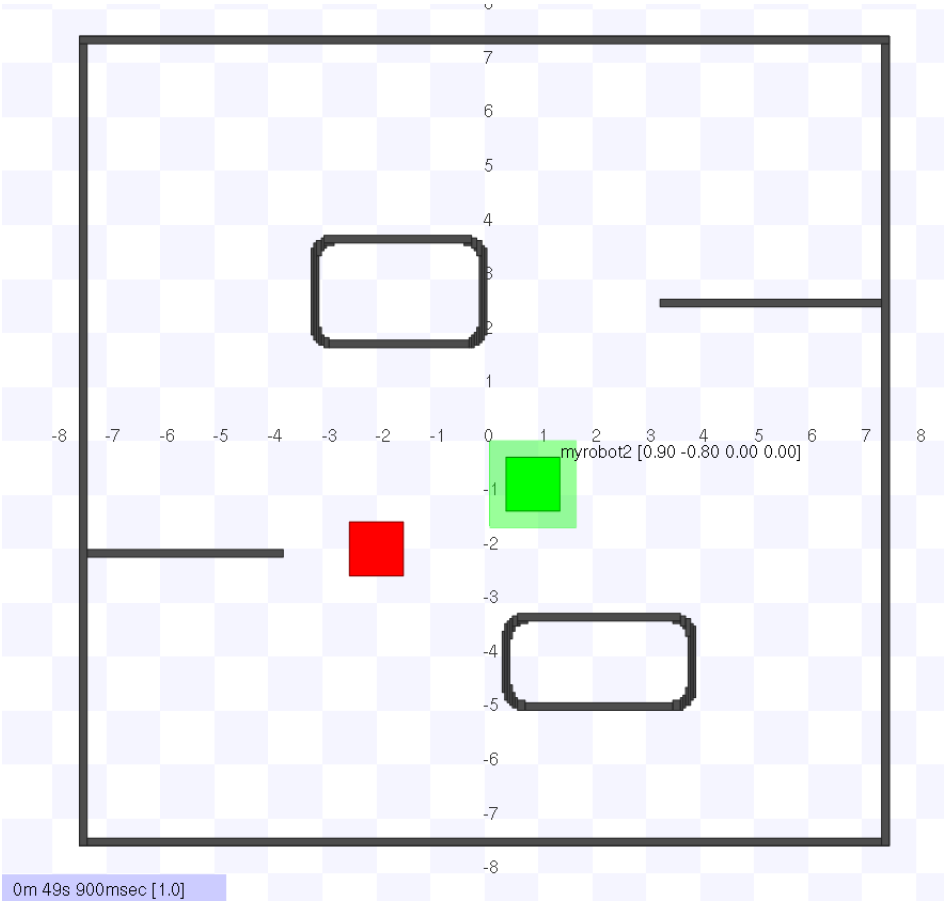


Figure 6.20: Arena for the first prey- experience. A hardcoded hunting algorithm in the red robot will try to catch the prey(green robot).

The Configuration will be:

- `(EANTLearningRate)` "0.3"
- `(EANTinitialPop_m)` "3"
- `(EANTMutationProb)` "0.3"

- (EANTNewSubNetInputProbability) "0.4"
- (CONTROLLER_numberofnets) "14"
- (CONTROLLER_exploitationlenght) "6"
- (CONTROLLER_crossoverchance) "0.5"

The Hunter is controlled by the following algorithm:

When there is no green robot in sight(H_status = 1), the hunter explores its surroundings by driving curves with a great deal of random influence to make it less predictable. If he sees a green blob, he accelerates towards it, and random influence is reduced to 1/3 of the original value, and the curveing stops.

```
 double front = (robotSonar->GetScan(0)+robotSonar->GetScan(1))/4;
double left  = (robotSonar->GetScan(7)+robotSonar->GetScan(0))/4;
double right = (robotSonar->GetScan(1)+robotSonar->GetScan(2))/4;
// robotBlobF->GetCount() = 1 if there is a green robot in sight
double O_0 =((front-0.3))+robotBlobF->GetCount();
double O_1;
if (H_status==0){//standard procedure
  O_1 =left-right+((drand48()-0.5)/6);
}
if (H_status==1){//if has lost green blob
  O_1 =left-right+(drand48());
}
if (robotBlobF->GetCount()==0){
  H_status=1;
}else{
  H_status=0;
}
 robotPos->SetSpeed(O_0,O_1);
```

6.4.2 Second Experiment

The angle of the robots is variated in +45Â° to -45Â°, the starting position is also variated by +-1 in x and y direction. With this, the difficulty is increased. The resulting nets show quite interesting behavior, the prey is ready for the hunt.

6.5 Step 4: Create the scenario

When I tried to start two full- grown neural net driven controllers for the Player-Stage Simulation, some strange IP-problems occured. I was not able to solve them in time.

6.6 Step 5: Porting it to the robot

After solving some strange problems, the framework was ported sucessfully on the robot. It was necessary to add a strict memory management to the framework, because the available memory space was reduced dramatically (6GB on the Simulation computer to 54MB on the Robot). I was not able to make a mapping between the differential drive of the Player-Stage Robot and the screw drive of the Karlsruhe-Robot, so the portation of the pre-evolved nets also wasn't possible.

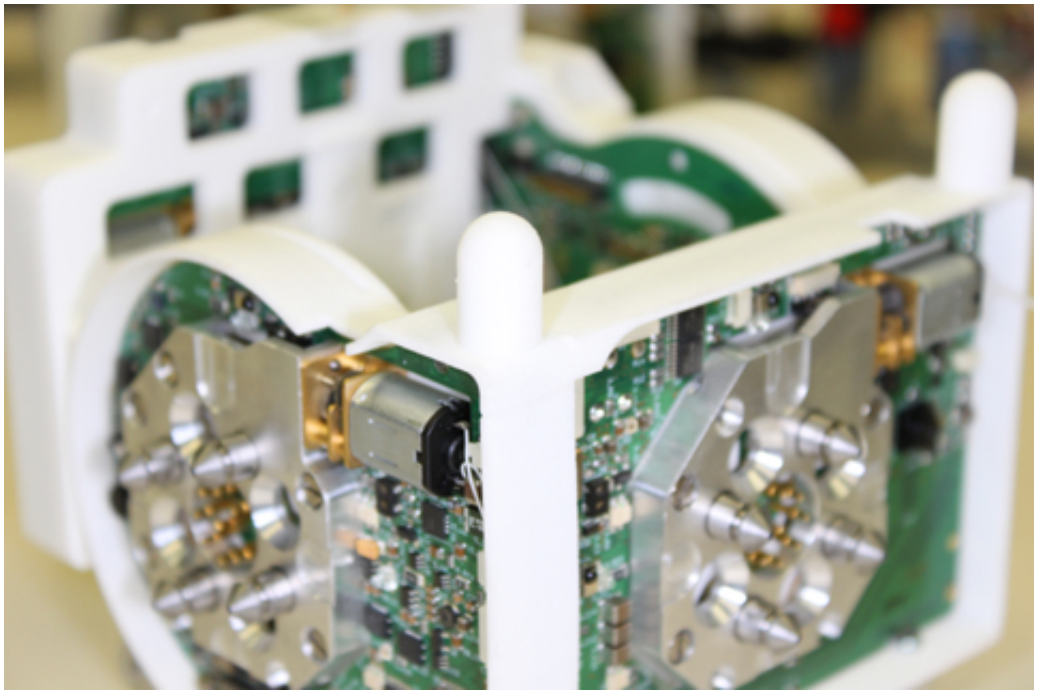


Figure 6.21: A picture of a real KaBot- robot used in the experiments.

6.6.1 First Run

The first on-line evolutionary experiment, evolving Collision Avoidance, ran quite sucessful: After 2 Generations, a net occurred, which drove the robot away from

any obstacle. An error which occured randomly made it impossible to have a long run without reseting and restarting the robot. The structure of the framework saved every progress though, so the only thing lost was the time for restarting.

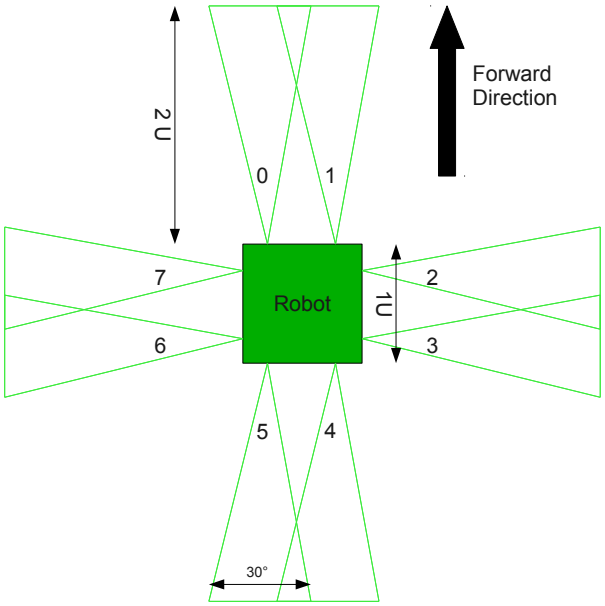


Figure 6.22: The schematics of the KaBot- robot used in the experiments.

The Configuration was:

- `(EANTLearningRate)` "0.3"
- `(EANTinitialPop_m)` "3"
- `(EANTMutationProb)` "0.3"
- `(EANTNewSubNetInputProbability)` "0.4"

- (CONTROLLER_numberofnets) "14"
- (CONTROLLER_exploitationlenght) "6"
- (CONTROLLER_crossoverchance) "0.5"

The output neurons were mapped directly to the two screws. There were three input neurons, (as already used in the collision avoidance tests before):

- InputData[0]= (SI.IR[0].Distance+SI.IR[7].Distance)/2;
- InputData[1]= (SI.IR[1].Distance+SI.IR[2].Distance)/2;
- InputData[2]= (SI.IR[0].Distance+SI.IR[1].Distance)/2;

The fitness function equaled the minimum measured distance of all IR-Sensors to an obstacle.

6.6.2 Results

The best performing net(in CGE) of the first run with a fitness of 0.467 was:

N 0	N 2	I b	I c	I a	I b
1.56735	0.0739657	-1.98729	1.29426	1.02371	-0.721217
4	2	-	-	-	-
I c	N 1	I a	I b	I c	
-0.877162	0.218482	-2.25123	-0.944345	0.805553	
-	3	-	-	-	

Table 6.8: The best net of the first experiment.

6.6.3 The second experiment

The purpose of the second experiment is to make a very long running evolution on the symbrion robot platform. The number of nets is reduced to 8, so more generations can be evolved in the same time. A forth input data is added, containing the average value of the rear IR-sensors. The experiment is run twice to lessen the effect of coincidence.

6.6.4 Results

A problem that occured at the long-term experiment was, that the hardware was not very reliable: The strength of the battery varied, the MSP's containing the infrared sensors went out of order, one of the screws stoped turning, etc. . Because of that, sometimes the best nets didn't survive a generation. After 12 generations

of structural mutations and additional 36 generations of parametric mutations the following two nets were the best of these experiments.

N 0 0.573426	N 4 0 2	I b 0 -	I d 0 -	N 3 0.3206643	JR 3 -0.197967 -
I b -0.685574 -	I c 1.09614 -	I a 0.447226 -	I b -1.23115 -	I c 0.853186 -	I d -0.919949 -
N 1 1.272075	N 2 -1.357933	JR 0 0.847707 -	I b 0.670806 -	I c 0.670806 -	I a 0.615895 -
I b -0.93937 -	I c 0.03579 -	I d 0.00211 -			

Table 6.9: The best net of the first experiment.

N 0 1.851445	N 2 -0.6392312	I b -0.304295 -	I c 0.998921 -	I a -0.364638 -	I b 0.87643 -
I c -3.80329 -	I d 1.45204 -	N 1 -0.1933294	I a 4.10256 -	I b -0.00263234 -	I c 0.222868 -
I d 0.579833 -					

Table 6.10: The best net of the second experiment.

6.7 Test for multi- controller capability

To verify the capability to integrate evolutionary and classical controllers a simple experiment is used: The Robot uses the hardcoded exploration algorithm 'Explore1' until the camera recognizes the target. Then the robot changes to hunting mode 'Hunter1', which uses the resulting nets from Step 2 and tries to catch it with an evolutionary created net. The experiment is made 5 times.

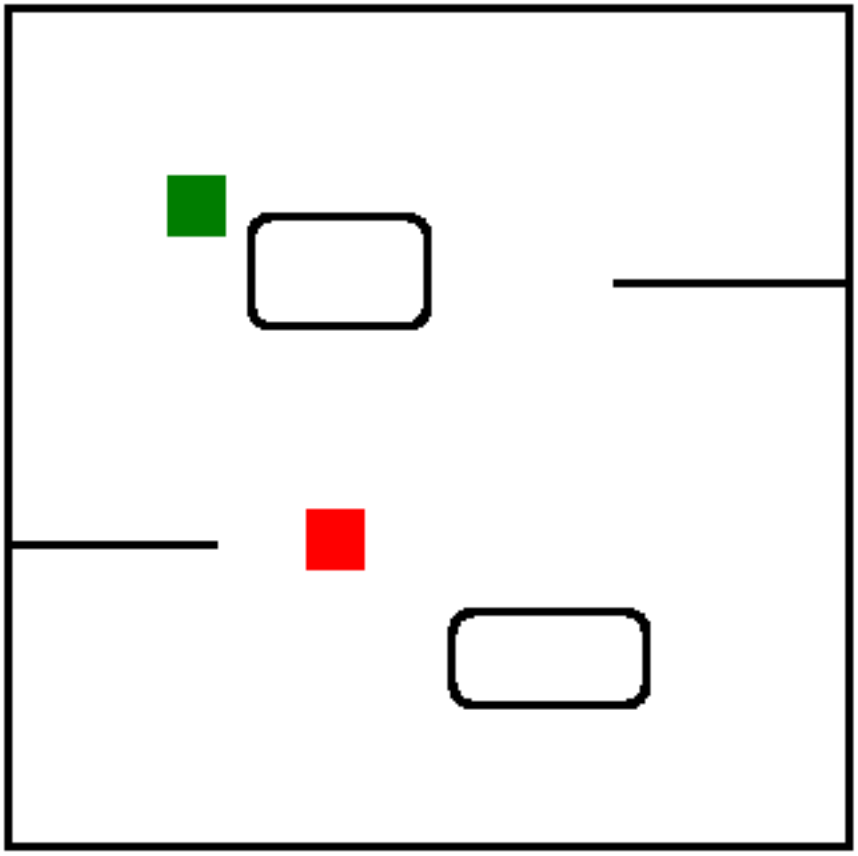


Figure 6.23: Experimental setup for the test of multi- controller capability.

The PetriNet of the experiment looks like this:
Places:
1;Explore1;1;
2;Hunter1;0;
3;END;0;

Transitions:
01;SeenColor;2;
02;TickTimer;200;

Links:
1;01;
01;2;
2;02;
02;3;

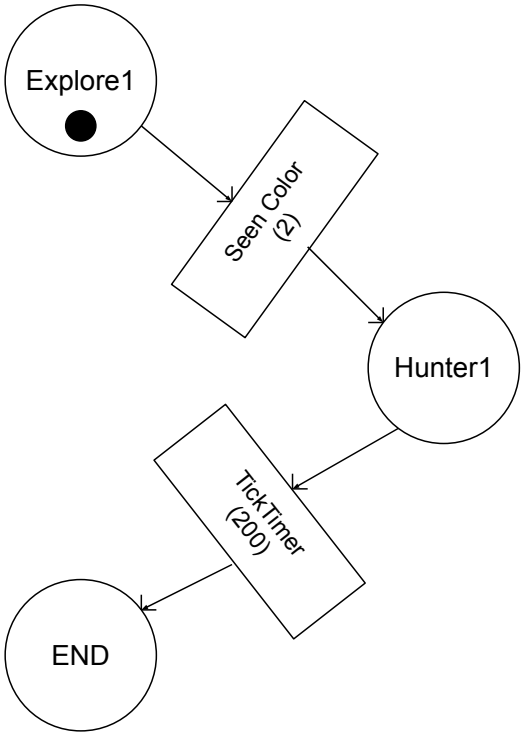


Figure 6.24: Graphical display of the used Petri Net.

6.7.1 Results

The red robot reached the green one after 30 to 60 seconds. The change between the exploration and the hunting part worked well, with no problems at all. This prooves the framework is capable of integrating evolutionary and classical controllers.

7 Conclusion

7.1 Platform-independent sensors and actors

One of the goals of this work was to find abstractions for sensors and actors, so that a given controller could be used by different platforms. This goal was only achieved partly:

- The sensor abstraction worked quite well. It was possible to find mappings between simulation, framework and real robot which allowed at least qualitative portability.

- The actuator abstraction didn't work. I was not able to find a mapping between the differential drive of the simulation and the screwdrive of the Karlsruhe Robots.

If there would have been another working robot with differential drive, the platform-independance would have been achieved. It could be part of further work to port the framework to other platforms.

7.2 Hierarchical approach

When implementing the framework, I encountered a problem: To make structural mutations in the top level of the hierarchical Framework, it is necessary to develop controllers dynamically. For this, there is the need for a sensor- input neuron mapping, a output neuron- actuator mapping, and the generation of a fitness function to get low-level evolution working.
When a top-level mutation generates a new controller it is not possible to guess what purpose this should have. So it is also not possible to guess a corresponding fitness-function.
Because of this, the top level part of the hierarchical approach had to be purely hand-made. With this reduction of the original plan the hierarchical approach, as implemented, workes.

7.3 Integration of classical and evolutionary controllers

Another important goal was the integration of classical and evolutionary controllers in one framework. This was achieved, as proved by the last experiments. The Change between classical and evolutionary controllers made by events workes, even if the full possibilities of this construction, as parallel execution of controllers and more complex petri-nets, were not tested yet.

7.4 Useability of the Framework

The framework as percieved during the experiments shows, that it is an easy-to-use framework, which allows to simulate complex behaviour by putting together basic parts. It may be a step towards the solution of complex real-world challenges.

Bibliography

[Ant90] P.J. Antsaklis. Neural networks in control systems. *IEEE Control Systems Magazine , Vol.10, No.3, pp.3-5*, 1990.

[BS08] et al. B. Siciliano, O. Khatib. *Handbook of Robotics*. Springer-Verlag Berlin Heidelberg, 2008.

[Cla07] Prof. Claus. Vorlesung: Evolutionre algorithmen, 2007.

[CM06] Nikolaus Correll and Alcherio Martinoli. Towards optimal control of self-organized robotic inspection systems. 2006.

[ea09] Michel Diaz et al. *Petri Nets: Fundamental Models, Verification and Applications*. Wiley-ISTE, 2009.

[HJH98] Robert P. Reid H. James Harrington, Glen D. Hoffherr. *Statistical analysis simplified : the easy-to-understand guide to spc and data analysis*. New York : McGraw-Hill, 1998.

[Kir09] Y. Kassahun' Jan Hendrik Metzen' Mark Edgington' Frank Kirchner. Incremental acquisition of neural structures through evolution, 2009.

[Kön07] Lukas König. A model for developing behavioral patterns on multi-robot organisms using concepts of natural evolution. *Diplomarbeit*, 2659, 2007.

[Mat10] Michael Mattes. Design and implementation of a framework for online evolution of robotic behaviour. *Diplomarbeit*, 2684, 2010.

[McC43] Pitts McCulloch. A logical calculus of the idea immanent in nervous activity. *Bulletin of Mathematical Biophysics, 5, 115-133*, 1943.

[Pet81] James Lyle Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall, 1981.

[Poh11] Hartmut Pohlheim. Geatbx: Genetic and evolutionary algorithm toolbox for use with matlab, 2011. http://www.geatbx.com/docu/algindex.html.

[Rei00] Heinrich Reichelt. *Neurobiologie*. Thieme, Stuttgart, 2000.

[RN03] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: a modern approach*. Prentice Hall, 2nd international edition edition, 2003.

[SK] H. P. Schwefel and F. Kursawe. Künstliche evolution als modell für natürliche intelligenz. *Technische Biologie und Bionik 1, Proceedings 1. Bionik-Kongress, BIONA report 8*.

Declaration

All the work contained within this thesis, except where otherwise acknowledged, was solely the effort of the author. At no stage was any collaboration entered into with any other party.

(Jochen Haag)