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

Diplomarbeit Nr. 3291

Evolution of Coordinated Behavior in a Heterogeneous Robot Swarm

Katja Deuschl

Course of Study:	Computer Science
Examiner:	Prof. Dr. rer. nat. habil. Paul Levi
Supervisor:	Dipl.-Inf. Florian Schlachter
Commenced:	27.09.2011
Completed:	29.03.2012
CR-Classification:	I2.2, I.2.6, I.2.9, I.2.11

Abstract

In cooperative and collective robotic systems, a swarm as a whole can handle situations and solve problems that single robots cannot, although each of the single robots of the system is completely autonomous. Single robots can exploit their physical conjunctions like docking elements for example to join together into artificial organisms which are capable of dealing with a changing environment and challenges that are too complex for a single robot.

This thesis is part of the Symbrion and Replicator project funded by the European Commission and investigates whether robots can be forced to develop cooperative behavior in a swarm by using an evolutionary approach. For this purpose we consider a task that can only be performed by two or more robots cooperatively, like a rescue scenario, bilateral docking or coordinated locomotion. Because the fitness function according to this task can be very complex, we first break the whole task into simpler ones and examine them by the use of related work and modern approaches as for example CGE, the Common Genetic Encoding or EANT, the Evolutionary Acquisition of Neural Network Topologies. All applied approaches are biologically motivated and increase the level of realism in neural simulations. Afterwards, these approaches will be applied to more complex scenarios. Because a serial artificial evolution of individual robots on a physical robot might require quite a long time, the system is first tested using a simulated scenario. Based on the existing evolutionary framework, the controllers will be evolved and evaluated online and onboard. This thesis concludes with presentations and sample scenarios which illustrate the evolved coordinated behavior in solving a variety of different tasks.

Contents

List of Figures

List of Tables

1 Introduction

In cooperative and collective robotic systems, a swarm as a whole can handle situations and solve problems that single robots can not, although each of the single robots of the system is completely autonomous. Single robots can exploit their physical conjunctions like docking elements for example to join together into artificial organisms which are capable of dealing with a changing environment and challenges that are too complex for a single robot. "An important goal of collective robotics is the design of control systems that allow a group of robots to accomplish common tasks by coordinating without a centralized control."[BTB+07]

This thesis is part of the Symbrion and Replicator project funded by the European Commission and investigates whether robots can be forced to develop collective behavior in a swarm by using an evolutionary approach. For this purpose we consider a task that can only be performed by two or more robots cooperatively, like a rescue scenario, bilateral docking or coordinated locomotion. Because the fitness function according to this task can be very complex, we first break the whole task into simpler ones regarding the deliberations of Nolfi and Floreano in [NF00] and examine them by the use of related work and modern approaches as for example CGE, the Common Genetic Encoding [KSE+07] or EANT, the Evolutionary Acquisition of Neural Network Topologies [KS05b]. All applied approaches are biologically motivated and increase the level of realism in neural simulations. Afterwards, these approaches will be applied to more complex scenarios. Because a serial artificial evolution of individual robots on a physical robot might require quite a long time, the system is first tested using a simulated scenario. Based on the existing evolutionary framework, the controllers will be evolved and evaluated online and onboard. This thesis concludes with presentations and sample scenarios which illustrate the evolved cooperative behavior in solving a variety of different tasks.

1.1 Symbrion Replicator

This section gives a brief introduction into the two European Commission funded open source and open science projects Symbrion and Replicator. Both projects share common research fields and some common problems. The key idea of these projects originates from a biological observation of symbiotic organisms. A symbiosis is a cooperative behavior between members of different species. For instance, individual elements can couple up and build more complex organisms with different functionalities. In such a union individual elements specialize or share resources such as energy. Further more, these organisms can disaggregate and exist further as stand-alone elements. "The main focus of the Symbrion and the Replicator project is to investigate and develop novel principles of adaptation and evolution for symbiotic multi-robot organisms based on bio-inspired approaches and modern computing paradigms. Such robot organisms consist of super-large-scale swarms of robots, which can dock with each other and symbiotically share energy and computational resources within a single artificial-life-form."[Sym]

Symbrion [Sym]

The expression "Symbrion" stands for "Symbiotic Evolutionary Robot Organisms". The Symbrion project is focused on investigation and development of novel principles of adaptation and is exploring artificial evolution in robotic population based on bio-inspired paradigms. The robotic population consists of super-large-scale swarms of robots building an organism. Like in a biological organism the members can interconnect with each other by means of docking elements or symbiotically share energy and computational resources.

Replicator [Rep]

"Replicator" is the acronym for "Robotic Evolutionary Self-Programming and Self-Assembling Organisms". This project is closely related with the Symbrion project and focuses on the problem of reconfigurability of actuators and sensors, learning strategies for symbiotic robot systems as well as adaptive control structures. Thus, an advanced robotic system, consisting of a swarm of small autonomous mobile micro-robots are capable of self-assembling into large artificial organisms which is based on modular sub-systems that can be autonomously reconfigured.

1.2 Outline

This thesis is divided into 9 chapters which are outlined in this section.

Chapter 1 - Introduction The introduction chapter covers briefly the Symbrion and the Replicator project which this thesis is part of.

Chapter 2 - Problem Statement This chapter describes the problem statement covered in this diploma thesis.

Chapter 3 - Nature as a Role Model Chapter 3 describes how nature can be used as a role model for modern approaches by examining the functionality of the human brain and the swarm behavior of ants. At the end of this chapter a possible mapping from biological patterns to artificial methods is presented.

Chapter 4 - Evolutionary Approaches The fourth chapter gives an overview of evolutionary approaches such as evolutionary algorithms. Furthermore, related work already done in this field of research so far which implements the methods introduced in the previous chapter is presented.

Chapter 5 - Implementation Chapter 5 is separated into three section. The first section covers the applied software such as the specification and description of the simulation tool Player/Stage as well as tools used for the visualization and documentation of the code and the results. You can read about the *Evo-RoF* framework this thesis is based on in the second section. This is followed by the presentation of implemented extensions and developed approaches. Chapter 5 concludes with information about how the task is executed in simulation.

Chapter 6 - Experiments During the research for this thesis challenges arose that have to be met such as the choice of the fitness function or the mapping from real robots into the simulation. These and more preliminary considerations are pointed out in chapter 6 which ends with a detailed description of four experimental setups and their execution steps.

Chapter 7 - Results and Evaluation Firstly, the simulation results are being represented. Also, a discussion of the results and evaluations in simulation is revealed in chapter 7.

Chapter 9 - Conclusion The last chapter concludes with an outlook on possible future research topics and on further development.

2 Problem Statement

In this thesis, the artificial evolution of cooperation between two or more robots is in the main focus. Using evolutionary processes, the robots should learn to cooperate and fulfill a common task together, like bilateral docking, rescue scenario or coordinated locomotion. Based on an existing evolutionary framework, the controllers should be evolved and evaluated online and onboard. Finally, experiments and demonstrators should show the evolved cooperation in different tasks.

To achieve those goals we are breaking down the problem statement into the following steps:

Improve and Extend Evolutionary Framework

As a first step we need to extend the existing evolutionary framework, created and developed by Schlachter et al. in [SADL12], to support the evaluation of swarm based robotics. Those extensions are based on the existing techniques described in the related work in chapter 4.2.

Validate Extended Framework

Once those extensions are implemented, this extended framework was validated using a few less complex tasks such as the collision avoidance as test scenarios.

Cooperative Behavior

The final step is to show cooperative behavior of two or more robots using this extended framework. The preliminary step here is to navigate cooperatively to a specific reference point. Here the reference point is marked by surrounding light barriers which can only be passed together. Together in this context means, that the robots can only have a certain distance to each other. In the extended version of this scenario - motivated by the foraging of ants - the robots should navigate back and forth between more than one reference points.

3 Nature as a Role Model

"That which is not good for the swarm, neither is it good for the bee."
(Marcus Aurelius *Meditations*, book VI. 167 AD)

Even in early history, man gained inspiration from nature. A growing number of disciplines in the area of computer science take nature as a role model, when coping with challenges in an ever more complex world. Nature amazes scientists over and over when it comes to the highly complex behavior of animals in huge groups or swarms. For example swarms of birds can help calculating the optimization of air resistance in the field of aerodynamics. Another common example are ants, helping to solve the Traveling Salesman Problem (TSP)[1] [JM97].

Figure 3.1: A colony of ants [The].

Therefore scientists are using the rules every single member of a swarm simply follows. To illustrate this fact, let's consider birds searching for a place to rest. Let's further assume that every bird is following one animal, which receives the highest air resistance. If one individual slightly alters its trajectory - because of detecting a resting place - all the other birds in the swarm will change their direction accordingly. Therefore, the swarm of birds reaches the two goals: they found a place to rest and they experienced as little as possible air resistance and hereby saved energy.

Ants are also a perfect example to illustrate how in the evolution of insects ants manage to survive even in great confusion or chaos by adapting to their environment. The ants' high level of self-organization ensures, that the complete colony

[1]TSP focuses on the problem of finding the shortest path between a set of cities, without visiting one of the cities twice.

continues to function in a stable way even if several participants of the colony (figure 3.1) have been removed.

In this context, self-organization means there exist no central point or leading member, which will be used to control the entire group. [Scia] Or in keeping with Bonabeau et al.:

Self-organization "[...] is a set of dynamical mechanisms whereby structures appear at the global level of a system from interactions of its lower-level components." [BDT99]

During forage, ants leave their marks in form of pheromones along the search paths. Succeeding members of the swarm are using those marks on the ground for orientation. If there is more than one possible path between the anthill and the food source the highest pheromone concentration will indicate the shortest path. Subsequent ants do not automatically follow the attractant, there is just an increased likelihood for the ants to follow the more highlighted route. However, it is important to note, that yet those statistical outliers are very important because without them, the ant colony would only take one standard route which might not be the most efficient path. It is the outlier which makes it possible to find a shortest path. Mathematicians and computer scientists are trying to map these biological functions to mathematical formulas and that way model the problem of the traveling salesman. [Scia]

This chapter covers the motivation behind the approach to replicate models in nature and biology. Techniques are discussed how to transfer biologically swarms to artificially developed swarms of robots. In other words, the attempt to supply natural behavior to artificial organisms. The last sections concentrate on further nature models such as the human brain or the nervous system and introduce into the existing approaches teaching a computer of how to "think".

3.1 Swarm Behavior

Nature thereby offers many samples showing that animals in a group are able to achieve performance a single animal is not capable of. First and foremost the swarm solves survival problems of an individual such as foraging, timely detection of enemies and reproduction. Interesting insights can be drawn from cohabitation of animals in a swarm. Scientists study how members in a swarm-like organism collect, share and process data to mutually reach their most important objective to pass on as much live as possible to the next generation. [Scib]

Such a conduct is referred to as swarm intelligence.

Swarm intelligence: "The emergent collective intelligence of groups of simple agents." (Bonabeau et al, 1999)

As an example for the swarm intelligence of ants see figure 3.2. As reported by Miller in [Nat] a "single ant or bee isn't smart, but their colonies are. As individuals, ants might be tiny dummies, but as colonies they respond quickly and effectively to their environment."

Swarm intelligent organisms are characterized by the following features:

- adaptability
- self-organization
- robustness[2]

In this context "intelligence" does not mean the ability to conceive facts and coherences or the competences to acquire knowledge for problem solving. Rather, the term "intelligence" is used to reveal the ability to establish a group memory and to mutually figure out problems within a group where every member contributes to the solution. Contrary to humans, the cooperation within the group is of essentially importance to achieve the common purpose.

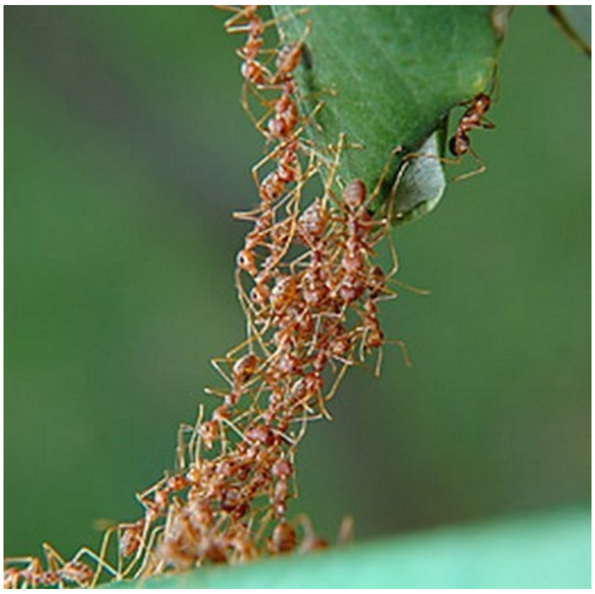


Figure 3.2: An example of the swarm intelligence of ants [Tec].

So far, swarm intelligence applies in computer science and related technical fields in terms of swarm robotics. Swarms are of special interest because they comprise of simple components which are relatively easy to program and to be manufactured

[2]Here, "robustness" means the fulfillment of a task despite the failure of one or more individuals within an organism[Nat]

but are capable of coping with complex tasks. If one part of the organism fails or even drops out does not lead to a complete breakdown of the mission or a failure to perform. The rest continues to fulfill the tasks and replaces the members no longer functioning.

3.1.1 Swarm Robotics

Swarm robotics is the approach of employing swarm behavior in the field of robotics. A large group of relatively simple robots are able to move, perceive environmental information and communicate with other swarm members. They collaborate and interact with each other for the purpose of jointly solve tasks that a single robot cannot cope with because of physical constraints or limited behavioral abilities [DCP+11]. Figure 3.3 depicts an exemplary scenario where several robots of a swarm self-assembled to overcome an obstacle.

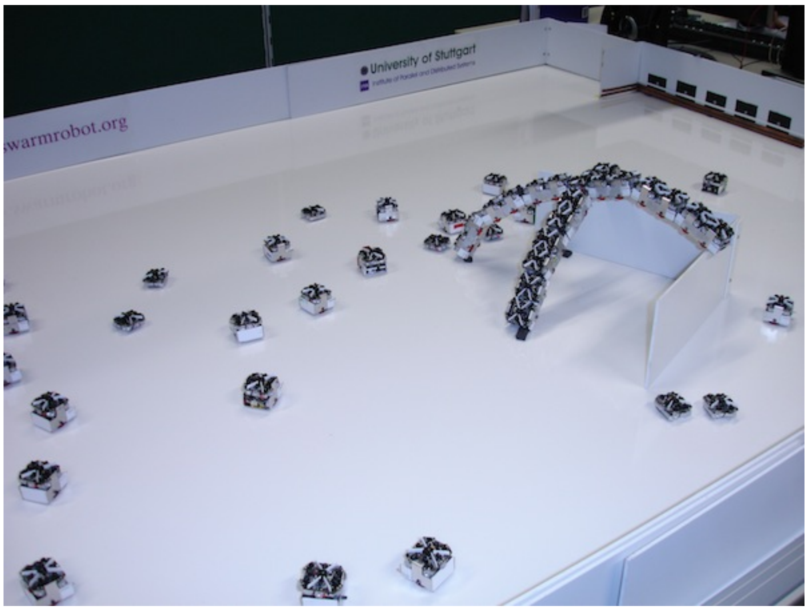


Figure 3.3: Several swarm robots self-assembled to overcome an obstacle in the arena [Sym].

Moreover, swarm robotics emphasizes aspects such as distribution of data, decentralization of control, local communication among robots and the emergence of global behavior. [BTB+07]

"Everything regarding swarm robots is very distributed: They don't all talk to each other. They act on local information. And they're all anonymous. I don't care who moves the chair, as long as somebody moves the chair. To go from one robot to multiple robots, you need all three of those ideas." [Nat]

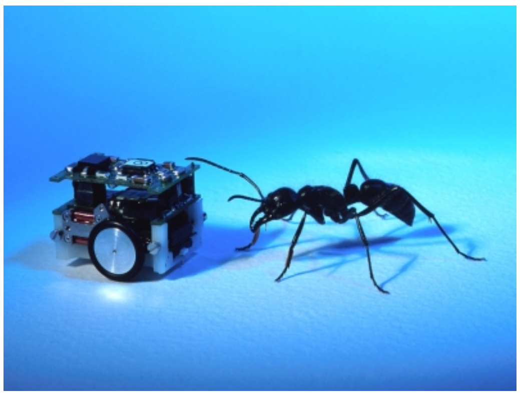


Figure 3.4: Nature versus technology [Sim].

In evolutionary robotics, the design of the robot controller is driven by bio-inspired approaches such as Artificial Neural Networks (ANNs). ANNs play an important role hence their close relationship to natural systems. The next chapter gives a detailed overview of how a neural network works and how it can be mapped onto Artificial Neural Networks.

3.2 Biological Neural Network

3.2.1 Motivation

In the human brain myriads of various neuronal cells are connected in a network processing information in parallel.

When trying to replicate the human brain using a computer, there are certain features which have to be taken into account. Here we have to solve the question of which characteristics of neural information processing are important and how they can be implemented by an artificial system.

Regarding the hardware, there is a need of very well networked neurons with a extreme packing density, slow but highly parallel components without a shared clocking and a low energy consumption. In addition, several tasks have to be achieved simultaneously with a very high fault tolerance. The system needs a high level of adaptivity and thorough design (in nature this design has evolved over the past 500 million years). Regarding the software, a pattern recognition should be considered as well as a complex coordination performance, good real-time features and low precision and computation accuracy for everyday accomplishments. Programming is replaced by learning and the system should be able to process vague data.[DBH01].

According to [Scia] researches are not yet able to totally imitate the human brain,

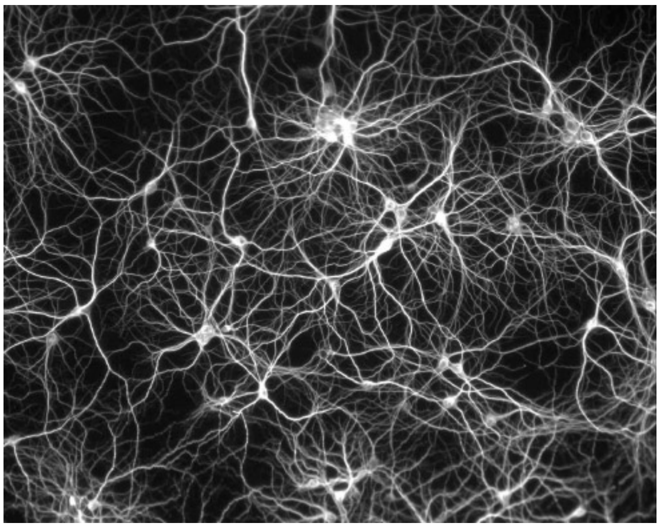


Figure 3.5: Exemplary illustration of the human Neural Network [All].

more precisely, a creatures neurons. Human intelligence with its high form of parallelism can still not be recreated by a computer. If one compares computers with the human brain, the latter processes many data simultaneously while computers handle information consecutively and sequentially. Indeed, individual nerve cells can be understood at the lowest level and simple chemical processes can be visualized by modern imaging techniques. However, to comprehend the concepts lying in-between and to intrinsically clarify networking and activation patterns are still open challenges.

3.2.2 Neural Networks

Neural networks form the information architecture and the structure of the human brain and the nervous system. A population of up to 10^{11} physically interconnected and electrically excitable nerve cells are combined in such a network. The main constituents of such a nerve cell, also called neuron ("neuron" is derived from the Greek *νευρον* meaning "nerve"), are its soma (the cell body), its axon, its synapses and its dendrites as depicted in figure 3.6. Additionally to the functionalities every cell provides, a nerve cell is able to transmit and process information via chemical and electrical signaling. As a receiver element, the neuron collects signals from other neurons converts those signals into electrical activity and forwards them to another place where it transfers them as a transmitter element to other neurons. [Rei00]

Soma The cell body or soma contains the cell nucleus. The dendrites and the axon arise from the soma.

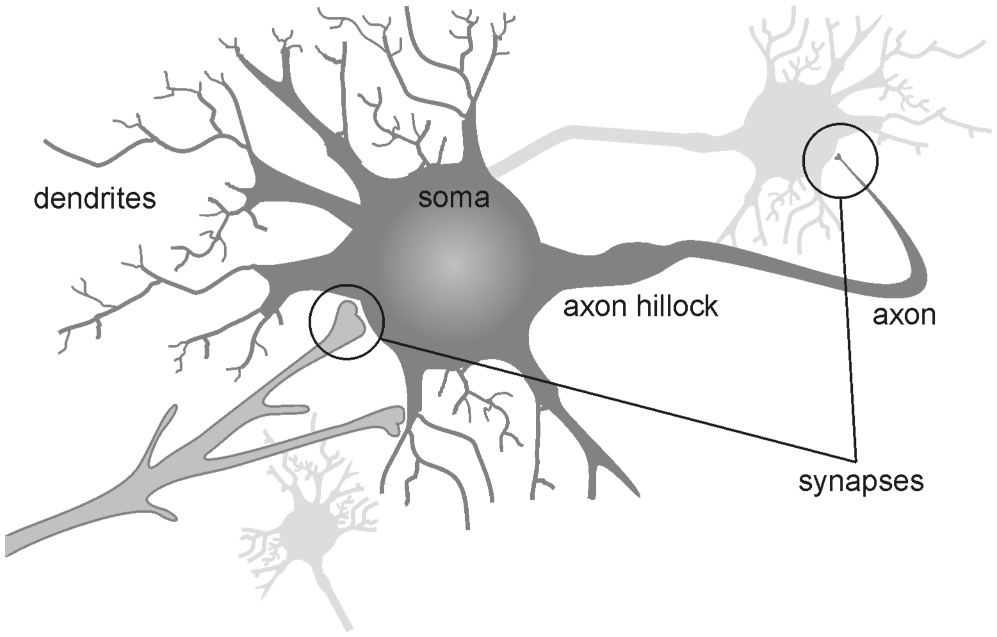


Figure 3.6: Schematic representation of nerve cell (neuron) following [Liv11].

Axon An axon is a particular cellular extension that arises from the soma and serves as electrical conduction. Every neuron has exactly one axon. Close to its target cells an axon splits up into smaller ramifications ending in synapses with other cells.

Axon hillock The axon hillock is situated between the soma and the cell's axon and is responsible for summating all inputs arriving from the dendrites. If a specific threshold value is exceeded, the axon hillock produces the action potential that is transmitted via axon to the recipient cell. This transmission is also called 'firing' of the neuron.

Dendrite Such as the axon the dendrites arise from the cell body, giving rise to a complex dendritic tree. Multiple dendrites are attached to the soma but never more than one axon.

Synapse Synapses are specialized connections to other cells that handle the chemical signaling and therefore the destination to which the neurons transfer their information. In other words, the synapse serves as a chemical contact which receives the axon. In the majority of cases, signals are sent from one neuron's axon to a dendrite of another.

3.2.3 Information Transfer in Neural Networks

A neuron collects signals via synapses from other neurons, which dock to the neuron, its axon or its dendrites. When a voltage variation occurs in a specific area of the dendritic tree the dendrites collect the excitations and transfer them to the cell's axon hillock. Every input alters the cell's membrane potential. Almost simultaneously incoming stimuli add up and establish a potential at the axon hillock that decides whether or not the action potential being triggered or not. In the case of exceeding a particular activation threshold the neuron "fires", that is, the neuron sends out impulses via the axon to the target cells it is connected to. After firing the cell requires time to rebuild its membrane potential and to be able to fire again. This is called repolarization. [Rei00]

3.2.4 Types of Neurons

Between thousand and ten thousand different types of neurons exist in a human brain thereof about a hundred types in the eye alone. Neuron types are classified by shape, location, range of ramification (their dendrites), kind of used neurotransmitter and more. As an example figure 3.7 shows a human Purkinje cell with its dendritic tree. Irrespective of the huge diversity of types, what all neuron

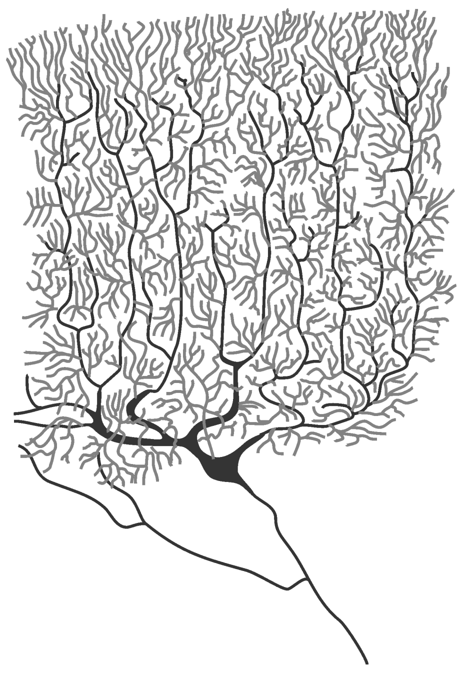


Figure 3.7: Schematic representation of human Purkinje cell following [Rei00].

types have in common is the soma, the synapses and the axon. Generally, a neuron is physically interconnected with about thousand to ten thousand other neurons. [Rei00]

3.3 Artificial Model of Neural Network

One evolutionary approach for the replication of the human brain is the development of an Artificial Neural Network (ANN). It interconnects artificial neurons and transmits information between them as in biological neural networks. It takes an input vector, processes them inside the network and outputs the corresponding result. With regard to robotics, sensor values are entered into the network as input and the output values are transferred to the existing actuators.

3.3.1 Artificial Neurons

As its biological role model an artificial neuron consists of input connections which collect signals from other neurons, a unit for adding up the incoming stimuli thus serving as synaptic summation, another unit responsible for the 'firing' conforming to a specific activation threshold and finally the output which is at the same time the activation for other neurons. Figure 3.8 depicts a schematic representation of an artificial nerve cell. [Liv11]
$x_1 ... x_n$ describe the input to the neuron j, y_j its output. The weights w_{ji} determine

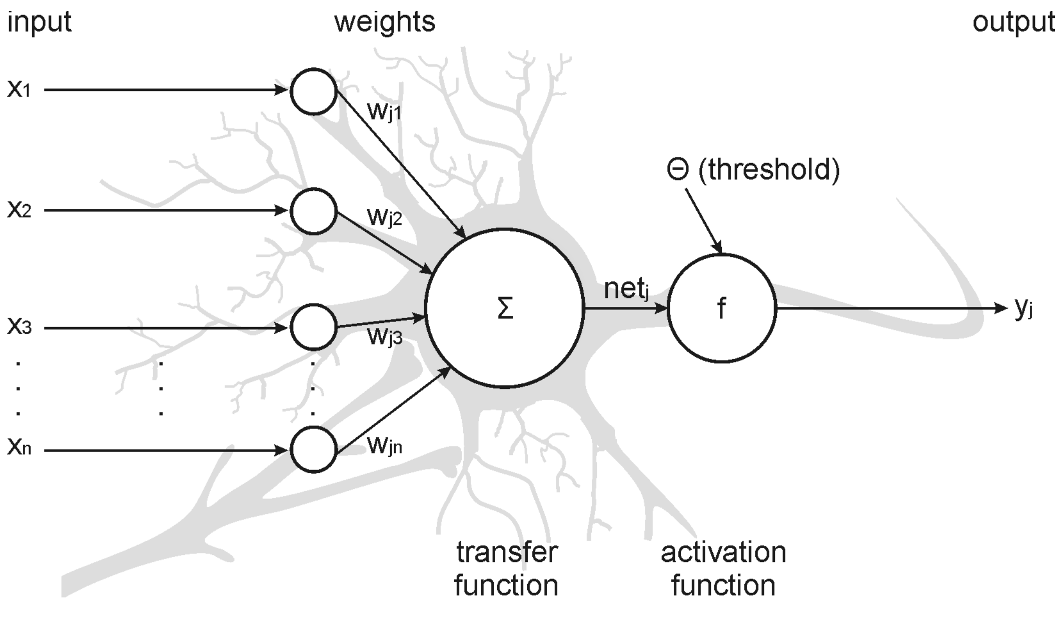


Figure 3.8: Schematic representation of Artificial Neuron j following [Liv11].

the grade of influence that the inputs adopt in later computations of the output. A different sign of a weight effects a different impact. So a positive signed weight $+w_{ji}$ means an exhibitory connection between neuron j and i and a negative signed weight $-w_{ji}$ means an inhibitory connection, respectively. If a weight has zero value, there is no connection between the corresponding neurons. By means of weighting the input values, the transfer function calculates the input for the activation function net_j by

$$net_j = \sum_{i=1}^{n} x_i w_{ji}. \tag{3.1}$$

Commonly used activation functions include the step function, the linear function, the sigmoid function as well as the hyperbolic tangent as listed in figure 3.9 and given by table 3.1.

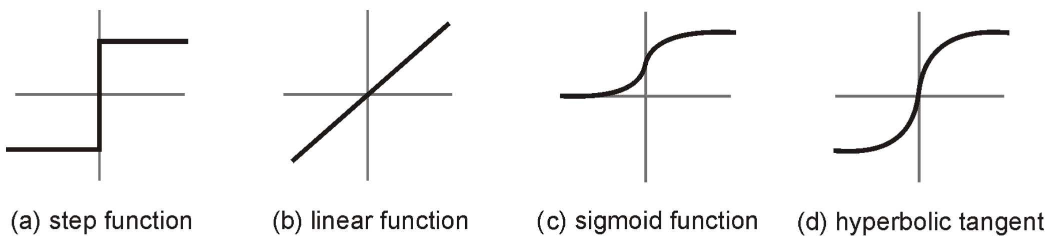


Figure 3.9: Selection of existing activation functions.

Table 3.1: Equations according to the activation functions of figure 3.9.

step function	linear function	sigmoid function	hyperbolic tangent
$f(x) = \begin{cases} -1 & \text{if } x < 0, \\ 1 & \text{if } x \geqq 0 \end{cases}$	$f(x) = x$	$f(x) = \frac{1}{1+e^{-x}}$	$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$

A neuron's output is determined by f_j and derives from net_j and the threshold value of Θ_j by

$$y_j = f_j(net_j - \Theta_j) \tag{3.2}$$

By assigning random values to the connection weights and the threshold and adjusting these values afterwards via learning algorithm the neuron learns the function to be represented. [Yao99]

3.3.2 Artificial Neural Network

Artificial Neural Networks (ANNs) are useful in evolving the control systems of robots, because they provide a straightforward mapping between the motors and the sensors of the robot, enabling them to represent directly the function to be learned.

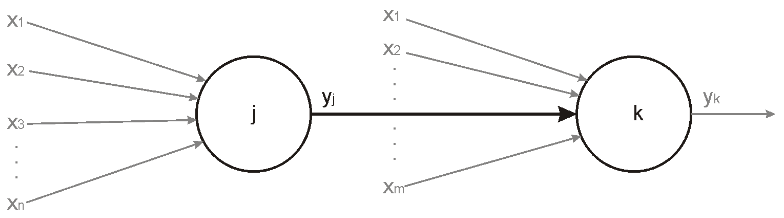


Figure 3.10: Schematic representation of a two-layer Artificial Neural Network consisting of two neurons j and k.

Figure 3.10 represents the network topology called "feedforward". Neurons are grouped into different layers as follows: The input layer contains every input neuron, the output layer is composed of the output nodes, respectively. There exists one more layer that is referred to as the hidden layer. All neurons in a neural network except the inputs and outputs result in being hidden neurons, that arise in the corresponding hidden layer. In a feedforward network each node of a specific layer has only directed connections to the nodes of the following layer towards the output layer. In contrast to another type of network topology called "recurrent network", a feedforward topology must not include any recurrent connection between two neurons. That is for instance a link combining an output neuron with a neuron of a hidden layer. More information about network topologies a neural network can be classified in can be found in [CŁ00].

3.3.3 Learning in Artificial Neural Networks

Learning in ANNs is achieved by adjusting the connection weights iteratively so that a trained ANN can perform a specific task. In most cases, synaptic strengths are initially set to small random values. By repeatedly presenting pairs of input and the corresponding output patterns to the network, learning takes place. After the presentation of each pattern or after the presentation of the entire training set, the modification of weight Δw_{ji} is calculated. The first modality is referred to as "online learning", while the second procedure is designated "offline learning". Updating the old weights w_{ji}^{t-1} is made by adding the newly computed change of weight

Δw_{ji}^t by means of equation 3.3.

$$w_{ji}^t = w_{ji}^{t-1} + \Delta w_{ji}^t \tag{3.3}$$

t is a certain time step, η corresponds to the learning rate and prevents wide oscillations of the weights between consecutive modification steps.

Therefore, learning algorithms focus on the determination of the modification term Δw_{ji}^t as you can see below. [NF00]

In accordance with [Yao99] there exist three common types of learning as follows.

Supervised Learning

Supervised learning is learning from examples provided by an external supervisor. The training set consists of input patterns x with its corresponding correct answer o so that after the output an precise error vector E can be presented to the network.

Reinforcement Learning

Reinforcement learning comprises of input patterns as well, but receives only a rating as answer instead of the correct solution. After passing a run a value is passed back to the network whether the result was right or wrong and how right or wrong. Hence, there is less information to deliver than in supervised learning.

Unsupervised Learning

In unsupervised learning the training set only consists of the input patterns. The network itself attempts to find similarities or categories. The learning objective is implicitly included in the learning rule, thus unsupervised learning requires the least amount of information.

After the training has taken place an ANN is able to find plausible solutions to similar problems of the same class without having explicitly trained them. Thus, an ANN implements the desired adaption derived from biological sources like

- learning ability,
- fault tolerance, and
- generalization ability.

Essential for learning is the learning rule. Frequently used examples of learning rules are the Delta Rule or the Hebbian Rule. The psychologist D. Hebb hypothesizes in 1949 that "the laws of classical conditioning reflected the functioning of the nervous system" [NF00]. Classical or respondent conditioning is a form of behavioral learning. It is based on the combination between response and the inbound stimulus. From this it follows that the Hebb Rule reveals that when linked neurons are stimulated at the same time, their connecting weight is strengthened. Both, the Delta Rule as well as the Hebbian Rule, are weight updating rules which determine how connection weights are adapted.

A generalization of the Delta Rule is Backpropagation. Backpropagation is a supervised learning method and means as much as the propagation of errors back into the net. The Backpropagation algorithm transcribes the following steps [Tok]:

Step 1: the input x is fed into the network, is forward propagated through the network to calculate the output y

Step 2: by means of the expected set values o the error E is computed as follows:

- in the case where neuron j is an output node:

$$E_j = f'(x_j)(o_j - y_j) \tag{3.4}$$

- in the case where neuron j is a hidden node (where m is in the range of every neuron downstream of j):

$$E_j = f'(x_j)\sum_m (E_m - w_{mj}) \tag{3.5}$$

Step 3: compute the modification of weights with learning rate η by

$$\Delta w_{ji} = \eta E_j y_j \tag{3.6}$$

Step 4: repeat steps 1 to 3 until E is less than a specified value ϵ

Consider the activation function $f(x)$ to be sigmoid, that leads to equation 3.7.

$$f'(x) = f(x)(1 - f(x)) \tag{3.7}$$

Thus, the calculation of E for an output neuron j can be simplified by equation 3.8 and for a hidden neuron through equation 3.9, respectively.

$$E_j = (1 - y_j)(o_j - y_j) \tag{3.8}$$

$$E_j = y_j(1 - y_j)\sum_m (E_m - w_{mj}) \tag{3.9}$$

3.3.4 Mutation of Artificial Neural Networks

Mutation of a neural network is performed by either of the three methods:

Parametric mutation By parametric mutation the weights of the network are perturbed usually randomly.

Structural mutation Mutation of the structure refers to the topology of the network, so to the nodes and interconnections. By structural mutation links may be added or deleted or the genome is extended by new neurons.

Crossover Crossover consults two different genomes and merges them into a new one. The so called *special crossover* remains the common parts of two networks and extends the resulting genome by the particular disparate parts. A more detailed description follows in chapter 4.2.4.

3.3.5 Reinforcement Learning vs. Other Kinds of Learning

Considering a goal-oriented agent in interaction with its changing environment reinforcement learning is learning how to map situations to actions so as to maximize its reward. In doing so it explicitly considers the whole problem instead of focusing on subproblems what can lead to considerable limitations. As in other kinds of learning the agent is told which actions to carry out, however, in this case the agent must observe which actions result in the most reward by trying them. Another difference is the trade-off between exploration and exploitation. Simultaneously, the agent has to exploit what it has already learned so far so as to gain reward and to explore a diversity of actions and incrementally favor those that appear to be best so as to perform better selections in future. Obviously, the actions an agent tries may include failures as well. [SB98]

Note that in the following chapters the term "evolution" refers to artificial evolution.

4 Evolutionary Approaches

This chapter introduces the basics of evolutionary robotics and points out the state of research in this topic up to now. Then a selection of related work is presented.

Today's Modern Evolutionary Synthesis is based on the following three pillars [Nis97]:

- Darwinian idea of an interplay between variation and selection
- classical genetics
- population genetics

Here we can identify the following three evolutionary factors: mutation, selection and recombination. Mutations serve as a material supplier for evolution and effect spontaneous changes in the genetic material what sometimes causes the occurrence of different phenotypical variants of the basic type of a kind. At this point natural selection takes place. Functionally most suitable individuals are preferred, which have the best chances in passing their hereditary dispositions. Selection is a stochastic process that also takes less well adapted individuals with low reproduction probability into account. During recombination the genetic material is newly shuffled. By means of the mixture of paternal and maternal chromosomes, more advantageous alleles[1] can be combined in an individual achieving a selective advantage. Therefore, new phenotypes[2] not only arise from genotypes[3] through mutation, but through recombination as well. [Nis97]

4.1 EA - Evolutionary Algorithms

Evolutionary Algorithms (EA) correspond to a class of stochastic search algorithms that were developed from ideas and principles of natural evolution. [Yao99] The main trends in Evolutionary Algorithms are Evolutionary Strategy (ES), Genetic Algorithms (GA) and Evolutionary Programming (EP). One important feature of all

[1]the form or manifestation, a gene can take on (from the Greek $\alpha\lambda\lambda\eta\lambda\omega\nu$ (*allélon*) meaning " each other, mutual ")
[2]phenotype is the collection of observable traits of an individual [Les07]
[3]genotype means the complement of genes of an individual [Les07]

three is the population-based search strategy instead of a point-based search strategy. Intelligent search processes use stochastic procedural elements that create and evaluate new solutions over and over again. The target-oriented search is focused on adequate solutions that are searched for from different points simultaneously and only needs the fitness value of the considered solutions.

Fitness The fitness of an individual considers not only survival of the best individual but also its success with regard to producing offspring compared to other members of the same species. Combination of the features of an individual determines its fitness. Therefore, the fitness function is used to evaluate the performance of individuals and to select the fittest. The form of a fitness function affects the result of an evolutionary sequence significantly [NF00].

Entering the features of the individuals into a coordinate system leads to a fitness landscape which contains the individuals of a population at different points. An example of a fitness landscape for two features is shown in figure 4.1. This fitness

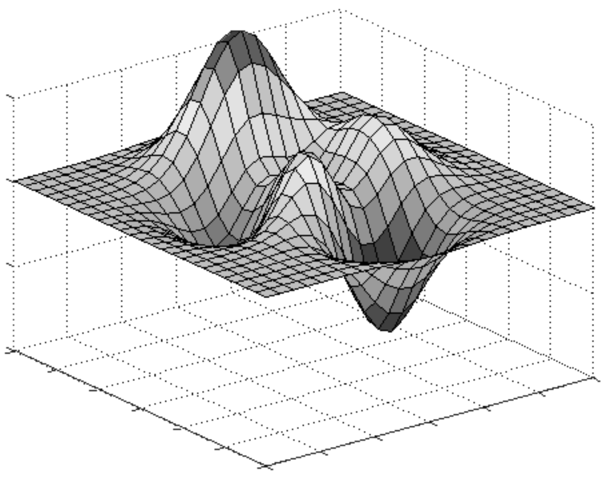


Figure 4.1: Exemplary representation of a fitness landscape.

landscape alters with a changing environment. Artificial evolution is considered to be a hill climbing in this landscape.

By parametric or structural mutation an individual reaches the optimum in the fitness landscape. Mutation can lead to jumps in the landscape. Thus an individual which is stuck in a local optimum has the chance to advance to the global maximum. Additionally the evolutionary based approach has the following advantages compared to the gradient-based approach [Yao99]:

- can handle global search problems better in vast, complex and non-differentiable surface

- does not depend on gradient info that is sometimes unavailable or costly to obtain
- generally much less sensitive to initial conditions
- always search for global optimum solution - the gradient descent algorithm can only find a local optimum in a neighborhood of the initial solution

Algorithm 4.1 and figure 4.2 show the common flow diagram of an EA and its basic cycle, respectively following [Nis97]. The starting population P(0) is generated stochastically. A possible break condition is a maximum time t_{max} or the variance between the fitness of two consecutive populations is less than a specified value ϵ. Step 5 to 11 describes one generation cycle.

```
 1  choose strategy parameters
 2  initialize starting population P(0)
 3  t = 0
 4  evaluate the individuals of population P(0)
 5  while a break conditions is not fulfilled
 6    t = t + 1
 7    selection (choosing the parents)
 8    replication and variation (producing offspring)
 9    evaluate the offspring
10    create new population P(t)
11  loop
12  show the results
13  stop
```

Algorithms 4.1: Common scheme of an EA [Nis97].

EA's basic operators are variation and selection, which are the driving forces of the evolutionary process.

Variation In this context variation means stochastic deviation causing offspring to receive different fitness/suitability. Variation includes mutation and crossover.

Selection Selection denotes natural selection, which is one of the main forces driving evolutionary change. Members of a population that are better suited to their environmental conditions due to their features have an increased chance of producing offspring and of passing on their genetic material.

Advantages of EA as an optimization method are:

- wide-scale application of the basic procedures
- adaptation to problem definitions

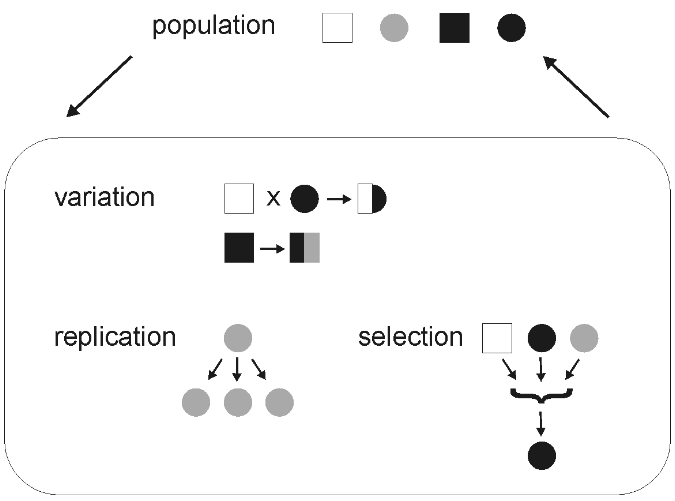


Figure 4.2: Basic EA cycle [Nis97].

- useful for managing large and complex problems that generate many local optima
- objective function does not have to be differentiable or continuous
- well combinable with other approaches
- less likely to be trapped in a local minima than a traditional gradient-based search algorithm
- does not need gradient information, therefore suitable for problems, where gradient information is not available or costly to obtain
- is able to handle problems where no exact objective function is available

Regarding these advantages, EA is shown to be much more robust than many other search algorithms.

4.2 Related Work

This section covers existing and topic-related work, which has a major influence on the further development of the existing framework.

4.2.1 CGE - Common Genetic Encoding

The Common Genetic Encoding approach is used to encode a linear genome, that represents the genotype of a neural network.

Encoding Schemes of Architectures

There exist two different schemes of encoding the architecture of a neural network. The indirect approach encodes only the most important parameters as for example the number of hidden nodes. In the direct encoding, every link and every node can be specified by it's chromosome.

As in [Yao99], an indirect encoding is biologically more plausible than the direct version, "because it is impossible for genetic information encoded in chromosomes to specify independently the whole nervous system according to the discoveries of neuroscience."

Encoding with CGE

"As a direct encoding method, CGE allows the implicit evaluation of an encoded phenotype without the need to decode the phenotype from the genotype."[KSE+07] That implies an easier implementation using only float datatypes when computing the network's output. Also less complex operations are necessary when evaluating the linear genome because there is no need to work on tree structures or graphs.

There exists a competing convention problem caused by many-to-one mapping from the genotype representation to the actual phenotype (neural network) [Yao99]. The same artificial neural networks with the same functionality but different order of hidden neurons have varying chromosomes. This makes the use of crossover operators inefficient as well as the production of good offspring when using the indirect encoding method. For these reasons, the direct encoding scheme is used in this thesis.

In CGE, a genotype corresponds to a string of genes also called a linear genome.

Linear genome
In a linear genome, the topology of a neural network encoded by it is implicitly represented by the ordering of the genes in the genome. In the array-like representation, neuron nodes come before the input nodes and the jumper connections (see table 4.1 as an example). In CGE a linear genome is defined to be complete, compact and closed. Complete in the way that it can be used to represent any type of neural network, compact in that the length of the linear genome is equal to the number of synaptic weights of the network and closed since all genotypes produced can be mapped into a valid set of phenotype networks.[KSE+07]

Each gene can take on a specific allele. That is either a vertex gene (N), an input gene (I), a forward jumper gene (JF) or a recurrent jumper gene (JR). The depth of a node in a network is defined as the minimal number of connections to be traversed from a given output to the node itself. In addition, the path must contain no explicitly defined connections.

N Vertex genes encode a node of the network and are either a hidden or an output neuron. An identification number, a weight, as well as the number of incoming connections are assigned to each vertex gene.

I Sensory signals are introduced into the network by means of the input genes. Every input node gets a specific label, therefore input genes with the same label refer to the same input.

JF Forward jumpers can be implicitly or explicitly encoded in the linear genome. Implicitly in the way a forward jumper gene is ordered in the genotype, then the forward jumper is not explicitly visible in the genotype but would be displayed as an edge in the corresponding phenotype. As well as the vertex genes the explicit representation of forward jumpers contains a unique ID, the number of input connections and additionally the global identification number of its source gene. A forward jumper starts from a node with higher depth and ends at a vertex with a lower depth.

JR The only difference between a recurrent and a forward jumper gene being the kind of the starting and ending vertex. Thus, the recurrent jumper starts and ends at vertices with the same depth or starts at a node with a lower depth and ends at a node with a higher depth.

Table 4.1 shows an example of a genotype that encodes the corresponding valid phenotype in figure 4.3. In addition to parameters like allele, label, id, weight and depth of a gene the table contains information about a gene's parent, the source of a jumper gene as well as the amount of incoming links d_{in}. v computes the number of produced outputs minus the number of expected inputs (1 - d_{in}). It should be noted, that each node in the network must have exactly one output (corresponding to the blue edges in figure 4.3). In the special cases of an vertex node being an input node, a recurrent or a forward jumper, the integer assigned to its value for v is 1. Since every vertex gene must have at least one input (one criterion, a genotype in CGE has to fulfill to be considered a valid genotype according to [KS05a]), the maximum value of v of a neuron node is zero, which is true for all vertex genes with one input. "One important property of a linear genome is that the sum of the integer values assigned to each of the nodes in a linear genome encoding a neural network is the same as the number of outputs of the neural network." [KS05a]

With the values of v it is possible to detect a subgenome of the linear genome in that starting from a vertex gene i and summing up the integer values assigned to the genes following i in the linear genome until the sum of all considered genes is calculated to one. The identification of subgenomes might be a very useful feature when applying genetic operators to a genotype.

Genetic operators
An example of genetic operators to be used in CGE are the parametric mutation,

Table 4.1: Table containing the genotype, that encodes the phenotype in figure 4.3.

allele	N	N	N	I	I	I	N	JF	I	I	JR
label	-	-	-	x	y	y	-	-	x	y	-
id	0	1	3	-	-	-	2	-	-	-	-
weight	0.5	0.5	0.3	0.2	0.6	0.3	0.4	0.1	0.4	0.1	0.7
source	-	-	-	-	-	-	-	3	-	-	0
d_{in}	2	2	2	-	-	-	4	-	-	-	-
v	-1	-1	-1	1	1	1	-3	1	1	1	1
parent	∅	0	1	3	3	1	0	2	2	2	2
depth	0	1	2	-	-	-	1	-	-	-	-

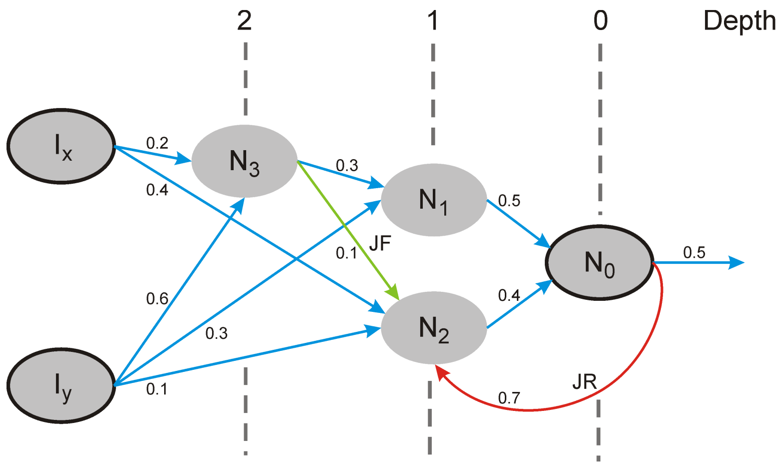


Figure 4.3: Example of a valid phenotype.

the structural mutation as well as a special crossover. Parametric and structural mutation is used by GNARL and is presented in the following subsection. All three search operators are used and described in the Evolutionary Acquisition of Neural Network Topologies (EANT) at the end of this chapter.

4.2.2 GNARL - GeNeralized Acquisition of Recurrent Links

In [ASP94] Angeline et al. describe an approach of simultaneously training a recurrent network structurally and parametrically by means of evolutionary programming (EP). GNARL (acronym for GeNeralized Acquisition of Recurrent Links) is an evolutionary algorithm that creates recurrent neural networks non-monotonically to solve a given task.

Genetic operators

When using genetic algorithms (GA) the main evolutionary operator is crossover. To apply crossover as valid search operator the interpretation function has to be extremely complex to avoid any misleading that happen to appear in neural network when using crossover. Thus, the interpretation function's complexity competes with the real learning problem. In comparison, EP searches in space of neural networks by using structural and parametric mutation and therefore manipulates the network directly. Offspring resemble their parents to a certain extent, hence, the parent's functionality will not get completely lost. For this reason, Angeline et al. have chosen to use EP instead of GA and no crossing over at all.

Parametric mutation Each weight w is mutated parametrically by perturbing it with gaussian noise and the application of a so called instantaneous temperature $\hat{T}$ (see equation X). $\hat{T}$ compensates undesirable side effects, that may happen during large parametric mutations. η represents the network, $N(\mu, \sigma^2)$ is a gaussian random variable and α is a user-defined proportionality constant.

$$w = w + N(0, \alpha\hat{T}(\eta)) \qquad \forall w \in \eta \tag{4.1}$$

Structural mutation The connectivity between nodes and the number of hidden nodes are altered through structural mutation used by GNARL. To avoid radical changes from parent to child, new links are initialized with zero and hidden nodes are added without any connection. "Links must be added by future structural mutations to determine how to incorporate the new computational unit." [ASP94] In GNARL it is possible to remove a node with all it's incident links as well as to delete a link or links, respectively.

Algorithm of GNARL

Construction of networks
The number of inputs m_{in} and outputs m_{out} is determined by the task and therefore set by the user and are not changed by the algorithm. Additionally the maximum number of hidden nodes h_{max} is set by the user as well. Every input node applies the sigmoid function as activation. All links between two nodes are represented by real numbers and must observe the following rules R_1 to R_3:

R_1: there can be no links *to* an input node

R_2: there can be no links *from* an output node

R_3: given two nodes x and y, there is at most one link from x to y

The networks created can have no connections, can be partially connected or fully connected. Formula S shows the search space of GNARL.

$$
\begin{aligned}
\mathrm{S} = \{\, \eta \quad : \quad & \eta \text{ is network with } \mathrm{w} \in \mathbb{R} \\
& \eta \text{ follows } R_1 - R_3 \\
& \eta \text{ has } m_{in} \text{ input nodes} \\
& \eta \text{ has } m_{out} \text{ output nodes} \\
& \eta \text{ has i } \leq h_{max} \text{ hidden nodes } (0 \leq i \leq h_{max}) \,\}
\end{aligned}
$$

Initialization
In the beginning, the population is initialized with randomly generated networks. An example of such a network created by GNARL is illustrated in figure 4.4. The number of initial links and the number of hidden nodes is chosen from uniform distributions over two different user-supplied ranges. Every link's starting and terminal node are chosen in compliance with the structural mutation used by GNARL. In the example below nodes with identifier I are input neurons, H stands for "hidden" node and N corresponds to a output neuron.

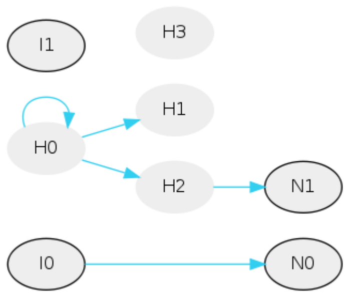


Figure 4.4: Example of initial network created by GNARL algorithm.

Note, that there is nothing in that initialization forcing a node to have any incoming or outgoing link, let alone for a path to be present between the input and output nodes.

Search operation
In each generation of search, the new parents are chosen in accordance to the following two steps:

Step 1: evaluate networks by means of a user-supplied fitness function

Step 2: networks achieving results in the top 50% are marked as the parents of the next generation (all remaining networks are rejected)

In evolving networks, all that GNARL needs is the feedback given by the fitness function. By exploring the space of networks by selection and mutation, the choice of fitness function does not change the mechanics of the GNARL algorithm.

Generating offspring
GNARL executes three steps for generating the offspring:

Step 1: copy the parents

Step 2: define severity of mutation to be performed

Step 3: mutate the copy according to one of the mutations described earlier

4.2.3 NEAT - Neuroevolution of Augmenting Topologies

In [SM02] Stanley and Mikkulainen represent a novel method of Neuroevolution (NE), that is artificial evolution of artificial neural networks using genetic algorithms. Their NeuroEvolution of Augmenting Topologies aims to demonstrate that evolving both, structure and connection weights of a neural network instead of just evolving the weigths and using a fixed topology, can significantly enhance the performance of NE. NEAT is assigned to address the three following issues, that may arise when evolving structure incrementally:

Issue 1: "Is there a genetic representation that allows disparate topologies to crossover in a meaningful way?"

Issue 2: "How can topological innovation that needs a few generations to optimize be protected so that it does not disappear from the population prematurely?"

Issue 3: "How can topologies be minimized throughout evolution without the need for a specially contrived fitness function that measures complexity?"

Initialization

NEAT starts out with a uniform population of networks with zero hidden nodes where all inputs are connected directly to the outputs. New structures are introduced incrementally as structural mutation occurs. By doing so only those structures survive that are considered to be useful through fitness evaluations.

Genetic encoding

The encoding scheme of NEAT is similar to the CGE, described earlier in subsection 4.2.1. Thus, genes are listed in an array-like representation of connection genes. Each of the genes refers to two nodes being connected specifying the source and the target node and the weight of the connection. In contrast to CGE, every connection gene stores information about the connection being enabled or disabled and a so called innovation number. "The innovation numbers are historical markers that identify the original historical ancestor of each gene. New genes are assigned new increasingly higher numbers." [SM02] By means of the innovation number, NEAT is able to crossover genomes in a sensible way (as will be explained below).

Genetic operators

Parametric mutation
As parametric mutation NEAT uses the common approach in NE systems meaning the connection weights to be perturbed or not at each generation by means of a previously determined mutation probability.

Structural mutation
There exist two kinds of structural mutation in NEAT, adding a node or adding a connection. An example of the latter is shown in tables 4.2 and 4.3. As can be seen in the example, each mutation step increases the size of the genome by adding one or more genes. When adding a new node between to nodes to the existing network the old connection gets disabled and two new connections are added to the genome. Thereby, new nodes are integrated into the network immediately.

Table 4.2: Table containing the genotype before adding a new node.

innov 1	innov 2	innov 3	innov 4	innov 5	innov 6
in 1	in 2	in 3	in 2	in 5	in 1
out 4	out 4	out 4	out 5	out 4	out 5
enabled	disabled	enabled	enabled	enabled	enabled

Table 4.3: Table containing the mutated genotype after adding a new node.

innov 1	innov 2	innov 3	innov 4	innov 5	innov 6	innov 8	innov 9
in 1	in 2	in 3	in 2	in 5	in 1	in 3	in 6
out 4	out 4	out 4	out 5	out 4	out 5	out 6	out 4
enabled	disabled	disabled	enabled	enabled	enabled	enabled	enabled

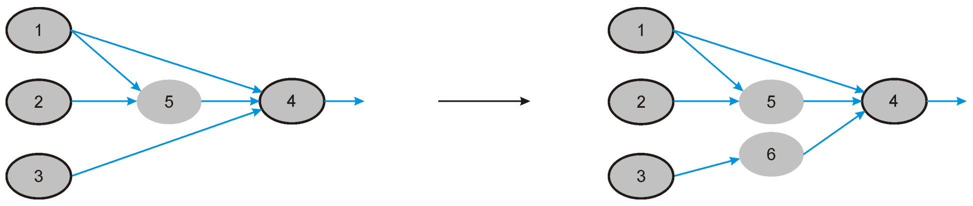


Figure 4.5: According phenotype to tables 4.2 and 4.3 before and after adding a new node.

Special crossover
The special crossover operator is already introduced in subsection 4.2.4 of the EANT section. This time, the innovation number is used to line up the common parts of two linear genomes. Genes having the same innovation number are called *matching* genes. With the help of the innovation number it is easy to track the historical origins, because the system knows exactly which genes match up.

Speciation

"Because smaller structures optimize faster than larger structures, and adding nodes and connections usually initially decreases the fitness of the network, recently augmented structures have little hope of surviving more than one generation even though the innovations they represent might be crucial towards solving the task in the long run." [SM02] To clarify this last open issue NEAT protects innovation by speciation of the population.

"Speciation allows organisms to compete primarily within their own niches instead of with the population at large. This way, topological innovations are protected in a new niche where they have time to optimize their structure through competition within the niche. The idea is to divide the population into species such that similar topologies are in the same species." [SM02]

Because the experiments in this thesis are conducted with a population size of two robots the approach of NEAT can not be applied. For future experiments or evaluations with a greater population size NEAT is considered to be a promising approach.

4.2.4 EANT - Evolutionary Acquisition of Neural Network Topologies

With EANT (acronym for Evolutionary Acquisition of Neural Network Topologies) Kassahun et al. [KS05b] show a method to learn neural networks by evolutionary reinforcement learning. The most important characteristics of EANT are described as follows:

- applies evolutionary reinforcement learning
- evolve structures and weights of neural networks
- compact and efficient encoding by means of Common Genetic Encoding (CGE)
- exploitation of existing structures and automatic exploration of new ones

Evolutionary reinforcement learning means a combination of neural networks, reinforcement learning and evolutionary algorithms. Neural networks are used to represent an approach and evolutionary methods are applied to search for the optimal approach directly in space of approaches. EANT starts with a minimal structure of a network and complexifies it along the evolutionary path by exploiting existing structures on a lower time-scale and exploring new structures on a larger time-scale. A further interesting aspect of EANT is it's genetic encoding of a network onto a linear genome using the Common Genetic Encoding described earlier. This allows the evaluation of a given network encoded by a genome without decoding it. It has been shown, that EANT is consistent in finding minimal neural structures for solving a given task.

Initializing the linear genome

Like Angeline et al. in [ASP94] the input and output nodes are fixed and previously determined by the user. When initializing the linear genome it only consists of input and output neurons I and N respectively. Recurrent jumpers and forward jumpers are introduced in later steps by structural mutation and added to the linear genome along the evolution path.

Kassahuan et al.[KMEK09] describe two kinds of methods for initializing the initial genome, the full method and the grow method. Both approaches remain some problems in that the full method leads to symmetric structures or the grow method generates repeated input. Those issues can be solved by editing the genome afterwards.

Evaluating the linear genome

The evaluation of a linear genome is realized by a first in last out stack and implemented as follows:

- start from right most node of the linear genome and move to the left
- in case the current node is
  I push current value and weight on stack

N pop n values with weights from stack (n = number inputs to neuron N)
compute result according to neuron's activation function
push result with weight

JR get last value of N where ID(N) = ID(JR)
push value and weight

JF copy sub-network starting from N where ID(N) = ID(JF)
compute response of sub-network
push result and weight

- after traversing completely pop resulting value or values (number of outputs determines the number of resulting values)

Genetic operators

Structural mutation on the topology and parametric mutation on the weights serve as search operators. Additionally EANT employs a special crossover operation.

Structural mutation
New forward or recurrent jumpers and new subnetworks are added with initial weight of zero to remain the performance or the behavior of the network, respectively. EANT only allows the deletion of jumpers. Nodes can not be removed by mutation. Structural mutation operates only on neuron nodes as follows:

- Step 1: test for each neuron, if it is going to be mutated
- Step 2: determine the kind of structural mutation

The first step is to check, whether a randomly generated number $\in [0, 1]$ is lower than the structural mutation probability p_m. If so, the neuron is going to be mutated in the next step. Step two determines the kind of structural mutation by drawing another random number also in the range of $[0, 1]$. Forward jumpers, recurrent jumpers as well as subgenomes receive equal mutation probabilities.

Parametric mutation
EANT's parametric mutation perturbs the synaptic weights w according to uncorrelated mutation. Additionally, every node obtains an associated mutation step size also called learning rate σ. The learning rate and the weight of each neuron are updated by equations 4.2 and 4.3.

$$\sigma_i' = \sigma_i * e^{\tau' N(0,1) + \tau N_i(0,1)} \qquad \text{where} \quad \tau' = \frac{1}{\sqrt{2n}} \quad \text{and} \quad \tau = \frac{1}{\sqrt{2\sqrt{n}}} \tag{4.2}$$

$$w_i' = w_i + \sigma_i * N_i(0, 1) \tag{4.3}$$

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

$N(0,1)$ results in a random number drawn from Gaussian distribution of unity standard deviation and zero mean. Equation 4.4 forces the learning rate σ not to be lower than a given threshold value ϵ_0.

Special crossover
The crossover operator is said to contradict the basic ideas behind artificial neural networks, because crossover works best when there are kind of blocks in the network. Since neural networks emphasize distributed representation it is not clear what such a block might be in a neural network. Therefore, recombining a part of a network with another part of the other network is likely to destruct both networks.

EANT employs a special crossover, that is first introduced by Stanley [Sta09]. This approach exploits the fact that structures which are formed from the same initial and minimal structures have some common parts. For example if two randomly selected structures S_1 and S_2 have a part called P_1 in common and the disjoint parts of structure one and two (here called D_1 and D_2, respectively), by aligning the common part P_1 it is possible to generate a third structure consisting of the common and the disjoint parts of S_1 and S_2. For better visualization see figure 4.6.

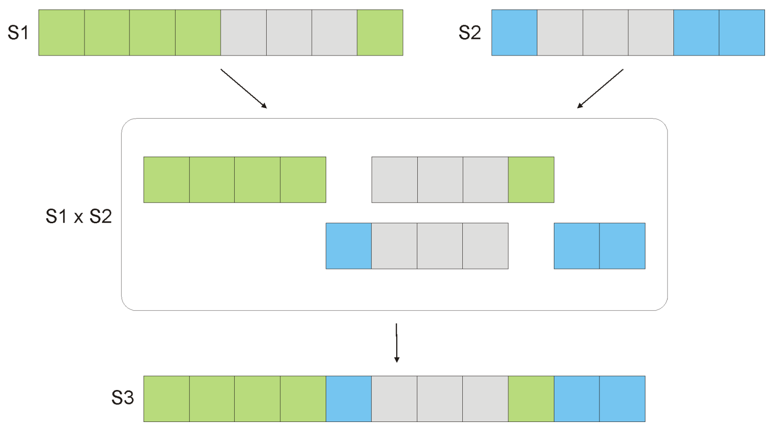


Figure 4.6: Example of special crossover of structure.

The input nodes of the newly created structure S_{12} are updated by

$$n(S_{12}) = n(S_1) + n(S_2) - n(S_1 \cap S_2) \tag{4.5}$$

where $n(X)$ is the number of input nodes to the neuron node X. Weights of the nodes of the resulting structure S_{12} are inherited randomly from both parents S_1

and S_1.

This section covered four topic related approaches for the artificial evolution of artificial neural networks. The CGE encodes a linear genome, which represents the genotype of a neural network. Its direct encoding method allows the evaluation of the corresponding phenotype without the need to decode the phenotype from its genotype. GNARL, NEAT and EANT are evolutionary approaches of simultaneously training neural networks by parametric and structural mutation. Note that, compared to EANT, GNARL does not offer cross over as search operator. NEAT introduces speciation, that allows organisms to compete within their own niches instead of with the population at large. Thus, NEAT requires a large population size and can not be applied to the experiment in this thesis. Regarding this summary, EANT is the most promising approach and thus is applied in the following.

5 Implementation

5.1 Applied Software

5.1.1 Simulation Tool

For the simulation of the robots' behavior and the execution of the task in a simulated environment there were two approaches to choose from. Firstly Player/Stage, a robot simulation tool, that consists of the program Player, a Hardware Abstraction Layer and the corresponding Plugin Stage that listens to what Player is telling it to do and turns these instructions into a simulation of the robot. And secondly Robot3D, a robot simulator as well built on top of the open-source game engine Delta3D. Because the simulation of the tasks used in this thesis is less complex and simpler to implement by means of Player/Stage and does not necessarily need a representation in full 3D, all the experiments are carried out by using Player and Stage.

Player/Stage

"The Player Project creates Free Software that enables research in robot and sensor systems. The Player robot server is probably the most widely used robot control interface in the world. Its simulation backends, Stage and Gazebo, are also very widely used. Released under the GNU General Public License, all code from the Player/Stage project is free to use, distribute and modify. Player is developed by an international team of robotics researchers and used at labs around the world." [Pla]

Player "Player provides a network interface to a variety of robot and sensor hardware. Player's client/server model allows robot control programs to be written in any programming language and to run on any computer with a network connection to the robot. Player supports multiple concurrent client connections to devices, creating new possibilities for distributed and collaborative sensing and control. Player supports a wide variety of mobile robots and accessories. Look here for a list of currently supported components." [Pla]

Stage "Stage simulates a population of mobile robots moving in and sensing a two-dimensional bitmapped environment. Various sensor models are provided, including sonar, scanning laser rangefinder, pan-tilt-zoom camera with color blob detection and odometry. Stage devices present a standard Player interface so few or no changes are required to move between simulation and hardware. Many controllers designed in Stage have been demonstrated to work on real robots." [Pla]

Simulation with Player/Stage

The simulation using Player/Stage is composed of three parts as follows:

- Part 1: The code, that talks to Player.
- Part 2: Player, that processes the code and forwards instructions to the robots. It gets sensor data from the robots and sends it back to the code.
- Part 3: Stage, which receives instructions from Player and moves the robots in the simulation. All the sensor data it gets from the simulated robots it transmits to Player.

5.1.2 Documentation and Visualization Tool

Doxygen

Doxygen is a tool for generating documentation for multiple programming languages directly from the source code, which makes it easier to keep the documentation consistent with the code. It is free software, released under the terms of the GNU General Public License. It can cross reference code and documentation, therefore when reading a documentation it is easy referring to the actual code.

Doxygen has a built-in support to create inheritance diagrams for C++ classes. Additionally, it can use the "dot" tool from Graphviz (see subsection 5.1.2) to automatically output more sophisticated dependency graphs and inheritance or collaboration diagrams.

In this thesis, Doxygen is used to generate an on-line documentation browser in HTML and for visualization of the relations between several elements.

For further information please see http://www.stack.nl/ dimitri/doxygen/.

Graphviz

"Graphviz is open source graph visualization software. Graph visualization is a way of representing structural information as diagrams of abstract graphs and networks. It has important applications in networking, bioinformatics, software engineering, database and web design, machine learning, and in visual interfaces for other technical domains." [Gra]

By reading every necessary instruction from a simple text file containing a description of the graph's vertices and edges, Graphviz can not only be used by people but by automatic processes as well to easily visualize objects and the relations among them. The specification of the graph to be displayed follows the rules of the DOT markup language. Often the structure definition by itself is sufficient for a reasonable output. Existing graphs may easily be extended or modified, that is not straightforward for standard graphics program. For further information see http://www.graphviz.org/.

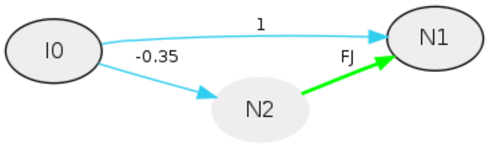


Figure 5.1: Simple example of a graph visualized via Graphviz.

In this thesis Graphviz is applied to display neural networks. Figure 5.1 shows a simple example of a graph visualized by means of Graphviz. I0 and N1 represent input and output nodes, respectively. The hidden node N2 is connected to the output neuron N1 via the forward jumper FJ colored in green. As shown in figure 5.2 edges displaying recurrent links are colored in red and labeled with RJ. Implicit links are represented in blue. For a more transparent and clear presentation every input and output node is displayed with a border.

Listing 5.1 demonstrates the corresponding DOT file for the simple graph.

```
1  digraph G {
2  graph [rankdir=LR]
3  node [fontname=Verdana,fontsize=12]
4  node [style=filled]
5  node [fillcolor="#EEEEEE"]
6  node [color="#EEEEEE"]
7  edge [color="#31CEF0"]
8  edge [fontname=Verdana,fontsize=10]
9  N1 [color="#000000"]
10 N2->N1 [style="bold", color="green", label="FJ"]
11 I0[color="#000000"]
12 I0->N1 [label="1"]
13 I0[color="#000000"]
14 I0->N2 [label="−0.35"]
15 }
```

Algorithms 5.1: Corresponding DOT file to the simple graph example.

Line 2-8 determines the graphical representation of every node and edge occurring in the graph like font style and size and or background and border color of each node. "rankdir=LR" provides the graph to be laid out from left to right. The lines 9-14 set the connectivity or the interrelations among the nodes and additionally handle special guidelines for selected nodes or edges, for example the input and output nodes have black borders compared to every other node. For further details and descriptions please visit the Graphviz website.

A more complex graph is shown in figure 5.2. It's representation is still very clearly arranged, but for further visualizations and even more complicated neural networks all outputs and all inputs, respectively, are displayed in one column or line to improve clarity.

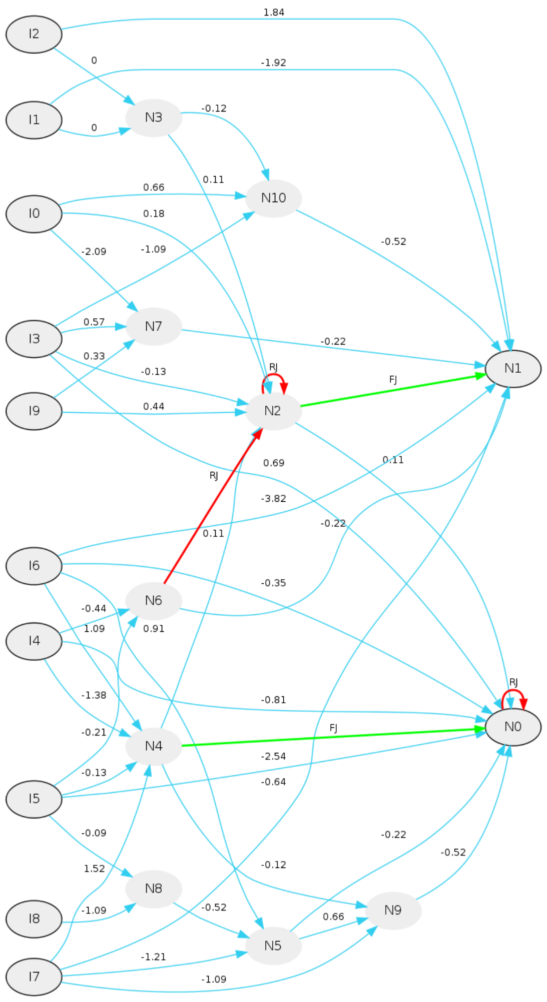


Figure 5.2: Example of a more complex graph representing a neural network.

5.2 The Framework

5.2.1 EvoRoF

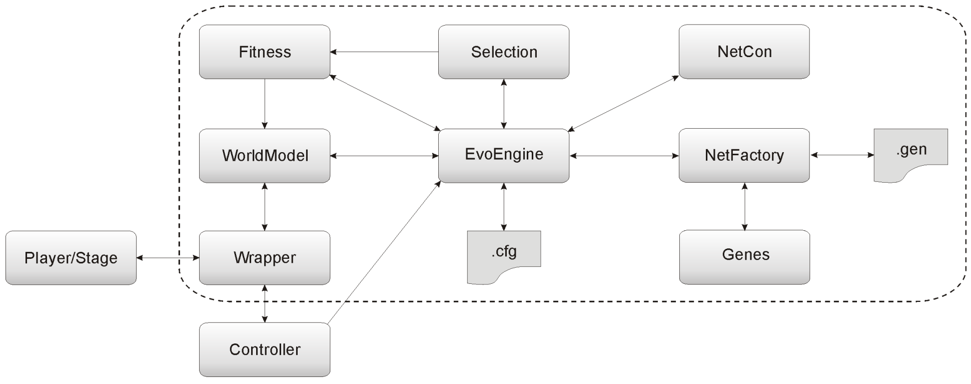


Figure 5.3: Overview of the classes the EvoRoF framework consists of.

Evolutionary **Ro**botics **F**ramework is an existing framework created and developed by Schlachter et al. [SADL12] that is extended by this diploma thesis. Its most important components are listed in the following:

Genes

Derived from the class *Genes* is on the one hand the class *Links* and on the other hand the class of neurons *AbstractNeuron*. The inherited classes are further divided as depicted in figure 5.4. The class *Links* is made up of input links, recurrent jumper and forward jumper. There exist two types of neurons, the artificial neurons and the spiking neurons. In this diploma thesis the former kind of neurons will be applied. The *NetFactory* builds the linear genome out of genes.

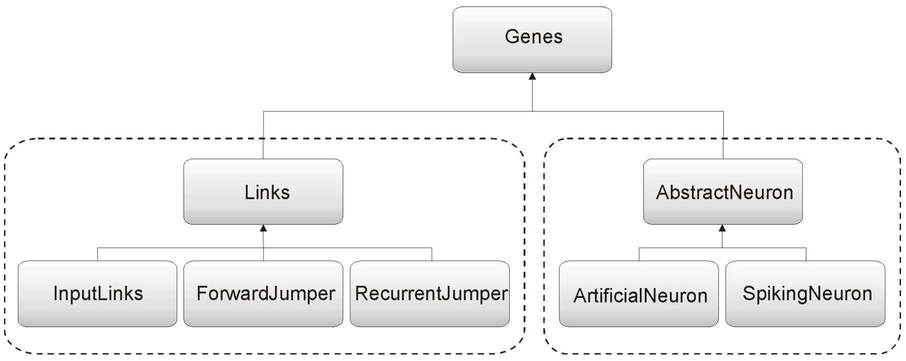


Figure 5.4: Inheritance diagram for the class *Genes*.

An artificial neuron comprises the parameters listed below:

- Identifier:
  As in EANT the identifier of an artificial neuron is the letter N.
- ID:
  The ID corresponds to a consecutive number starting with zero
- Weight:
  Weight of the neuron. Note that output nodes always have a weight of 1.0.
- Learning rate:
  In this thesis the learning rate of a neuron is constantly set to a value of 0.3.
- Depth:
  Every neuron is located in a specific layer in the network. Output neurons for example are in the output layer with a depth of zero.
- Number of inputs:
  For a valid genotype to phenotype mapping and for the evaluation of the linear genome the number of inputs a neuron has got is important.
- Mutation probability:
  The mutation probability determines how often or whether a neuron is going to be mutated at all.

Input links, recurrent jumper as well as forward jumper consist of the same parameters as an artificial neuron except the depth, the mutation probability and the number of inputs. As an example the following subsection depicts a genome file that represents a linear genome with nine input links and two output nodes.

The genome file *.gen*

A genome file such as listing 5.2 is structured to the pattern of the gene class and its derivatives. In this example the corresponding neural network includes two output neurons N0 and N1 and nine input nodes from I0 up to I8.

```
N 0 1 0.3 0 9 0.1
I 0 0.583091 0.3 --
I 1 -0.246647 0.3 --
I 2 0.588356 0.3 --
I 3 0.704877 0.3 --
I 4 0.749209 0.3 --
I 5 -0.992314 0.3 --
I 6 0.928373 0.3 --
I 7 1.042293 0.3 --
```

```
I 8 0.134652 0.3 --
N 1 1 0.3 0 9 0.1
I 0 0.659219 0.3 --
I 1 0.201941 0.3 --
I 2 0.894955 0.3 --
I 3 -1.014724 0.3 --
I 4 -0.018947 0.3 --
I 5 0.270701 0.3 --
I 6 -0.059452 0.3 --
I 7 -0.380017 0.3 --
I 8 -0.717569 0.3 --
```

Algorithms 5.2: An example of a genome file (*.gen*).

The configuration file *.cfg*

The configuration file sets several parameters which are read in and processed by the *EvoEngine*. The first parameter defines the type of the network that is used. There are two types to choose from: ANN - Artificial Neural Network or SNN - Spiking Neural Network. This diploma thesis examines the implementation of ANN. The second parameter determines the amount of members in a population of an island that will be evaluated. You can choose to load genomes from files by means of the variable *LoadGenomesFromFiles*. If set to "NO" the genomes are created randomly with random initial weights and fully connected from the inputs to the output nodes. *NumberOfInputNeurons* and *NumberOfOutputNeurons* are self-explanatory, such as the default learning rate and the default mutation probability. *NumberOfStepsForEvaluation* defines how many steps each controller is evaluated in simulation before switching to the next population member. The evaluation can be stopped automatically when reaching the number of generations by *NumberOfGenerations*. Artificial evolution is implemented following EANT. Thus, *NumberOfExploitationSteps* specifies the number of exploitation steps to perform before an exploration step follows. The last three parameters describe the kind of fitness and selection function to be applied. So far there is a choice for the fitness functions between DIST (distance), DISTaCOLL (distance and collision) and COLLECTIVE. The latter function is presented in subsection 6.1.1. Two slightly different modes for the selection can be set by *SelectionMode* and the *SelectionParameter*. ELITISM and ELITISMaREMAIN (elitism with a remaining part) are introduced in section 6.1.2.

```
EvolutionaryAlgorithmType     ANN
PopulationSizeOfIsland        10
LoadGenomesFromFiles          NO
GenomeDirectory               data/
GenomePrefix                  genome

```

```
NumberOfInputNeurons          9
NumberOfOutputNeurons         2

NumberOfStepsForEvaluation    100
NumberOfGenerations           200
NumberOfExploitationSteps     100

DefaultLearningRate           0.3
DefaultMutationProbability    0.1

FitnessFunctionType           DISTaCOLL
SelectionMode                 ELITISMaREMAIN
SelectionParameter            30.0

EnableParametricMutation      ON
EnableStructuralMutation      ON
```

Algorithms 5.3: The configuration file (*.cfg*).

EvoEngine

The evolutionary engine is a central component in the EvoRoF framework that links all the classes. All data in the simulation is provided by the *WorldModel* the Evo-Engine accesses. The *Controller* initializes the EvoEngine and forces the next steps which trigger artificial evolution.

The main tasks of the EvoEngine result in the following:

- read the configuration file and store the parameters in a list
- create the initial neural networks or linear genomes using the *NetFactory*
- initiate the evolutionary step
- generate the next generation
- carry out the parametrically and structurally mutation using *NetCon*
- calculate the fitness of an individual through a specified fitness function in *Fitness*
- determine the individuals of the next generation by means of the *Selection* function

Thus, the evolutionary engine is responsible for the artificial evolution.

NetFactory

The network factory creates the genomes either randomly or reads the necessary information from the configuration file. A parameter in the configuration file determines the kind of genes the genome is going to consist of. When randomly creating a genome, every input node is connected to every output node. There exist no jumpers and all the weights are initially set to a random value. The code fragment below illustrates how an artificial neural network is created randomly with *numberOfInputs* input nodes, *numberOfOutputs* output neurons and given default values for the learning rate and the mutation probability.

```
NetCon* NetFactory::createRandomANN(int id, int numberOfInputs,
    int numberOfOutputs, float defaultLearningRate, float
    defaultMutationProbability) {
  std::vector<Genes*> genome;
  int numNeurons = 0;
  float learningRate = defaultLearningRate;
  float mutProb = defaultMutationProbability;

  // create outputs
  for (int i = 0; i < numberOfOutputs; i++) {
    ArtificialNeuron * an = new ArtificialNeuron("N", i, 1.0,
        learningRate, 0.0, numberOfInputs, mutProb);
    an->initNeuron();
    genome.push_back(an);
    numNeurons++;

    // create fully connected random network
    int k = 0;
    for (k = 0; k < numberOfInputs; k++) {
      // initialize weights with random values between -1..1
      InputLink * l = new InputLink("I", k, (2.0*rng.rand()-1.0),
          learningRate);
      genome.push_back(l);
    }
  }
  NetCon *net = new NetCon(id, numberOfInputs, numberOfOutputs,
      genome, numNeurons, defaultLearningRate,
      defaultMutationProbability);
  return net;
}
```

Algorithms 5.4: Randomly creating an ANN in the *NetFactory*.

Netcon

Netcon contains the class for the network controllers. All evolutionary operators such as parametric and structural mutation are part of the tasks of the network controller. In addition, it implements the stack operations according to EANT and CGE such as push, pop and evaluate the sub-genome. The code fragment below implements the parametric mutation introduced in subsection 4.2.4. Note that weights of output neurons are not mutated by the parametric mutation. Those weights keep a value of 1.0. In this thesis the learning rate *sigma* remains constant at a value of 0.3.

```
void NetCon::parametricMutation() {
  ...
  N_it = rng.randNorm(0, 1.0);
  tau = 1 / sqrt(2 * sqrt(genome.size()))
  tau_tick = 1 / sqrt(2 * genome.size());
  sigma = (*it)->getLearningRate();
  weight = (*it)->getWeight();

  // compute and set new weight
  weight_tick = weight + sigma * N_it;
  (*it)->setWeight(weight_tick);
}
```

Algorithms 5.5: Implementation of parametric mutation in *NetCon*.

According to the formula of the parametric mutation in EANT the variables used in listing 5.5 denote the following:

- N_it: random number drawn from Gaussian distribution with zero mean and unity standard deviation, newly calculated in every iteration
- tau: $1/\sqrt{2n}$ with n equals to the size of the genome
- tau_tick: $1/\sqrt{2\sqrt{n}}$ with n equals to the size of the genome
- weight: weight that is going to be mutated
- weight_tick: new weight after mutation
- sigma: learning rate
- sigma_tick: new learning rate after calculation

Fitness

The fitness of a robot corresponds to its weighting function and determines the reward the robot receives for it's actions taken. By means of the values of the fitness function the *Selection* function elects the individuals for the next generation. The distance of the current position to the starting position is one exemplary fitness function depending only on the robot's position. Furthermore the fitness can be calculated by a combination of several parameters.

In this thesis the fitness function denoted as COLLECTIVE is applied and introduced in subsection 6.1.1.

Selection

After an evolutionary cycle has happened the *Selection* function determines the individuals of the next generation according to a certain schema. One of the easiest ways of selection is to replace the half of the population that performed the worst with the other half that performed the best.

This thesis uses the selection function named ELITISMaREMAIN meaning elitism selection with a part of the population that remains and is not replaced by elite individuals. Section 6.1.2 demonstrates the applied selection mechanism.

Logger

The *logger* class allows visualization of the generated neural networks or logged data such as the fitness or the position. A linear genome is either presented in table format in the console or in the form of a graph in a dot file. The latter is converted into a png file via Graphviz representing the associated graph or phenotype to the linear genome. Input and output nodes are displayed with black borders, whereas hidden neurons are illustrated without a border. Recurrent as well as forward jumper are differentiated by color from implicit links. Moreover, edges are labeled with the appropriate weights and neurons with its IDs. An example of such generated phenotypes is depicted in chapter 5.1.2 and in the chapter presenting the results (chapter 7).

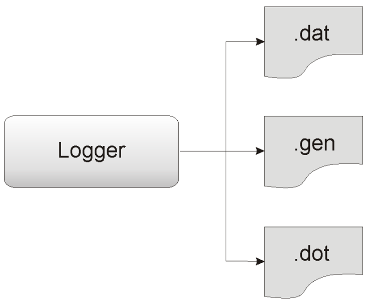


Figure 5.5: Overview of the files the logger can creates.

WorldModel

The *WorldModel* manages the current sensor values, the actual position of the robot, the fitness values and the starting position. Thus, all the data is stored centrally and is accessible through the *WorldModel*. The parameters of the *WorldModel* and their according data types are presented in table 5.1. Additionally, the *WorldModel* provides a function for resetting the robot to it's initial position. This can be useful for offline evaluation or the synchronous start of the robots in simulation. After a controller is evaluated the *EvoEngine* calls the function *resetRobotPosition()*. The *Controller* retrieves the *doRobotPositionReset()* and initiates the action to put the robot back to the start position.

Table 5.1: Table showing the variables of the *WorldModel*.

variable	**description**	**data type**
numberOfIrSensors	contains the number of sensors the robot is assembled with	integer
ir	here, contains all the sensor values of a robot	float vector
startPosition	stores the x, y, and z coordinate of the robot's initial position	float vector
currentPosition	stores the x, y and z coordinate of the current position of the robot in each simulation step	float vector
variableData	this vector can take variable data of type float	float vector
doReset	setting this flag to true effects the reset of the robot to the start position	boolean

Current infrared sensor values are stored in the vector *IR*, the current position of the robot in *currentPosition* and the start position in emphstartPosition, respectively. Because the task of this thesis requires fiducial sensor in addition to the infrared sensors the *IR* vector stores the fiducial sensor data as well. The combining of the *IR* vector takes place in the *PlayerWrapper* as described in the next subsection.

PlayerWrapper

The wrapper class for the simulators Player and Robot3D is derived from the superior class *Wrapper* as illustrated in figure 5.6. The *PlayerWrapper* class inherits from *Wrapper* which interacts as an interface between the *Controller* and the simulator. Thus, the *PlayerWrapper* realizes the interface between the *Controller* and Player. Additional tasks of the *PlayerWrapper* are as follows:

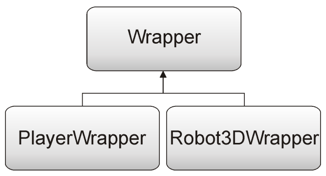


Figure 5.6: Inheritance diagram for the *Wrapper* class.

- set up the connection to Player and initialize the Stage proxies as well as the actuators and sensors defined in Stage
- receive all sensor values and the current position of the robot from the simulation
- transmit all data obtained to the *WorldModel*

```
void PlayerWrapper::init(unsigned int portNumber, unsigned int
    numberOfSensors, unsigned int fidRangeTop, unsigned int
    fidRangeBottom)
{
  port = portNumber;
  numberSensors = numberOfSensors;
  fiducialRangeTop = fidRangeTop;
  fiducialRangeBottom = fidRangeBottom;
  robot = new PlayerClient("localhost", port);
  robotPos = new Position2dProxy(robot,0);
  robotFidBottom = new FiducialProxy(robot,0);
  robotFidTop = new FiducialProxy(robot,1);

  //enable motors
  robotPos->SetMotorEnable(1);
  moved = false;

  // read current sensor data
  robot->Read();

  // update all sensor data
  updateSensorInput();
  ...
}
```

Algorithms 5.6: Implementation of initialization of the *PlayerWrapper*.

When updating the sensor data the *IR* vector that is transmitted to the *WorldModel* is combined from IR data as well as fiducial sensor values such as the distance to another robot or the target object and also the angle to the target object.

Every robot possesses two fiducial sensors. One on the top of the robot and the other one at the bottom. Both fiducial sensors detect all objects which have the *fiducial_return* flag set to 1 and return an array containing the IDs of the perceived fiducial.

To obtain the position of a certain fiducial, for example the position of the other robot with ID 1 which is detected by the top fiducial sensor, one have to verify the array containing the particular ID. In the case the other robot is within range it's current position is returned, otherwise the maximum distance is assumed. The maximum distance corresponds to the maximum range of the fiducial sensor. If no fiducial is detected the maximum distance is returned as well. Note that the distance to a fiducial is given relative to the robot's position.

The distance between the robot and a fiducial is computed by the Euclidean distance with formula 5.1 where A is the position of the robot and B refers to the position of the detected fiducial.

$$d(A,B) = \sqrt{(x_B - x_A)^2 + (y_B - y_A)^2} \tag{5.1}$$

Because the position of the fiducial is returned relative to the robot's location the values given back can directly be used for the calculation of the Euclidean distance. Afterwards the result is subtracted from and divided by the maximum distance to invert the value and receive a result in the range of 0 and 1. The reason for the inverting is the activation of the neuron in the underlying network. If no fiducial is within range, the neuron should not perceive any activation thus a value of zero is returned to the input of the neuron. In the contrary case the input should activate the neuron proportional to the distance to the fiducial. From this it follows that the closer a fiducial is the more activation a neuron obtains and that way the stronger the behavior turns out. Another consequence of the inverting of the sensor values is the need of a bias input neuron. Otherwise in the case of no fiducial is detected all the inputs receive a zero value resulting in no movement or turning of the robot at all.

The computation of the angle between a robot and a fiducial is carried out using polar coordinates in the interval of $(-\pi, \pi]$ and equation 5.2.

$$\phi = (signum(y) + 1 - |signum(y)|) \arccos \frac{x}{r} \tag{5.2}$$

As mentioned earlier the *PlayerWrapper* performs the task of resetting a robot to it's initial position. Because setting a robot directly to a given position is not yet implemented in the Stage driver the *GoTo* command is used to put a robot back to the start position. By means of the *GoTo* instruction the robot moves with a constant speed back to the specified position (startPosX, startPosY). Depending on the robot's actual position this can take more or less time.

A differential steering model is realized in the *PlayerWrapper* by means of a alternate moving and turning as illustrated in listing 5.7.

```
void PlayerWrapper::setActuatorOutput(float speed, float turnRate)
{
 if (moved) {
   robotPos->SetSpeed(0, turn);
   moved = false;
 }
 else
 {
   robotPos->SetSpeed(drive_forward, 0);
   moved = true;
 }
}
```

Algorithms 5.7: Implementation of differential steering

Controller

Firstly, the *Controller* initializes the *PlayerWrapper* passing the port number, the amount of the sensors, and the ranges of the fiducial sensors (see code fragment 5.8). The initial sensor data is stored in the *inputVector*. Then, the *EvoEngine* is initialized.

```
void Controller::initialize(int port, int numberOfSensors)
{
 // initalizes the PlayerWrapper
 init(port, numberOfSensors, fidRangeTop, fidRangeBottom);

 // write sensor data into input vector
 inputVector = sensorData;

...

 // initialize the evolutionary engine
 evo->initEvoEngine((char*)"config/island.cfg", port, worldModel, &inputVector, &outputVector);
}
```

Algorithms 5.8: Initialization of the *Controller*.

After the initialization steps, the *Controller* initiates the evolutionary step by calling *nextStep()* of the *EvoEngine* in line 7 of listing 5.9. In a final step, the *Controller* transmits the calculated actuator values to the simulation by means of the *PlayerWrapper* (see line 18). Line 10-17 implement the scenario of a light barrier that can only be overcome if another robot is within a certain range. This is put into action by the logical implication $(inLight \Rightarrow robotNearby)$.

```
void FirstController::run() {
  for (;;)
  {
    updateSensorInput();
    evo->nextStep();

    check if robot reached the light barrier;
    check if other robot is near;

    if (!inLight || robotNearby) {
      drive_forward = outputVector[0]*2;
      turn = dtor(outputVector[1]*30);
    }
    else {
      drive_forward = 0.0;
      turn = 0.0;
    }
    setActuatorOutput(drive_forward, turn);
  }
}
```

Algorithms 5.9: Looping the execution in simulation.

5.3 Execution in Simulation

A Player/Stage project fundamentally consists of a *inc* file, a *world* file and a configuration file (*cfg*). The implementation in simulation via Player/Stage comprises the subsequent files. The according implementations can be found in the appendix:

karobot.inc contains a detailed description of the KA robot as set out in the appendix in listing 9.1. The robot's size, shape and its initial position are determined, as well as the number of sensors the robot is assembled with. Player/Stage provides two steering modes, the differential drive and the omnidirectional drive.

simple.world This file (see appendix, listing 9.2) describes the floor plan, the size of the arena and determines all boundaries given by the according png file. By including karobot.inc additional robot features can be set in this world file. Furthermore, all objects such as the robots or obstacles are set into the arena by a given name and position.

fiducialfinder.cfg includes the world file. In this configuration file (see appendix, listing 9.3), all applied drivers are listed and adjusted accordingly. The starting port in Player/Stage is 6665 and is continued in ascending order for each further robot.

obstacleRectangle.png Every black pixel in this exemplary image file (see figure 5.7) is considered as obstacle or boundary by the world file. This allows to easily change the boundaries of the arena by including another image file.

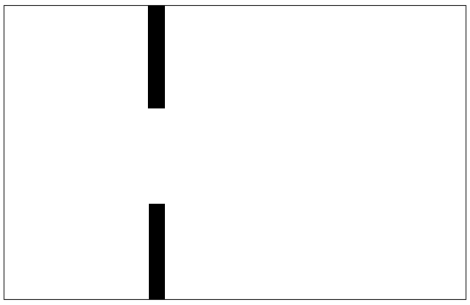


Figure 5.7: Image file for the boundaries.

5.3.1 Mapping the Scene

Arena

The arena in Stage is determined by the world file. Figure 5.9 and 5.8 show the associated arena with the bounding box, obstacles, fiducials and robots as created from the *simple.world* file described earlier. The bounding rectangle measures 120x80cm. The height of the walls measures 3cm. The small green object with an elongated shape in the centre of the yellow circle corresponds to the target object. The yellow circle serves as a "light barrier" surrounding the target object.

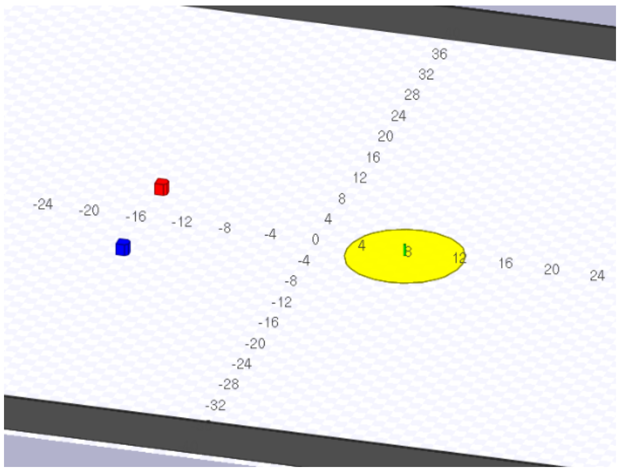


Figure 5.8: Perspective view of the arena of figure 5.9.

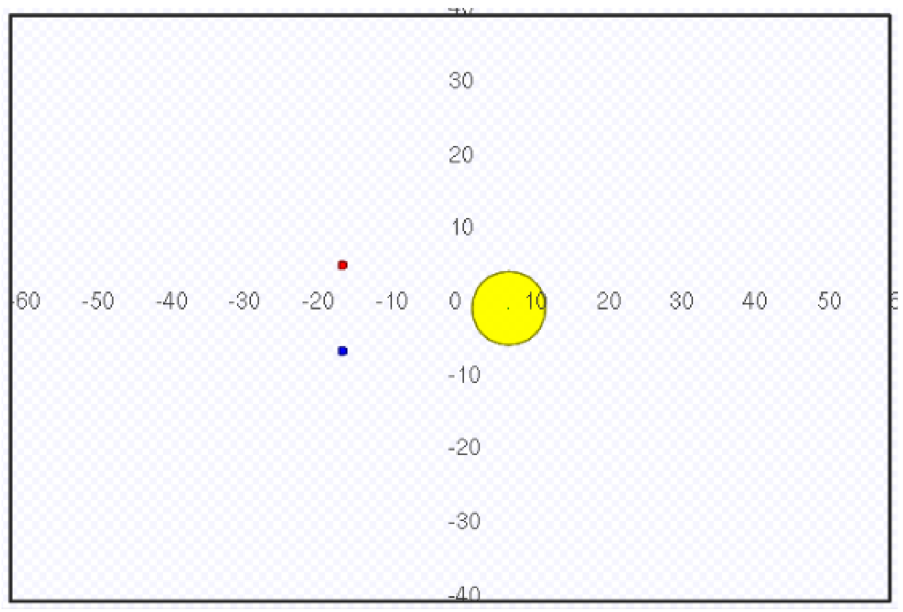


Figure 5.9: Exemplary initial state of the arena.

Lightsource

A fiducial object in Stage is detected by a robot, if the robot provides a fiducial sensor and whether the fiducial's flag *fiducial_return* is set to a value greater than zero. In this thesis, the *fiducial_return* value of the light object is set to 2, the value of the other robot is set to 1 and the value of the target object is set to 3. There exist two ranges for the fiducial sensor, namely *range_min* and *range_max*. The former determines the minimal range a robot's fiducial sensor detects the fiducial. If set to zero it is the distance between the fiducial's centre and the robot's sensor. The latter, *range_max*, specifies the distance from which downwards the robot's sensor is able to detect the fiducial. If the actual distance lies between *range_min* and *range_max* the robot's sensor receives two different values. Either a zero value, when the distance is less than the value set by *range_max_id* or the specified fiducial ID (here for example 1, 2 or 3) if the sensor value is greater than *range_max_id*. For the following experiments both *range_max* and *range_max_id* are given a value of 40.

Robots

Size and Shape
In simulation the robot is depicted in abstract as a 1x1x1cm cube with a bevel on the front edges as shown in figure 5.10. By means of the bevels one can distinguish if the robot moves forwards or backwards. By way of distinction the robots are

visualized in different colors. However, this does not effect the robot's behavior or the execution of the task.

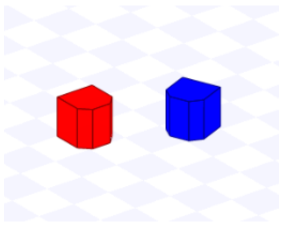


Figure 5.10: Shape of the robots in simulation.

The shape of the robots derives from their real role model or the hardware the evolved controller should be applied on in further experiments. As an example figure 5.11 depicts the SCOUT Robot (PISA Robot) of the Symbrion Replicator project.

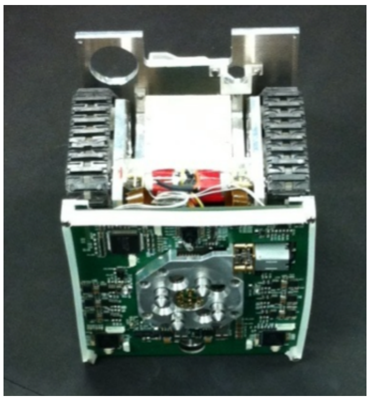


Figure 5.11: SCOUT Robot (PISA Robot)

Sensors and Actors
The robot is mounted with eight range sensors and two fiducial sensors. Two range sensors are attached on each of the robot's four sides. The range sensors allow the robot to detect obstacles in near distance with an angle of 30 degrees. One of the two fiducial sensors is located centrally on the top of the robot, the other one at the bottom. Thereby the robot is able to recognize fiducials on the floor as well as fiducials that are situated in the height of the robot.

As illustrated in figure 5.12 and 5.13 range sensors are displayed in green and the range of the fiducial sensors is shown by a red circle. The red dotted line corresponds to the distance between the robot's fiducial sensor and the fiducial, in this case the light source or the target object. If the fiducial is within the sensor

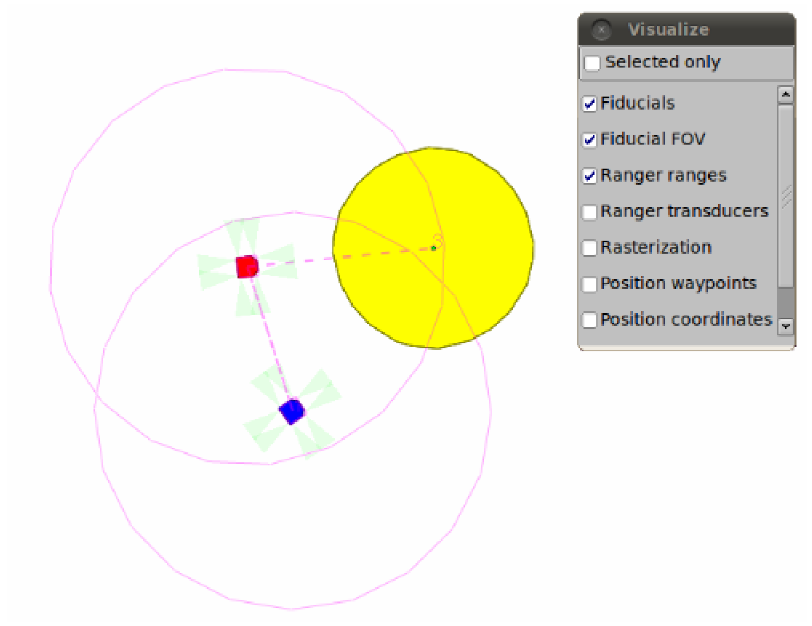


Figure 5.12: Range and fiducial sensors of the robot.

range and the distance is less than a predefined value the fiducial sensor receives the fiducial ID as in figure 5.13. Otherwise the response is zero.

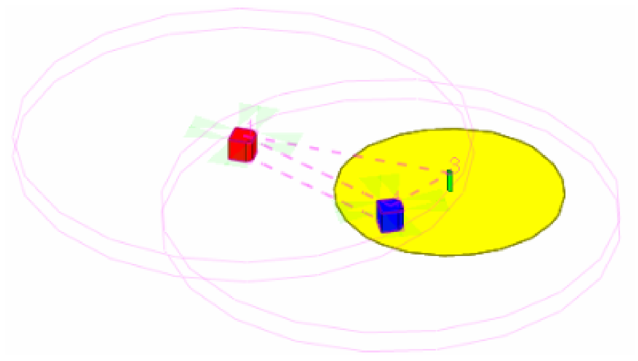


Figure 5.13: Range and fiducial sensors of the robot in perspective view.

6 Experiments

6.1 Preliminary Considerations

6.1.1 Choice of Fitness Function

The fitness function is used to determine the behavior of an individual or the actions to be performed preferably.

When developing a fitness function one important question which should be considered is how to avoid the design of specific behavior, but still evolve individuals that are able to successfully achieve a desired goal. [CN10].

The following sections describe several fitness functions according to different scenarios. It is shown that varying functions may lead to the same goal. The choice of the fitness function has a significant influence on the results and/or the behavior of the robot.

Collision Avoidance with Random Walk

A preliminary stage of the fitness function applied in this thesis is the function for the so called collision avoidance scenario. A usual task of a robot is to randomly move through the arena without hitting any obstacle or, in other words, reliably avoid collisions. There exist many approaches in developing the fitness function for collision avoidance. In this approach, the basic procedure is to reward individuals that never collided during a run and punish those, that hit at least one obstacle. Punishing can take place by resetting the individual's fitness value to zero at the time of a collision. Or the entirely fitness is set to zero. A reward can be given by adding those time steps in which an individual was free from clashes.

Note that avoiding walls can also be implicitly contained in a fitness function such as covering the longest distance in a specific period of time. An individual getting stuck when hitting a wall cannot move along and thus does not cover much meters. What leads to a lower fitness compared to other individuals which successfully avoided any obstacle.

In the following, we consider both approaches and also combinations of them and present a possible implementation that is applied during development in this diploma thesis.

Approach 1

The code fragment in listing 6.1 illustrates an exemplary fitness function for collision avoidance. Each update step in the method *updateFitness* refreshes the robot's position as well as it's sensor data, calculates the distance the robot moved during the last step and examines whether the sensor values reveal a collision. The distance is added up to the tracks of the last steps and thus specifies the fitness. In doing so, the robot is forced to increase it's distance in every step, which is equivalent with moving along.

The method *getFinalFitness* punishes or rewards the individual whether a collision has happened or not. In the first case the fitness value is reduced by a factor of 10 whereas in the latter case an individual is rewarded by doubling it's fitness. Consequently, the robot is told that it is good to avoid obstacles.

```
void FitnessCollision::updateFitness(void)
{
  // counter for the simulation steps
  count++;
  // get current position and current sensor values of robot
  curPos = getCurrentPosition();
  iRData = getIR();
  // calculate distance the robot covered during one simulation step and set fitness value
  float distance = sqrt((curPos[0]-lastPos[0]) * (curPos[0]-lastPos[0]) + (curPos[1]-lastPos[1]) * (curPos[1]-lastPos[1]));
  fitness += distance;

  // reset lastPos for next calculation step
  lastPos = curPos;

  // if last controller gets stuck at a wall, give next controller a few steps to move away from the wall
  if (count > 10){
    for (unsigned int it=0; it<iRData.size(); it++) {
      // check all sensors if distance to an obstacle is too close
      if (iRData[it] < 0.1)
        collision = true;
    }
  }
}

float FitnessCollision::getFinalFitness(void )
{
  if (!collision){
    // if the individual never collided, extra reward it's
        fitness by doubling the fitness
    fitness += (2*fitness);
  }
  else {
    // if a collision took place, reduce the fitness by a
        factor of 10
    fitness = fitness/10;
  }
  collision = false;
  return fitness;
}
```

Algorithms 6.1: Exemplary fitness function for collision avoidance.

Approach 2

The second approach aims at the same objective: avoid obstacles and move through the arena. Following another modality there is no explicit testing for a collision but two approaches measuring the fitness of an individual and thus telling it what the preferred actions are. Instead of predicating the fitness on the distance, the robot is rewarded or punished by constant values. A slow motion is rewarded by a value of 0.1 whereas a faster movement is rewarded by 0.5. If the robot stops or hardly moves it gets penalized by a value of -1.0 as depicted by line 19 in listing 6.2. Again, by this means the robot is forced to move constantly.

The lines 23 to 28 describe the obstacle detection within a certain range. If there is no obstacle the sensor values transmit a zero value. In case of a collision the corresponding sensor sends a value of 1.0. The robot is penalized by 0.1 within a distance value starting from 0.1 to 0.5 and by 0.5 if an obstacle becomes too close (distance > 0.5).

```
 1 void FitnessCollision::updateFitness(void)
 2 {
 3   // get current position and current sensor values of robot
 4   curPos = getCurrentPosition();
 5   iRData = getIR();
 6   // calculate distance the robot covered during one
        simulation step
 7   float distance = sqrt((curPos[0]-lastPos[0]) * (curPos[0]-
        lastPos[0]) + (curPos[1]-lastPos[1]) * (curPos[1]-
        lastPos[1]));
 8   // reset lastPos for next calculation step
 9   lastPos = curPos;
10
11   // reward intermediate movement with +0.1 and faster
        movement additionally with +0.5
```

```
  // punish with a value of -1.0 if there is (nearly) no
      movement
  if (distance >= 0.02)
  {
    fitness += 0.1;
    if (distance >= 0.2)
      fitness += 0.5;
  }
  else fitness -= 1.0;

  // punish, if obstacles are detected within a range of 0.1
      and 0.5 with -0.1
  // if an obstacle is close (sensor value > 0.5) punish with
      -0.5
  for (unsigned int it=1; it<iRData.size(); it++) {
    if (iRData[it] <= 0.5 and iRData[it] > 0.1)
      fitness -= 0.1;
    else if (iRData[it] > 0.5)
      fitness -= 0.5;
  }
}

float FitnessCollision::getFinalFitness(void )
{
  return fitness;
}
```

Algorithms 6.2: Varying fitness function for collision avoidance.

Remain Close

In many swarms spatial proximity plays an important role. For example in a swarm of birds those members save energy that fly behind another bird because in this position they experience less aerodynamic resistance. Considering a school of fishes, by remaining close to each other the nearness serves as protection against predators. The following fitness function forces the robots to remain as close as possible.

```
void FitnessDist::updateFitness(void)
{
  // get current position of robot and current sensor values
  curPos = getCurrentPosition();
  sensorData = getSensorData();

  // calculate distance the robot covered during one
      simulation step and set fitness value
```

```
 8    float distance = sqrt((curPos[0]-lastPos[0]) * (curPos[0]-
         lastPos[0]) + (curPos[1]-lastPos[1]) * (curPos[1]-
         lastPos[1]));
 9    fitness += distance;
10
11    // reset lastPos for next calculation step
12    lastPos = curPos;
13
14    // maxDistance is the range of the fiducial sensor set in
         Stage and determines the maximum distance a robot is
         able to detect another robot
15    fitness += (maxDistance - currentDistanceToRobot)/10;
16  }
```

Algorithms 6.3: Fitness function for the distance to another robot.

As in the previous example, the robot is told by the fitness function to move through the arena (listing 6.3 line 8). The instruction in line 15 rewards close proximity to other robots by adding the difference of the maximum measurable distance and the distance to the other robot to the current fitness value.

Moving to a Reference Point

Regarding the scenario of the ants moving back and forth between the ant hill and a food source, the swarm robots should have the same behavior. The corresponding fitness function for this scenario needs a reference point the robot should navigate to. If the distance between the robot and the reference point decreases the robot is rewarded, otherwise the robots is penalized. The implementation for this fitness function is illustrated in the listing 6.4 below.

```
void FitnessFiducial::updateFitness(void)
{
  // reward, if distance to fiducial decreases
  fitness += curDistanceToFiducial - lastDistanceToFiducial;

  // give extra reward if robot is very close to the fiducial
  if (curDistanceToFiducial < 0.1) {
    fitness += 2;
  }

  // reset lastDistanceToFiducial for next calculation step
  lastDistanceToFiducial = curDistanceToFiducial;
}
```

Algorithms 6.4: Fitness function for navigation to a fiducial.

Collective Behavior

Topic of this diploma thesis is to show coordinated behavior within a swarm of robots. For simplification in the following the robot swarm consists of two robots, which have to fulfill a task they can only solve cooperatively. For this purpose, the robots should move to a target object that is surrounded by a barrier. A robot is only able to overcome the barrier if another robot is within a certain distance. The corresponding fitness function for this task is a combination of some of the functions listed above and is shown in listing 6.5. For the implementation of the barrier the robot controller queries the robot position and the position of the barrier. If the positions match the robot's speed and turning are set to zero and will not be changed before the distance to the other robot is reduced to a certain value.

The fitness function for the evaluation of cooperative behavior is an extension of the fitness function in listing 6.4 and is depicted in listing 6.5.

```
void FitnessCollective::updateFitness(void)
{
  // get current sensor data
  sensorData = getSensorData();

  // reward, if distance to fiducial decreases
  fitness += curDistanceToFiducial - lastDistanceToFiducial;

  // give extra reward if robot is very close to the fiducial
  if (curDistanceToFiducial < 0.1) {
    fitness += 2;
  }

  // reset lastDistanceToFiducial for next calculation step
  lastDistanceToFiducial = curDistanceToFiducial;

  // reward with inverted distance value
  fitness += (maxDistance - currentDistanceToRobot)/10;
}
```

Algorithms 6.5: Fitness function for collective behavior with one reference point.

In the following experiments two fiducials are placed in the arena as depicted in figure 6.1 and 6.2. The robots should move to the fiducials one after the other. Again, this scenario is motivated by the foraging of ants or the navigating back and forth between two food sources. If the first food source is reached the supply is exhausted and the robot should set out to the next food source. Both food sources are surrounded by a barrier which can only be overcome in twos. The corresponding fitness function for this approach, that is applied in the following experiments, is listed in listing 6.6.

```
if bias neuron has positive value
  reward if distance to reference point 1 decreases
  and robot is moving

if bias neuron has negative value
  reward if distance to reference point 2 decreases
  and robot is moving

if a fiducial is reached
  reward with double fitness and an extra value of 50

reward if other robot is close
```

Algorithms 6.6: Pseudo code of fitness function for collective behavior.

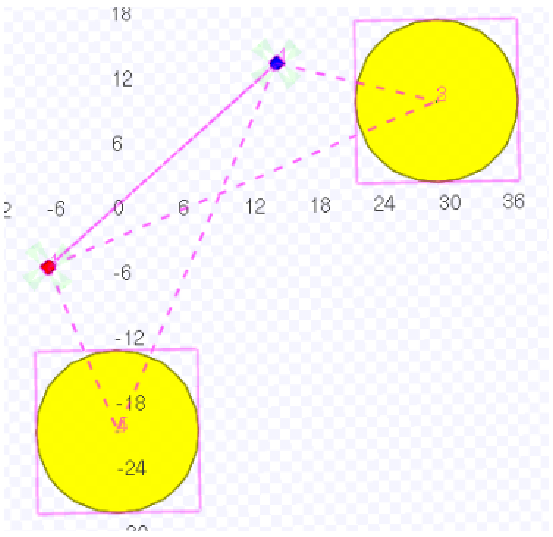


Figure 6.1: The arena containing two reference points surrounded by a light barrier.

6.1.2 Choice of Selection Function

Another challenge in order to gain successfully individuals is to apply an appropriate selection process. Selection is the moving power behind evolution. After an evolutionary cycle all individuals obtain a fitness according to their achieved performance. The selection mechanism keeps those individuals alive, that performed best, regarding a task which has to be fulfilled, and discards the contrary case.

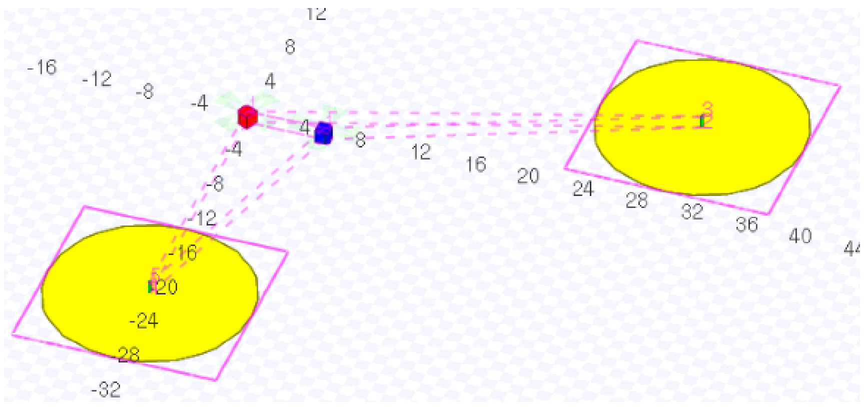


Figure 6.2: Perspective view of the arena.

Elitism

The probably simplest way to execute selection is to order all individuals of a population in ascending order, drop out the bottom half and replace it by the other 50 percent of the population which performed better. In this diploma thesis we call such kind of screening *elitism selection*.

A variant of the elitism selection is to choose a percentage of the population less than 50 percent that will be declared as elite. The, for example, 10 percent of individuals of a population that performed best (the elite) replace the 10 percent that performed the poorest. The remaining 80 percent of the population survive and, as well as the elite, get another chance to gain better fitness in the next generation. Algortihm 6.7 shows the implementation of the elitism selection that remains a part of the population.

```
void ElitismRemainSelection::performSelection
        (std::vector< NetCon* > & population)
{
  evoplog("Select next generation");

  unsigned int discardNumber =
        ceil((population.size() * percentageOfElite) / 100);

  std::vector<NetCon*>::iterator eliter = population.begin();
  std::vector<NetCon*>::reverse_iterator discarder =
        population.rbegin();

  for (unsigned int i=0; i <= discardNumber; i++){
    delete *discarder;
    *discarder = (*eliter)->clone();
    discarder++;
17      eliter++;
18    }
19  }
```

Algorithms 6.7: Elitist selection function that remains individuals with a moderate performance.

At first, the variable *discardNumber* stores the amount of individuals that are going to be discarded or, in other words, the part of the population which is going to be replaced by the elite. The percentage of the elite in the population is set by the parameter *percentageOfElite*. In the for loop from line 13 to line 18 one individual with a low fitness after another is deleted and replaced by an elitist member.

Additionally the elite can be excluded from mutation and thus is able to perform in the following cycle at another position again to demonstrate that it belongs to the elite group.

6.1.3 Choice of Team Composition and Level of Selection

"The design of control rules for multi-agent systems is challenging because agent behavior depends not only on interactions with the environment, but also on the behavior of other agents. As the number of interacting agents in a team grows, or when agent behaviors become more sophisticated, the design of suitable control rules rapidly becomes very complex. This is especially true when agents are expected to coordinate or cooperate to collectively achieve a desired task." [WKF09]

Thus the artificial evolution of a team of robots must address the following two major issues:

- Selection may either operate on robot teams or on individuals.
- Individuals of a population may either employ different control rules or share their control rules.

As in [WKF09] teams with identical control rules are referred to as *genetically homogeneous*. They share the same genes. *Genetically heterogeneous* individuals have different control rules. Waibel et al. concluded that heterogeneous teams with team selection performed inefficient and cannot be recommended for cooperative tasks. It follows the application of heterogeneous teams evolved with individual level selection for the experiments in this thesis. "Team heterogeneity allowed to evaluate a high number of different genomes in parallel, and individual selection allowed efficient selection of good genomes." [WKF09].

6.1.4 Online and Onboard Evolution

The task of this thesis requires online and onboard evolution of robot controllers. As Bredeche states in [Bre04] evolutionary algorithms are usually applied offline, meaning that evolutionary processes develop one or several possible solutions to the given problem and after that one solution is selected for further deployment and application. In situations where the optimization process must be run on-board, and self-adaptation is performed online during the actual exploitation of the robot, for example when there is no possibility to retrieve the robot for further training, offline evolution cannot be applied. Another example for the application of online learning or optimization: "the supervisor may not be able to re-initialize the environment and robot location inbetween evaluations, it may not be possible to identify a priori what is the structure of the environment prior to development or it may be possible that the environment may change during the course of experiment. In such cases, a "self-adaptation" mechanism is required to ensure survival and task-fulfilling in the long term. [Bre04]"

In [BHE09] Bredeche et al. show the viability of online, onboard evolution in autonomous robots. "We have presented the (1+1)-ONLINE evolutionary algorithm to provide continuous adaptation in autonomous robots in unknown and/or changing environments, without help from any supervisor [...]" [BHE09]. Further information about the (1+1)-ONLINE evolutionary algorithm is provided in [MB11]. Further on, they present two categories of related work on the online and onboard evolution of robot controllers as follows:

Distributed online onboard evolution approach "Each robot carries one genotype and is controlled by the corresponding phenotype. Robots can reproduce autonomously and asynchronously and create offspring controllers by recombination and/or mutation." [BHE09]

Encapsulated online onboard evolution approach "A robot has an EA implemented on-board, maintaining a population of controllers inside itself. The EA is running on a local basis and perform the fitness evaluations autonomously. This is typically done in a time-sharing system, where one member of the inner population is activated (i.e., decoded into a controller) at a time and is used for a while to gather feedback on its quality." [BHE09]

Both approaches employ heterogeneous populations of robot controllers.

Summarizing the previous considerations, online and onboard evolution is motivated by the fact that it is not always possible to train the robot from the start in a controlled environment because of several constraints or to provide adaptation in autonomous robots in changing environments, without the help from a supervisor.

Regarding the results of Karafotias et al. in [KHE11], online and onboard evolution of robot controller is successful with small population and within short times. Additionally, Karafotias emphasizes the advantages of distributed and of encapsulated evolution as follows: "a distributed evolutionary algorithm exploits the presence of multiple robots by effectively implementing concurrent evaluation of the population, allowing evolution to progress rapidly in real time" [KHE11], on the other hand the "encapsulated approach can offer an advantage when dealing with tasks or environments that involve competition or require various skills: here, the separate evolutionary algorithms of robots can promote co-evolution, speciation and/or specialisation" [KHE11]. They concluded that a combination of both approaches, leading into an island model algorithm, may be the most promising approach.

Consequently, this thesis is running an island consisting of several controllers one after the other on one robot, thus applying the encapsulated approach within each robot and the distributed approach between the two robots.

6.1.5 Execution of the Task in Simulation

Different test cases require various conditions such as the size of the arena, the number of obstacles, the start position of the robots and many more. For the task of this thesis the arena is chosen to be relatively large compared to the size of the robots. The reference points are placed approximately in the centre of the arena and in the middle of the surrounding barrier as shown in figure 6.3.

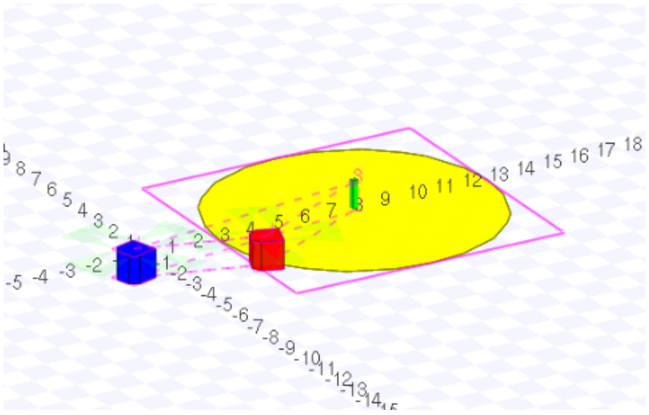


Figure 6.3: Representation of the fiducial sensors.

The green target object, the yellow barrier as well as the other robot are recognized by the robot's fiducial sensor. One fiducial sensor detects objects, that are located on the floor, the other sensor detects all objects, that have at least the size of the robot. It had to be taken into account here, that an object being placed between the robots hinders the view of one robot to the other. In the case of both robots having reached the target object at the same time, the target object should not inhibit the detection of the other robot. Thus, the shape of the reference object is narrow and elongated. In addition, the height of the target object must have at least the height of a robot to be detected by it's fiducial sensor as object and not as barrier.

Another challenge was the influence of the object to the barrier, because in Stage both items share the same reference point. The solution is a lift of the target object by a few centimeters.

Another characteristic of Stage is the uncertainty in the measured pose of a fiducial. As illustrated in figure 6.4 the bounding box of a fiducial sometimes flips from its original position, thus injecting noise into the simulation.

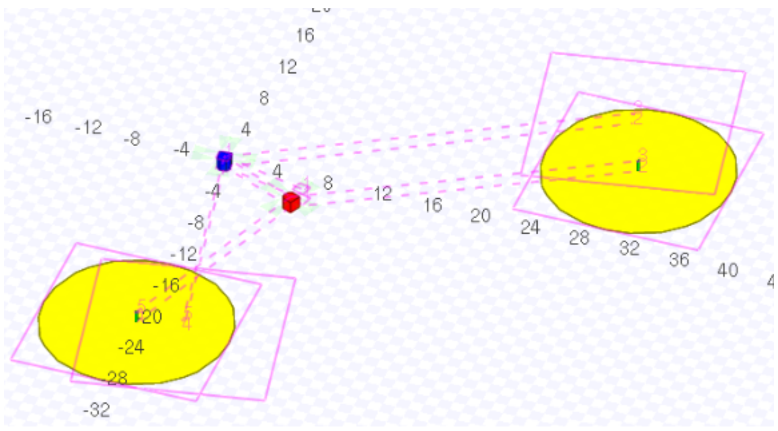


Figure 6.4: Illustration of the uncertainty in the measured fiducial pose in Stage.

During the execution in simulation the *Controller* controls the barriers surrounding the green target objects by activating and deactivating the function of the barriers. In the beginning the robots should drive to reference point 1 with fiducial ID 3 (see figure 6.4), then start out to the second reference point with fiducial ID 5 and afterwards move back and forth between the two green goals. In the initial state, only the barrier around reference point 1 is activated, thus the robots are able to reach the second reference point. Anyway, the robot only gains a reward if moving to the right object, regarding the correct order. In this case, the right object means the object with an activated light barrier. If a robot reaches an object the corresponding light barrier is deactivated and the barrier of the other object is activated. Thus a robot cannot get stuck at a barrier of an object it should not head out to.

6.2 Experimental Setups

The next subsections describe the setups of the four experiments carried out in this thesis. Evolution is executed online and onboard on two robots following a combination ot the encapsulated and the distributed approach of Bredeche [BHE09] introduced above. Every robot maintains an island of individuals with the size of 10 inside itself. Every such individual is activated one after the other for 160 evaluation steps in simulation to gather fitness for its performance. After a population is evaluated completely the robot controller resets the robot to its start position and switches to the next generation. The robot reset is forced by reproducibility and simplifies the comparison of different parameter values. Neither individuals of the same population nor individuals from different populations share any kind of information such as their fitness or current state.

So, all experiments work on two populations of size 10 with linear genomes consisting of 12 input neurons and two output neurons. In all four experiments 60 generations are executed in one run and the fitness value of the best individual and the average fitness are printed into a file in every generation. Forced by computational limitations we only tested 10 runs in every experiment. In experiment 1 to 3 structural mutation is carried out once after 50 generations. The fitness function as well as the selection function stay the same in all four setups. For the computation of the fitness, the fitness function for collective behavior illustrated in listing 6.6 is applied. Selection is carried out by the selection function 6.7 which remains that part of the population that performed best.

Table 6.1: Table showing the mapping of sensor values to input neurons.

input	**corresponding sensor type**	**value range**
I0	bias neuron initial state is 1.0; when a robot reached the first fiducal the input to the bias neuron switches to -1.0 and vice versa when a robot reached the second fiducial	[-1.0, 1.0]
I1	front range sensor left	[0, 1.0]
I2	front range sensor right	[0, 1.0]
I3	rear range sensor left	[0, 1.0]
I4	rear range sensor right	[0, 1.0]
I5	fiducial sensor for distance to fiducial with ID 1 (other robot)	[0, 1.0]
I6	fiducial sensor for distance to fiducial with ID 3 (reference point 1)	[0, 1.0]
I7	fiducial sensor for distance to fiducial with ID 2 (barrier 1)	[0, 1.0]
I8	fiducial sensor for angle between robot and fiducial with ID 3 (reference point 1)	[0, 1.0]
I9	fiducial sensor for distance to fiducial with ID 5 (reference point 2)	[0, 1.0]
I10	fiducial sensor for distance to fiducial with ID 4 (barrier 2)	[0, 1.0]
I11	fiducial sensor for angle between robot and fiducial with ID 5 (reference point 1)	[0, 1.0]

Table 6.1 describes the meaning of each input neuron of the artificial neural networks used in the following experiments. The distance sensors deliver a value of 0, if the measured object is not within range. The closer an object, the greater the value the input receives. The two output neurons N0 and N1 correspond to the speed and the turn angle of the robot, respectively.

6.2.1 Setup 1 - Simple Neural Network

The first experiment starts with a randomly created fully connected artificial neural network with randomly generated weights. An example initial network is depicted in figure 6.5. This simple neural network has no hidden nodes. Input node I0 serves as a bias neuron. The initial value of the bias neuron is 1.0. When a robot reached the first fiducial the robot controller flips the bias value to -1.0. Reaching the second fiducial the bias input switches back to 1.0. Thus, the bias acts like a switch. The corresponding configuration file is shown in listing 6.8.

```
EvolutionaryAlgorithmType     ANN
PopulationSizeOfIsland        10
LoadGenomesFromFiles          NO

NumberOfInputNeurons          12
NumberOfOutputNeurons         2

NumberOfStepsForEvaluation    160
NumberOfGenerations           60
NumberOfExploitationSteps     50

DefaultLearningRate           0.3
DefaultMutationProbability    0.1

FitnessFunctionType           DISTaCOLL
SelectionMode                 ELITISMaREMAIN
SelectionParameter            30.0
```

Algorithms 6.8: Main parameters of the configuration file for experiment 1.

6.2.2 Setup 2 - Network with Hidden Neurons

In the second experiment a user generated artificial neural network is applied. As in experiment 1 the weights are generated randomly in the range of -1 to 1. As illustrated in figure 6.6 the network contains four hidden neurons. The first hidden node N2 builds a sub-genome with the bias neuron I0, input nodes I6 and I7 as inputs and N0 as output. Thus, the hidden neuron N2 influences the speed of the robot according to the distance to reference point 1 and the surrounding barrier.

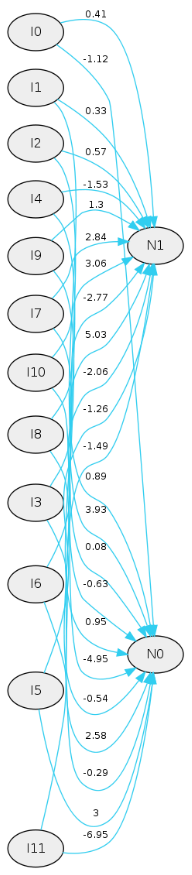


Figure 6.5: Example of initial fully connected graph with random weights.

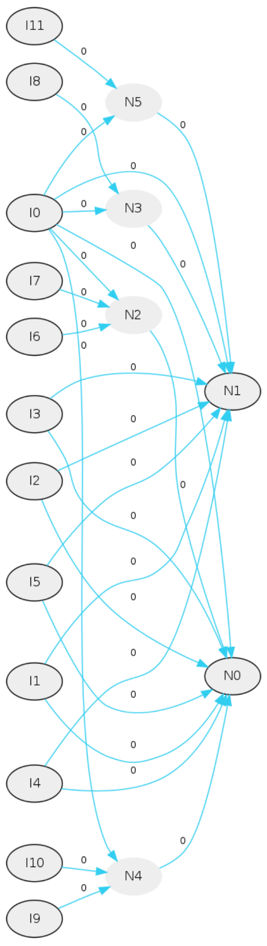


Figure 6.6: Initial graph with zero weights.

Hidden node N4 has the same functionality regarding the second reference point. N3 and N5 receive the values of the angle between the robot and reference point 1 or 2, respectively, as well as the bias input and transmit their values to output neuron N1, thus adjusting the angle of the robot. Listing 6.9 shows the corresponding configuration file for the setup of experiment 2.

```
EvolutionaryAlgorithmType     ANN
PopulationSizeOfIsland        10
LoadGenomesFromFiles          YES

NumberOfInputNeurons          12
NumberOfOutputNeurons         2
```

```

NumberOfStepsForEvaluation   160
NumberOfGenerations          60
NumberOfExploitationSteps    50

DefaultLearningRate          0.3
DefaultMutationProbability   0.1

FitnessFunctionType          DISTaCOLL
SelectionMode                ELITISMaREMAIN
SelectionParameter           30.0
```

Algorithms 6.9: Main parameters of the configuration file for experiment 2.

6.2.3 Setup 3 - Trained Network with Hidden Neurons

The applied network of experiment 3 is the most developed network regarding the desired task. The topology of the network is the same as of the previous experiment. However, this initial networks are already trained in moving to a reference point, which means that the initial individuals are capable of solving the given task. It is particularly interesting here to examine whether the initial genotypes remain along the evolutionary path.

6.2.4 Setup 4 - Simple Network with Structural Mutation

Because the experiments 1-3 do not investigate the influence and effect of structural mutation, the fourth experiment applies structural mutation every fifth generation starting with the same neural network as of the first experiment. Listing 6.10 shows the main part of the corresponding configuration file for this setup. The rest of the configuration file remains the same as in the previous listings.

```
EvolutionaryAlgorithmType    ANN
PopulationSizeOfIsland       10
LoadGenomesFromFiles         NO

NumberOfInputNeurons         12
NumberOfOutputNeurons        2

NumberOfStepsForEvaluation   160
NumberOfGenerations          60
NumberOfExploitationSteps    5
```

Algorithms 6.10: Main parameters of the configuration file for experiment 4.

7 Results and Evaluation

This chapter covers the results of the conducted experiments introduced in chapter 6.2. In the following results and diagrams a fitness value exceeding the value of 50 corresponds to collectively reaching the first of two reference points. A fitness value greater than 200 meets the achievement of navigating to both fiducials one after the other within the predefined, available 160 evaluation steps. Also, the solid lines show the average of the best individuals over all 10 runs. The dotted lines show the average of the average of all 10 individuals of the population over all 10 runs. Red lines correspond to the red robot on port 6665 and analogue the blue lines to the blue robots on port 6666. At the end of each experiment, we extract the linear genome of the individual of an island which gained the best fitness value over all 10 runs. These two genomes (the best individual of the island on the red robot and the best individual of the island on the blue robot) are evaluated in simulation one more time. However, this time there is no restriction of evaluation steps and the population size of the island on each robot is one. Herewith, we examine the evolved behavior more detailed by repositioning the fiducials and their light barriers to find out if the evolved controllers are able to adapt to the changing environment.

7.1 Result of Simple Neural Network

Figure 7.1 shows the results of the first experiment. The individuals starting with a simple fully connected and randomly created initial network increase their fitness continuously, regardless of the fitness value corresponding to the best individuals or the average of all 10 individuals of the population. On average, after about 40 generations all the island members accomplished to reach the first reference point successively. In the best case, the robots succeeded to reach the first fiducial after about 15 generations. Considering the mean value, within the 60 generations of a run no individual achieved the goal of moving back and forth between reference point one and two. An impact of the structural mutation that was carried out after 50 generations cannot be detected in the results of experiment 1.

Figure 7.2 shows the best individuals of the run that contains the highest fitness value of all 10 runs for the red and the blue robot, respectively. Here and in the following experiments the term "best" means the individual with the highest fitness value, not the individual that reached one or two fiducials most often. In this run,

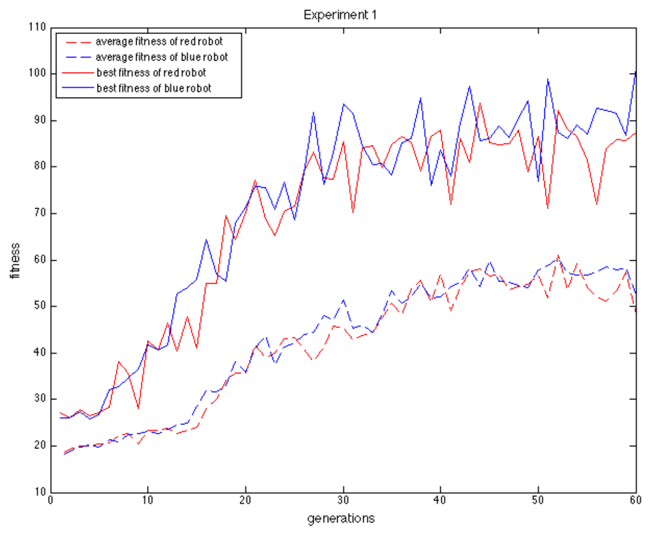


Figure 7.1: Results of experiment 1.

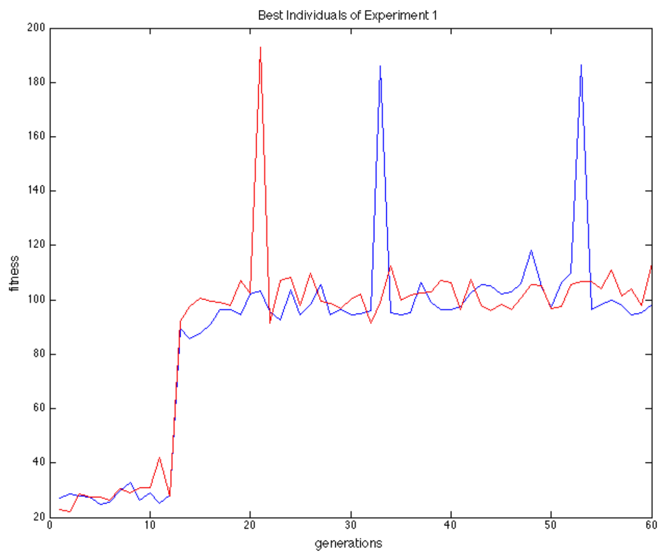


Figure 7.2: Run that contains the individuals with the highest fitness value of experiment 1.

the best individuals on the red robot almost reached both fiducials one after the other once, the best individuals on the blue robot twice. During evaluation, it was observed that either there was not enough time and thus the robot was reset before reaching fiducial number two or the robot with the lower fitness value passed by the second barrier (and thus the second reference point) while the other robot was stuck at the barrier being too far away from the passing robot to be able to overcome the barrier.

The graphs in figure 7.3 and 7.4 show the artificial neural networks of the individuals which gained the highest fitness values of their population.

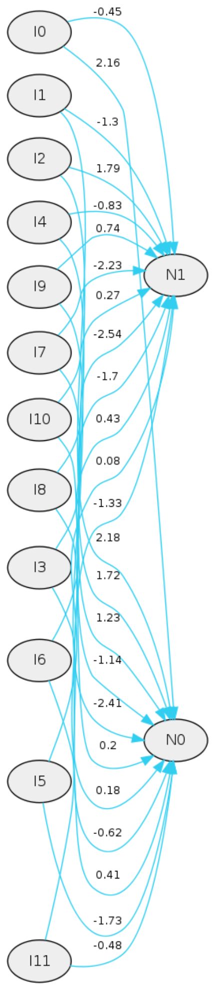


Figure 7.3: Graph of the individual with the highest fitness of the island on the red robot.

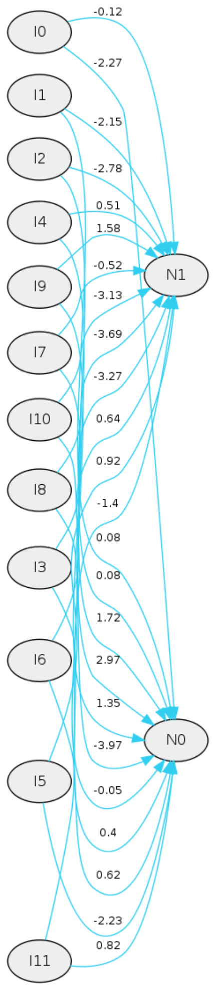


Figure 7.4: Graph of the individual with the highest fitness of the island on the blue robot.

The picture sequences of table 7.1 and 7.2 illustrate the trajectories of the evolved robots according to their linear genomes depicted in figure 7.3 and 7.4. Pictures a-h show the common movement to reference point 1 in the right upper corner of the arena. Picture i of table 7.2 shows the red robot setting out to the second reference point on the left. The blue robot moves to reference point 2 as well (picture k and l) and get stuck at the barrier of fiducial 2 (picture m and n) until the red robot is within reach. Then both robots drive back to reference point 1.

Instead of moving on together to the second reference point both robots make a U-turn in front of the second barrier and drive back to object 1. Thus, the evolved robots of this experiment only fulfilled the subtask of jointly moving to reference point 1.

Table 7.1: First part of the picture sequence illustrating evolved robots of experiment 1.

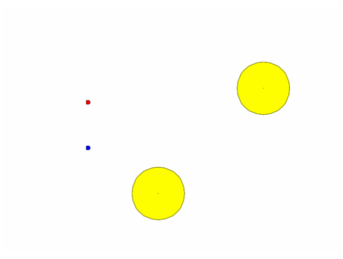
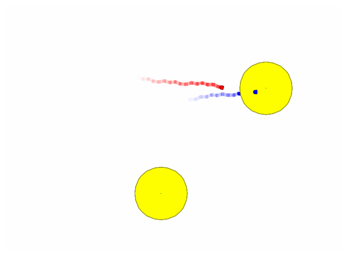
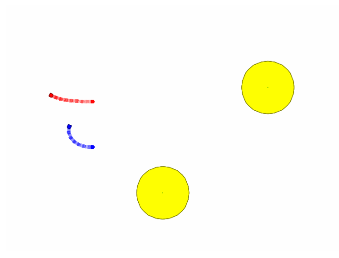
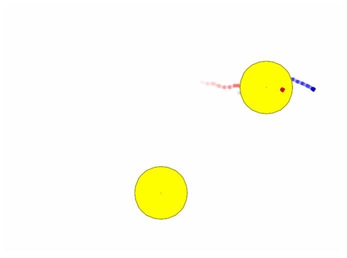
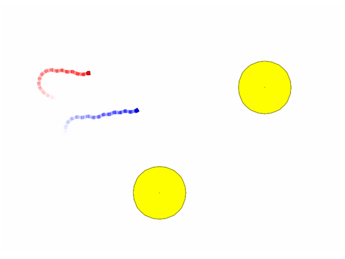
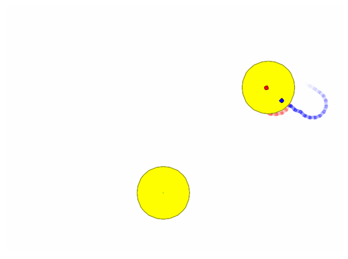
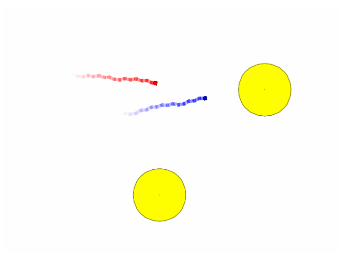
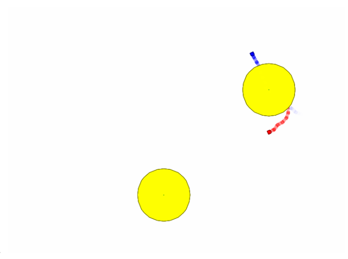
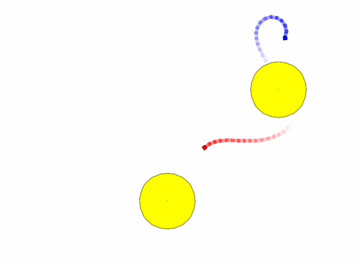
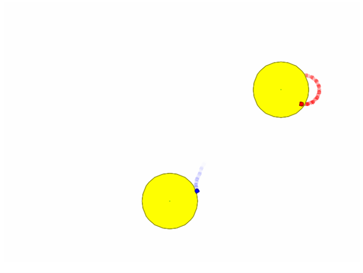
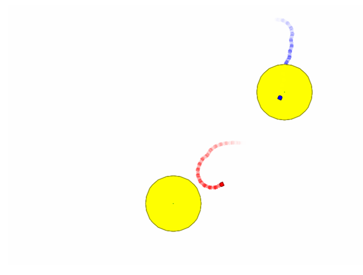
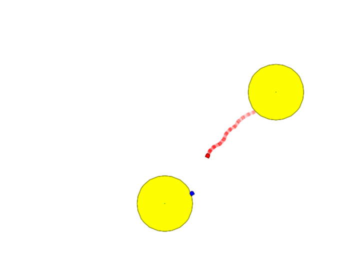
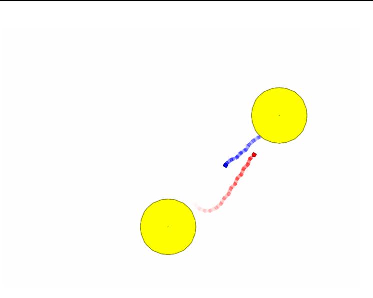
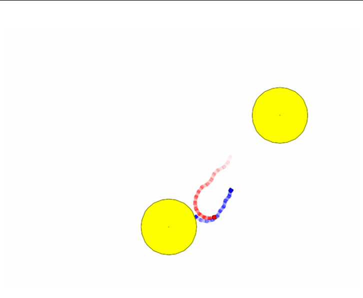
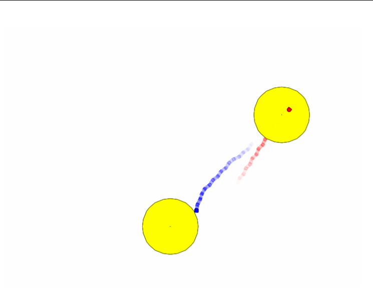
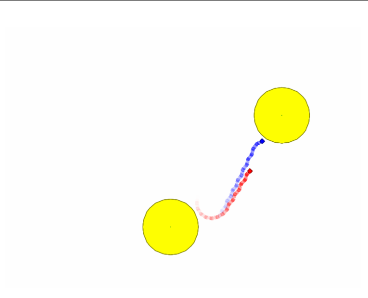
 a	 e
 b	 f
 c	 g
 d	 h

Table 7.2: Second part of the picture sequence illustrating evolved robots of experiment 1.

 i	 m
 j	 n
 k	 o
 l	 p

7.2 Result of Network with Hidden Neurons

Compared with the first experiment, the fitness in experiment 2 (see figure 7.5) starts to increase a little later after about 20 generations. However, the fitness value then rises more steeply to a higher value at the end of the 60 generations. Regarding the average fitness, this time an effect of the structural mutation at generation 50 can be noticed by a short decline of the fitness for a few generations.

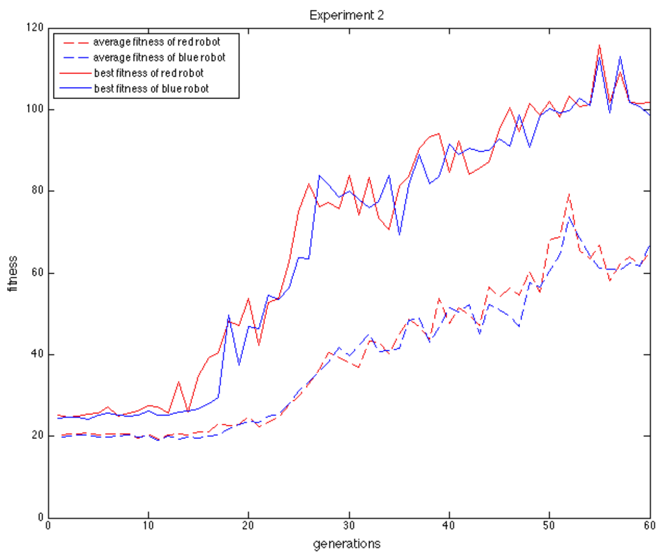


Figure 7.5: Results of experiment 2.

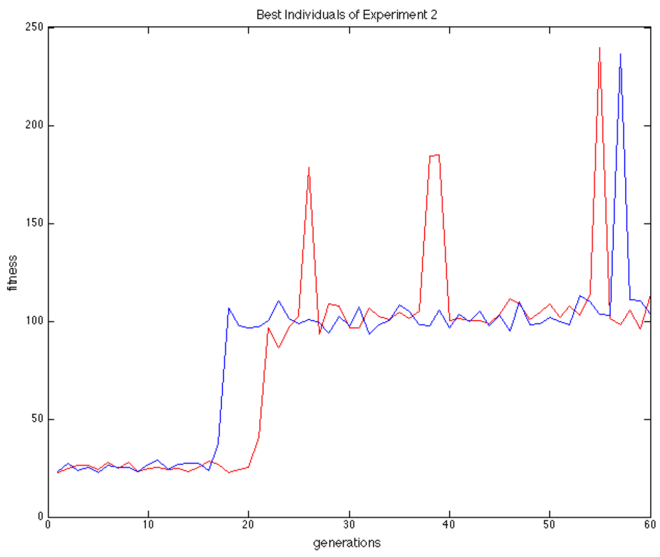


Figure 7.6: Run that contains the individuals with the highest fitness value of experiment 2.

In this experiment using artificial neural networks which contain hidden nodes, after about 20 generations the best individuals of the population reached the first reference point in common. The average achieved that subtask after about 40 generations.

In the second experiment also the fitness of the best individuals in figure 7.6 exceeded those of experiment 1. The best individual on the red robot almost succeeded in reaching both fiducials for the first time after approximately 25 generations with a fitness value of 175, the second time after about 37 generation and then even a few generations in a row. After about 55 generation the red robot reached the second reference point with a fitness value of 240. The best individual on the blue robot succeeded once at the end of the run in moving to both fiducials.

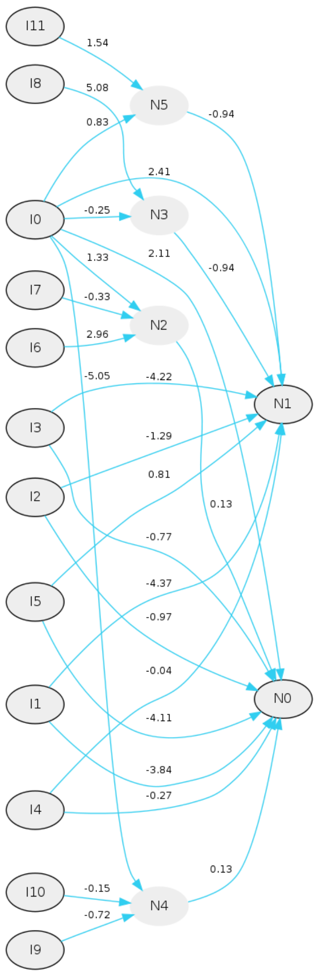


Figure 7.7: Graph of the individual with the highest fitness of the island on the red robot.

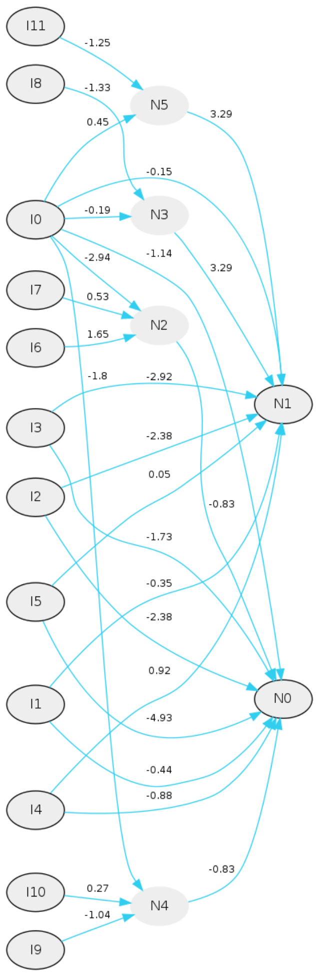


Figure 7.8: Graph of the individual with the highest fitness of the island on the blue robot.

Note that in this example the both robots did not reach the second reference point together and/or simultaneously in the given 160 evaluation steps. Also note that in experiment 1 as well as in experiment 2 the fitness of the best individuals of each population is leveling off at a value of about 100 in the first third of the whole run of 60 generations. That corresponds to the joint achievement of the robots subgoal to move to the first reference point. Consequently, the controller which is able to solve the first subtask remains in the population further on.

Figure 7.7 and figure 7.8 show the graphs of the individuals which gained the highest fitness in the second experiment.

Table 7.3: Picture sequence illustrating evolved robots of experiment 2.

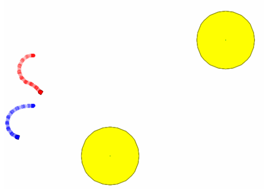
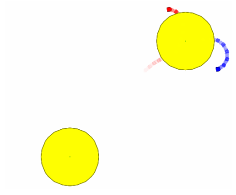
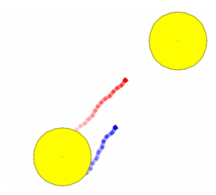
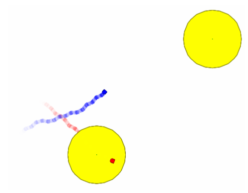
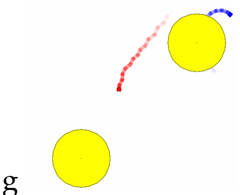
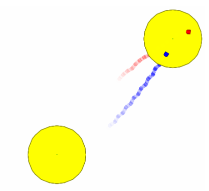
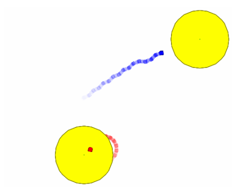
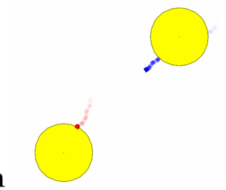
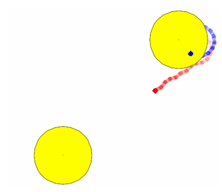
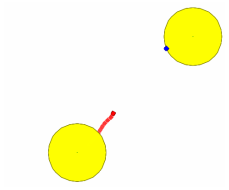
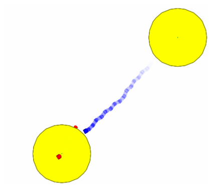
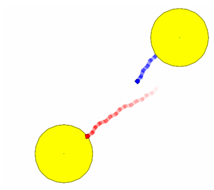
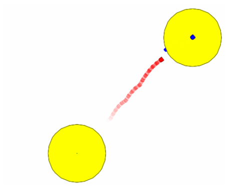
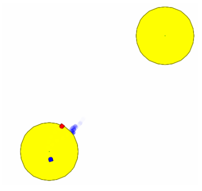
a 	f 	k 
b 	g 	l 
c 	h 	m 
d 	i 	n 
e 	j 	

Table 7.3 demonstrates the trajectory of two evolved robots of the second experiment. At the beginning both robots make a turn and then drive backwards as can be seen in picture a. The blue robot sets out directly in the direction of reference point 1, whereas for the moment the red robot is attracted by the second reference point (picture b and c). After circling around the second target object for a few steps the red robot starts out to reference point 1 where the blue robot is already waiting, or in other words, the blue robot is stuck at the light barrier of fiducial 1 (see picture d). In picture e and f both robots reach the first fiducial and start moving to reference point 2. The remaining pictures g to n show the robots moving back and forth between both fiducials.

7.3 Result of Trained Network with Hidden Neurons

As depicted in figure 7.9 the individuals started with a higher fitness compared to the population members of the previous experiments. As well as in experiment 1 and 2 the average of the best and the average fitness increased continuously. However, in experiment 3 the increase turned out to be significantly smaller and would at best stagnate, if not decline, regarding the average of the best individuals. Additionally, the fitness range of figure 7.9 is much lower compared to the fitness range of the first two experiments. Thus, all in all, the average fitness of the individuals which initially were able to perform well turns out to be lower compared to the experiments with the untrained artificial neural networks.

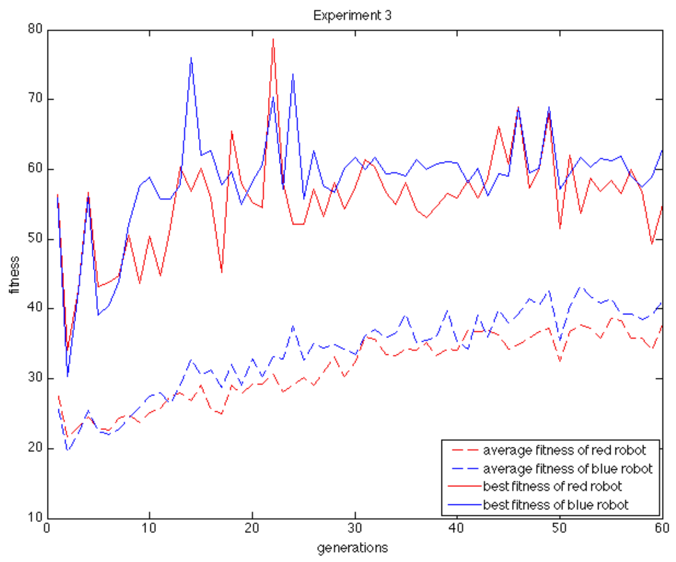


Figure 7.9: Results of experiment 3.

On the other hand, regarding the run containing the individual with the highest fitness value of all runs, figure 7.10 reveals the two individuals with the maximum fitness of all 4 experiments with a fitness value greater than 250. In addition, those two individuals reached the second fiducial simultaneously. Otherwise, as well as in the experiments before, the fitness leveled off at a fitness value of about 100. But this time, there was no such kind of jump in the first third of the generations but a fluctuation up to generation 18.

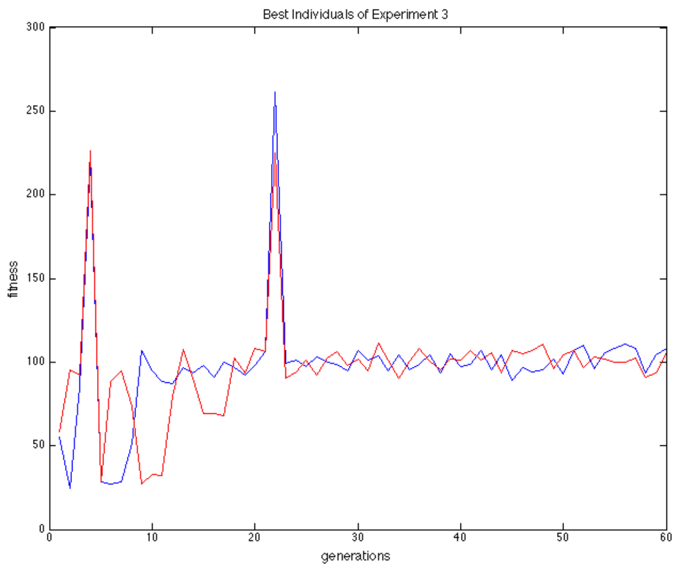


Figure 7.10: Run that contains the individuals with the highest fitness value of experiment 3.

Considering the average fitness functions at generation 50, structural mutation caused a slight and short decline of the fitness. In conclusion, the results of experiment 3 show that the initial 10 individuals of a population did not remain in no run, although they performed quite well.

Figure 7.11 and 7.12 show the phenotypes of the linear genomes of the individuals with the highest fitness. When one examines the two graphs, it can be ascertained that the sign of the edge weights of both graphs match. Also the size of the weights are proportionally very similar.

The picture sequence of table 7.4 depicts the trajectories of evolved robots by experiment 3. Picture a shows the initial position whereas the pictures b, c and d illustrate the trajectories of the red and the blue robot from the start position to reference point 1. After reaching the first object both robots move straight to the second target object (see picture e-i). Pictures j-p show another cycling between reference point 1 and 2. This time both robots take another route when moving to

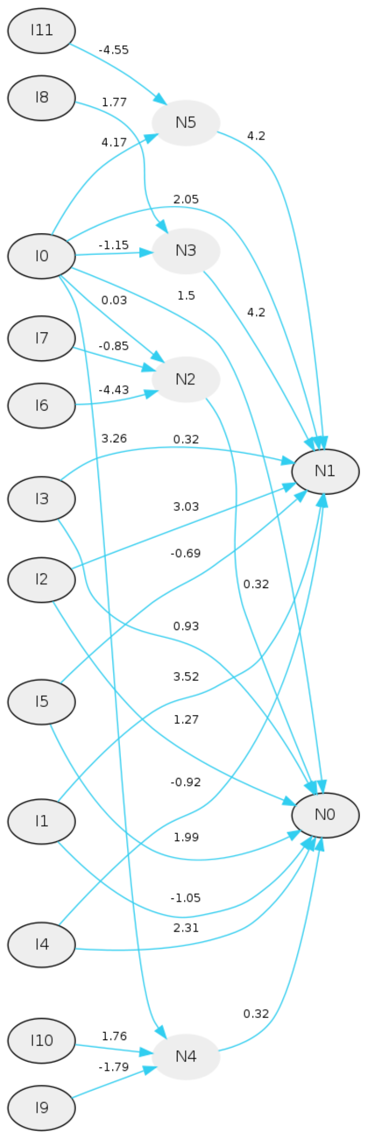


Figure 7.11: Graph of the individual with the highest fitness on the red robot for experiment 3.

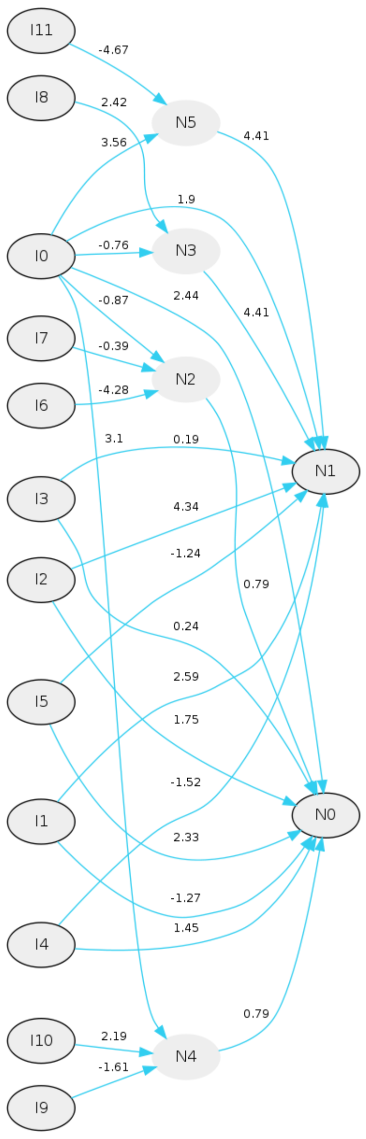
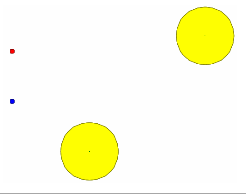
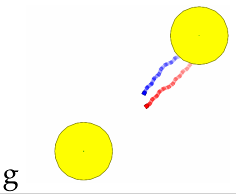
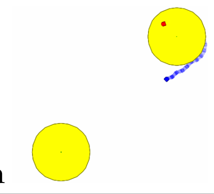
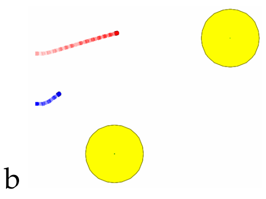
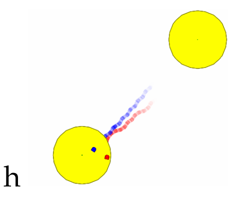
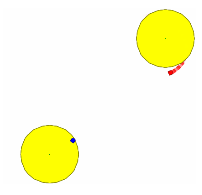
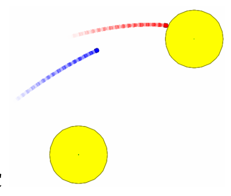
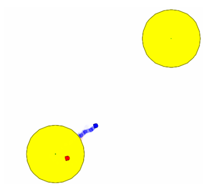
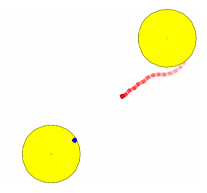
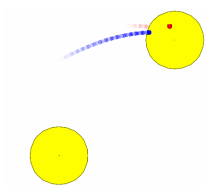
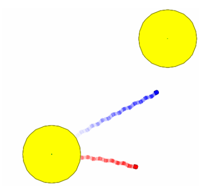
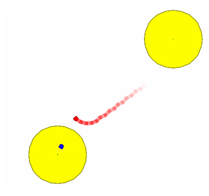
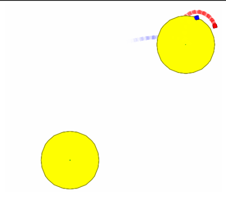
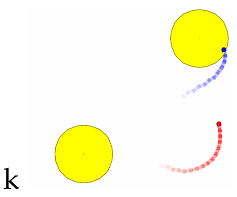
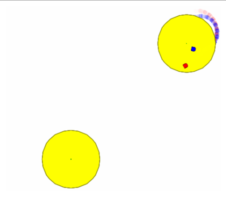
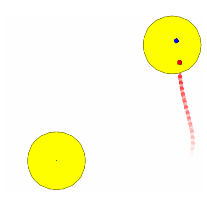


Figure 7.12: Graph of the individual with the highest fitness on the blue robot for experiment 3.

reference point 1 than in the beginning. In particular the red robot drives a bend before setting out straight to the centre of the barrier around reference point 1.

The second table of picture sequences demonstrate another experiment with the same individuals as of the previous experiment. This time the initial positions of

Table 7.4: Picture sequence illustrating evolved robots of experiment 3.

 a	 g	 m
 b	 h	 n
 c	 i	 o
 d	 j	 p
 e	 k	
 f	 l	

the target objects and their surrounding barriers are changed (see picture a in table 7.4). Reference point 1 and the barrier are shifted upwards in the arena by 20cm from coordinate (30,10) to (30,30). The second reference point and the barrier are repositioned from (0,-20) to (-30,30). The start position of the robots remains the same as before.

Table 7.5: Second picture sequence illustrating evolved robots of experiment 3 with repositioned fiducials.

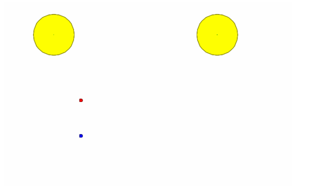
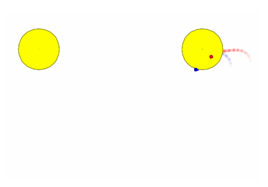
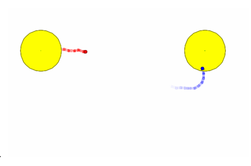
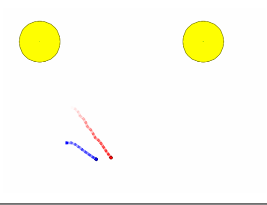
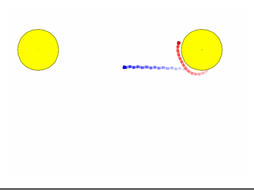
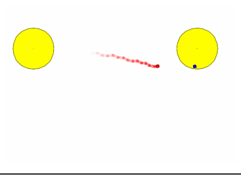
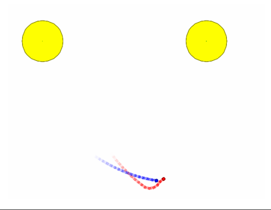
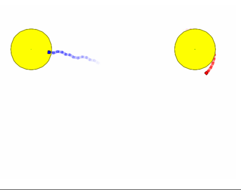
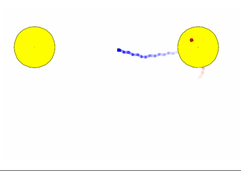
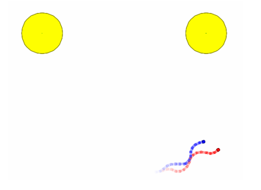
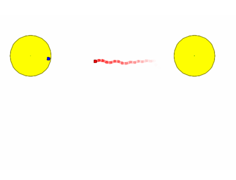
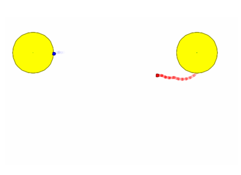
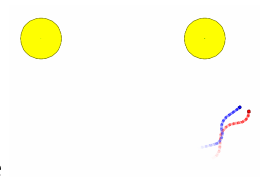
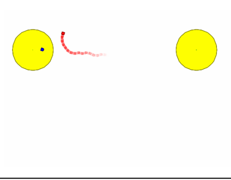
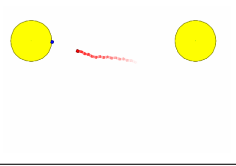
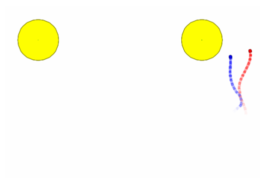
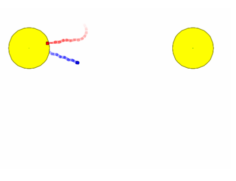
 a	 g	 m
 b	 h	 n
 c	 i	 o
 d	 j	 p
 e	 k	 q
 f	 l	

Table 7.4 shows the corresponding trajectories of the robots. Both robots drive a right-hand bend in the direction of reference point 1 as depicted in the pictures

b-f. They reach the first object from the east. The blue robot directly starts out to reference point 2 (picture g) while the red robot firstly circles around object 1 before setting out straight to object 2 as well (pictures h-j). The remaining figures k-q illustrate the further commute back and forth between object 1 and 2. It will be noted that the blue robot is a few steps ahead. But still the two individuals manage to reach the reference points one after the other.

7.4 Result of Simple Network with Structural Mutation

The fourth experiment is oriented on the investigation of structural mutation. Thus, structural mutation is carried out after every fifth generation, hence 10 times more frequent than in the previous experiments.

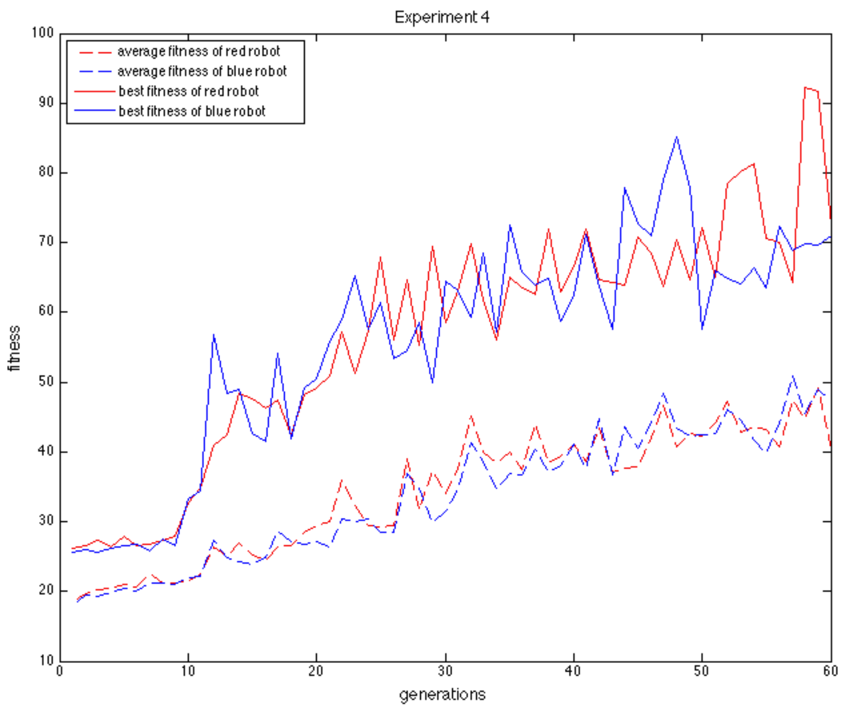


Figure 7.13: Results of experiment 4.

When comparing the results of the average fitness of experiment 4 in figure 7.13 with the results of experiment 1 shown in figure 7.1, it is noticeable that the curves of the fitness functions resemble each other regarding the first third. However, the fitness values of experiment 4 fluctuate more strongly and increase slower. Note that experiment 1 and 4 start with the same initial parameters (except the frequency of structural mutation) and the same artificial neural network, randomly created

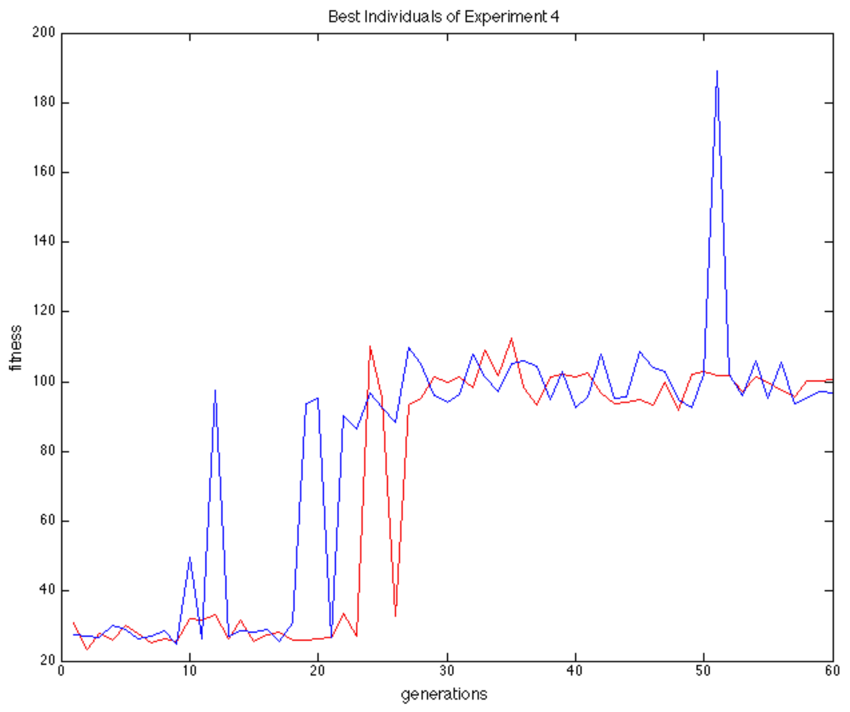


Figure 7.14: Run that contains the individuals with the highest fitness value of experiment 4.

and with a full connection between the input and output neurons of the net. As a result of the comparison one can say, that the structural mutation is responsible for the slightly stronger fluctuation in the fitness curves of experiment 4.

Figure 7.14 illustrates the best individual of a generation for all 60 generations for the red and the blue robot, respectively. The blue robot achieved the subtask of moving to the first reference point within the first 20 generations a few times and almost reached the second reference point after about 50 generations. The best individuals on the red robot performed worse in this run. However, over all 10 runs, no individual, neither on the red robot nor on the blue robot, reached the second fiducial.

The graphs in figure 7.15 and figure 7.16 show the artificial neural networks of evolved individuals of experiment 4. Although the linear genomes of graph 7.15 and graph 7.15 appeared in the last third of the run (in generation 50) there are almost no changes regarding structural mutation except a recurrent jumper and one hidden node in the network of the red robot. Remember that this experiment started with randomly created and fully connected networks with no hidden nodes, recurrent jumper or forward jumper.

However, the graph of figure 7.17 illustrates a more complex network with some

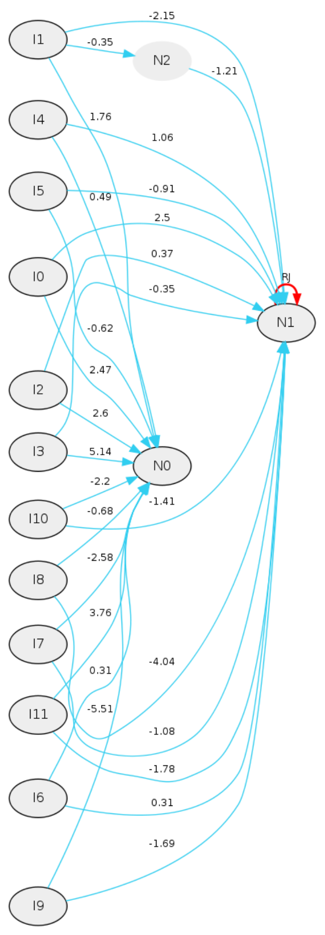


Figure 7.15: Graph of an individual of the red robot of experiment 4.

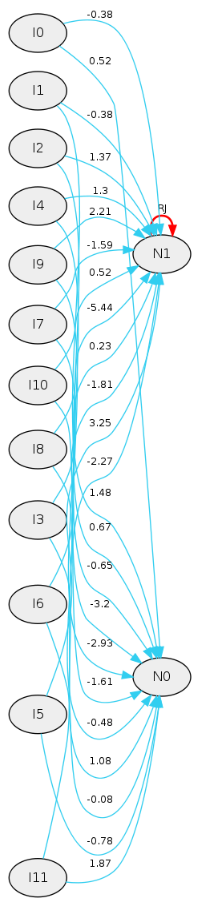


Figure 7.16: Graph of an individual of the blue robot of experiment 4.

hidden nodes and a recurrent jumper at output neuron N1. This net appeared in generation 55 of experiment 4, but - regarding 60 generations - could not reach a fitness value as high as the fitness of the individuals corresponding to the graphs of experiment 4 introduced above. Even so structural mutation was carried out after every fifth generation most of the generated graphs show almost no change.

Note that, although weights are perturbed by parametric mutation only with small values, nevertheless weights with a value greater than 5 had occurred in the artificial neural network shown in figure 7.15. For example, the edge from input node I6 to output neuron N0 is weighted by a value of -5.51.

The trajectories of two evolved robots of the last experiment can be seen in table 7.6 Both robots start out driving backwards in picture b. The blue robot moves

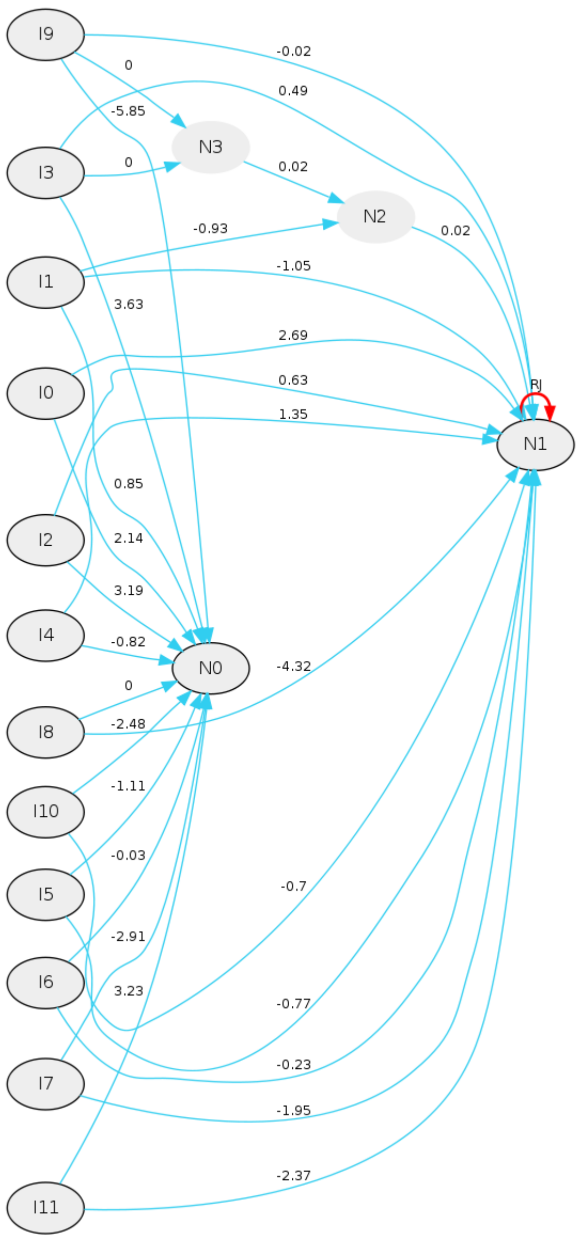
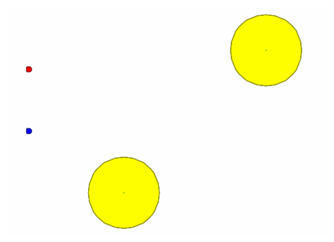
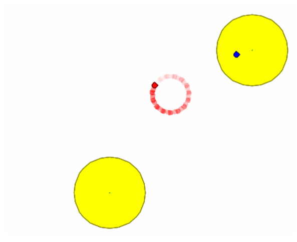
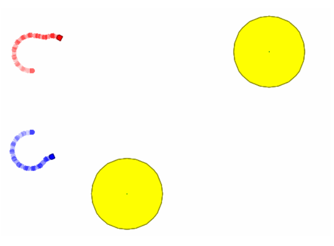
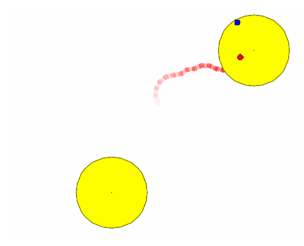
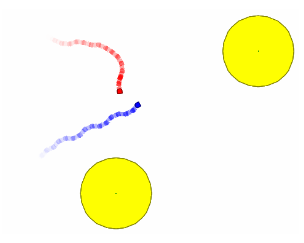
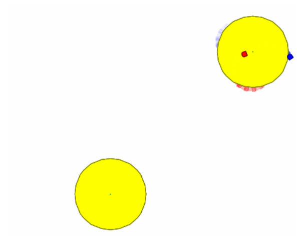
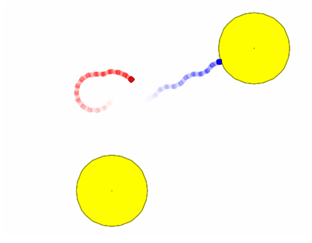
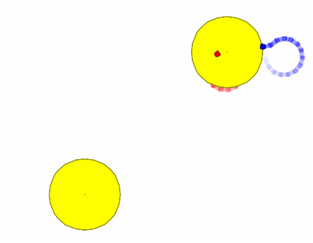
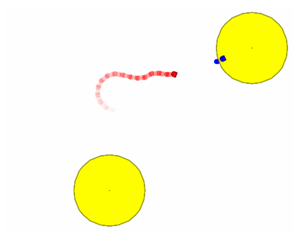
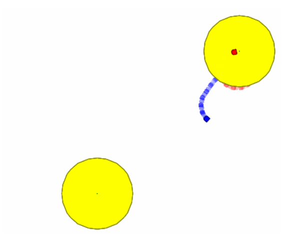


Figure 7.17: Results of experiment 4

straight to the first reference point until it get stuck at the barrier as shown in the figures c-f. The red robot turns in a few circles before moving to object 1 (picture c-g). After both robots reached reference point 1 they circle around the first green object. They continue doing so and do not commute to the second target object.

Table 7.6: Picture sequence illustrating evolved robots corresponding to the graphs of figure 7.15 and 7.16 of experiment 4.

 a	 f
 b	 g
 c	 h
 d	 i
 e	 j

8 Conclusion

In this thesis, it was examined whether members of a swarm of robots are able to develop cooperative or coordinated behavior to collectively solve a common task. For this purpose our implementations were build on the EvoRoF framework. Existing functionalities were enhanced to facilitate the evaluation of robots online and onboard in the simulation. In order to test the framework after the implementation, smaller experiments with two robots were executed such as collision avoidance with random walk and jointly moving to a reference point. Afterwards, 4 experiments were carried out to share the common goal but start with different initial starting conditions. This can be seen as an increase of the appropriateness or the power of the underlying artificial neural networks. Thus, in experiment 1 a simple net was chosen which simply connects the inputs and the output neurons whereas experiment 2 is based on a predefined user generated network and experiment 3 is based on the same predefined network, which was, trained by previous experiments to be able to solve the task. The fourth experiment investigated the effect of structural mutation on the development of swarm robotics. Comparing the results of these experiments revealed, that the robots of experiment 2 have developed to become the most efficient individuals. Contrary to our expectations, the robots of experiment 3 have the worst record regarding the average of all runs. But on the other hand the individuals that gained the best fitness over all experiments appeared relatively early in this experiment. The results of experiment 4 are showing that networks generated through structural mutation improve as well, but not as fast as the networks of the previous experiments. This shows the importance of the right balance between parametric and structural mutation. Furthermore we can say that completely untrained networks develop faster and perform better at the end as predefined and partly trained networks.

All in all, it has been shown that the existing framework this thesis is based on works well, meets all requirements and provides a promising approach for further investigations in the field of evolutionary approaches. The task of this thesis to develop coordinated behavior could be solved by using the EvoRoF framework. For future work it might be interesting to investigate more complex scenarios such as moving to several reference points using a larger population or more than two robots in the simulation. When the associated hardware is available, it will be very interesting to observe how the evolved simulated robots in this thesis perform on their real counterpart.

9 Appendix

karobot.inc

```
define karobot_ir ranger (
  # number of ir sensors
  scount 8

  # define the pose of each transducer
  # [xpos ypos heading]
  spose[0] [ 0.5 0.4 0]
  spose[1] [ 0.5 -0.4 0]
  spose[2] [ 0.4 -0.5 270]
  spose[3] [-0.4 -0.5 270]
  spose[4] [-0.5 -0.4 180]
  spose[5] [-0.5 0.4 180]
  spose[6] [-0.4 0.5 90]
  spose[7] [ 0.4 0.5 90]

  # define the field of view of each transducer
  # [range_min range_max view_angle]
  sview [0.05 2.0 30]

  # define the size of each transducer
  # [xsize ysize] in meters
  ssize [0.01 0.05]
)
define karobot position (
  # actual size
  size [1 1 1]

  block(
    points 6
    point[0] [0.75 0]
    point[1] [1 0.25]
    point[2] [1 0.75]
    point[3] [0.75 1]
    point[4] [0 1]
    point[5] [0 0]
  )

  drive "diff"      # differential steering model

  # sensors attached to karobot
  karobot_ir()
  karobot_bl()
  obstacle_return 1   # can hit things
  laser_return 1      # reflects laser beams
  ranger_return 1     # reflects sonar beams
  blob_return 0       # seen by blobfinders
)
```

Algorithms 9.1: The Stage file *karobot.inc*.

simple.world

```
include "karobot.inc"

# milliseconds per update step
interval_sim 100

# real-time milliseconds per update step
interval_real 100

window (
  size [700.000 600.00]
  scale 20
  show_data 1
  show_flags 1
)
define floorplan model (
  color "gray30"
  boundary 1      # enable the bounding box

  gui_nose 0
  gui_grid 0
  gui_move 0
  gui_outline 0
  gripper_return 0
  fiducial_return 0
  laser_return 1
)
floorplan (
  bitmap "empty_ellipse.png"
  size [120 80 3]
)
define beacon model(
  size [ 0.200 0.200 1.000]
  gui_movemask 0
  gui_nose 0
  obstacle_return 0
  laser_return 0
  ranger_return 1
)

# the target objects
beacon( pose [ 30.0 10.0 0.1 0 ] color "green" fiducial_return 3)
beacon( pose [ 0.0 -20.0 0.1 0 ] color "green" fiducial_return 5)
#beacon( pose [ 30.0 30.0 0.1 0 ] color "green" fiducial_return 3)
#beacon( pose [ -30.0 30.0 0.1 0 ] color "green" fiducial_return 5)

define barrier position (
  size [10 10 0]
  sorigin [0 0 0]
  boundary 0

  # coordinates for a circle serving as barrier around the target object
  block(
    points 24
    point[0] [1 0]
    point[1] [0.97 0.25]
    point[2] [0.85 0.55]
    point[3] [0.75 0.675]
    point[4] [0.55 0.85]
    point[5] [0.25 0.97]
    point[6] [0 1]
    point[7] [-0.25 0.97]
    point[8] [-0.55 0.85]
    point[9] [-0.75 0.675]
    point[10] [-0.85 0.55]
    point[11] [-0.97 0.25]
    point[12] [-1.0 0]
    point[13] [-0.97 -0.25]
    point[14] [-0.85 -0.55]
    point[15] [-0.75 -0.675]
    point[16] [-0.55 -0.85]
    point[17] [-0.25 -0.97]
    point[18] [0 -1]
    point[19] [0.25 -0.97]
    point[20] [0.55 -0.85]
    point[21] [0.75 -0.675]
    point[22] [0.85 -0.55]
    point[23] [0.97 -0.25]
  )
  gui_movemask 0
  gui_nose 0
  gui_outline 0
  obstacle_return 0
  laser_return 0
  ranger_return 0
)

# the light barriers around the target objects
barrier( pose [ 30.0 10.0 0 0 ] color "yellow" fiducial_return
   2)
barrier( pose [ 0.0 -20.0 0 0 ] color "yellow" fiducial_return
   4)
#barrier( pose [ 30.0 30.0 0 0 ] color "yellow" fiducial_return
   2)
#barrier( pose [ -30.0 30.0 0 0 ] color "yellow"
   fiducial_return 4)

# define the two fiducial sensors for the robot
define myBot karobot (
  fiducial (
    pose [0.000 0.000 -1.000 0.000]
    range_min 0
    range_max 100
    range_max_id 100
    fov 360      # sensor range in degrees
  )
  fiducial (
    pose [0.000 0.000 0.000 0.000]
    range_min 0
    range_max 100
    range_max_id 100
    fov 360      # sensor range in degrees
  )
  localization "gps"
  localization_origin [ 0 0 0 0 ]
)

# position the red robot in the arena
myBot (
  name "myrobot1"
  pose [-20 6 0 0]    # initial position
  color "red"
  fiducial_return 1
```

```
)

# position the blue robot in the arena
myBot (
  name "myrobot2"
  pose [-20 -7 0 0]   # initial position
  color "blue"
  fiducial_return 1
)
```

Algorithms 9.2: The Stage world file *simple.world*.

fiducialfinder.cfg

```
driver (
  name "stage"
  plugin "stageplugin"
  provides ["6665:simulation:0"]
  # load the named file into the simulator
  worldfile "simple.world"
)
driver (
  name "stage"
  provides ["6666:simulation:0"]
)

# driver for the red robot
driver (
  name "stage"
  provides ["6665:position2d:0" "6665:sonar:0" "6665:fiducial:0" "6665:fiducial
     :1"]
  model "myrobot1"
)

# driver for the blue robot
driver (
  name "stage"
  provides ["6666:position2d:0" "6666:sonar:0" "6666:fiducial:0" "6666:fiducial
     :1"]
  model "myrobot2"
)
```

Algorithms 9.3: The Stage configuration file *fiducialfinder.cfg*.

Bibliography

[All] allmystery. `http://www.allmystery.de/themen/gw59952`. Accessed: 01/08/2012.

[ASP94] Peter J. Angeline, Gregory M. Saunders, and Jordan B. Pollack. An evolutionary algorithm that constructs recurrent neural networks. *IEEE Transactions on Neural Networks*, 5:54–65, 1994.

[BDT99] Eric Bonabeau, Marco Dorigo, and Guy Theraulaz. *Swarm Intelligence: From Natural to Articial Systems*. Oxford University Press, Inc., 1999.

[BHE09] N. Bredeche, E. Haasdijk, and A.E. Eiben. On-line, on-board evolution of robot controllers. In *Proceedings of the 9th international conference on Artificial Evolution (Evolution Artificielle - EA'09)*, 2009.

[Bre04] Nicholas Bredeche. *Contributions to Evolutionary Design of Embodied Agents*. PhD thesis, Univ. Paris-Sud XI, INRIA, CNRS, 2004.

[BTB$^+$07] Gianluca Baldassarre, Vito Trianni, Michael Bonani, Francesco Mondada, Marco Dorigo, and Stefano Nolfi. Self-Organized Coordinated Motion in Groups of Physically Connected Robots. *Systems, Man and Cybernetics, Part B, IEEE Transactions on*, 37(1):224–239, 2007.

[CŁ00] Ernest Czogała and Jacek Łęski. *Fuzzy and neuro-fuzzy intelligent systems.* Heidelberg: Physica-Verlag, 2000.

[CN10] Sarah Chasins and Ivana Ng. Fitness functions in neat-evolved maze solving robots, 2010.

[DBH01] Georg Dorffner, Horst Bischof, and Kurt Hornik, editors. *Artificial Neural Networks - ICANN 2001, International Conference Vienna, Austria, August 21-25, 2001 Proceedings*, volume 2130 of *Lecture Notes in Computer Science*. Springer, 2001.

[DCP$^+$11] Frederick Ducatelle, Gianni A. Di Caro, Carlo Pinciroli, Francesco Mondada, and Luca Maria Gambardella. Communication assisted navigation in robotic swarms: Self-organization and cooperation. In *IROS*, pages 4981–4988. IEEE, 2011.

[Gra] Graphviz - graph visualization software. `http://www.graphviz.org/`. Accessed: 12/15/2011.

[JM97] David S. Johnson and Lyle A. Mcgeoch. *The Traveling Salesman Problem: A Case Study in Local Optimization*. 1997.

[KHE11] Giorgos Karafotias, Evert Haasdijk, and Agoston Endre Eiben. An algorithm for distributed on-line, on-board evolutionary robotics. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, GECCO '11, pages 171–178, New York, NY, USA, 2011. ACM.

[KMEK09] Yohannes Kassahun, Jan Hendrik Metzen, Mark Edgington, and Frank Kirchner. Incremental acquisition of neural structures through evolution. In *Design and Control of Intelligent Robotic Systems*, pages 187–208. 2009.

[KS05a] Y. Kassahun and G. Sommer. Evolution of neural networks through incremental acquisition of neural structures. Technical Report Number 0508, Christian-Albrechts-Universität zu Kiel, Institut für Informatik und Praktische Mathematik, Juni 2005.

[KS05b] Yohannes Kassahun and Gerald Sommer. Efficient reinforcement learning through evolutionary acquisition of neural topologies. In *ESANN'05*, pages 259–266, 2005.

[KSE$^+$07] Yohannes Kassahun, Gerald Sommer, Mark Edgington, Jan Hendrik Metzen, and Frank Kirchner. Common genetic encoding for both direct and indirect encodings of networks. In *In Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2007*, pages 1029–1036. ACM Press, 2007.

[Les07] Arthur M Lesk. *Introduction to Genomics*. Oxford University Press, Oxford, 2007.

[Liv11] D.J. Livingstone. *Artificial Neural Networks: Methods and Applications*. Methods in Molecular Biology. Humana Press, 2011.

[MB11] Jean-Marc Montanier and Nicolas Bredeche. Embedded Evolutionary Robotics: The (1+1)-Restart-Online Adaptation Algorithm. In Springer Series: Studies in Computational Intelligence, editor, *New Horizons in Evolutionary Robotics*, pages 155–169. Springer, 2011.

[Nat] National geographic - "the genius of swarms" by peter miller. `http://ngm.nationalgeographic.com/2007/07/swarms/miller-text/7`. Accessed: 01/06/2012.

[NF00] Stefano Nolfi and Dario Floreano. *Evolutionary Robotics: The Biology, Intelligence, and Technology of Self-Organizing Machines*. MIT Press, Cambridge, MA, USA, 2000.

[Nis97] Volker Nissen. *Einführung in evolutionäre Algorithmen: Optimierung nach dem Vorbild der Evolution*. Vieweg, 1997.

[Pla] Player/stage. `http://playerstage.sourceforge.net/`. Accessed: 01/03/2012.

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

[Rep] Replicator - robotic evolutionary self-programming and self-assembling organisms. `http://www.replicators.eu/tiki-index.php`. Accessed: 01/07/2012.

[SADL12] Florian Schlachter, Patrick Alschbach, Katja Deuschl, and Paul Levi. Evorof - a framework for evolutionary robotics, 2012. to be published.

[SB98] Richard S. Sutton and Andrew G. Barto. Reinforcement learning i: Introduction, 1998.

[Scia] Sciencegarden. `http://www.sciencegarden.de/content/2008-08/die-natur-als-vorbild-fuer-die-wissenschaft`. Accessed: 01/06/2012.

[Scib] Sciencegarden. `http://www.sciencegarden.de/content/2008-08/schwaermen-und-schwaermen-lassen-gemeinsam-ans-ziel`. Accessed: 01/06/2012.

[Sim] Simon Garnier swarm behaviours in natural and artificial systems. `http://www.simongarnier.com/swarm-intelligence-journal/`. Accessed: 01/08/2012.

[SM02] Kenneth O. Stanley and Risto Miikkulainen. Efficient reinforcement learning through evolving neural network topologies. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2002)*, page 9, San Francisco, 2002. Morgan Kaufmann.

[Sta09] Kenneth O. Stanley. *Efficient Evolution of Neural Networks Through Complexification*. PhD thesis, Department of Computer Sciences, The University of Texas at Austin, 2009.

[Sym] Symbrion - symbiotic evolutionary robot organisms. `http://symbrion.org/tiki-index.php`. Accessed: 01/07/2012.

[Tec] Tech-faq. `http://www.tech-faq.com/swarm-intelligence.html`. Accessed: 01/08/2012.

[The] The guardian. `http://www.guardian.co.uk/technology/2009/sep/30/anti-virus-software-ants`. Accessed: 01/08/2012.

[Tok] Milan Paris Tokyo. Raul rojas neural networks a systematic introduction.

[WKF09] Markus Waibel, Laurent Keller, and Dario Floreano. Genetic Team Composition and Level of Selection in the Evolution of Cooperation. *IEEE Transactions on Evolutionary Computation*, 13(3):648–660, 2009.

[Yao99] Xin Yao. Evolving artificial neural networks, 1999.

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.

(Katja Deuschl)