Efficient Neuro-Evolution of Hole-avoidance and Phototaxis for a Swarm-bot

Anders Lyhne Christensen

IRIDIA – Technical Report Series

Technical Report No.
TR/IRIDIA/2005-014

October 2005

IRIDIA – Technical Report Series
ISSN 1781-3794

Published by:

IRIDIA, *Institut de Recherches Interdisciplinaires et de Développements en Intelligence Artificielle*
Université Libre de Bruxelles
Av F. D. Roosevelt 50, CP 194/6
1050 Bruxelles, Belgium

Technical report number TR/IRIDIA/2005-014

Contents

1 Introduction **1**

1.1 Evolutionary Robotics . . . 1

1.2 Goal . . . 3

1.3 Overview . . . 3

1.4 The Swarm-bots Project . . . 3

1.4.1 An S-bot . . . 4

1.4.2 A Swarm-bot . . . 6

1.5 Related Work . . . 6

2 Theoretical Foundations **9**

2.1 Artificial Neural Networks . . . 9

2.1.1 Background . . . 9

2.1.2 Definitions and terminology . . . 10

2.1.3 Artificial Neural Network Types . . . 11

2.1.4 Artificial Neural Networks in Robotics . . . 16

2.2 Evolutionary Computation . . . 17

2.2.1 Searching in Complex Spaces . . . 17

2.2.2 Introduction to Evolutionary Computation . . . 18

2.3 Evolutionary Algorithms, Neural Networks, and Robots . . . 23

2.3.1 Advanced Neuro-Evolution Strategies . . . 23

2.3.2 A Simple Example . . . 25

2.4 Summary . . . 28

3 Building a Simulator **29**

3.1 Major Requirements . . . 30

3.1.1 Existing Simulators . . . 31

3.1.2 Why Yet Another Simulator? . . . 32

3.2 Design . . . 34

3.2.1 The Simulator: TwoDee . . . 34
3.2.2 Evolutions . . . 45
3.3 Putting Everything Together . . . 47
3.4 Benchmarks . . . 48
3.5 Future work . . . 50
3.6 Summary . . . 50

4 Evolving Controllers **51**
4.1 Methodology . . . 51
4.1.1 Interdependence . . . 51
4.1.2 Summary . . . 55
4.2 Experimental Setup . . . 55
4.2.1 The Arena . . . 56
4.2.2 The S-bot's Sensors . . . 56
4.2.3 Running Evolutions . . . 58
4.2.4 Fitness Functions and Notation . . . 58
4.2.5 Baseline Sets . . . 59
4.3 Fitness Function . . . 60
4.3.1 A Simple Fitness Function . . . 60
4.3.2 Escaping a Local Optimum . . . 61
4.3.3 Generality and Transfer of the Controller . . . 62
4.4 Neural Network Type and Structure . . . 68
4.4.1 Results . . . 70
4.5 Neuro-Evolution Method . . . 74
4.5.1 Standard GA vs. $[\mu, \lambda]$ vs. CCGA . . . 75
4.5.2 Incremental Evolution . . . 80
4.5.3 Evolving Neural Arrays . . . 86
4.5.4 Conclusion . . . 88
4.6 Test on Physical Robots . . . 89
4.6.1 Results . . . 90
4.7 Discussion . . . 93

5 Conclusion **95**
5.1 Future Work . . . 96

References **99**

Introduction

CHAPTER 1

This DEA thesis is on the subject of evolving controllers for cooperative mobile robots. We evolve controllers for a specific task, namely hole-avoidance and phototaxis for a group of physically connected robots each with limited sensing capabilities. Aside from the task itself we are concerned with evolving capable controllers in an efficient manner. Therefore, we explore and compare a number of different neuro-evolution techniques and outline a methodology for evolving robot controllers. In the remainder of this chapter we discuss some of the advantages and disadvantages of evolutionary robotics, we introduce our reference hardware platform, the s-bot, and we discuss some of the work previously conducted on this platform.

1.1 Evolutionary Robotics

Robotics, evolutionary algorithms, and neural networks are each well-established research fields in their own respect. "Evolutionary robotics" is the name frequently used for research in the hybrid field combining the three. In this field the goal is to use evolutionary computation to evolve robot controllers based on artificial neural networks, thus a nature-inspired approach for developing controllers for autonomous robots. One of the major advantages of this approach is that artificial evolution can find solutions that a human developer would not have considered or predicted. When a robot is situated in an environment, solutions found by evolution can exploit features in the environment *as they are perceived through the robot's sensors* to solve a given task, whereas a human developer to some degree will be limited by his or her understanding of the task and possible solutions [Nolfi and Floreano, 2000]. Therefore, an evolutionary approach can find simpler, more robust, general, and scalable solutions to novel tasks compared to hand-written controllers. Moreover, researchers and implementers are freed from writing controllers by hand and can let evolution find good solutions.

From the paragraph above it could seem that evolutionary robotics is a universal method for developing controllers for autonomous robots and alike. However, there is at least one major issue with this approach, namely that designing, implementing, and testing various evolutionary setups, neural network structures, and so on is a tedious and time-consuming process. Evolving controllers involves a great deal of trial-and-error and only controllers for relatively simple tasks have been evolved so far. This can be due to problems related to evolution bootstrapping, fitness function development, and of course running the evolutions themselves. In some cases, evolution can be performed directly on physical robots, [Krohling et al., 2002], [Nolfi et al., 1994], but often evolutions are run in software simulators, because of the speed, no risk of damaging hardware, and the ease of changing setups (e.g. the environment, the number of robots and so on). Even if evolutions are run in software, they can sometimes become complex to the degree where it takes several days and weeks before results are obtained, even when run on mid-range PC clusters. The complexity arises from the need of simulating the dynamics of a virtual world to a fidelity and with an accuracy such that the inputs and the outputs of the robot(s) are sufficiently true to the real world and the robot hardware to allow for the evolved controllers to be transferred to real robots.

Finding a suitable fitness function that asserts the correct evolutionary pressure for all but the simplest tasks can be a non-trivial effort. For instance, assume that we require a robot to perform hole-avoidance while moving around in an enclosed arena that contains holes; our first idea on a fitness function could be that we simply penalize individuals who fall into holes. Such a strategy is likely to fail, however, because a robot which does not move at all does not fall into any holes. Thus, an immobile robot maximizes the fitness function but does not perform the task that we had in mind. A revised fitness function could contain a component related to movement of the robot. Thus, the more a robot moves the higher the fitness it receives, while still being penalized for falling into holes. The amount of movement of an individual can be computed easily by taking the sum of the distances covered from one control cycle to the next. The revised fitness function expresses our objectives: Move and do not fall into holes. However, one likely outcome of an evolution could be that the best individuals move around in circles, thereby minimizing their chances of falling into holes, while moving at full speed, thereby maximizing the fitness function. The value of the fitness component related to movement for such an individual would be close to maximum since the path covered between individual control cycles is almost linear (assuming that the control frequency is sufficiently high and that the diameter of the circular path is sufficiently large). As this example shows, it can take some experimentation before our ideas expressed in natural language can be translated into a fitness function that asserts the evolutionary pressure we had in mind.

Artificial neural networks come in many flavors and they have some desirable properties, namely that they are universal approximators and versatile. Basically, any function can be approximated with arbitrary precision by some neural network. Additionally, neural networks can be extended with memory by introducing recurrent connections. However, all of this power comes at a price: For a given task there is no way of analytically determining the best or even a good neural network, and often a suitable one must be found experimentally.

In the event that a good neural network is found and that weights are evolved under a fitness function that shapes the behavior as we expected, we might want to add more sensors, actuators or change some other aspect of the robot, the environment or the task. In that case we could be in a situation where we would practically have to start from scratch - with a different artificial neural network, a different fitness function and/or a different evolution strategy.

Given the time-consuming nature of running simulations and given the amount of trial-and-error involved in finding a suitable fitness function as well as a good artificial neural network for the task at hand, evolutionary robotics can hardly be considered a universal method for developing controllers for autonomous robots. Taking these virtues into account one could argue that evolutionary robotics is more art than science. Regardless of whether we argue for or against this view, it is clear that a methodical approach to evolving robot controllers, if such can be developed, would be beneficial, since less time would be spent on trial-and-error and more on science.

In this study we will pay attention not only to the results obtained by the evolved controllers but also to the process as a whole. We suggest and discuss a methodology for evolving robot controllers and we explore the performance of various neuro-evolution strategies, such as genetic algorithms [Goldberg, 1989], [Goldberg, 2002], [Mitchell, 1996], cooperative coevolutionary genetic algorithms [Potter and De Jong, 1994], [Potter and De Jong, 2000], $[\mu, \lambda]$ evolutionary algorithms [Schwefel, 1995], incremental evolution [Harvey et al., 1994], [Nolfi et al., 1994], [Gomez and Miikkulainen, 1997], and evolving neural arrays [Corbalán and Lanzarini, 2003]. It is not our ambition to flesh out a complete recipe for evolving robot controllers based on artificial neural networks. Some - possibly major - parts of evolutionary robotics are likely to always depend on a great deal of experimentation. On the contrary, our ambition is to take the first small steps on the road leading to a more structured and methodical approach for evolving controllers for autonomous robots. In order to move in the direction of a structured approach to evolving controllers, we have chosen a non-trivial task, which a swarm-bot should perform. A description of the task can be found in the following section.

1.2 Goal

The goal of this study is to develop robot controllers based on artificial neural networks using evolutionary methods. The controllers should enable a swarm-bot to perform phototaxis while avoiding holes. Moreover, different approaches for evolving such controllers are investigated. This includes designing, running, and testing different evolutionary setups, different artificial neural network structures, and different neuro-evolution methods. A swarm-bot consisting of multiple s-bots is used as the hardware platform. The outcome of this project should be a controller working on real robots and an outline of a general methodology for evolving controllers for robots.

1.3 Overview

This DEA thesis is organized as follows: In the following section we provide an overview of the Swarm-bots project with focus on the robots hardware platform and its capabilities. This is followed by a section on related research and studies done within the context of the Swarm-bots project. In Chapter 2 we provide a brief introduction to artificial neural networks, evolutionary computation, and evolutionary robotics. An overview of the custom-built software simulator is provided in Chapter 3. In Chapter 4 we suggest a methodology for evolving robot controllers and apply it to obtain controllers for our main task, namely phototaxis and hole-avoidance as described in Section 1.2. Finally, a summary, concluding remarks, and future directions are found in Chapter 5.

1.4 The Swarm-bots Project

The main objective of the Swarm-bot project is to study, design, and implement self-organizing and self-assembling artifacts. The project comprises three main activities:

1. Hardware - a number of mobile robots called *s-bots*[1].

2. Simulation - a software simulator called Swarmbots3D has been constructed[2].

3. Control software - the software that controls the robots[3].

The project focuses on studying novel approaches to the design and implementation of self-organizing and self-assembling artifacts. Much of the work within the project has been concerned with the construction and control of a number of small, mobile robots, called *s-bots*. Each s-bots is equipped with a *rigid gripper*, which can be used to physically connect multiple s-bots and/or other objects. In this way a number of s-bots can assemble into a *swarm-bot*. Inspired by insects and other swarming animals various tasks have been studied such as coordinated motion [Trianni et al., 2004b], prey-retrieval [Labella et al., 2004a], cooperative transport [Groß and Dorigo, 2004b], chain formation [Nouyan and Dorigo, 2004], autonomous self-assembly [Trianni et al., 2004c], and adaptive task allocation [Labella et al., 2004b].

Below we present the hardware platform, while in Chapter 3 we take a closer look at simulation issues. For further information on the Swarm-bots project and its objectives, see the following web page: `http://www.swarm-bots.org`.

1.4.1 An S-bot

A photo of an s-bot is shown in Figure 1-1. An s-bot is mobile due to the combined tracks and wheels labeled *Differential Treels* © *Drive*. The treels are attached to the *chassis*. A *turret* containing the majority of the sensors is mounted atop the chassis and can rotate independently up to 360 degrees from one extreme to the other[4]. The rigid gripper is located on the turret as can be seen in Figure 1-1 on the next page. This gripper is quite powerful and enables near-rigid connections between s-bots. The s-bot has a camera located at the lower end of the transparent, plastic tube pointing upwards and a mirror is situated in the upper end of the tube providing the s-bot with omni-directional vision.

Its sensors and actuators, listed in Table 1-1, make the s-bot a very flexible platform for studies on several topics and allow researchers to experiment with different approaches to solving various tasks. However, no single study has yet used all of the capabilities of an s-bot at once (and we do not intend to do so in this work either). Therefore, we will describe only those sensors and actuators used in the appropriate sections later in this DEA thesis, while we refer the reader to one of the following publications by Mondada et al. [2004] and Mondada et al. [2003] available on the Swarm-bots project web page, for a more detailed description of the s-bot hardware.

[1]See Section 1.4.1
[2]See Chapter 3 on page 29
[3]See the publications on the Swarm-Bot web page: `http://www.swarm-bots.org`
[4]It is actually possible to for the turret to rotate slightly beyond 360 degrees with respect to the chassis, however as this increases the chance of damaging the hardware, we restrict ourselves to 360 degrees.

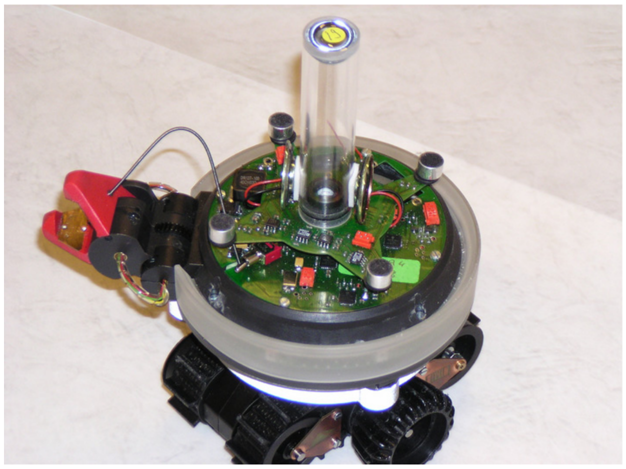


Figure 1-1: An S-bot without the optional flexible arm. The diameter of the chassis is 11.5 cm.

Actuators	**Sensors**
Treels	Camera
Motorized rotation of the turret	Four microphones
Gripper	Traction sensors
Two loud-speakers	Humidity sensor
8x3 colored leds	Temperature sensor
Flexible arm (optional)	Inferred proximity sensors
	Inferred ground sensors
	Accelerometer
	Gripper torque sensor
	Rotate torque sensor
	Treel torque sensor
	Light sensors

Table 1-1: Overview of the sensors and actuators installed on each s-bot.

1.4.2 A Swarm-bot

When a number of autonomous s-bots are physically connect they compose an artifact referred to as a *swarm-bot*. In swarm-bot formation multiple s-bots can cooperate directly on tasks that stand-alone s-bots cannot solve, for instance passing over a gap wider than one robot and transporting heavy objects. In some studies, like this one, the s-bots are in swarm-bot formation from the beginning till the end of an experiment, while in other studies one or more swarm-bots are formed as needed[5]. Figure 1-2 shows a swarm-bot composed of 8 autonomous s-bots.

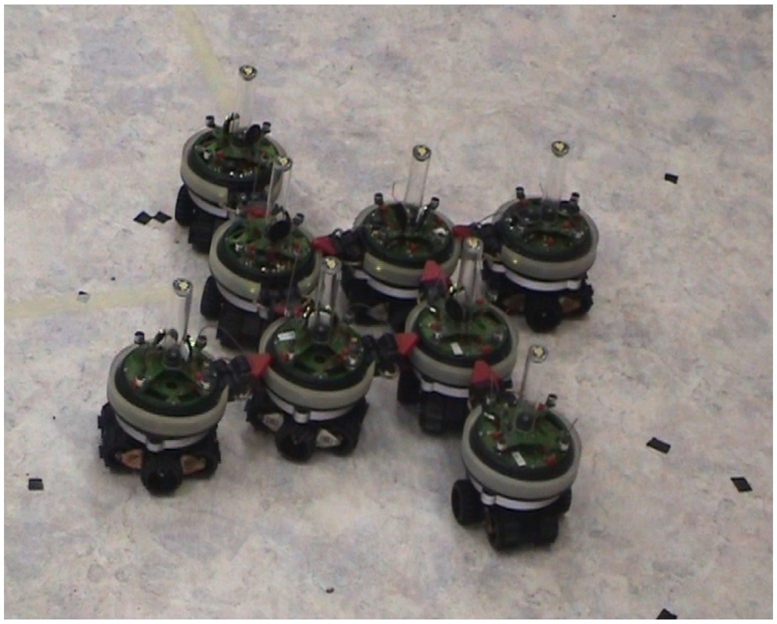


Figure 1-2: A swarm-bot composed of 8 autonomous s-bots.

1.5 Related Work

In this DEA thesis we cover a number of theoretical and practical topics, namely neural networks[6], evolutionary computation[7], simulation and controller transfer[8], and a number of neuro-evolution strategies[9]. Work related specifically to these topics are found in the appropriate sections of the thesis. Below we briefly describe some of the work related to the s-bot platform and work concerned with phototaxis and/or hole-avoidance. A general introduction to evolutionary robotics can be found in [Nolfi and Floreano, 2000].

In the context of the Swarm-bots project a body of research on evolving artificial neural network controllers such as [Baldassarre et al., 2002] and [Trianni et al., 2003] has been done. Moreover,

[5]When s-bots assemble in swarm-bot formation as needed, the s-bots are said to perform *functional self-assembly* [Trianni et al., 2004c].
[6]See Section 2.1.
[7]See Section 2.2.
[8]See Chapter 3.
[9]See Section 2.3.1 and Section 4.5.

specific studies on evolving controllers for coordinated-motion and hole-avoidance has been performed, see for instance [Trianni, 2003] and [Trianni et al., 2005]. The hole avoidance studies performed to date have not included moving towards a specific target such as a light source, but only on coordinated movement while avoiding holes. However, other studies, such as [Groß and Dorigo, 2004a] and [Groß and Dorigo, 2004c], have been concerned with collective transport of an object towards a light source, albeit in obstacle-free environments. Thus, combined photo-taxis and hole-avoidance has not been attempted for swarm-bots prior to the study presented in this thesis.

Miglino et al. [1998] has studied evolution of neural network controllers for a Kephera robot equipped with a directional camera and proximity sensors. The robot should move from its initial location towards a light source in an arena with walls. The light could always be seen by the camera if the robot was facing in direction of the light, even if walls were located in between the light and the robot. The walls were placed in such a way that it was necessary for the robot to take a detour and momentarily loose sight of the target in order to go around them. The authors found that a recurrent neural network solved the task better than a simple perceptron; however, given their relatively simple setup and small arena (55x40cm) it is hard to draw general conclusions based on their experimental results.

Although the study by Miglino et al. [1998] bares some resemblance to the task that we are trying to solve, there are some fundamental differences: We operate with a swarm of robots that have to learn to cooperate on solving the task. Our robots are physically connected and a robot can usually only sense the presence of a hole once it is directly above it. This means that a robot would in many cases fall in if it was not for the other s-bots in the swarm-bot. Moreover, we do not use a directional camera, but instead a number of light sensors which sense light from all directions, and every s-bot can always sense the direction of the light source.

Regarding our secondary objective, namely, studying a methodical approach for engineering robot controllers through artificial evolution, we have been unable to find any related work. The reason for this is likely that evolutionary robotics is a relatively new, hybrid field and that only little is known about the dynamics of evolutionary algorithms applied to situated, embodied robots.

Theoretical Foundations

Chapter 2

In this chapter we give a brief introduction to the theory behind artificial neural networks and evolutionary algorithms as well as to how these techniques are applied in evolutionary robotics. We devote the majority of the chapter to the concepts and methods used in the remainder of the report, therefore this is not meant to be a general introduction to neither of the fields. Readers unfamiliar with these topics are advised to seek introductory material elsewhere, such as [Haykin, 1998], [Fausett, 1994], [Goldberg, 1989] and [Coley, 1997]. Section 2.1 and Section 2.2 on artificial neural networks and evolutionary algorithms, respectively, can safely be skipped by a reader familiar with these topics. Section 2.3 touches on more advanced neuro-evolution techniques such as evolving neural arrays, SANE, SAGA and CCGA.

2.1 Artificial Neural Networks

Artificial neural networks represent an information processing paradigm that is inspired by the way biological nervous systems, such as the brain, process information. The key element of this paradigm is the novel structure and semantic of interconnected nodes or *neurons*. Artificial neural networks is a versatile and robust paradigm for processing noisy inputs, such as those produced by imperfect robot sensors and for agents who navigate in environments that contain elements of randomness and/or uncertainty.

Below we give a brief introduction to the history of artificial neural networks, examples of different network structures, and to the applicability of neural networks in robotics.

2.1.1 Background

The history of neural networks began in 1943 when McCulloch and Pitts published a mathematical model of a neuron [McCulloch and Pitts, 1943]. In their model, a neuron is a linear threshold computing unit with multiple inputs and a single output of either 0, if the nerve cell remains inactive, or 1, if the cell *fires*. While some still use binary neurons, many researchers in robotics today use a more sophisticated firing scheme, where the output is a continuous function of the neuron's inputs. In the late 1950s and early 1960s the first artificial neural networks where implemented on a computer [Rosenblatt, 1959] and [Rosenblatt, 1962]. The computational limitations[1] of the first class of artificial neural networks, *perceptrons*, where not overcome before the 1980s. A new type of artificial neural network was proposed and Hornik et al. [1989] showed that these networks, referred to as *multi-layered perceptrons*, are in fact universal function approximators. After this discovery new network structures and extensions

[1]A perceptron cannot represent functions whose inputs are not linearly separable, this is for instance true for the binary exclusive-or (XOR) function [Minsky and Papert, 1969].

have been suggested and studied and the field has attracted a lot of interest from researchers in various disciplines. As a result artificial neural networks are today used in many different areas such as pattern recognition, control, detection, and prediction.

2.1.2 Definitions and terminology

In this section we first give a formal definition of a neuron and discuss general properties artificial neural networks (ANNs).

Definition 1 - *Neuron*

A neuron, n, a number of inputs, $x_1 \ldots x_i$, and one output y. Associated to each input x_i is a weight w_i, moreover w_0 specifies the weight to a special bias *input $x_0 = 1$.*

The activation, *a_n, of n is given by the following expression:*

$$a_n = \sum_i w_i x_i \tag{2-1}$$

The output, y, of a neuron is computed based on its activation, a_n:

$$y = f(a_n) \tag{2-2}$$

We call f the neuron's activation function *while y is called the neuron's* activity.

□

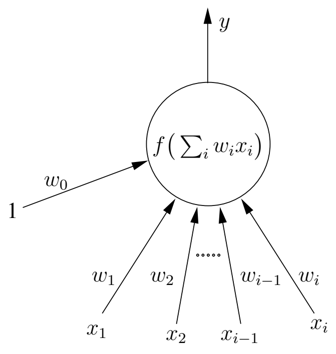


Figure 2-1: A graphical representation of a neuron. A neuron has a number of inputs $x_1, \ldots, x_i$, one special bias input, $x_0 = 1$, and one output, y. The activation of the neuron is the sum of the inputs multiplied by their respective weights.

A graphical representation of a neuron is shown in Figure 2-1. The weight w_0 on the bias input for the neuron and determines the threshold for the activation and allows the neuron to output a value different from $f(0)$ in case all external inputs $x_1, \ldots, x_n$ are all 0.

A neuron's activation function can be freely chosen. However, for a neural network to be able to represent non-linear functions, it is necessary to use non-linear activation functions. Moreover, some learning techniques, like back-propagation, require the activation function to be differentiable. Examples of three popular activation functions are shown in Figure 2-2. The linear and the hyperbolic tangent are classic mathematical functions, while the *sigmoid activation function*, which we use, is given by the following expression:

$$f(a_n) = \frac{1}{1+e^{-a_n}} \tag{2-3}$$

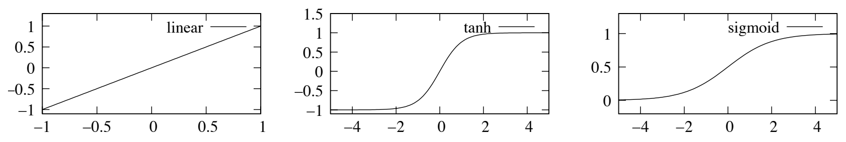


Figure 2-2: Common neuron activation functions.

A neural network consists of one or more connected neurons. Some or all neurons are connected to an external environments through *inputs* and *outputs*. The neurons are connected by letting the outputs of some neurons function as inputs for others. In this way, a neural network forms a directed graph. For what is called *feed-forward networks* such a graph is acyclic, while for *recurrent networks* such a graph contains cycles. Some of the inputs for one neuron can come from the environment and the outputs from some neurons can be output to the environment. For robot controllers based on ANNs the environmental inputs are usually sensory readings, while the outputs control the robot's actuators.

A common, formal definition for all types of artificial neural networks does not exist to our knowledge and we shall not attempt to state one here. We instead present some of the common types of ANNs that we will use in later chapters. The various types of network differ in terms of their structure and with respect their *update rule*, hence the order in which the neurons' activities are computed.

2.1.3 Artificial Neural Network Types

The first type of ANN proposed was called a *perceptron*. Perceptrons belong to the class of ANNs called *feed-forward networks*:

Definition 2 - *Feed-forward Network*

Let the neurons from a neural network be the nodes in a graph, and let every non-zero weight represent a directed edge going from the outputting neuron to the neuron receiving the input. Then a neural network is called a feed-forward network if and only if such a graph is acyclic.

Perceptrons

A perceptron consists of two *layers*: An input layer and an output layer. Each input from the environment is received by one and only one neuron in the input layer. In turn all neurons in the input layer are connected to all the neurons in the output layer. An example of a perceptron is shown in Figure 2-3. The neurons in the input layer of a perceptron are special in the sense that their activities are set externally. In this way each neuron in the input layer works as a separate input to the perceptron.

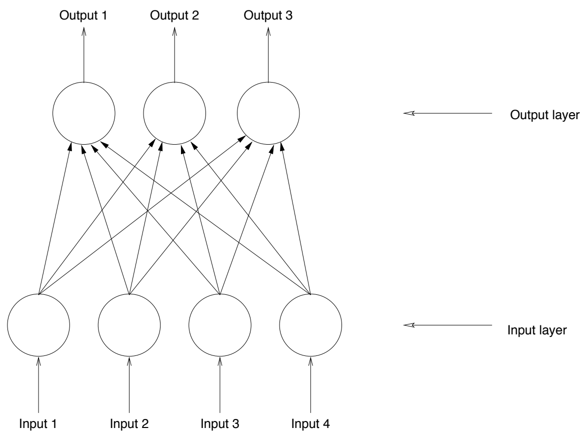


Figure 2-3: An example of a perceptron. The inputs from the environment enter the network through the neurons in the bottom layer: The input layer. The input layer is fully connected to the output layer. There is one neuron in the input layer for each input and likewise there is one output neuron for each of the network's outputs.

The update rule for perceptrons is straight-forward: The activities of the input neurons are set and the activities of the output neurons are computed and output to the environment. Thus, perceptrons are updated in discrete input/output cycles.

Multi-layer Perceptrons

Perceptrons have some basic limitations. For instance, they cannot represent the function exclusive-or (XOR)[2]. These limitations can partly be overcome by adding one or more *hidden layers* of neurons. Each neuron in a hidden layer receives the outputs of the neurons in the previous layer and provides inputs to the neurons in the next layer. Thus, the neurons in a hidden layer do not receive input directly from the environment and their outputs are only connected to other neurons and not to the environment. The neurons in the hidden layer are also called *hidden neurons*. Feed-forward networks with one or more hidden layers are called *multi-layer perceptrons (MLP)* or generally *multi-layer neural networks*. An example of a MLP is shown in Figure 2-4.

Although multilayer networks can have multiple hidden layers it is uncommon to see networks with more than two hidden layers.

[2]More generally, they cannot compute any logical function, which is not linearly separable.

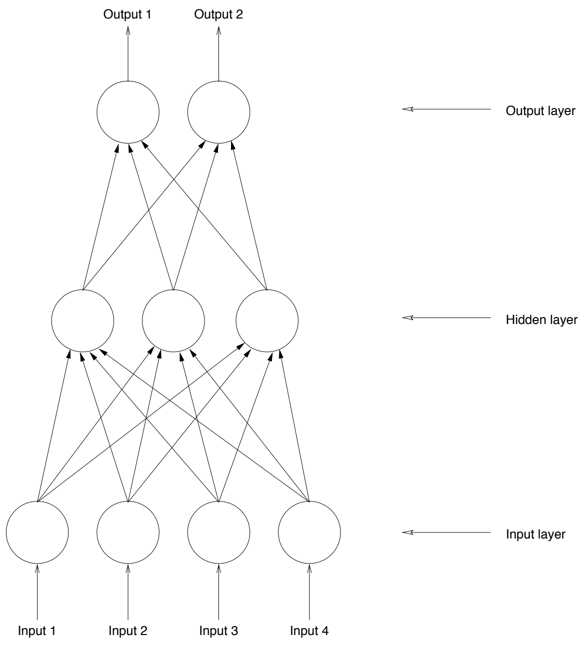


Figure 2-4: An example of a Multilayer Perceptron (MLP). An MLP as a layer of input neurons, one or more layers of hidden neurons, and one layer of output neurons.

Notice that the number of neurons in one layer can be chosen independently of the number of neurons in the other layers. E.g. the number of neurons in the input layer corresponds to the number of sensory readings, the number of output neurons correspond to the actuator controls, while the number of hidden nodes can be chosen arbitrarily[3].

The update rule for MLP is similar to that of perceptrons: The activities of the neurons are computed in a bottom-up fashion, see Figure 2-4.

Recurrent Neural Networks

While single and multilayer perceptrons are sufficient to represent static functions, many interesting problems require *memory*, e.g. to recognize temporally extended patterns in input sequences in for instance speech recognition. Memory is implemented in neural networks by introducing a new type of neuron known as a *context unit* or *memory unit*. A number of extensions to MLPs have been proposed, such as *Jordan networks* and *Elman networks*, see [Jordan, 1990] and [Elman, 1990], respectively. In Jordan networks each output neuron has a corresponding *context unit* to which its output is copied after each evaluation. The value of the

[3]The number of hidden nodes influences the performance of an ANN: A simply network, with only few hidden neurons, might not be powerful enough to represent a given function with sufficient accuracy (it is said to *underfit* the problem), while the reverse is true for a network with too many hidden nodes, hence it can *overfit* a given problem. An additional disadvantage of using a network with too many nodes is that more resources are in general needed to find suitable weights (train the network) for more complex networks given the increase in the number of weights present in a more complex network.

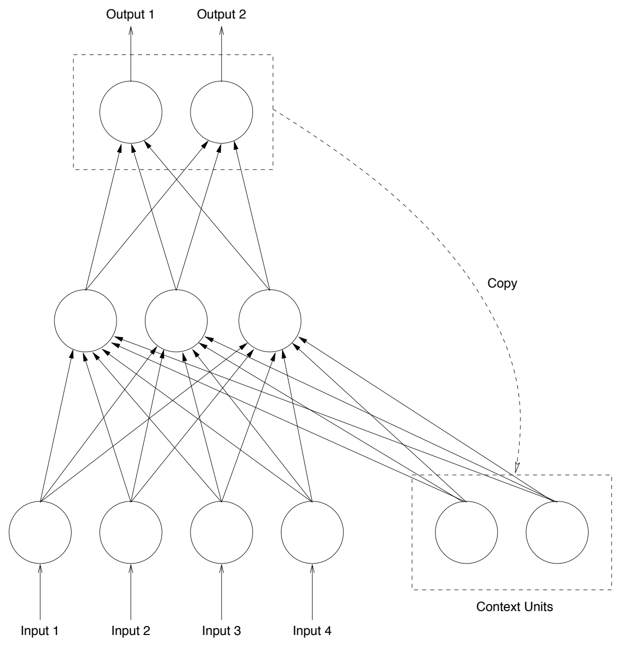


Figure 2-5: An example of a Jordan network. A Jordan network has recurrent connections from the output neurons to each neuron in the hidden layer.

context units is then used as inputs to the hidden layer(s) on the next input/output cycle. In this way context units work as a memory by saving the output state of a neural network across consecutive time steps. Figure 2-5 shows an example of a Jordan network. A layer of context units can even be added for the context units themselves (self-recurrent connections), which allows for finer control over the decay of the network's memory. Without such self-recurrent connections the network rapidly forgets previous states [Jordan, 1990].

Elman networks build on the same concept. However, instead of copying the activity of the output neurons, the hidden layer(s) are copied into a set of context units. An example of an Elman network with one layer of hidden neurons (and therefore one set of context units) is shown in Figure 2-6. According to Elman [1990] the recurrent connections on the hidden layer(s) allow for a network to build internal representations of generalizations when a network is trained.

The update rule for recurrent neural networks is bottom-up as for MLPs. However, the context units are only updated *after* all other neurons have been updated, including output neurons, and they keep their activity between input/output cycles.

Dynamic Neural Networks

So far we have only discussed discrete-time networks; that is, a network receives some input at time step t_i and computes the outputs for that time step. At time step t_{i+1} the network receives a new set of inputs and computes a new set of outputs. In case the input is of a discrete form, for instance a string of letters or a stream of temporally even-spaced samples,

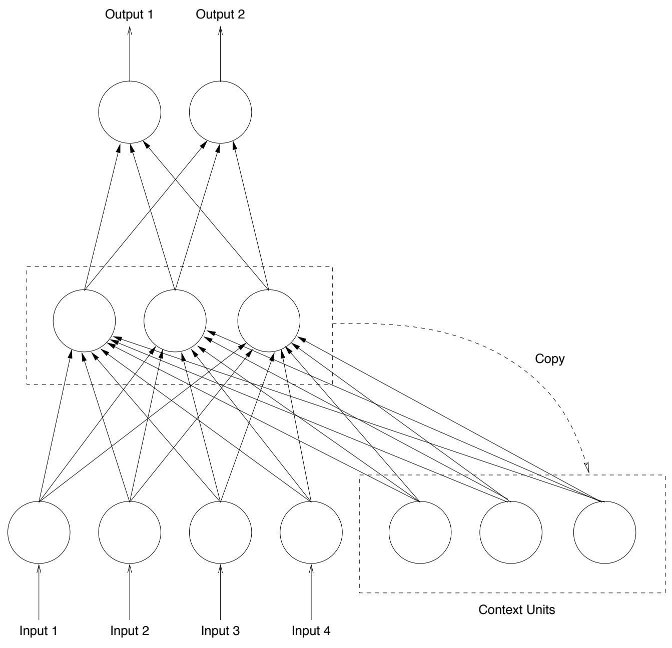


Figure 2-6: An example of an Elman network. An Elman network has recurrent connections from the output neurons to each neuron in the hidden layer.

this model works fine since both the network and the underlying phenomenon are inherently discrete.

Many phenomena are, however, not discrete in their nature and *time* plays a central role. Assume, for instance, that a robot's camera input is processed asynchronously with respect to the decision-making controller and that new input is available at irregular intervals depending on the complexity of an image anywhere between every 20 to 50 milliseconds. If we require the ANN controller of the robot to track a moving object and to act accordingly, we need to take the temporal dimension into account as the output should depend on the time that passes between each input.

By replacing the activation functions in our neurons with time-dependent, differential equations we obtain a continuous-time neural network [Yamauchi and Beer, 1994], [Beer, 1995]. The activity of a neuron is computed in the following way:

$$a_n(0) = 0, \tag{2-4}$$

$$\frac{da_n(t)}{dt} = \frac{1}{\tau_n}\Big(- a_n(t) + \sum_i w_i x_i \Big) \tag{2-5}$$

where $a_n(t)$ is the activation of neuron n at time t, τ_n is a time constant specific to neuron n, w_i and x_i are the weights and inputs, respectively, from other neurons (including the input neurons).

In this manner, perceptrons, multilayer perceptrons, and recurrent neural networks can be extended with continuous time as a parameter. Such networks are known as *dynamic neural* *networks* or *continuous-time neural networks*.

The update rule in a dynamic network is often similar to that of its non-dynamic counterpart - in this case it is called a *synchronous network*. In other cases neurons are updated at different times and the network is then called an *asynchronous network*.

Other Neural Network Types

A number of other ANN structures and types exist, which we have not touched upon here. These include Hopfield nets [Hopfield, 1982], self-organizing maps [Kohonen, 2001], and Long Short-Term Memory networks (LSTM nets) [Hochreiter and Schmidhuber, 1996]. Although it could be interesting to study the applicability of for instance LSTM nets in this work, we consider this as being outside the scope of this study. We therefore resist the temptation of going into further details with these variants.

2.1.4 Artificial Neural Networks in Robotics

ANNs is an interesting paradigm for researchers in the field of autonomous robotics, since robots have to act in a non-deterministic world perceived through noisy sensors and neural networks are in general quite robust against noise. A complete controller can consist simply of a neural network, where the sensor readings correspond to the inputs of the network and the ANN's outputs control the actuators of the robot.

If we want to use ANNs for robot controllers (or for any other problem/task), we have to determine the type of neural network to use, its structure and weights. A great amount of research has been dedicated to this problem, which often involves training sets consisting of a number of inputs and corresponding outputs (supervised learning), or simply a set of inputs (unsupervised learning). These training sets are then applied to various types of neural networks with different structures. The structure more capable of generalizing over the training set(s) with respect to some benchmark is then chosen. However, these popular learning methods do not currently seem applicable in autonomous, mobile robotics, as training sets are hard to construct. The right thing to do when faced with certain sensory inputs is not always obvious, even to humans. Therefore, we treat this as an optimization problem and apply evolutionary algorithms in order to *evolve* suitable ANNs.

2.2 Evolutionary Computation

In this section we first give a definition of what an optimization problem is and discuss various strategies for solving such problems. We then describe the optimization problem which we want to solve - optimizing weights in neural networks controllers - and go into depth with the heuristic which we adopt, namely evolutionary computation, including extensions and common issues.

2.2.1 Searching in Complex Spaces

An instance of an optimization problem can be defined in the following way:

Definition 3 - *Optimization Problem*
Formally, an ***instance of an optimization problem*** *is a pair* (S, f) *where* S *is a generic set of solutions called the* feasibility space *and* $f : S \to \mathbb{R}$ *is a function that associates a* cost *to each solution. The problem is to find a* $\bar{s} \in S$ *for which:*

$$f(\bar{s}) \leq f(s) \qquad \textit{for all } s \in S. \tag{2-6}$$

Such an element $\bar{s}$ *is called an* optimal solution *to the given instance.*[4]

As a shorthand notation, we write:

$$\bar{s} = \arg\min_{s \in S} f(s), \tag{2-7}$$

where with "$\arg\min$*" we denote the element of the set* S *for which the minimum is attained*[5]*.*

□

Optimization problems can be grouped into classes, such as discrete, combinatorial optimization problem (graph, network, scheduling, etc. usually fall in this category), and general unconstrained problems where a global minimum (or maximum) of a non-linear function over reals is sought.

A number of methods have been developed for solving optimization problems. These methods can be divided into two categories, namely *exact methods* and *heuristic strategies*.

Exact methods

Exact methods guarantee to find a global optimum within the feasibility space. If the cost function is differentiable the global optimum can in some cases be determined analytically. Methods relying on this technique are called *calculus-based*. Other exact methods include

[4] *It is customary to formulate optimization problems as* minimization problems. *Clearly, any minimization problem* (S, f) *can be transformed into a maximization problem* (S, g) *by simply considering* $g = -f$ *and reversing the inequality in Equation 2-6. In this case, it is said that the function* g *assigns a* value *to each solution in* S*.*
[5] *Under the assumption that* S *is closed the minimum of* f *on* S *does exists. If more than one element of* S *attain such minimum it is indifferent which one is chosen.*

enumeration, branch and bound, Bayesian search algorithms, successive approximation, and homotopy. Common to all known exact methods is that they are impractical to apply unless either the feasibility space, S, is sufficiently small or that the structure and/or behavior of the cost function, f, is known and can be exploited.

Heuristic strategies

Unlike exact methods, heuristic strategies do not guarantee to find a global optimum of a given cost function, instead they aim at finding *good* solutions for problems to which exact methods are not practically applicable. Heuristic methods have proven successful for many theoretical and real-world problems. Like some of the exact methods, such as branch and bound and Bayesian search, heuristics often have to be adapted to a particular class of optimization problems in order to perform well. Heuristic strategies include methods such as simulated annealing, tabu-search, ant-colony optimization, and evolutionary computation.

Optimizing ANN weights

The optimization problem that we are faced is to find good sets of weights for artificial neural networks. Although we will evaluate different ANN types and structures, we will not consider the number and types of neurons and connections as parameters of our optimization problem and our focus is on optimizing the weights in ANNs. Thus, in this study we want to optimize the weights of a neural network for s-bots which move around in a virtual and real world. The value cost function for some ANN reflects the performance of a group of s-bots controlled by the ANN in a virtual world with respect to some performance criteria, e.g. how close do the s-bots get to the target area.

The number of different states[6] of the virtual world is practically *infinite* and partly *non-deterministic* since noise is added to sensors and actuators. Moreover, the virtual world, and therefore also the cost function, lacks a well-defined structure, which can be exploited by any known exact method. The problem is of relatively high dimensionality, that is one dimension for each weight we want to optimize, and the size of the feasibility space therefore makes the use of exact methods impractical.

Of the available heuristic strategies we have chosen evolutionary computation since extensive research has been done on applying this method to the evolution of artificial neural networks. In Section 2.3 we discuss some of this research in more detail, while in the next section we give an overview of evolutionary computation.

2.2.2 Introduction to Evolutionary Computation

The concept behind evolutionary computation is inspired by natural selection. Natural selection prescribes that the fitter insects, plants, animals and humans have better chances of reproducing compared to lesser fit specimens. Thus, fitter specimens rear more offspring than less fit

[6]A state of the virtual world is defined by the positions, orientations, momentums, and configurations of the s-bots, and their sensors and their actuators. Since the number of different possibilities for these parameters is infinite for all practical purposes, we consider the number of states infinite.

specimens. When individuals reproduce new combinations of their genetic material are created, which may lead to new blending of good genes, which results in even fitter individuals. Moreover, genetic mutations in individuals can lead to improvements or regressions that in turn result in higher and lower chances of reproduction, respectively. In this way species undergo constant changes where the fittest members are favored by Nature.

Evolutionary computation is built around this idea and they are to various degrees inspired by the underlying biological mechanisms governing reproduction. The basic concept behind evolutionary computation can be stated as follows:

> *Let a population of individuals solve a given problem and let those individuals who solve the task best form the basis for a new population. Let the process continue until some termination criterion is reached.*

The termination criterion can for instance be that no improvement has been seen for a number of successive populations or *generations*, a solution of a sufficient high quality is reached, or simply that a certain number of generations have been evaluated.

Evolutionary methods often rely on some compact representation of an individual, inspired by the role and function of DNA in Nature. Each set of chromosomes, present in every cell of the every living organism, is a compact blue-print for the entire being. In Nature the information contained on a chromosome is written in a four-letter base-pair alphabet: $\{A, T, G, C\}$. In evolutionary computation a different encoding scheme is often chosen for the equivalent of chromosomes, such as the alphabet $\{0, 1\}$. A string of 0s and 1s can serve as a compact representation of an "individual", i.e. a candidate solution to a particular problem such as a set of neural network weights for a robot controller. In the same way as a gene is a section of consecutive elements on the double-helix DNA molecule in living organisms, a set of consecutive letters on our virtual chromosome code for a *feature* of the solution. Thus, our string or virtual chromosome works very much like DNA in humans, animals and insects: The representation is general, but the decoding of a particular section is problem-dependent (specie-dependent). A feature consisting of a single element in our string could for instance code for the presence or absence of an edge in a graph, the next feature on the string could consist of 16 elements (or bits) and could be interpreted as a floating-point weight on the edge.

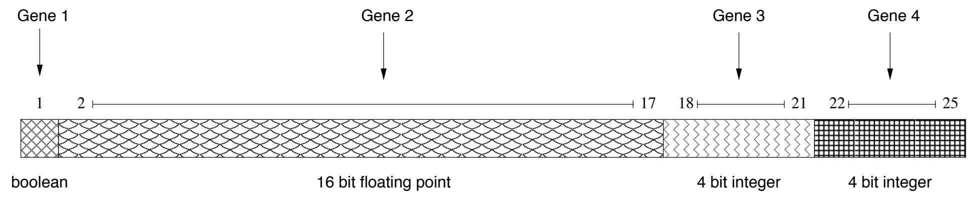


Figure 2-7: An example of a *string* over the alphabet $\{0, 1\}$. In this example the string has a length of 25 and encodes four genes or *features*: One boolean (true/false), one floating point value, and two integers.

An example of a string is depicted in Figure 2-7. In this example the string over the alphabet $\{0, 1\}$ has a length of 25 elements, which encodes four features: One boolean (true/false) one floating point value, and two integers. For another optimization problem we could have chosen a different length for the string and another decoding. However, once we have a string of the correct length we can decode and evaluate it for a particular problem. For instance, we can evaluate the length of the tour it describes if we are optimizing an instance of the traveling salesman problem, or in our case, we can see how well a robot performs a certain task when the string is interpreted as a set of weights for a neural network controller. In this way we can assign a *fitness value* (or simply a *fitness*) to each string[7].

Once all strings in a population have been evaluated, we know how the solution encoded in each string performed and we can select the best strings for reproduction. Several selection schemes exist, for instance roulette wheel selection where each individual has a chance proportional to its fitness of getting selected for reproduction. A random number generator "spins the wheel". For each string in the new population two parent strings, P_1 and P_2, are selected and their strings are combined through the operation known as *crossover*: A position, x, between two elements on one parent string is randomly chosen and the new string, C, is composed of elements $e_1 \dots e_x$ from P_1 while elements $e_{x+1} \dots e_l$ are copied from P_2. In this way C is a combination of solutions P_1 and P_2. Besides crossover, we introduce *mutation* so that each element of in C has a small probability of being randomly selected from alphabet instead of one of the parent strings. Mutation ensures that new solutions are explored from time to time.

When enough individuals have been created a new generation is available for evaluation. To summarize this type of evolutionary computation, know as a *genetic algorithm* (GA), the pseudo-code for a general GA is shown below:

```
1.  Create the initial population* $\mathcal{P}$
2.  Evaluate($\mathcal{P}$)
3.  while (termination criterion not met)
4.      $\mathcal{P}' = \varnothing$
5.      for each individual to be created
6.          $P_1, P_2$ = SelectParents($\mathcal{P}$)
7.          $C$ = Crossover($P_1$, $P_2$)
8.          Mutate($C$)
9.          $\mathcal{P}' \leftarrow C$
10.     endfor
11.     $\mathcal{P} = \mathcal{P}'$
12.     Evaluate($\mathcal{P}$)
13. endwhile
```

*The initial population is usually randomly generated.

[7]In the case of the travelling salesman problem each candidate solution can easily be assigned a numerical value corresponding to the quality of that solution; the shorter, the better. In autonomous robotics, on the other hand, it is not always straight-forward to assign such values to patterns of behavior as we shall see in later chapters.

The pseudo-code shown above illustrates the basic concept behind GAs. The basic principle behind GAs is fairly simple. Due to this simplicity and due to GAs generality, they are widely used for combinatorial and general optimization problems alike in both industry and research. Many variations and extensions have been proposed, such as: Diploidy, inversion and reordering strategies, multi-objective evolution, elitism, etc. [Goldberg, 1989], [Goldberg, 2002], [Mitchell, 1996], [Michalewicz, 1999], [Vose, 1999]. Furthermore, for some problems more elaborate feature encoding schemes and variable length strings are used, for instance for problems where graph structures are evolved. In such cases, the crossover and mutation operators often have to be modified to ensure meaningful offspring. For instance, the crossover operator could select sub-graphs from each parent and join them in a problem-specific manner, while the mutation operator could add or remove edges and nodes.

Fitness Landscapes

The *fitness landscape* is a metaphorical term used to describe the appearance of the landscape defined by the fitness function when function values are interpreted as "heights". Such a landscape often has several valleys and peaks. An example of a fitness landscape for a one-dimensional fitness function is shown in Figure 2-8. If we start with a random population, the starting points will be spread more or less evenly in the fitness landscape. A GA will start moving the strings in successive generations uphill towards peaks by continuously favoring individuals placed higher in the fitness landscape. For example, in Figure 2-8 individuals in the valley D are not as likely to be selected for reproduction as individuals on the hill towards C or E. Because of crossover and mutation there is a chance that one or more strings are created that are further up the hill than in the previous generation.

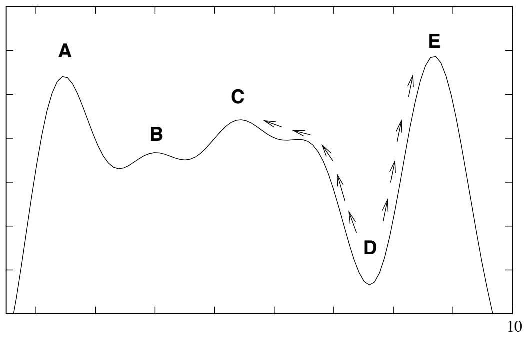


Figure 2-8: An example of a fitness landscape with several peaks and valleys. A population starting around D is likely to start *climbing* towards C and/or E.

It is not uncommon for a fitness landscape to have multiple peaks. This phenomenon poses a common risk associated with the use of GAs and heuristics in general; namely, the chance of finding a local optimum from which evolution cannot escape. In Figure 2-8 the peak E is the global optimum, but since the section of the fitness landscape around A, B, and C in general has a relatively high fitness, there is a chance that a GA would start climbing the wrong hill. Even though crossover, mutation, and the selection scheme all contribute to population diversity and ensure that new solutions are explored, there is no general method for avoiding getting stuck in a local and not global optimum.

Another problem can arise in case the fitness landscape has few and/or narrow peaks. In this case it can be difficult for a GA to find a hill to start climbing. This is known as the *bootstrapping problem*. The fitness landscape in Figure 2-9 illustrates this problem. If none of the strings in the initial population are on a hill leading to a peak, all strings will receive roughly the same fitness, and it is therefore impossible for a GA to tell which individuals are closer to a good solution. The result is that in these cases the GA works as an inefficient random search.

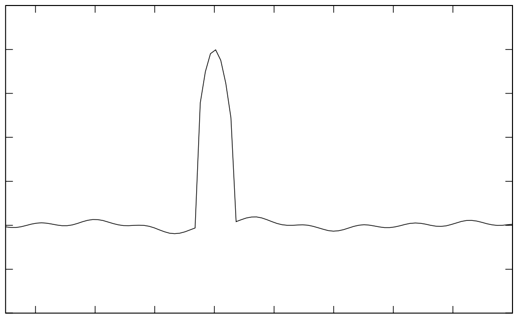


Figure 2-9: An example of a fitness landscape with only one peak. Bootstrapping (getting a GA to move towards better solutions) can be a problem in this case since the area surrounding the peak in the fitness landscape is more or less flat.

There are a number of problem-dependent strategies for overcoming the bootstrapping problem. We return to these in later chapters.

2.3 Evolutionary Algorithms, Neural Networks, and Robots

Neural networks and evolutionary computation can be combined in to evolve controllers for autonomous robots. In some cases both the structure, activation functions and weights are evolved by evolutionary algorithms. We first present four recently proposed, advanced neuro-evolution strategies, followed by a simple example in which neural networks and genetic algorithms are used in conjunction.

2.3.1 Advanced Neuro-Evolution Strategies

In the previous section we saw how an evolutionary algorithm could relative easily find a good set of weights for the perceptron, allowing a robot to find and move towards a light source. For a simple task and a simple ANN, a straight-forward approach can be sufficient. For more complex tasks it is unfortunately not always so. Issues related to bootstrapping and premature convergence to local optima combined with the amount of processing required to evaluate each individual have sparked research in more efficient strategies for evolving robot controllers based on ANNs. Below we list four such strategies:

SAGA: Species Adaptation Genetic Algorithms

SAGA is an extension of classic genetic algorithm, where the dimensionality of the search space is under evolutionary control [Harvey, 1992]. The fundamental principle behind SAGA is to let an evolutionary algorithm optimize not only weights but also structure and size of ANNs. This is done by letting variable length strings encode the morphology as well as weights of networks.

SAGA is an incremental evolution strategy, where the first generations consist of relatively simple ANNs. It is up to the evolution to produce increasingly more complex networks if this is useful to solve the task in question. The incremental strategy can help solve the bootstrapping problem because the search space is relatively limited during the initial generations and only grows when and if evolution finds an advantageous way of doing so.

In [Harvey et al., 1992] the authors apply SAGA to the problem of robot navigation in an environment containing obstacles. In this case each variable length string has two fixed sections; one for the input neurons and one for the output neurons, while the section of the string corresponding to the hidden nodes is of variable length. The authors use non-layered, continuous-time recurrent neural networks, with both excitation and inhibitor links and where noise is added to the activation of each neuron. Although these ANNs are rather complex, the SAGA approach could easily be adapted for simpler types of networks, such as those introduced in Section 2.1.3 on page 11.

SANE: Symbiotic, Adaptive Neuro-Evolution

So far we have only discussed genetic algorithms where each string defines an entire ANN. A method known as Symbiotic, Adaptive Neuro-Evolution (SANE) takes a different approach, namely that each population member represents only a partial solution [Moriarty and Miikkulainen, 1996]. In SANE each string corresponds to *a single neuron* in a MLP, that is, each string defines a set of connections and weights from the input neurons and to the output neurons.

Fitness evaluation takes place by constructing a MLP where the hidden layer neurons and their connections to the output layer are defined by a number of strings selected from the current population. In this way multiple strings are evaluated at the same time and they are therefore rewarded for their generality. Each string participates in a number of different networks constructed in this manner and its fitness is the average performance of these networks.

Since a MLP with only a single type of hidden neuron seldom is sufficient to solve a particular task, SANE results in populations where several specialized sub-groups emerge. In this way SANE results in implicit niching and is therefore not as prone to premature convergence as classic neuro-evolution strategies. In Moriarty and Miikkulainen [1996] the authors show that for some problems SANE can find better solutions faster than other popular reinforcement learning[8] strategies.

CCGA: Cooperative Coevolutionary Genetic Algorithms

Potter and De Jong [1994] and Potter and De Jong [2000] have suggested a symbiotic evolution strategy much like SANE described above. However, instead of letting specialized subgroups form implicitly, CCGA evolves specialized neurons on a set of islands, whose members are not interbred with members from other islands. The rationale is that this allows for faster evolution, than with SANE, because hapharzards - destructive recombinations between members of different specializations - are avoided. The disadvantage is that the number of islands and therefore specializations is decided a priori, whereas in SANE implicit subpopulations are formed by the evolution. Potter and De Jong [1995] have suggested a method for adjusting the number of specializations based on changes in fitness: If evolution stagnates and there is no improvement in fitness over a certain number of generations, a new island (and thus a new hidden node) is introduced.

Given that both CCGA and SANE take a divide-and-conquer approach to neuro-evolution, there are hopes that these methods will allow for evolution of networks for previously intractably complex tasks.

ENA: Evolving Neural Arrays

Evolving neural arrays were proposed by Corbalán and Lanzarini [2003] as a novel mechanism for learning complex actions sequences. The principle behind Evolving Neural Arrays is quite simple: Instead of having one complex neural network to solve a task, a number of MLPs are used to solve various sub-tasks. Hence, ENA is a divide-and-conquer approach on a different level than SANE and CCGA.

A set of ANNs are organized in an ordered list called a network array: $NA = (\mathcal{N}_n, \mathcal{N}_{n-1}, \ldots \mathcal{N}_1)$. Every $\mathcal{N}_i$ has the same input and the same output neurons. One output neuron of each ANN is

[8]"Reinforcement learning is learning what to do–how to map situations to actions–so as to maximize a numerical reward signal. The learner is not told which actions to take, as in most forms of machine learning, but instead must discover which actions yield the most reward by trying them. In the most interesting and challenging cases, actions may affect not only the immediate reward but also the next situation and, through that, all subsequent rewards. These two characteristics–trial-and-error search and delayed reward–are the two most important distinguishing features of reinforcement learning." [Sutton and Barto, 1998].

special: It represents the networks confidence in handling the current situation defined by the inputs. The ANNs in the network array are evaluated in a left-to-right order until a network, $\mathcal{N}_i$, outputs a confidence above some threshold and $\mathcal{N}_i$'s other outputs are applied to robots actuators. In case the right-most ANN is the list, $\mathcal{N}_1$, is reached its outputs are used. Thus, the $\mathcal{N}_1$ defines the default behavior if no other network display a confidence high enough to handle the current situation.

The evolutionary algorithm used with ENA operates in a staged manner, where some task, t, is split into multiple sub-tasks, $t_1 \ldots t_l$. Each stage S_i scores individuals based on their ability to solve the task t' comprised of $t_1, \ldots, t_i$. At stage S_i there are exactly i ANNs in the network array. When evolution moves from one stage to the next, a new network is inserted as the first ANN in the network array, and only the newly inserted network is evolved. The insertion order is then the inverse of the evaluation order, and a newly inserted ANN must "learn" to delegate previously learnt tasks to the other networks in the array.

The authors have compared the ENA method to SANE and show some impressive results in [Corbalán and Lanzarini, 2003], where they evolve robot controllers for obstacle avoidance and target reaching in a maze-like environment. However, unlike SANE, ENA is an incremental evolution strategy that requires a task to be broken into sub-tasks, which makes the technique problem-dependent and therefore difficult to generalize.

2.3.2 A Simple Example

Assume that we have a robot with eight light-sensors pointing in evenly distributed directions in the horizontal plane and two sets of wheels, one on the left-hand side and one on the right-hand side of the robot, respectively, and that we want to evolve a controller that moves the robot towards a light source[9]. If we want to evolve a neural network controller for this task, we might convince ourselves that a perceptron is sufficient to solve the task. The sensor readings correspond to perceptron's inputs while the perceptron's two outputs control the wheels of the robot, see Figure 2-10.

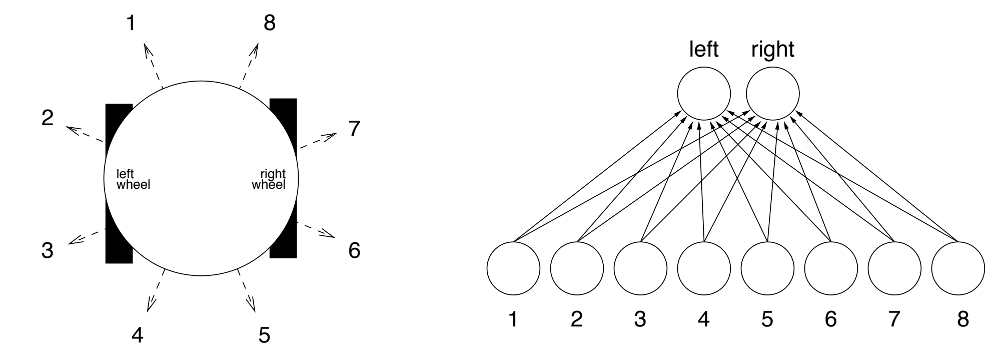
In this case the perceptron has 18 weights to optimize[10]. When a string is evaluated, we translate it into the 18 weights and let a robot move around in a virtual arena with a light source. We keep track of the shortest distance the robot reaches within a certain amount of time, t, and let the fitness:

$$fitness(s) = \frac{D_{start} - D_{shortest}}{D_{start}} \tag{2-8}$$

where D_{start} is the initial distance from the robot to the light source, and where $D_{shortest}$ is the shortest distance between the robot and the light source recorded during t, see Figure 2-11.

[9] Hand-coding a controller for this simple task is likely to be both a lot easier and faster than evolving the desired behavior, but here we use it as a simple example.

[10] There are 18 weights since each input is connected to both outputs (a total of 16 connections) plus both of the output neurons have a bias weight (+2). The input nodes do not have any incoming weights as their activity is set directly by the environment (scaled sensor values).



Robot seen from above with the directions of the 8 light sensors shown

The perceptron controller. Each of the light sensors is connected to the two wheels

Figure 2-10: A robot with 8 light sensors, two wheels, and a perceptron controller.

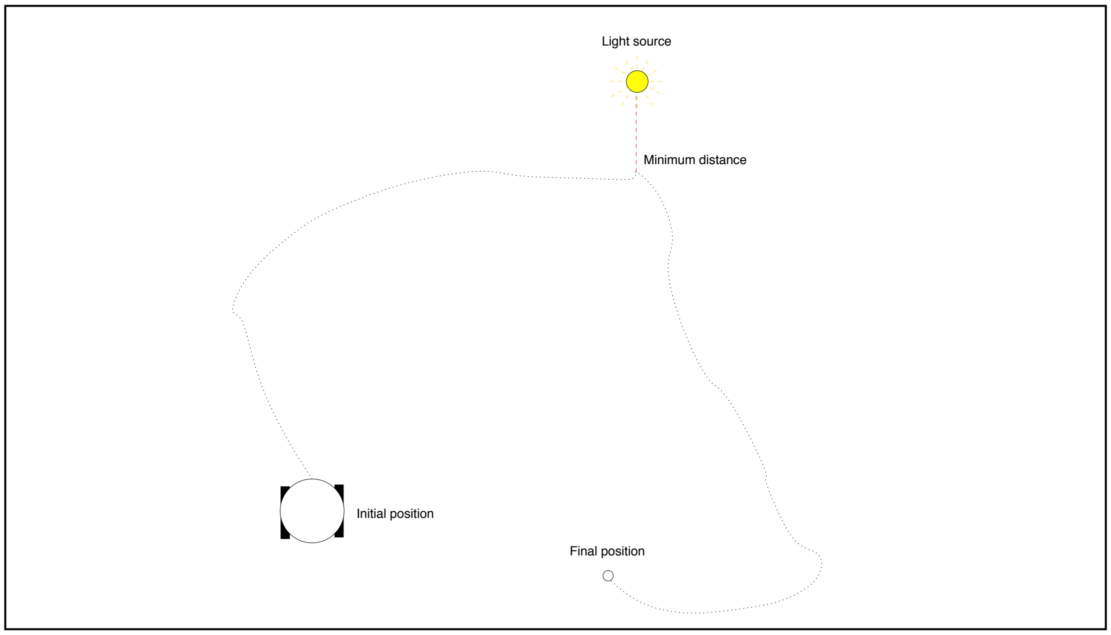


Figure 2-11: The robot starts in an initial position and then moves around the arena. The fitness of the robot is computed based on the point of path closest to the light source. The evolution was performed in a software simulator and the arena was obstacle-free and infinite.

This is repeated for a number of times with the robot initially oriented in different directions with respect to the light source. The reason for performing multiple samples is that we want to evolve controllers which are general, e.g. can move towards a light source no matter where the light is coming from with respect to the robot. If we only perform fitness evaluations where robots initial orientation and position are fixed, we are likely to evolve a controller that is specialized for that particular situation, i.e. perceive the light source on sensor 7 and 8, turn-right and then move forward, without being able to use of any other sensory readings for completing the task.

The results of one evolution are shown in Figure 2-12. The figure shows the fitness of the best string of each generation. As can be seen, the fitness reaches peaks at generation 2, 5, and 9 before reaching the maximum at generation 18 and beyond[11]. These peaks are likely due to the randomization of the initial position and orientation of the robot, which can give some individuals in early generation a high fitness by chance[12].

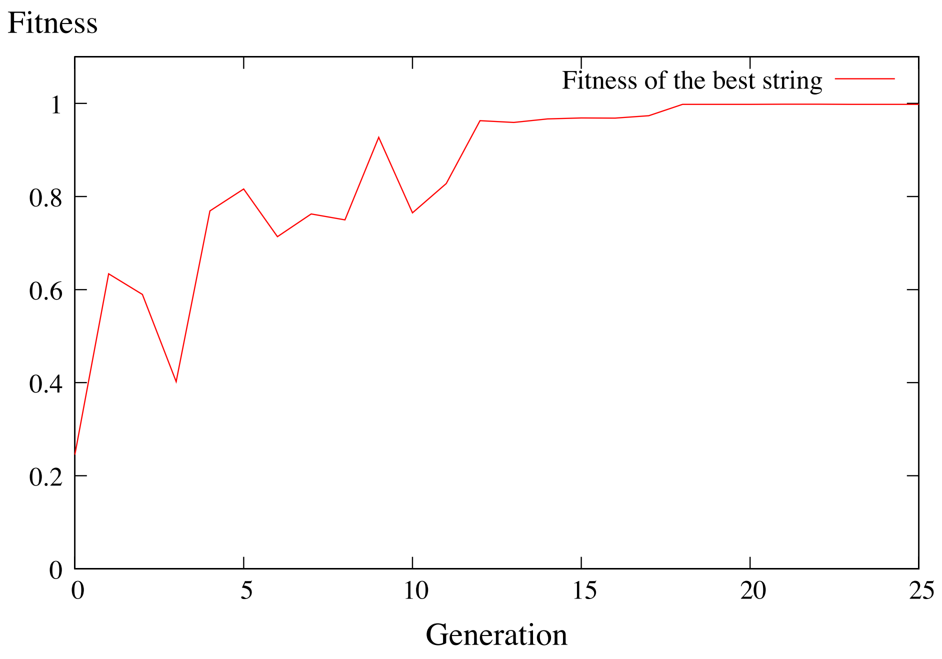


Figure 2-12: The fitness of the best individual in each generation.

These uncertainties combined with the issues mentioned in the previous section regarding the GA getting stuck in local optima and bootstrapping means that evolutions often have to be run a number of times with different initial populations before any conclusions can be drawn.

[11]We let the evolution continue till the 1000th generation and we only recorded a couple of drops in the fitness of the best individual from generation 551 to 559, after that the fitness stabilized around 0.99 again.

[12]If a robot simply moves relatively straight there is 50% chance that its trajectory takes it closer to the light source than its initial position. Furthermore, if it by chance starts facing (or facing away from) the light source and moves forward (backward) the high fitness it receives does not reflect the way it solves the task in general, but simply a coincidence.

2.4 Summary

In this chapter we introduced various types of artificial neural networks, a nature-inspired information processing paradigm, in which several artificial *neurons* or nodes are connected to each other and to the environment from and to which input and output is provided. We saw how another nature-inspired approach, namely evolutionary computation, can be used to find solutions to optimization problems, such as the problem of assigning suitable weights to edges between neurons in artificial neural networks for a given task. Finally, we discussed some recently developed methods for combining neural networks and genetic algorithms, relying on divide-and-conquer strategies, evolution of weights as well as morphologies, and the use of entire arrays of neural networks in robot controllers simultaneously.

Now that we have discussed the theoretical foundations for our study we will move on to some of the practical issues related to evolutionary robotics. In the next chapter we will present a custom-built simulator that allows us to perform rapid prototyping of ideas and to test various evolutionary setups.

Building a Simulator

Chapter 3

Artificial evolution of controllers is often done in simulation for practical reasons. It is not uncommon that thousands of strings (or individuals) have to be evaluated before a good solution is found and doing so in software can be orders of magnitude faster than on real robots. Furthermore, there is no risk of damaging hardware, no need to re-calibrate sensors and actuators, and robots do not run out of battery[1] in simulation.

Evolving controllers in software simulators is not a perfect solution, however, since transferring controllers, which have been evolved in software simulators, successfully to physical robots is a non-trivial effort. An evolved controller might rely on subtle cues and symmetries present in a simulated environment, which are slightly different in the real world, real sensors and actuators are not ideal, meaning that they are not always as accurate as one could hope for, and finally the complex dynamics of the real world cannot be simulated perfectly. All these aspects have to be taken into account if the controllers evolved in software are meant to be used on physical robots. Unfortunately, there is no general method to ensure that controllers evolved in software simulation are transferable to physical robots. An obvious approach is to try to make the differences between reality and simulation as small as possible, thus to develop or use complex software that attempts to simulate reality faithfully. This is typically done by modeling the s-bots in high detail and by using a software dynamics engine, such as Vortex and Open Dynamics Engine (ODE), to simulate the behavior and interaction between virtual objects.

A complementary approach is to use sensory readings sampled from physical robots. For some sensors, like for instance infra-red proximity sensors, it is possible to record reading for various conditions, such as angles and distances to obstacles, and subsequently use those in simulation. However, this is likely to involve tedious, error-prone, manual work in order to obtain a sufficient number of samples and the approach requires that sensors to behave in the exact same way as when samples were recorded after the controller has been transferred. Moreover, if multiple robots are used, they might be slightly different and even if only robots from which samples have been recorded and used in simulation are employed in experiments, the calibration of the sensors might have drifted or changed. For some sensors, like a camera sensor, it can be practically infeasible to use real samples for simulation.

Another widely used, relatively inexpensive, strategy for easing the transfer of controllers from simulation to reality involves adding noise to sensory reading and to actuator outputs [Jakobi, 1997], [Miglino et al., 1995]. The effect of adding noise is two-fold: Real sensors and actuators

[1]Unless, of course, the battery power plays a role for the controllers being evolved and therefore is modelled explicitly in software; for instance if a robot should learn to move to a special area to recharge when its battery level gets low.

are noisy by nature and controllers should be robust enough to handle this; furthermore, slight differences between the simulated and real versions of the sensors and actuators are smoothened by the noise. While adding noise to sensors and actuators does not influence the performance of simulators significantly, the use of detailed robot and environment models and rigid body simulators has a huge impact on the number of CPU cycles required for running evolution in simulation.

During the life-time of the swarm-bots project a number of simulators have been built and a quick web search shows that several generic and architecture-specific mobile robot simulators exist. Yet we argue that none of these are ideal for our purpose and therefore we develop a custom simulator. In this chapter we present a custom built simulator for performing various neuro-evolution experiments on simulated s-bots.

3.1 Major Requirements

Below we list and discuss some of the major requirements for a software simulator.

Description: A simulator is needed for evolving neural network controllers. The simulator should allow us to compare different neuro-evolution strategies for evolving a controller, which solves the task of moving a swarm-bot from an initial position to a target while avoiding holes.

Functional requirements: In order to experiment with different evolutionary approaches one of the most important requirements is that the simulator is versatile, that is, it should be possible (and easy) to change s-bot features, arena layout, evolutionary algorithm, controller types, and so on. If for instance it is necessary to evaluate how an individual evolved in one arena performs in another, it should be easy to do so, meaning that it should not involve recompilations or changing configuration files. Likewise, trying to evolve controllers using different fitness functions and with different neuro-evolution methods should not require recompilation either. Given that a large amount of trial-and-error is necessary, it is important that solutions at various stages can be visualized (e.g. see the robot(s) move around in the virtual world). Noise on sensors and actuators and possibly other features of a simulator depend on a random number generator. It is important that the random generator can be controlled so that a simulation run is deterministic. This means that if two or more simulations are started with identical initial conditions (including identical random seeds), they always produce the exact same results[2].

Non-functional requirements: Since we are not only interested in a controller that performs the task, but intend to compare different neuro-evolution strategies, the performance of the simulator is important because a large number of evolutions are necessary in order to show statistical significance of our results. Due to the performance requirements (and

[2]This requirement generalizes to entire evolutions, such that evolutions started with the same initial conditions should always produce exactly the same results.

good software engineering practices in general); it is important that the simulator can be compiled and run on various platforms and run on clusters and grids, to allow for utilization of various computing resources. The majority of these architectures run some flavor of UNIX, and we therefore require that the simulator at least compiles and runs on a POSIX.1 compliant operating systems.

Physics requirements: We do not use environments (arenas) with rough terrain or obstacles. Our requirements for a simulator in terms of environment modelling are therefore limited to plain environments containing holes. We do not need to handle s-bot/s-bot collisions nor s-bot/wall collisions. However, given that the s-bots are assembled in swarm-bot formation the simulator has to model the dynamics of such a body correctly. The output of the traction sensor, which allows s-bots to detect forces acting on the turret in the horizontal plane, is dependent on how the dynamics are computed. The traction sensor plays a central role in enabling s-bots to move coordinately in swarm-bot formation. Each s-bot can, for instance, align itself to the motion of the rest of the group using the readings of the traction sensor [Trianni et al., 2004b].

This lax requirement concerning the dynamics represents a strong limitation in terms of the type of experiments for which the simulator can be used. However, some of these limitations can easily be overcome. Extending the simulator with collisions should be straight-forward and the reason behind it not being included here is that collisions are not relevant for our study. Support for rough terrain, on the other hand, is not a priority at this stage, since many robot platforms do not support such and limiting the simulator in this way is likely to have a significant, positive impact on the performance.

Thus, what is needed is a flexible, high performance simulator capable of running evolutions on UNIX and Linux machines. The simulator provides features for visualization of the performance of evolved controllers and it should allow for experimenting with various setups and evolutionary algorithms.

3.1.1 Existing Simulators

The research conducted in our lab related the s-bots platform has spun off a number of different simulators; some are built on top of Vortex, a commercial rigid body dynamics simulator, and a more recent simulator is based on Open Dynamics Engine (ODE) and free, open-source alternative. These simulators aim at generality and correctness in terms of physical dynamic behavior and response of virtual objects. Several versions of these simulators exist, going from minimal simulators, in which each s-bot is modelled as a box on wheels, to more complex setups, where every major part and virtually all joints of every s-bot are modelled in detail. Due to their generality, the configuration and setup task is relatively tedious; a number of parameters for every object in the virtual world have to be specified, such as weight, material, friction, position, etc. For the existing simulators this is currently done by hand-editing multiple low-level configuration files (XML).

At the other end of the scale a couple of simple, specialized simulators have been developed where one or few s-bots move around in a 2D world. These simulators handle flat terrain and some even s-bot/s-bot collisions.

There exist a multitude of freely available robot simulators developed by various academic institutions. Many are specific to certain robots and certain types of robots, such as the Khepera robot and industrial robots, respectively. A few simulators aim at being general for mobile, cooperative robotics. One of the more well-known is Player/Stage, which is an open source projected with contributors from prestigious institutions such as Stanford University and University of Southern California. Player/Stage actually consists of two separate but interoperable projects.

Player is a robot device interface:

> *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.*
>
> `playerstage.sourceforge.net`

While Stage is a multi-robot simulator:

> *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. Several controllers designed in Stage have been demonstrated to work on real robots.*
>
> `playerstage.sourceforge.net`

From the sections above, Player/Stage seem to meet most the requirements for the simulator, however Stage does not provide the dynamics that we require and the simulation logic of an s-bot and its sensors and actuators, would need to be implemented in this simulator. Thus, it would take a significant amount of work to add these features to Player/Stage and there would be an overhead associated with sending sensory readings and control action through the network protocol stack. Moreover, even though Stage can simulate multiple robots, each robot has to connect to Stage separately, making the simulation setup cost quite high (remember that since we want to run evolutions we have to setup a new simulation at least once for each individual that we want to evaluate).

3.1.2 Why Yet Another Simulator?

We have chosen to develop an entirely new simulator for multiple reasons, the two major ones being *performance* and *flexibility*. When the work described in this thesis began a number of simulators where tested. We, for instance, evaluated one of the more mature simulators called *MISS* which uses the Vortex dynamics engine. However, with the available hardware resources[3] an average evaluation consisting of 100 individuals per generation, each evaluated for 400 control cycles (40 seconds), for 100 generations, takes an average of 10-12 hours. Now in order to draw any general conclusions on the feasibility of a given evolutionary setup, it is necessary to run multiple evolutions with different initial random seeds: Running at least 10 and often 20 or more evolutions is common practice in the literature, which means that a given evolutionary strategy takes many days to evaluate. If only few strategies need to be evaluated, this is not a major problem. However, a significant amount of trial-and-error is often necessary in order to shape evolution so that the solutions found are satisfactory and thus the evaluation time makes the whole process tedious.

Alternatively, the specialized 2D s-bot simulators perform orders of magnitude faster than any of the simulators built on dynamics simulators, but they are specialized for particular purposes and therefore they lack many of the features and the flexibility that we require. One of the simpler simulators could be adapted to our needs, but the adaptation would likely take as long or longer than developing a simulator from scratch. Moreover, none of the simple simulators meet our requirements for dynamics related to movement in swarm-bot formation and to computation of traction sensor readings.

In a first attempt to develop our own simulator we tried to combine the flexibility and correct dynamics of more complex simulators based on physics engines with the performance of the simpler 2D simulators. We tried various schemes for optimizing the use of a dynamics engine by exploiting the fact that we only need flat terrain with holes and that s-bots are always assembled in swarm-bot formation. If we make these assumptions each individual s-bot needs not to be represented in its own, separate collision space, but instead a whole swarm-bot can be represented as a single compound object in its own collision space, which greatly reduces the computational complexity for the dynamics engine. Similarly, only working with flat terrain with holes opens for some optimization possibilities as a swarm-bot only need to rotate in two dimensions (unless it is falling into a hole, but in this case we can stop the simulation, so we just need to determine when a robot would fall into a hole and not to simulate the actual fall).

The resulting simulator showed promising performance results when simulating few (<8) robots, about 2 to 3 times faster than MISS. However, the scalability of the approach proved poor. The reason for the scalability issues is that each connection, through collisions and/or joints, between bodies (or objects) represents a constraint, and for m constraints the exact algorithm used by ODE requires $O(m^2)$ space and $O(m^3)$ time for each cycle. Even when the number of joints is minimized by letting an entire swarm-bot be represented by a single body with multiple wheels, the performance and scalability was still was not satisfactory. We tried various approximation schemes built into ODE, however, simulations became unstable. Even a combination of the exact method and various approximation schemes proved unsuccessful.

Therefore, a decision was made to implement a 2D simulator, TwoDee, with a custom rigid body engine, specialized to handle only the dynamics for a swarm-bot on flat terrain with holes. Given that, even in some of the more complex simulators, connection between s-bots are assumed

[3]Available hardware resources: 15 single AMD Athlon 1800-2800+ nodes and 16 dual Opteron 1.8 GHz nodes shared between the researchers in our lab. Estimates are based on the assumption that we have exclusive access to one third of the processing power, that is, 16 CPUs.

to be rigid, the entire swarm-bot can be treated as a single body on which each s-bot exerts a force. This reduces the computational resources necessary for conducting simulations and evolutions significantly, and improves scalability because the time complexity becomes $O(n)$, where n is the number of s-bots. In the next section we present the design and some of the features of our simulator.

3.2 Design

In this section we present the major ideas behind the simulator and the implementation of evolutionary algorithms. This section, together with the source code, should provide a reader fluent in C++ with an overview sufficient to make changes and additions to the source code.

The design as been divided into three sections, namely one on the simulator, one on the design of the evolutionary algorithms component, and one on the overall principles and the finished piece of software - an efficient experimental platform for neuro-evolution of cooperative, mobile robot controllers.

The code is written in object-oriented C++ in Hungarian notation. It compiles and runs on POSIX.1 compatible operating systems such as Linux, and with relatively few change it should by compilable and runnable on other operating systems such as Microsoft Windows, although this yet to be tested. Autoconf and automake from the GNU Autotools are used for dependency checking and build management. g++ version 3.3 from the GNU Compiler Collection is the compiler that has been used during development.

3.2.1 The Simulator: TwoDee

The goal of the simulator is to simulate a virtual world and the interaction between a number of s-bots and static objects in an arena. Control programs running on physical robots often operate in discrete cycles. In each cycle sensory inputs are read, the controller decides how to act and sets the actuators accordingly. Like many other simulators, TwoDee follows this discrete sense-act-update pattern as an approximation of continuous time. The simulation cycle is shown in Figure 3-1; first the virtual world is set up, then a cycle is entered where first the sensors readings are computed and fed to the robot controllers which in turn determine the actions of the robots. The state and time of the virtual world is then updated by moving objects according to the forces currently acting on them[4]. Finally, the state of the virtual world can be visualized if the user is running the simulator in interactive mode. Typically s-bots run a control cycle every 0.1 seconds, thus per default one simulation cycle corresponds to 0.1 seconds of virtual time but the control frequency can be adjusted. Each update of the virtual world is referred to as a *simulation step*.

Structure

Internally the simulator consists by a number of objects of classes representing entities such as the static parts of the virtual world (the arena), s-bots, sensors, and actuators. All classes

[4]It is assumed that forces are constant between update cycles. This is also an assumption made in software dynamics engines like ODE and Vortex.

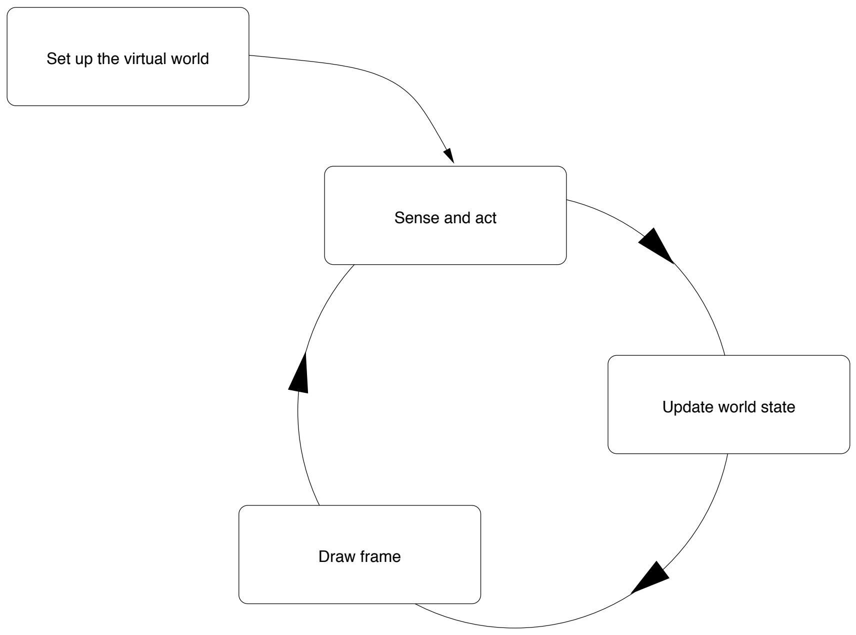


Figure 3-1: The simulator operates in discrete cycles shown above. Sensor readings are computed and actions are performed, the state of the virtual world is updated and the result is (optionally) visualized.

except from renderers, which are responsible for visualizing the simulation, inherit from a root class called `CSimObject`. The class `CSimObject` contains a few convenient methods and allows all objects to be tagged with a string denoting a human readable name, much like the `toString()` method in Java. Furthermore, since objects of the type `CSimObject` can be added to other `CSimObject` objects as children, an object tree can be build. This is convenient for passing events, such as the start of a new simulation cycle, user interaction, and screen refreshes to all objects without relying on some entity having a global view of the entire system. Decentralization in general is a good software design strategy whenever possible, since many changes and additions can be kept local and do not require complete, global knowledge. Objects of the class `CSimObject` has four important methods shown below:

```
CSimObject::
    virtual void AddChild(CSimObject* pc_child);
    virtual void SimulationStep(unsigned int n_step_number,
                                double f_time,
                                double f_step_interval);
    virtual void Draw(CRenderer* pc_render);
    virtual void Keypressed(int n_keycode);
```

The first method, `AddChild()`, add a child to the callee object. All `CSimObject` objects are responsible for forwarding events to their children, meaning that when `SimulationStep()` is called on an object, it should call the `SimulationStep()` method on all its children. This is done automatically by the methods implemented in `CSimObject`, thus a new class only has to explicitly forward events to children for the methods, which it overrides.

SimulationStep() is called each time the simulation takes a step. Objects in TwoDee are responsible for updating their own positions etc. and they should do so by overriding the SimulationStep() method and perform their actions each time it is called. The parameter n_step_number refers to the logical number of the current simulation step, while f_time denotes the virtual time in seconds from the beginning of the simulation, and f_step_interval denotes the virtual time interval between simulation steps.

The method Draw() is only called when the simulator is running in interactive mode, and it is called with reference to the renderer that should be used. A renderer if the type *CRenderer* has methods for drawing high-level primitives such as s-bots, arenas, and lights.

Keypressed() is called when the user provides keyboard input. When the simulator is running in visualization mode, the keyboard can for instance be used to control individual s-bots, fast-forwarding the simulation-time, adding and remove lights etc. Actually, since all instances of the class CSimObject are notified every time the user presses a key, interactivity can be added in any part of the simulator with little effort.

Although the idea about having a rooted class hierarchy with single super-class with methods ranging from drawing, to handling user inputs and to controlling the simulation might seem complex, the total code in the class amounts to less than 100 lines of code. Hence, the concept is quite simple, yet powerful given that it relies on decentralization and the recursion. To illustrate this, we provide an example: Assume that we need to draw a frame; the renderer calls Draw() method on an object of the class CSimulator, our main container for all other objects in the simulation (see Figure 3-2). If the simulator needs to put some status information on the frame, like a simulation timer, it does so and then calls the Draw() methods on all its children. The "children" of an object denotes the set of objects that the object aggregates, which in the simulator's case is an arena, a number of swarm-bots, possibly lights and so on. When the Draw() method of an s-bot is called, the s-bot draws itself, or rather calls the renderer to have the s-bot drawn, and calls all of its children's Draw() methods. The children of an s-bot are sensors, actuators, a controller and any other sub-class of CSimObject, which might be convenient to have as a child of an s-bot. Now assume that we at some point need to add a humidity source to our simulator and that it should be drawn whenever the simulator is run in visualization mode. In order to do so, we simple make our humidity source class inherit from CSimObject and add it as a child to our simulator (or to the arena if we prefer). If we override the Draw(), Keypressed(), and SimulationStep() methods in the humidity source class we can control the appearance and behavior of the source. No changes are needed in the simulator class itself, as it sees a humidity source as any other CSimObject object.

The class diagram of the basic structure is shown in Figure 3-2. The root class, CSimObject, and a few auxiliary classes have been omitted for readability. At the top of the figure three classes are shown: *CRenderer*, CSimulator and CFitnessFunction. The latter is drawn in dashed lines since the fitness function is logically part of the evolutionary algorithms component and not the simulator, but given that the fitness score is computed based the an individual's performance in simulation, the fitness function needs direct access to the state of the simulator. The CSimulator class encapsulates an entire simulation and consists of an arena in which a number of swarm-bots move. The class is relatively thin and besides from

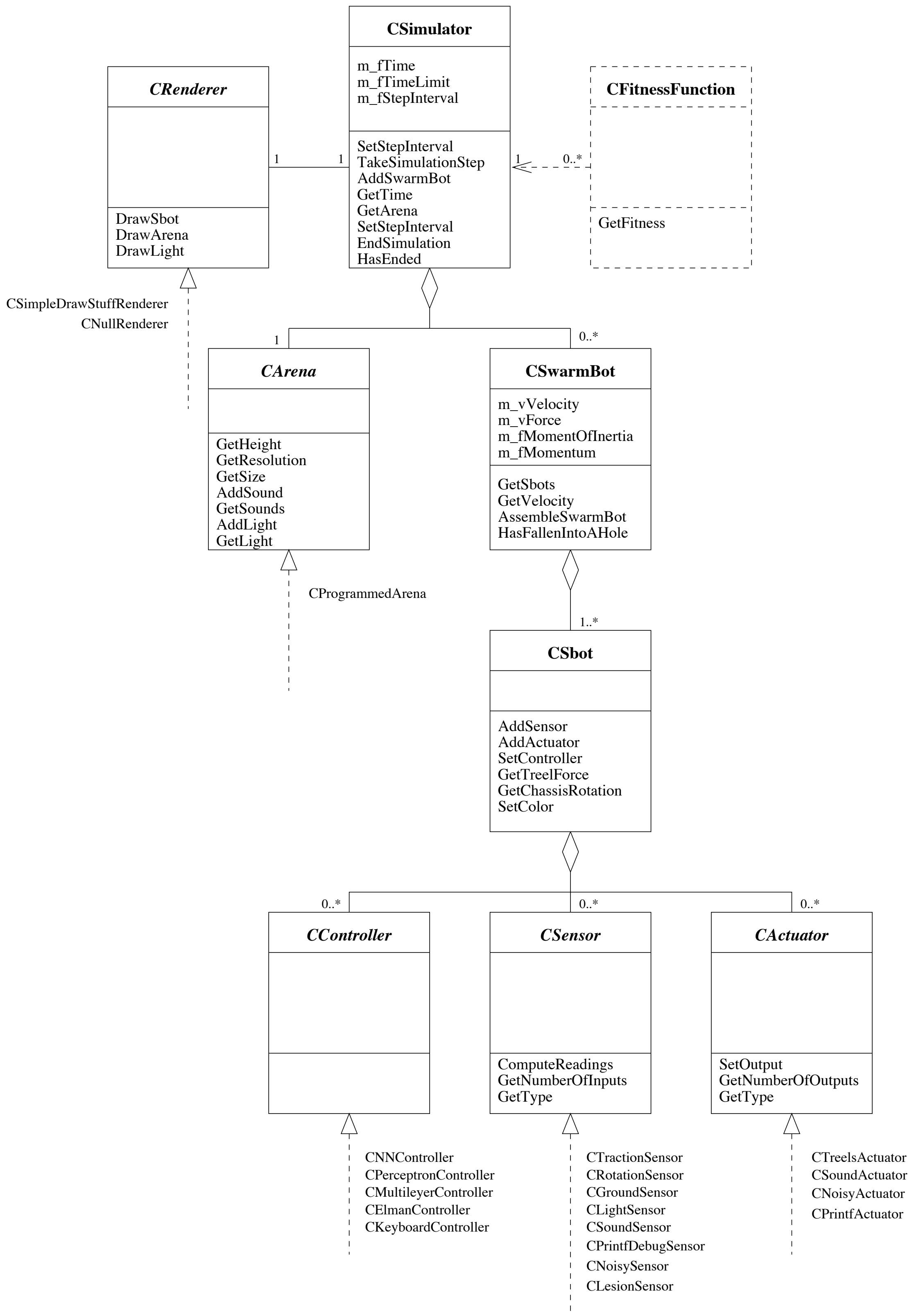


Figure 3-2: A UML diagram of the simulator. The figure shows the main classes and some of their specializations. All classes except from *CRenderer* inherit from the `CSimObject` class as described in the text.

keeping track of time and the state of the simulation (either running or not running), it is simply a container for the objects in the simulation.

The abstract *CRenderer* class defines a set of methods for visualizing simulations. Each time a frame should be drawn the renderer calls the `Draw()` method on the simulator, which in turn calls the draw method on all its children recursively as described above. Since the *CRenderer* only defines high-level primitives like `DrawSbot()` and `DrawArena()` and not low-level primitives for drawing triangles, rectangles or alike, implementations can range from 2D to 3D and from text-based to full-blown graphics visualization. At the time of writing, three implementations of renderers exist: A null renderer, an OpenGL renderer based on the DrawStuff library, which is a part of ODE, and a gnuplot renderer.

Arenas

The static parts of a virtual world, referred to as the arena, are encapsulated in the class *CArena*. Arenas are two-dimensional and each point of the arena surface can take three values (or heights): Normal, hole, and obstacle.

An arena has a size and a resolution. The size corresponds to its physical size within the virtual world (in meters), while the resolution denotes how many points there are in each dimension in case of discrete arenas. Continuous arenas have infinite resolutions, and they are usually defined analytically (this is convenient if, for instance, an arena with a circular hole has to be modelled), while discrete arenas are specified as a two-dimensional array of points, which each point corresponds to the height of a rectangular tile in the arena. An array of points can for instance be specified by an array of characters, where one type of character (a white space) specifies the presence of a hole on a particular tile, while another ("`X`") denotes an obstacles such as a wall and finally a character for the parts of the arena on which robots can move ("`#`"). An example of this scheme is shown below for an arena consisting of 41x15 tiles:

```
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
X#######################################X
X##  ###  #      #  #####  #####      ##X
X##  ###  #  #####  #####  #####  ##  ##X
X##       #    ###  #####  #####  ##  ##X
X##  ###  #  #####  #####  #####  ##  ##X
X##  ###  #      #      #      #      ##X
X#######################################X
X##  ###  #      #      #  #####     ###X
X##  ###  #  ##  #  ##  #  #####  ##  ##X
X##  ###  #  ##  #     ##  #####  ##  ##X
X##  # #  #  ##  #  #  ##  #####  ##  ##X
X###     ##      #  ##  #      #     ###X
X#######################################X
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

The arena contains the two words "HELLO" and "WORLD" written in hole tiles framed by a wall. The corresponding 3D rendered version of the arena can be seen in Figure 3-3.

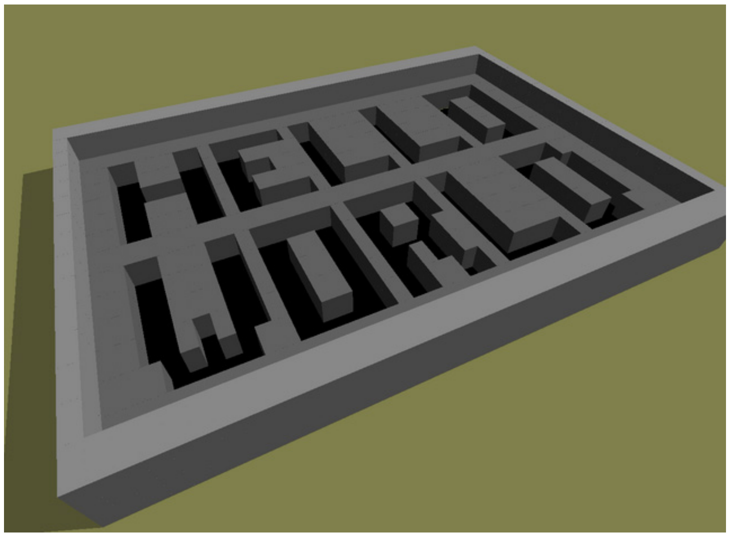


Figure 3-3: An example of a discrete arena rendered in 3D.

The physical size of an arena is independent of the resolution. In our example above, we have chosen the arena to have the physical dimension 4x3 meters; however, we could have chosen length and width arbitrarily.

This way of specifying arenas allows the user to rapidly try various arenas without having to specify a set of meshes and their 3D coordinates in XML files as it is the case with some other simulators. Actually, constructing an arena like the one shown in Figure 3-3 would be a tedious task without the aid of some 3D modelling tool if all coordinates had to be calculated and typed in by hand.

Another benefit of constructing from a number of equally-sized tiles with different heights arranged in a two-dimensional grid is that almost all computations related to robot/arena interaction and arena queries in general can be done in constant time. For instance, in order to determine if one of the treels of a robot is on the arena surface or if it is currently over a hole, it is enough to make a constant time lookup in the character array defining the arena[5]. Thus, the time complexity of running simulations is not affected by the complexity of the arena. The complexity of the arena only affects the run-time in cases where we use sensors dependent on ray-casting (e.g. a camera).

Swarm-Bots, s-bots, controllers, sensors, and actuators

The `CSwarmBot` and `CSbot` classes implement the logic concerning swarm-bots and s-bots, respectively. The `CSwarmBot` class implements the dynamics for swarm-bots. A swarm-bot

[5] In more general 3D simulators multiple objects often have to be checked, since they - due to their generality - lack a well-defined structure for the arena, which in TwoDee enables constant time operations.

can consist of one or more s-bots[6]. The CSbot class implements the concept of a bare-boned s-bot to which a number of sensors and actuators as well as a controller can be added. Hence, as such the CSbot class implements the geometrical properties of an s-bot, while its capabilities are added as children. Therefore, it is not necessary to have various specializations s-bots depending on which sensors, actuators and controller one wishes to use; the capabilities of the s-bot can be determined and adjusted at run-time.

Multiple controllers exist; one that allows the user to control one or more s-bots interactively via the keyboard and several artificial neural network controllers specialized for different types of networks. A number of sensors and actuators have also been implemented together with *meta-sensors* and *meta-actuators*, which adds features such as debugging and noise to existing sensors, see Figure 3-2.

Swarm-Bot Dynamics

The custom physics implemented in TwoDee is based on standard 2D rigid body physics without joints. We simply treat a whole swarm-bot as a single body and let each of the s-bots be a force on the body. The sum of the forces and torque in turn determines the acceleration and the angular acceleration of the whole swarm-bot. This fundamental design choice has one important consequence: Connections between s-bots are assumed to be completely rigid and infinitely strong. Obviously, this is a crude approximation since such connections are impossible in practice. The rigid gripper actually does move a little both horizontally and vertically with respect to the gripped s-bot when connected s-bots move around. However, some of the other more complex simulators makes this approximation as well, and even simulators where the gripper is modelled as a set of meshes and where connections are implicit through mesh collisions, do not simulate reality closely due to the continuous and complex nature of the physical interactions responsible for the connections between s-bots.

A concise, yet thorough, introduction to rigid body dynamics can be found in Hecker [1996a], Hecker [1996b], Hecker [1997a] and Hecker [1997b]. Below we present and discuss mathematical framework that we use to compute the dynamics of Swarm-Bots in TwoDee.

Linear Motion

Let $\mathbf{r}$ denote the 2D position vector of a given object, and let $\mathbf{v}$ be the objects velocity. If we let t denote time, then the velocity is the derivative of displacement:

$$\frac{d\mathbf{r}}{dt} = \mathbf{v} \tag{3-1}$$

The relationship between acceleration, $\mathbf{a}$, and velocity is similar:

$$\frac{d\mathbf{v}}{dt} = \mathbf{a} \tag{3-2}$$

Acceleration is in turn determined by the mass, m, of an object and the forces, $\mathbf{F}$, acting on it:

[6]Some might rightfully argue that it would take at least two s-bots to form swarm-bot, however, we generalize in the simulator so that a swarm-bot can also consist of a single s-bot in order to maintain the hierarchy: Swarm-bots are children of the simulator, while s-bots are children of swarm-bots.

$$\mathbf{a} = \frac{\mathbf{F}}{m} \tag{3-3}$$

If we know the power of the electric motors driving the treels of an s-bot, the orientation of the chassis and the mass of an s-bot, we can simulate the movement of an entire Swarm-Bot by summing the forces acting on it, computing the acceleration, velocity and finally displacement. It would be straight-forward to implement a dynamics engine if linear motion was all there was to it, however two important components are missing, namely orientation and friction.

Orientation

Physical objects ranging from elementary particles to stars spin around themselves. The *orientation* of an object and the quantities *angular velocity*, *angular acceleration*, and *torque* are related to this type of motion, which is complementary to linear motion. Changes in orientation are like changes in displacement dependent on the forces acting upon the object. The type of motion resulting from a given force being applied depends on the point of application and the direction of the force. Try for instance to push a pen on the desk in front of you, if you push it (apply a force) perpendicularly close to one of the ends, it simply rotates, but if you push it on the middle the whole pen moves linearly. We need to simulate both types of kinetic quantities, since s-bots moving in swarm-bot formation tend to often change their orientations with respect to one another.

Since we only need to deal with dynamics in two dimensions the orientation of an object can be described by a scalar corresponding to the angle between the orientation of object and the world coordinate system. We let Ω denote this angle. The quantity known as *angular velocity*, ω, e.g. how fast an object is rotating, is the derivative of the orientation:

$$\frac{d\Omega}{dt} = \omega \tag{3-4}$$

And the angular acceleration, α, is the derivative of the angular velocity:

$$\frac{d\omega}{dt} = \alpha \tag{3-5}$$

Before we can write down an expression for the relation between the forces applied to an object and angular acceleration we define two convenient measures: The center of mass and the moment of inertia. We can consider a swarm-bot as a set of points corresponding to the s-bots. The center of mass, $\mathbf{r}_{CM}$, is then given by:

$$\mathbf{r}_{CM} = \frac{\sum_i \mathbf{r_i} m_i}{\sum_i m_i}, \tag{3-6}$$

where $\mathbf{r_i}$ and m_i is the position and the mass of point i, respectively.

If we let the orientation of each point in an object be defined with respect to the center of mass, the kinetic quantity known as *moment of inertia*, is given by:

$$I = \sum_i m_i |\mathbf{r}_i - \mathbf{r}_{CM}|^2 \tag{3-7}$$

Thus, the sum of the mass of each s-bot, multiplied by the square of it distance from the center of mass. In the 2D case we then use the *perp-dot* product between two vectors to find the contribution of a force, $\mathbf{F}$, to the "angular force" known as torque and denoted by τ:

$$\tau = (\mathbf{r}_i - \mathbf{r}_{CM})\dot{\perp}\mathbf{F} \tag{3-8}$$

where $\dot{\perp}$ denotes the perpendiculized dot product or prep-dot product:

$$\mathbf{A}\dot{\perp}\mathbf{B} = -A_yB_x + A_xB_y. \tag{3-9}$$

Thus, we take the vector starting in the center of mass and ending in the point in which force, $\mathbf{F}$, is applied and *perpendiculize* this vector by rotating it $\frac{\pi}{2}$ counter-clockwise[7] and take the dot product between the perpendiculized vector and the force. An example is shown in Figure 3-4; the torque in resulting from $\mathbf{F}$ being applied in $\mathbf{r}_i$ is exactly the dot product between $\mathbf{F}$ and $\mathbf{A}_\perp$ or the perp-dot product between $\mathbf{A}$ and $\mathbf{F}$.

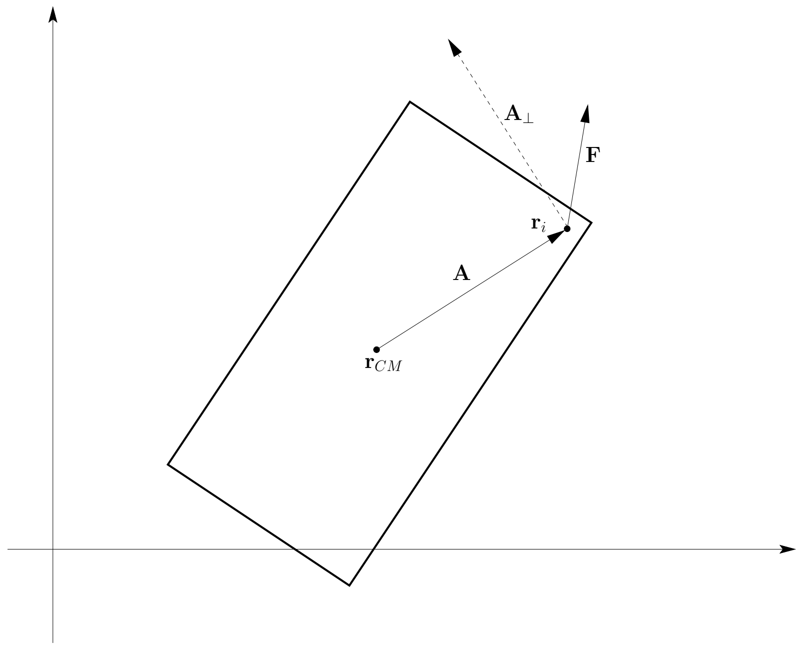


Figure 3-4: The figure shows an example of a force $\mathbf{F}$ being applied in a point $\mathbf{r}_i$. The vector $\mathbf{A}$ is the vector from the center of mass $\mathbf{r}_{CM}$ to the point of application $\mathbf{r}_i$, while $\mathbf{A}_\perp$ denotes $\mathbf{A}$'s perpendiculized cousin. The torque is computed as the dot product between $\mathbf{A}_\perp$ and $\mathbf{F}$.

Finally, we can write an expression for the relationship between the forces applied on an object and its angular acceleration:

[7] A 2D vector $\mathbf{A} = (A_x, A_y)$ is easily perpendiculized or rotated $\frac{\pi}{2}$ radians counter-clockwise: $\mathbf{A}_\perp = (-A_y, A_x)$

$$\alpha = \frac{\tau}{I_A}. \tag{3-10}$$

With this expression we can now compute the angular acceleration, angular velocity, and the orientation a swarm-bot based on the forces applied by the treels of each s-bot in the formation.

Friction

There are two types of friction, namely static friction and kinetic friction, and kinetic friction can again be divided into sliding, rolling and fluid friction. For your purposes it suffices to consider only one type of kinetic friction, namely sliding friction. Static friction determines how much force it takes to put an object into motion, while the kinetic friction applies once the object is in motion.

The friction force, F_f, is given by the following expression:

$$F_f = \mu_f F_p \tag{3-11}$$

where μ_f is the *friction coefficient* and F_p the *normal force*. The normal force is the force perpendicular to the contact surface, which in our case always equals the force of gravitation times the mass of a given object since we only consider flat terrain. Notice that the friction force, F_f, is a scalar and not a vector. This is because friction always acts in the direction opposite to the direction of motion for kinetic friction and static friction denotes the size of force necessary to put the object into motion and therefore has no direction.

In our case, however, the friction force is a bit more complicated, since we are not dealing with passive objects; but rather we have a swarm-bot that consists of a number of interconnected, active objects, which use the friction between the surface and their treels to propel themselves, possibly in different directions. Imagine, for instance, two s-bots connected to each other trying to move; if they try to move in complete opposite directions, the forces simply cancel each other out, while if they attempt to move in the same direction, we do not have to take friction, in the classical sense, into account. In all other cases the friction depends on the orientation and speed of the treels of each individual s-bot in the swarm-bot.

We divide friction into two components: The linear friction and angular friction. Moreover, we distinguish between the situation in which the treels of an s-bot is moving (kinetic friction) and when they are not moving (static friction) as practical experiments show that moving treels exert much less friction than non-moving teeels, since moving treels slip easier.

Linear friction

We assume a friction model where the friction caused by each s-bot is given by the following expression if the swarm-bot is moving:

$$F_f = (1 - \cos\theta)\mu_f |\mathbf{F}_t| \tag{3-12}$$

where μ_f, the friction coefficient, is either μ_m if the treels are moving and μ_n if the treels are not moving. θ denotes the angle between the current velocity vector and the orientation of the treels and $\mathbf{F}_t$ the force of the treels[8]. If the swarm-bot is not moving the friction of each s-bot is given by:

$$F_f = (1 - \cos\Theta)\mu_f|\mathbf{F}_p| \tag{3-13}$$

where μ_f, the friction coefficient, is either μ_m if the treels are moving and μ_n if the treels are not moving. Θ is the angle between the current sum of all the forces acting on the swarm-bot and the force exerted by the s-bot, and $\mathbf{F}_p$ denotes the normal force. If an s-bot is passive and not exerting any force on a static swarm-bot, we simply use the standard fiction model shown in (3-11) with $\mu_f = \mu_n$.

Angular friction
Angular friction is, like linear friction, divided into two cases, namely one for when the angular velocity is zero and one where the angular velocity is non-zero. An s-bot only causes angular friction if it does not exert a torque with the same sign as the angular velocity or the total torque of the swarm-bot if the angular velocity non-zero and zero, respectively. The friction on angular motion is given by:

$$F_\Omega = \mu_g|\mathbf{A}||F_p| \tag{3-14}$$

where μ_g, the angular friction coefficient, is either $\mu_{m'}$ if the treels are moving and $\mu_{n'}$ if the treels are not moving, $|\mathbf{A}|$ the length of the vector from the center of mass to the s-bot, and $|F_p|$ the size of the normal force.

We have based our friction models on intuition and experimentation, and tests with evolved controllers on physics robots show that this friction model is sufficiently close to reality to allow controllers to be transferred. If a more accurate or a different friction model is needed, then it can fairly easily be implemented, since the friction model is isolated in a single part of the code[9]. The fiction coefficients can and should be adjusted for a particular arena surface in the simulator.

Summary

The following steps are taken in order to compute the dynamics of a swarm-bot:

1. For a swarm-bot, compute the center-of-mass and the moment of inertia at the center-of-mass.
2. Set the initial position and orientation of the swarm-bot
3. Let the robots sense and act.

[8]Note that the force exerted on a swarm-bot by the treels of an s-bot depends on the difference between the orientation and speed of the treels *and* on the velocity of the swarm-bot.

[9]It is not only the friction code that is isolated in a single part of the code; the code concerned with the dynamics in general can be replaced, e.g. with calls to a dynamics engine like ODE or Vortex, although this would take some effort, since these engines require detailed 3D models of the objects present in the simulation.

4. Compute all the forces on the swarm-bot, including their points of application.
5. Sum all the forces and divide by the total mass of the swarm-bot to find the linear acceleration at the center of mass.
6. For each force, form the perp-dot product from the center of mass to the point of application and add the value into the total torque at the center of mass.
7. Divide the torque by the moment of inertia at the center of mass to find the angular acceleration.
8. Compute the friction and adjust the linear and angular acceleration accordingly.
9. Numerically integrate the linear and angular acceleration, update the linear and angular velocities, and finally the position and orientation of each s-bot.
10. Update the clock and go to step 3.

Units

All measures and units, such as time (seconds), weight (kg), forces (N), velocities (m/s), positions (m), etc. used in the simulator follow the International System of Units (SI); thus, the metric system used in continental European.

3.2.2 Evolutions

Strictly speaking, the code related to evolutionary computation is not part of the simulator. However, since our focus is on evolution of controllers, it makes sense to provide an overview of the implementation concerned with evolutionary algorithms and its interplay with the simulator.

A UML diagram of the evolutionary computation component is shown in Figure 3-5. The design of the component is straight-forward: A population consists of a number of individuals. The super-class, *`CPopulation`*, is an abstract class defining a common interface to different population types. Each separate type of neuro-evolution algorithm is represented by its own specialized class inheriting from this super class. The class `CStandardGAPopulation` implements a standard genetic algorithm population, with rank based selection, crossover and mutation. The class `CMuLambdaPopulation` implements a [μ,λ] population (pronounced *mu-lambda population*). The [μ,λ] evolution loop works as follows: The μ best individuals from the previous population are retained, and the others are discarded. These μ individuals are called the "parents". Each parent gets to produce $\frac{\lambda}{\mu}$ children: thus the total number of children is λ. These λ children form the next generation. In this way no cross-over is used and optimization is done solely through mutation [Schwefel, 1995].

Notice that the cooperative coevolutionary population class, `CCCGAPopulation`, aggregates a number of populations; one population object for each sub-population in the evolution. These populations can be of any type (even different types), which means sub-populations can even be recursive. So far, we have used standard populations with crossover and mutation, hence objects of the type `CStandardGAPopulation`, as sub-populations and the use of recursive sub-populations are yet to be studied.

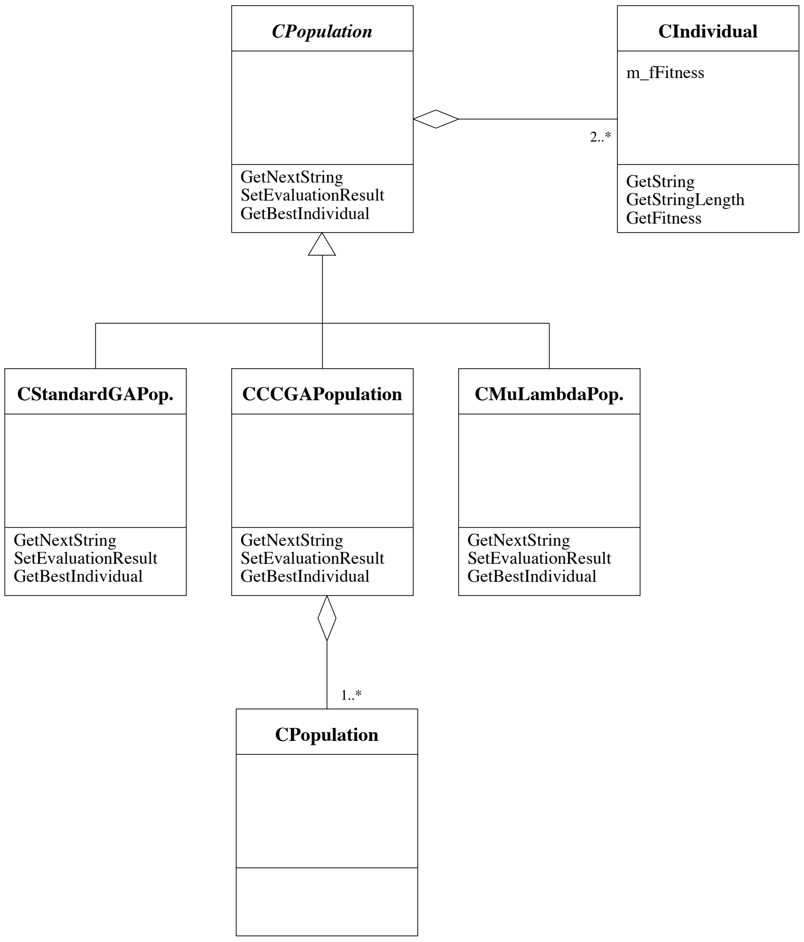


Figure 3-5: A UML diagram of the evolutionary computation component.

The evolution loop is started by either creating a new, random population or by loading an existing population. Then each individual or string is assigned a fitness score. Once all individuals have been evaluated, a new generation is created (unless the termination criterion is satisfied) and the individuals of the new generation are evaluated, and the loop is started from the beginning. Multiple individuals can be evaluated concurrently and in any order. These features allow for parallel implementations.

We have specialized the evolutionary algorithms so that strings are given over the alphabet of floating point values in the interval $[-10 : 10]$, since we are optimizing artificial neural network weights. However, the genetic algorithms component is not part of the simulator and it could easily be used in other domains and studies as well. In the following section we present how the components of simulator and the evolutionary algorithms have been joined together in order to provide an easy-to-use, flexible environment for studying neuro-evolution of robot controllers.

3.3 Putting Everything Together

The simulator and the genetic algorithms have been combined in a command-line driven application, which allows for configuration and execution of simulations, evolution and graphical presentation of the evolved behavior. The component that binds the simulator and the genetic algorithms together takes care of reading and writing status files, allowing for features such as restarting evolutions with different settings (e.g., a different fitness function and/or a different arena) and easy visual validation of the behavior of evolved controllers.

The main goals and features are presented below. In the next section we provide some examples to illustrate how some of the features listed are used.

Minimal configuration files: Configuration files, even when done in XML or another standard representation format, are often highly specialized and have a tendency to become bulky and complex over the life-time of a software project. New users are often frustrated by the cryptic configuration options and their interdependencies. For the experienced user large configuration files result in the software being flexible but tedious to setup. Therefore, one of the goals for TwoDee was to minimize the use of configuration files as much as possible, and we have succeeded in avoiding configuration files altogether, yet allowing TwoDee to be highly configurable through command-line parameters.

No recompilation: Configuring everything at compile time is an easy, but poor way of avoiding configuration files. Actually, recompilations should be avoided in general, since the process of maintaining and changing code frequently (e.g. each time an aspect of a simulation has to be changed) is error-prone. Some of the existing simulators suffer from this problem. In TwoDee code only have to be changed when bug are corrected and additions, such as new fitness functions, new sensors, controllers, etc. are made - and these additions often require only local changes. All fitness functions, arenas and previously developed sensors, actuators and controllers can be used in future simulations and evolutions without recompilations. Thus, this provides backward compatibility and the feature set of TwoDee is continuously expanded.

Flexibility: Many aspects of TwoDee are highly flexible and versatile, for instance the arena, fitness function, configuration of the s-bot, sensors, actuators, controller type, number of samples, etc. can all be configured and reconfigured during an evolution.

Works out of the box: Since the GNU Autotools have been used, the simulator compiles effortlessly on a wide variety of POSIX operating systems. So far it has been tested on various Linux distributions and on Mac OS X and compiles in the standard GNU way; decompress, run `configure` and then `make`. Furthermore, since no configuration files are needed the simulator can be started directly from command-line, and will provide the user with a visual 3D display of a standard arena and containing one s-bot, which can be controlled via the keyboard.

Easily experiment with evolved controllers: Given that trial-and-error is an inherent part of evolutionary robotics it has been made easy to validate the behavior of evolved controllers and to change various aspect of the simulation, e.g. to check how the controller performs in other environments, scalability, different initial configurations and even with other fitness functions.

3.4 Benchmarks

In this section we provide benchmarking results for a simulator built using ODE and compare its performance characteristics to those of TwoDee. We do not mean for this to be an extremely accurate benchmark, but merely to show trends in the performance of simulators that are built according to different designs. Table 3-1 shows the performance results for two simulators: A recently developed simulator built on ODE, which we shall call NS, and TwoDee. We use NS for our benchmarks instead of MISS, since NS was develop with the intention of replacing MISS and other Vortex-based simulators at our lab. The times shown in the table correspond to the time each of the simulators requires to simulate 100 seconds for a given number of connected s-bots moving randomly. All the benchmarks were conducted on an Athlon XP 2800+ with 512 MB running Debian Linux. Notice that for NS we have two sets of results, one for exact simulation and one for approximate simulation using the *quick-step* feature of ODE. The benchmarking methodology and the performance results for NS have been obtained from Dung [2005].

As it can be seen from the performance results TwoDee is orders of magnitude faster than NS for larger configurations and shows a close-to-linear relationship between the number of s-bots simulated and the processing time required to execute a simulation. NS running in approximation mode shows a similar linear relationship, although between 20 and 50 times slower than TwoDee. These performance results are a natural consequence of the methods used to compute the dynamics of the virtual world; while TwoDee and ODE running in approximation mode, employ methods linear in the number of objects, Vortex and ODE running in exact mode solve systems of equations with non-linear constraints which grow quadratic in the number of connected virtual objects. Due to this time and space complexity, NS is practically limited to around 12 robots when running in exact mode and as described in Dung [2005] ODE actually crashed due to memory shortage when the benchmarks were run for more than 12 s-bots on

# s-bots	TwoDee	NS exact	NS approx.
1	0.02s	0.09s	0.41s
2	0.03s	0.41s	0.84s
4	0.05s	2.72s	1.74s
6	0.06s	8.90s	2.61s
8	0.08s	24.87s	3.49s
12	0.10s	75.47s	5.27s
16	0.13s	-	7.06s
20	0.15s	-	8.88s
25	0.19s	-	11.09s
32	0.23s	-	-
64	0.44s	-	-
128	0.86s	-	-
256	1.71s	-	-
512	3.39s	-	-
1024	8.41s	-	-

Table 3-1: Performance benchmark for TwoDee, NS running in exact mode, and NS running in approximation mode. The numbers shown correspond to the time each of the simulators requires to simulate 100 seconds for a given number of connected s-bots moving randomly. NS running in approximation mode and TwoDee show a time complexity close-to-linear in the number of connected s-bots simulated, while NS running in exact mode appears to have a cubic time complexity. TwoDee is orders of magnitude faster than NS even when running in approximation mode.

the test machine. TwoDee has no such memory requirements and simulating 10.000 connected robots takes up only 13 MB of memory.

3.5 Future work

The current version of TwoDee is highly specialized for the study underlying this DEA thesis. However, we intend to make the simulator, or parts thereof, more general in the future versions. Specific features that we will include are collision detection and dynamics for multiple, disconnected robots, as well as support for new robot architectures such as the e-Puck.

Furthermore, a parallel version of TwoDee is planned, which allows for evaluation of different members of a population in parallel. Currently, multiple evolutions can easily be done in parallel, simply by starting multiple instances on different nodes on a parallel computer such as a cluster. However, a parallelized version of TwoDee itself would allow for multiple CPUs working on *the same* evolution concurrently. This in turn would decrease the time between an evolution is started and the results become available, thus it might speed up the trial-and-error phase of evolving robot controllers as results are made more rapidly available.

3.6 Summary

In this chapter we have presented the requirements for a software simulator and an environment for neuro-evolution of robot controllers. We have motivated why existing simulators do not meet these requirements and we have therefore chosen to develop our own, specialized software called TwoDee. We have presented the overall design of the simulator, the underlying dynamics and benchmarks for TwoDee and another recently developed simulator, NS, built on a general software dynamics engine. Our results show that TwoDee outperforms NS by several orders of magnitude, and that TwoDee scales close to linearly, both time and space-wise, in the number of s-bots simulated. We have successfully been able to simulate 1.000.000 connected robots on a normal workstation[10].

At the time of writing TwoDee comprises approximately 12.000 lines of code plus an additional 1.500 lines of code in the DrawStuff library.

[10] A normal workstation in this case is an Athlon XP 1800+ with 512 MB of RAM running Debian Linux. When TwoDee simulates 1.000.000 connected s-bots the memory usage is approximately 500 MB of RAM.

Evolving Controllers

Chapter 4

Evolutionary robotics is a field that is in its infancy. The problems tackled by researchers are often limited to simple tasks such as path finding, prey retrieval, and reaching a target location. The main reason why evolutionary robotics has not yet provided us with sophisticated and useful robots is that evolving complex behavior is *very hard*. No general guidelines exist for how one should design evolution so that it finds a good solution for the task at hand. Therefore, one is left with a trial-and-error approach where several strategies have to be tried before a successful one is found or before the task is given up. First of all, boot-strapping evolution can be a problem; secondly expressing our objectives in terms of a fitness function can be challenging, and finally shaping the fitness landscape so that evolution has a fair chance of reaching an acceptable solution is often non-trivial.

In this chapter we investigate different strategies for evolving controllers for semi-complex behavior; that is, a controller which allows a swarm-bot to move from an initial location to a target area marked by a light. In the process we suggest a method for finding a good evolutionary setup and we compare the performance of various types of neural networks and neuro-evolution methods.

We have structured the chapter in the following way: We first describe our methodology for obtaining a good evolutionary setup, namely by defining an appropriate order and way in which the different aspects of the artificial evolution of a controller are treated. We argue that the order should be the following: Fitness functions, neural network types, neuro-evolution methods, and evolutionary parameters. We then discuss our general experimental setup and the mathematical notation we use to express fitness functions. In the remaining sections of the chapter we apply the suggested method in order to evolution of a controller that enables a group of s-bots in swarm-bot formation to perform phototaxis and hole-avoidance.

4.1 Methodology

Essentially, an evolutionary setup can be divided into three different parts as shown in Figure 4-1: Neural network type and structure, fitness function, and the neuro-evolution method. The neuro-evolution method contains a number of evolutionary parameters such as mutation rate, selection scheme, population size, crossover rate, etc. In this section we discuss some of the issues related to finding an evolutionary setup and the main ideas behind our methodology.

4.1.1 Interdependence

Defining a method for finding a good evolutionary setup is not straight-forward. Most if not all components of the evolutionary setup are interdependent, which means that changing one easily affects the performance of another.

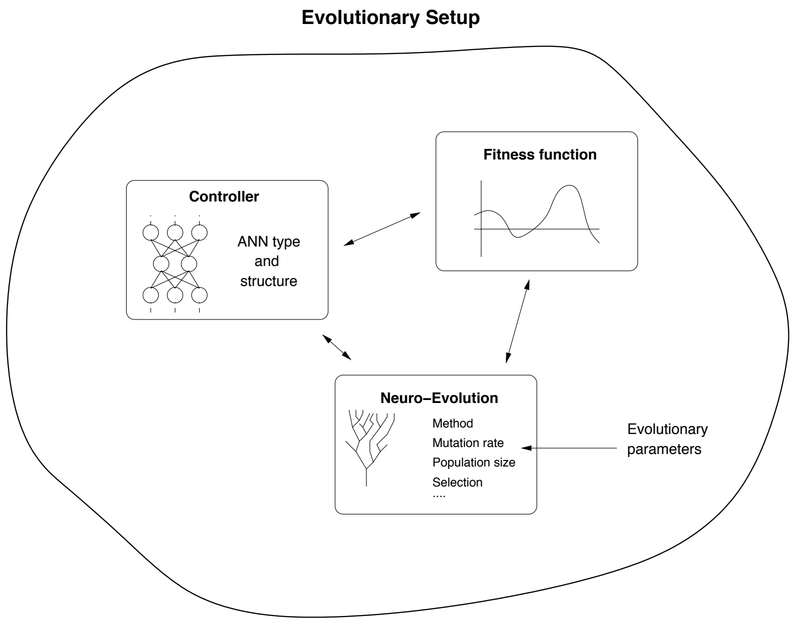


Figure 4-1: The three main components in an evolutionary setup: The controller, which consist of an ANN, a fitness function, and a neuro-evolution method. As the arrows in the figure indicate, the components are interdependent, which means that changing an evolutionary parameter can influence the performance of another.

Several papers have been written on very specific details concerning evolutionary computation such as the mutation operator. Although such studies increase our understanding of the subtleties of evolutionary computation, a single part of the evolutionary setup can hardly be isolated and analyzed independently without regard for the rest of the system. For instance, if the mutation rate is very high then a selection scheme that includes elitism could out-perform a selection scheme without elitism, whereas the reverse might be true if we choose a very low mutation rate. In turn, the quantitative translation of *very high* and *very low* in terms of mutation rate is problem-dependent and has to do, not only with the number and types of variables to optimize, but also the dependencies between the variables themselves, the chromosome encoding, and so on. Aside from the evolutionary parameters for a given neuro-evolution method, we also have to find both a suitable fitness function and an applicable artificial neural network type and structure. All of these components are interdependent since the fitness landscape is shaped by the fitness function and neural network type and structure, while the neuro-evolution method and its evolutionary parameters determines how the search for good solutions is conducted herein.

Thus, every component of the evolutionary setup depends on every other component, which is probably one of the reasons why no standardized or formally proved method for evolving robot controllers exists. In the following we will discuss the method, which we have used to find an evolutionary setup that allows us to evolve controllers for our task. Although the method has been developed with our task in mind, we hope that it can serve as the first small steps towards a structured approach for task-oriented evolutionary robotics[1].

Laying Down an Order

It is practically infeasible to perform an exhaustive search in the space of evolutionary setups, that is, all possible combinations of parameters values, fitness functions, and controllers types. For an evolutionary parameter like the population size, mutation rate, and number of elites to copy from one generation to the next, we can discretize the interval of reasonable values, run a number of tests, and choose the combination of values yielding the best results. For other components of the evolutionary setup, for instance the fitness function, such a discretization is not obvious.

To sum up, we are faced with the following situation: We have to find good values for evolutionary parameters such as mutation rate, population size, crossover rate, number of samples per individual, etc.; we have to find a suitable artificial neural network structure and a fitness function that applies the correct evolutionary pressure. All of these aspects are interdependent and there currently exist no method for finding a suitable evolutionary setup. On the one hand this is quite discouraging given the lack of any apparent structure or approach, while on the other it leaves a lot of room for improvements to the current situation.

One way of tackling this issue is to try a number of different combinations (or just one) based on intuition, pick the best combination and adjust evolutionary parameters, neural network structure, and fitness function, based on trial-and-error and experience until a good solution is found. This is probably how the majority of task-oriented evolutionary robotics is done nowadays.

An alternative would be to apply a phased approach in which only one aspect of the evolutionary setup is treated at the time and work towards a good solution by progressive locking more and more components and parameters of the evolutionary setup. Due to the interdependence between the various aspects of the evolutionary strategy, there is no obvious order and a given methodology should ensure, to some degree, that the parameters, fitness function, and the neural network structure and type, are not locked into suboptimal values early in the process. We try to minimize this risk by first focusing on the most uncertain and predominant aspects of the evolutionary setup and only afterwards tuning the less dominant parts. We regard the most important aspects as those that require most human intuition, are hardest to find in a brute force manner, and determine if a suitable controller can be evolved or not (as opposed to how fast an acceptable solution is found). Based on these criteria, we have to determine the order for finding good evolutionary parameters (neuro-evolution method, selection scheme, mutation rate, and so on), ANN type and structure, and fitness function.

The most important and uncertain ingredient of the evolutionary setup is arguably the fitness function. The fitness function represents the element for which our understanding of the problem

[1]"Task-oriented evolutionary robotics" refers to evolutionary computation being used in situations for which a task has been defined (formally or informally) and the controller as well as the evolutionary strategy can be chosen more or less freely by the engineer or researcher. Thus, the focus on evolving a controller that can solve a specific task. This is in contrast to situations where the aim is not so much to obtain a controller, which solves a given task, but more on other aspects of evolutionary robotics, such as for instance in studies where properties of a specific type of network are investigated.

plays the largest role and ultimately should be a mathematical description of our objectives. If everything else is perfect, but the fitness function is wrong, we are highly unlikely to obtain a satisfactory behavior. The same is not necessarily true for the other aspects of the evolutionary setup; if the mutation rate, for instance, is a bit too low, it could take longer than necessary to evolve the desired behavior in the average case, and if an artificial neural network has too many or too few hidden nodes, it might overfit or underfit the problem, but still an approximation of the behavior we had in mind can be evolved.

An important issue is that we cannot determine if a fitness function exerts the correct evolutionary pressure *unless we test it*. However, we cannot test it before we have chosen a neural network type and structure and unless we try to evolve the controller under an evolutionary algorithm and set all of its parameters to some values. This chicken-and-egg issue can only be overcome by testing a fitness function with a range of different controllers and with a set of evolutionary algorithm parameters. These parameters might of course not be optimal but as long as they are relatively *conservative* in the sense that if a solution can be found, the evolutionary algorithm has a good chance of finding one (although it might take several generations more than strictly necessary). This means that every fitness function is tested with a conservative and representative selection of neural network types and structure and evolutionary parameters.

We keep testing and modifying fitness functions until we achieve satisfactory solutions meaning that we obtain controllers that receive a relative high fitness *and* solve the task that we had in mind. Once it has been verified that the fitness function expresses our intuitive idea, it is important to test if the evolved strategies are general and whether the controller can be transferred to the real robots. When a fitness function, which meet these criteria, has been found, the range of neural network types and structures is expanded and tested. When a suitable network has been found the focus is shifted to the neuro-evolution method and the evolutionary parameters. An illustration of this concept is shown in Figure 4-2, where the horizontal axis denotes time and where the width of the bar shown for each component indicates the relative number (or range) of different elements tested for the component at a point in the process.

We call the set of representative elements the *baseline set* for a given component (in Figure 4-2 the baseline sets are illustrated as the thicker bars on the left-hand side of the triangle for each component). Hence, the baseline sets consist of the elements - neural network structures, neuro-evolution methods, and evolutionary parameters - that are used for a component of the evolutionary setup before the focus is on that component. The number of elements in the baseline sets is a trade-off between speed and coverage. The more elements in the baseline sets the more combinations we test, which on one hand decreases our chance of missing a good evolutionary setup, but also requires more evolutionary runs and therefore slows down the process.

During the phase in which the fitness function is determined we do not necessarily have to test a large number of different fitness functions; if one of the first ones that we test proves to mathematically express our intention and allows for controllers to be evolved and transferred to the real robots, there is no reason to go further. Likewise, if one of the neural networks in the baseline set shows adequate performance, we do not necessarily have to test a large number of other networks. Depending on the performance requirements of a component, more or less effort can go into finding optimal settings for that component. If a robot has limited

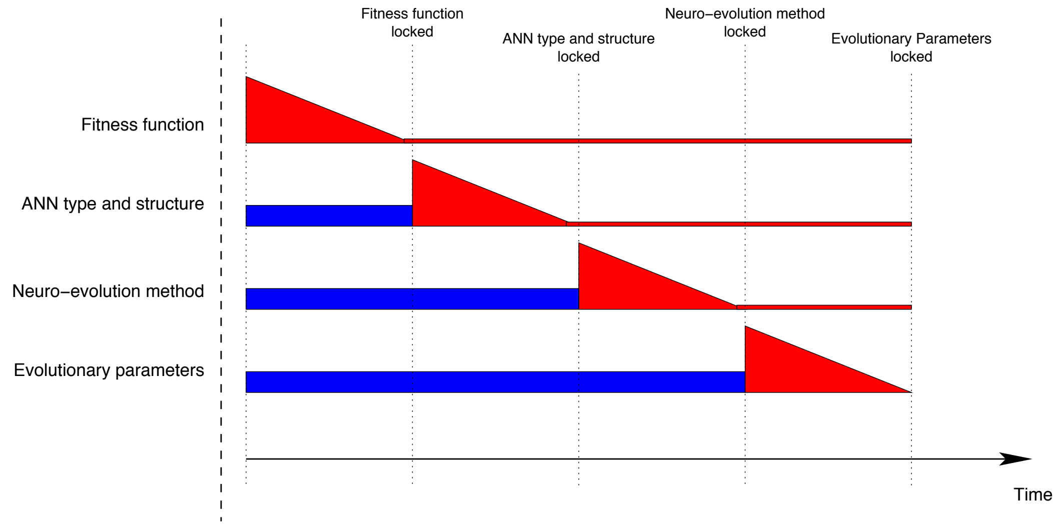


Figure 4-2: Overview of the method. First a suitable fitness function is found, by testing a large number of fitness functions with a limited number but broad span of neural network types, and evolutionary parameters. Once a good fitness function has been found the range of neural network types and structures is expanded and tested. When a suitable network has been found the focus is shifted to the neuro-evolution method and the evolutionary parameters.

processing capabilities, finding the smallest applicable ANN could be a priority, while in case several artificial evolutions have to be done in the future, energy would be better spent tuning the evolutionary parameters. Thus, the triangles on Figure 4-2 do not indicate that a large number of settings *have* to be tested for each element, but simply that the component is in focus during that phase and that elements not in the baseline set can be tested if necessary.

4.1.2 Summary

There currently exist no method or structured approach for performing artificial evolution of robot controllers. We have motivated a phased approach in which the evolutionary setup becomes progressively more specific. First we develop, test and modify fitness functions until a suitable one is found, thereafter we find an applicable neural network, and finally we focus on the neuro-evolution method, and tune its parameters.

4.2 Experimental Setup

In this section we describe our experimental setup, the configuration of the s-bot, and the sensors that we use, mathematical notation, and some general notes on how we conduct our artificial evolutions. Finally, we define the baseline sets, that is the neural networks, neuro-evolution methods, and the parameters for the chosen neuro-evolution methods in the baseline set.

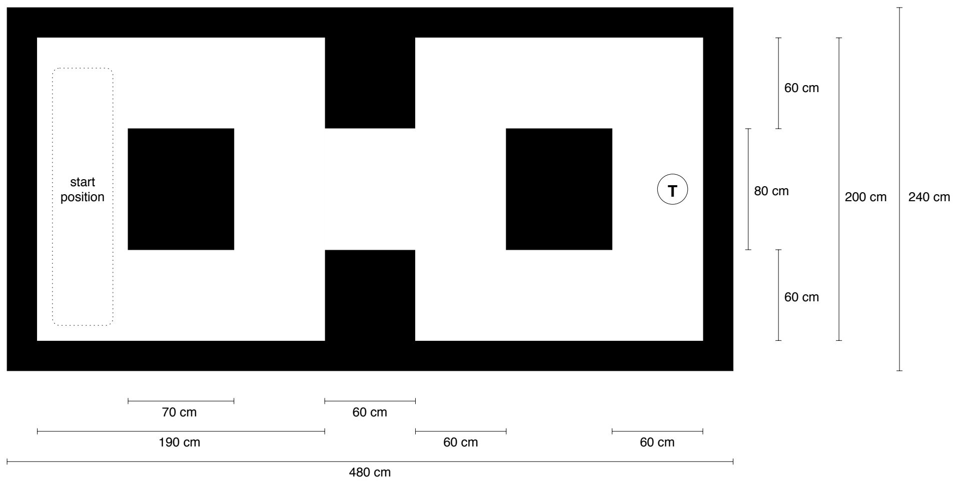


Figure 4-3: The arena: The swarm-bot start in the marked arena to the left and has to reach the target marked by the **T** on the right-hand side. The white areas represent the parts of the arena on which the swarm-bot can move, while the black areas represent holes.

4.2.1 The Arena

A sketch of the main arena is shown in Figure 4-3; the arena is shaped like an "8", where the swarm-bot starts in area marked "start position" in the left-hand side of the arena and has to move to the target area marked by a **T** on the right-hand side. A light source is located above the target area. The white areas on the figure represent the parts of the arena on which the swarm-bot can move, while the black parts represent holes.

In this chapter we are going to use other arenas as well. However, the arena shown in Figure 4-3 has a special importance as it can be built the facilities available at our lab. We have chosen it as our main arena and we use it for the tests on the physical robots described in the end of this chapter.

4.2.2 The S-bot's Sensors

Below is a description of the sensors on the s-bot, which we use:

Light sensors: The target area, to which the s-bots should move, is marked by a light source that can always be detected by all the s-bot. 8 light sensors are positioned evenly on the turret of each s-bot. In Figure 4-4(a) two light sensors can be seen under the semi-transparent rubber band next to the gripper. The reading of a light sensor is a a value indicating to the amount of light detected in the corresponding direction. Each light sensor is connected to an input neuron in the artificial neural networks used and the light sensors therefore provide a total of 8 inputs neurons.

Traction sensor: The traction sensor enables s-bots to sense forces acting on the turret, hence when they are being pushed and pulled by other s-bots. The traction sensor allows s-bots to coordinate their movement when they are in swarm-bot formation. The traction measures the difference in terms of force between the chassis and the turret in the two dimensions of the horizontal plane, and provides four inputs to a neural network: A positive and a negative force in each of the dimensions.

Ground sensors: Four ground sensors are mounted under each s-bot; two are inclined pointing $45°$ forward and backward, respectively, while the remaining two ground sensors point straight downwards to the ground under the s-bot. In Figure 4-4(a) three ground sensors can be seen between the left and the right treel, namely the inclined ground sensor in the front and the two sensors pointing straight down. The ground sensors measure the distance to the ground by emitting infrared light and recording the amount reflected. A hole or a dark surface will cause the ground sensors to record very little or no reflected light. We have hardwired the ground sensors so that when a hole is detected a sound is played, which allows the other s-bots to sense that a hole has been seen by one of the s-bot in the swarm-bot. Initial experiments have shown that playing a sound whenever a hole is detected greatly improves the evolvability and performance of controllers for hole avoidance [Trianni et al., 2004a]. Each of the ground sensors corresponds to an input to a neural network, and thus the ground sensors provide a total of 4 inputs.

Sound sensor: The s-bots are equipped with microphones and different sound frequencies can be distinguished real-time, however since we need only to discriminate between sound and no sound (one or more s-bots currently detect a hole, no s-bot detects a hole) our sound sensor is binary. Thus, the sound sensor provides only one input.

Rotation sensor: The turret of an s-bot can rotate a total of $360°$ from one extreme to the other with respect to the chassis. When an s-bot needs to turn, e.g. when a hole is encountered, it is only be able to turn in one direction if the turret is rotated to one of the extremes. Thus, we assume that the rotation of the turret with respect to the chassis at a given moment is a valuable input for the robot controller. The rotation sensor provides two inputs; one for the rotation from the initial position to each of the extremes.

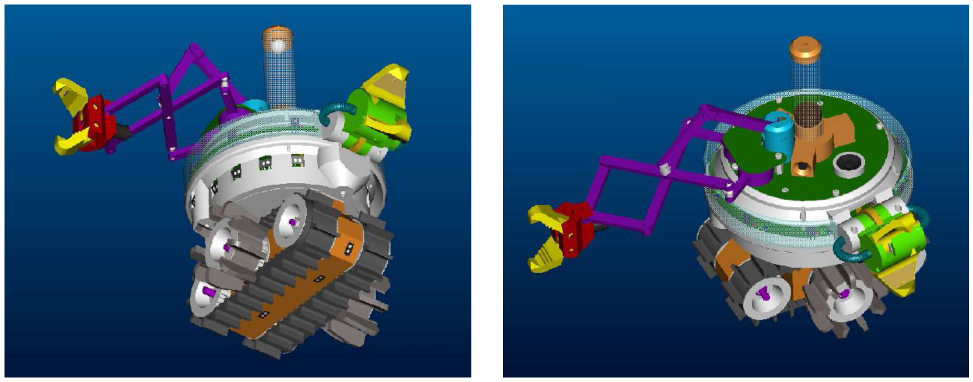

(a) (b)

Figure 4-4: The figure shows a 3D rendering of the s-bot seen from below (a) and from above (b).

Since we are using the 5 types of sensors described above, we have a total of 19 sensory inputs (8 from the light sensors, 4 from the traction sensor, 4 from the ground sensor, 1 from the sound sensor and 2 from the rotation sensor). As the s-bots start off in swarm-bot formation we do not have to control the gripper during our experiments and therefore we are limited to controlling the differential treels. Thus, we only have two actuator outputs, one the left and one for the right treel.

4.2.3 Running Evolutions

When evolutions are conducted it is common to evaluate each individual a number of times each with different initial conditions in order for the assigned fitness score to better reflect an individual's general performance. Thus, when an individual is evaluated it undergoes multiple *trials*. If each individual only undergo a single trial, the assigned fitness reflects only how well each individual performed from a single starting condition, while when more trials are used the final fitness score tends to reflect an individual's *general performance* better. However, the number of trials per individual is a trade-off between accuracy and speed. Unless otherwise stated, we use 5 trials for each individual and let the individual's fitness score be the average of the fitness scores obtained during all trials. Alternatively, one could use for instance the fitness score from the worst or from the best trial.

Evolutions are based on a random generator, in the sense that the initial population, mutation, crossover, selection, and so on, are partly based on random numbers. Thus, like multiple trials are required to evaluate an individual, multiple evolutionary runs are necessary in order for us to be able to draw any conclusions about the feasibility and performance of a given method and of a set of evolutionary parameters. Since our goal is to investigate and compare various methods and parameters, we are required to collect data from multiple evolutions for each setup with different initial conditions (different initial gene-pools and random generator seeds). In the field of evolutionary robots it is custom to conduct 10 or more evolutions for each evolutionary setup.

4.2.4 Fitness Functions and Notation

We use the mathematical notation described below for the concepts such as simulation step, time and position, to express fitness functions.

Time: Each individual is evaluated over a finite period of t_f simulated seconds. We let t_i denote a time in the interval $0 \leq t_i \leq t_f$, where t_f is the evaluation time limit for each trial for every individual. We let $t_0 = 0$ and let Δt denote the time between each discrete sense-act-update cycle (or simulation step) as described in Section 3.2.1 on page 34.

Simulation step: We let S denote the ordered list of all simulation steps, s_i, where s_0 corresponds to the initial step and s_f to the last simulation step, respectively. The time between each simulation step is Δt with s_0 being the first simulation step at time t_0.

Positions: The spatial position of a swarm-bot in the virtual world can be described by a two-dimensional vector. We let the position of a swarm-bot correspond to the position of its center of mass. We let $\mathbf{p}$ denote the position of the swarm-bot and $P = (\mathbf{p}_0, \mathbf{p}_1, \ldots, \mathbf{p}_f)$ be an ordered list of all positions reached at simulation steps $s_0, s_1, \ldots, s_f$, respectively.

Thus, to each simulation step, s_i, we can associate a unique time t_i and a position of the swarm-bot $\mathbf{p}_i$.

4.2.5 Baseline Sets

In this section we present and discuss the baseline sets for neural types and structures, neuro-evolution method, and evolutionary parameters, which have been selected.

Neural Network Types and Structures

One could argue that our task can be solved by a purely *reactive* agent, since the s-bots never loose sight of the target area and because it should never be necessary to back-track in order to solve the task. If a controller is reactive it means that any action performed is only based on the latest set of inputs (e.g., sensory readings). Thus, a reactive agent is *memory-less*, no context-units are needed and it should suffice to use a feed-forward network to control robots for such a task. Even so, it is unclear what type of network to use: A single layer perceptron or a multilayer perceptron. In case we choose a multilayer perceptron we also have to decide on the number of hidden neurons for the network. There are no theories or generally accepted rules of thumb regarding a suitable number of hidden nodes. A multilayer network with too few hidden nodes is prone to underfit a given problem, while a network with too many hidden nodes is likely to both be harder to evolve, given the increased number of weights to optimize, and to overfit the problem. Therefore, we have tried a number of different multilayer configurations going from a simple network where the number of hidden nodes equals the number of outputs (2) to a medium network with 10 hidden nodes and to a complex network, where the number of hidden nodes equals the number of inputs (19). In our baseline set of neural networks, we have the following four networks:

1. A single layer perceptron with 19 inputs and 2 outputs - a total of 40 weights to optimize.
2. A multilayer perceptron with 19 inputs, 2 outputs and 2 hidden nodes - a total of 46 weights to optimize.
3. A multilayer perceptron with 19 inputs, 2 outputs and 10 hidden nodes - a total of 222 weights to optimize.
4. A multilayer perceptron with 19 inputs, 2 outputs and 19 hidden nodes - a total of 420 weights to optimize.

A priori we have no way of knowing if this selection of ANNs is sufficient for the task; maybe a perceptron controller can solve the task satisfactory or maybe a multilayer perceptron with more than 19 hidden nodes is needed. Some initial experiments have, however, indicated that the task is solvable by some evolved controllers using some of the network described above.

Neuro-evolution Method and Evolutionary Parameters

The baseline set of neuro-evolution methods consists of three elements, which are all classic genetic algorithms, with rank-proportional selection, elitism, and one-point crossover. They differ only in terms of mutation rate. The evolutionary parameters are the following:

Population: Generations are of 100 individuals each.

Selection: A rank-proportional selection scheme is used and the best 5 individuals from a generation survive to the next (elitism).

Mutation: The mutation rate is set to 1%, 5%, and 10%.

Crossover: One-point crossover between parents is used.

In the neuro-evolution method phase, we experiment with other neuro-evolution methods as well.

4.3 Fitness Function

In this phase we attempt to find a fitness function that applies an evolutionary pressure and evolves controllers, which perform phototaxis and hole-avoidance. It should not be too hard to bootstrap the evolution and the evolved controllers should be transferable to real robots.

We start off with a simple fitness function and modify and extend it as necessary.

4.3.1 A Simple Fitness Function

Since we want the swarm-bot to reach the target area, a natural way of shaping the fitness landscape to avoid bootstrapping problems, is by rewarding individuals depending on how close they get to the target area. For this purpose we use the following fitness function:

$$f_{light} = \begin{cases} 1 - \frac{\min_{\mathbf{p}_i \in P} |\mathbf{p}_i - \mathbf{l}|}{|\mathbf{p}_0 - \mathbf{l}|} & \text{if target area is not reached;} \\ 2 - \frac{t_\mathbf{l}}{t_f} & \text{if the light is reached at some time } t_\mathbf{l}, \end{cases} \tag{4-1}$$

where $\mathbf{l}$ denotes the position of the light source and $|\mathbf{v}|$ the length of the vector $\mathbf{v}$. Thus, $|\mathbf{p} - \mathbf{l}|$ denotes the distance between some position of the swarm-bot reached during an evaluation and the position of the light source. Similarly, $|\mathbf{p_0} - \mathbf{l}|$ denotes the initial distance from the swarm-bot to the target. The fitness function f_{light} shown above, has two different parts; one concerning individuals who do not reach the target area marked by a light, and one concerning individuals who reach the light source. For the former part, individuals are scored based on the minimum distance they get within the light source, while in the latter case individuals are rewarded for how quickly they reach the target area. If the target area is reached we stop the evaluation of the individual in question and let $t_\mathbf{l}$ be the simulation time at which the target was reached.

Individuals who do not reach the target area will therefore have a fitness score in the interval $[0, 1[$, while individuals who reach the target area will have a fitness score in the interval $[1, 2[$.

Each individual undergoes a number of trials each with an evaluation time limit, t_f set to 4 minutes, hence each individual was given maximum 240 virtual seconds or 2.400 control cycles to reach the target area.

For each of the ANNs in the initial set listed in Section 4.1 we ran 10 evolutions with different initial random seeds and recorded the best, worst, and average fitness score for each generation.

Results

The results of the evolutionary runs varied in the sense that different strategies where found for runs with different initial random seeds. A few the evolutionary runs resulted in controllers relying on promising strategies. An example is shown in Figure 4-5(a). However, the majority of the evolutionary runs got stuck in a local optimum, where the swarm-bot would avoid the first hole and move as fast as possible towards to the second hole in front of the light and fall in. An example of this behavior is shown in Figure 4-5(b).

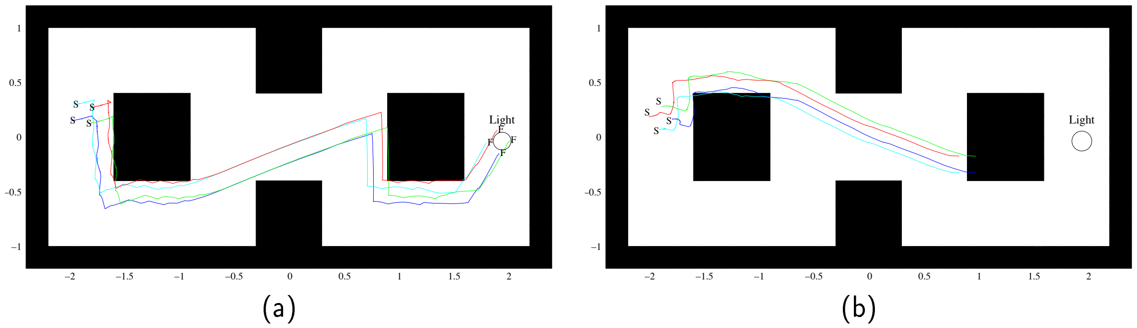


Figure 4-5: Examples of behaviors evolved using the fitness function, f_{light}. (a) shows a successful strategy, where the target area is reached, while (b) shows an example of the result similar to the majority of the evolved controllers. A few evolutions resulted in successful strategies, while it was common for evolution to get struck in local optima with a resulting behavior similar the one shown in (b).

4.3.2 Escaping a Local Optimum

The fitness function, f_{light}, used in the previous section represents our objective, namely that the swarm-bot should move towards the light source, but does not shape the fitness landscape in such a way that a good solution can be found easily. Therefore, we have to change the fitness function or add components, which change the fitness landscape and its features to make it easier for an evolutionary algorithm to find good solutions. It would be convenient if we could do so automatically, but to our knowledge no systematic or automatic method exist for evolutionary computation applied to the domain of mobile robotics. Hence, we have to use our understanding of the task and decompose it into a set of sub-tasks.

One of the problems with the relatively simple fitness function used in the previous section was that the evolutionary algorithm often ends up in a local optimum, because the swarm-bot would fall into the hole just in front of the light. Hence, a first step would be to extend the fitness function with a component that assigns a lower fitness to individuals who fall into holes. We shall call this fitness function $f_{stayalive}$:

$$f_{stayalive} = \begin{cases} 0.5 & \text{if the swarm-bot falls into a hole during the evaluation,} \\ 1.0 & \text{otherwise.} \end{cases} \tag{4-2}$$

This fitness function rewards individuals who stay clear of holes. When we combine it with the fitness function, f_{light}, which rewards individuals depending on the minimum distance to a light source obtained during a simulation run, we get the following fitness function:

$$f_{lightandstayalive} = f_{light} \cdot f_{stayalive} \tag{4-3}$$

Thus, we simply multiply the two fitness functions to obtain the third.

Results

The majority of the evolutionary run yielded promising results using the composed fitness function $f_{lightandstayalive}$. Figure 4-6(a) shows a typical strategy found by evolution, where the swarm-bot is able to move from its initial location to the target area while avoiding holes. Figure 4-6(b), on the other hand, shows the performance of the same controller as in (a), but in a different arena. As can be seen from the result, evolution has found a strategy that is *specialized* for the arena in which the controller has been evolved. This is a problem since the controller does not employ a strategy general enough to solve the task in slightly different settings. We discuss this issue and why it is important to evolve more general controllers in the next section.

4.3.3 Generality and Transfer of the Controller

We would like to evolve controllers relying on general strategies and the controllers should be transferable to physical robots. No simulator is perfect in the sense that it simulates reality with complete accurately, and therefore we must be aware of the relevant discrepancies between simulation and the real world.

If we only score individuals in the arena shown in Figure 4-6(a) we are prone to evolve controllers, which are highly specialized for this arena. In order to counter this the swarm-bot is positioned and oriented differently in the start area at the beginning of each trial[2]. However, in this case it appears not to be enough to evolve general strategies. An example of the performance of an evolved multilayer neural network controller with 10 hidden nodes using the $f_{lightandstayalive}$ fitness function can be seen in Figure 4-6. Notice how the evolved controller correctly steers the swarm-bot to the target area in the arena where it was evolved (a), while for the new

[2]All individuals from a given generation are evaluated under the exact same conditions, thus for a given trial all individuals are positioned and oriented in exact the same way.

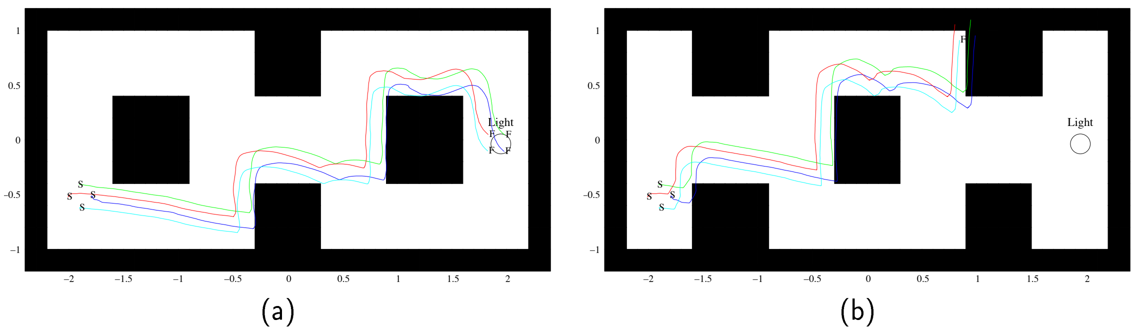


Figure 4-6: A controller evolved in the virtual arena shown in (a) can successfully steer a swarm-bot from the starting arena to the light source in (a). When the same controller is tested in a different arena (b) it fails and steers the swarm-bot over the side of the arena. Hence, the controller evolved for arena (a) appears to be highly specialized for (a) even though the initial conditions such as random positioning and orientation have been used throughout the evolution of the controller.

arena (b) it fails and steers the swarm-bot out of the arena. In particular notice how in (b) the swarm-bot is steered *away* from the light before it falls over the side of the arena. Thus, the strategy found by artificial evolution appears to be highly specialized for the arena in which evolution took place.

Obviously, we would like to evolve controllers that use more general strategies. Since there will always be differences between simulation and reality, a controller with a more general strategy is easier to transfer successfully to physical robots. A controller that uses a general strategy is not as prone to rely on features specific to a given arena, which might be different in a real arena, than a specialized controller. Moreover, if we wanted a controller that only has to work for a single arena, we would probably be able to develop one by hand faster.

In order to evolve a robust controller, which relies on a general strategy, we have to apply an appropriate evolutionary pressure. This can be done by evaluating each individual in several arenas instead of a single arena only [Thompson, 1998]. We have chosen the four arenas shown in Figure 4-7. These arenas should be different from one another to a degree, which makes evolution favor general strategies over controllers that are specialized for one of the arenas only.

The four arenas are used in the following manner: When a controller is being evaluated it is evaluated twice in each of the arenas and its fitness is recorded over a period of 240 seconds in simulation. For each trial the fitness is recorded, and the fitness used by the evolutionary strategy to determine an individual's rank is then the average of the fitness scores for all 8 trials[3].

[3]When we use multiple arenas for the fitness computation evolution can get stuck in a local optimum where all the individuals in the population specialize for one of the arenas and fail in the others. Thus, evolution quickly finds a strategy that works well for one of the arenas but does not generalize to other arenas. If only two arenas are used, we experienced this quite often, whereas if we use three or more arenas, this is less likely to happen. If we use the minimum fitness obtained instead of the average, we can avoid this specialization.

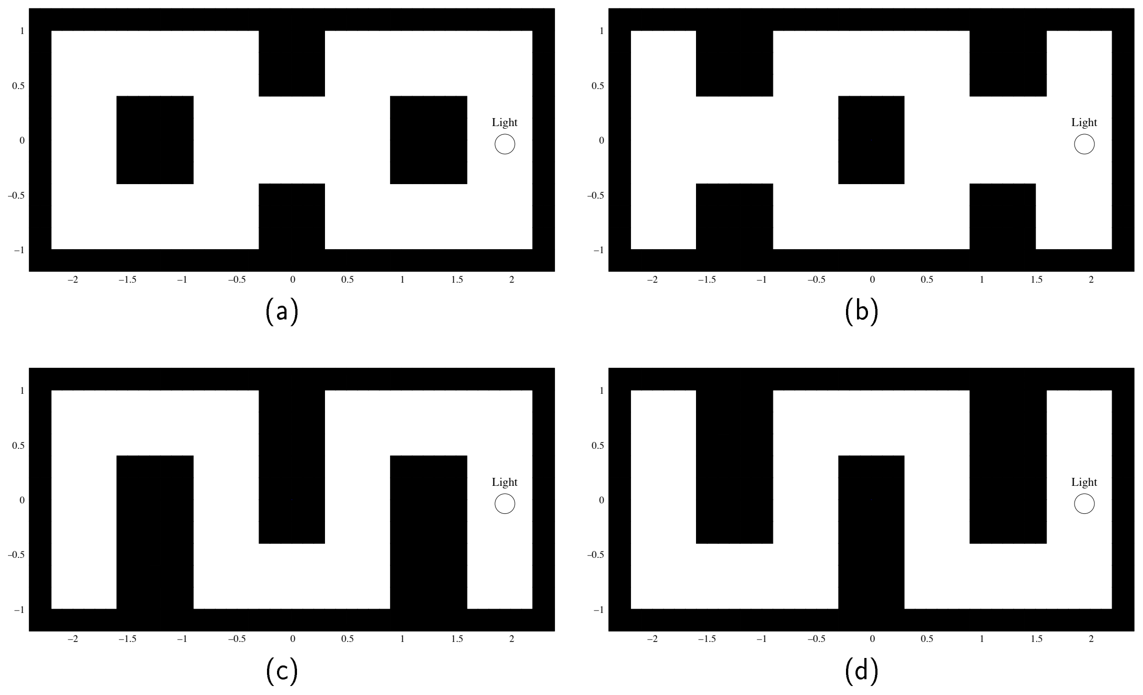


Figure 4-7: Four arenas used to evolve general controllers. When a controller is evaluated, it receives a fitness that reflects its performance in all of the arenas. Thus, in order to obtain a high fitness, a controller must implement a general strategy, which works in the most or all arenas.

A typical strategy found by evolution can be seen in Figure 4-8. This result indicates that evolution has indeed found a general strategy, namely: Go north until a hole is found and then follow the edge until the light is reached. This strategy is similar to the simple maze-solving strategy for mazes without loops: Put your hand on the left or right wall and follow the wall.

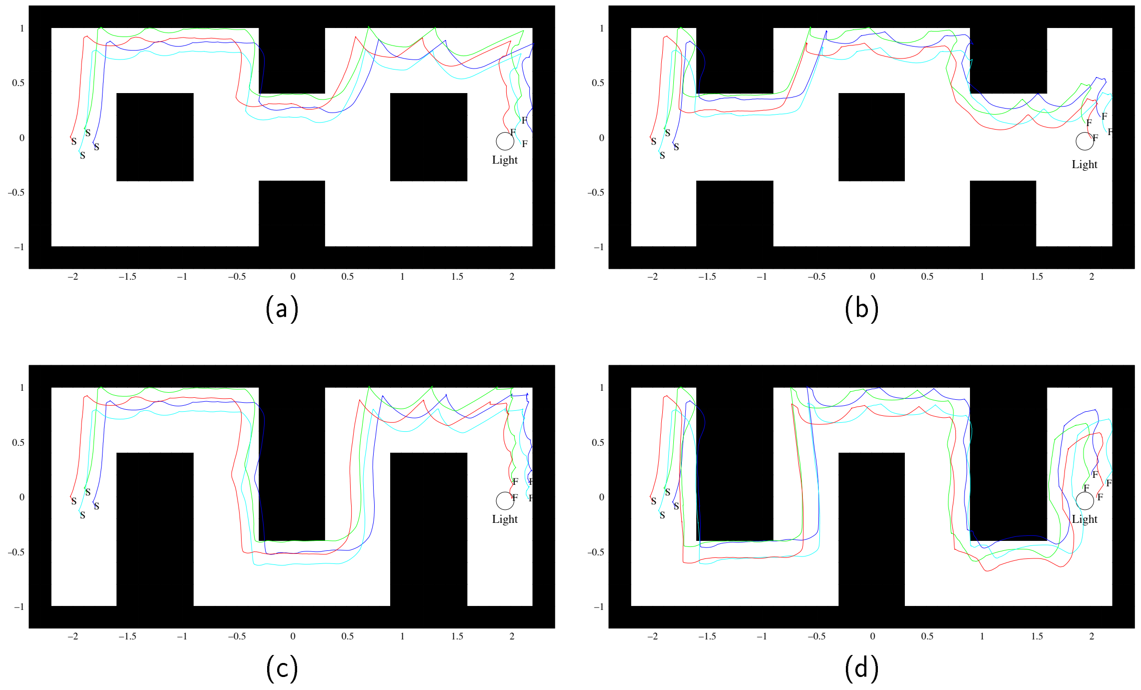


Figure 4-8: Example of how a controller behaves in four arenas using a simple, but general and successful, hole-following strategy. During evolution of the controller, each individual was tested in each of the arenas twice and the average of the fitness scores for all the arenas makes up the individual's final fitness.

Coordination and Discrepancies

Recall that we made the assumption in our simulator that connections between s-bots are completely rigid. This is obviously a crude approximation of reality. Our initial attempts to transfer controllers to real robots indicated that the difference between the physical behavior in simulation and reality is an issue that requires special attention. Basically, the physics of the simulator enables evolution to exploit some subtleties in the virtual world in a way which is not possible in the real world - or at least requires much more detailed models to simulate accurately. An example of the behavior of a controller, which uses a relatively successful strategy in simulation, is shown in Figure 4-9. In reality connections between s-bots are not rigid, the treels are not completely even, patches of dust change the friction, and so on. Not all of these novelties can be compensated for by adding noise to the sensors, actuators, and to the physics.

However, bootstrapping can then become problematic. In our case, assigning the fitness score based on the average performance in the 4 arenas proved to work well.

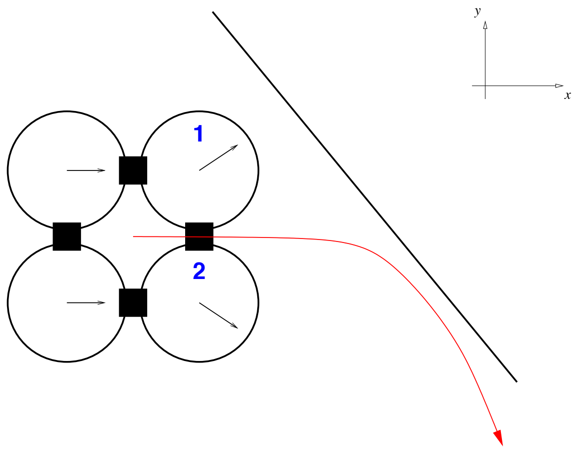


Figure 4-9: An example of how the physics and the rigid connection assumption can be exploited by evolution so that controllers, which cannot be transferred to real robots, are evolved. The figure shows a swarm-bot consisting of four s-bots currently moving from left to right. The arrows drawn inside each of the s-bot denote the direction of the treels for that s-bot and hence of the force exerted on the swarm-bot by the s-bot. The treels of s-bots 1 and 2 are not completely aligned with the direction of motion; s-bot 1's treels point slightly upwards, while s-bot 2's treels point slightly downwards. Since they cancel each other's movement out in the y-dimension, the swarm-bot only moves in the x-dimension. However, once the edge is reached s-bot 1 is the first s-bot to be without surface contact, and its treels therefore no longer exert any force on the swarm-bot, and as a result s-bot 2 pulls the swarm-bot downwards, away from the hole as the trajectory arrow shows. Although this strategy works relatively well for avoiding holes in simulation, it is not directly applicable in practice since s-bots pulling each other in the manner required by the strategy results in s-bots tipping, sliding, and even slight jumping across the surface. Thus, symmetries and basic assumptions, such as the treels of the s-bots always being in contact with the surface, can result in evolution finding non-transferable solutions.

When we attempted to use the controller, whose behavior is shown in Figure 4-8, which partly relies on the strategy for avoiding holes shown in Figure 4-9, some of the differences between simulation and reality became evident. A photo of four s-bots in swarm-bot formation steered by the controller is shown in Figure 4-10. Notice how the treels of the s-bot to the far right are not in contact with the ground surface. This happens because the other s-bots are moving in a different direction and therefore are pulling the s-bot to the far right effectively tipping it. While this is a common situation in reality, in simulation the treels are assumed to always be in contact with the ground surface unless they are currently over a hole.

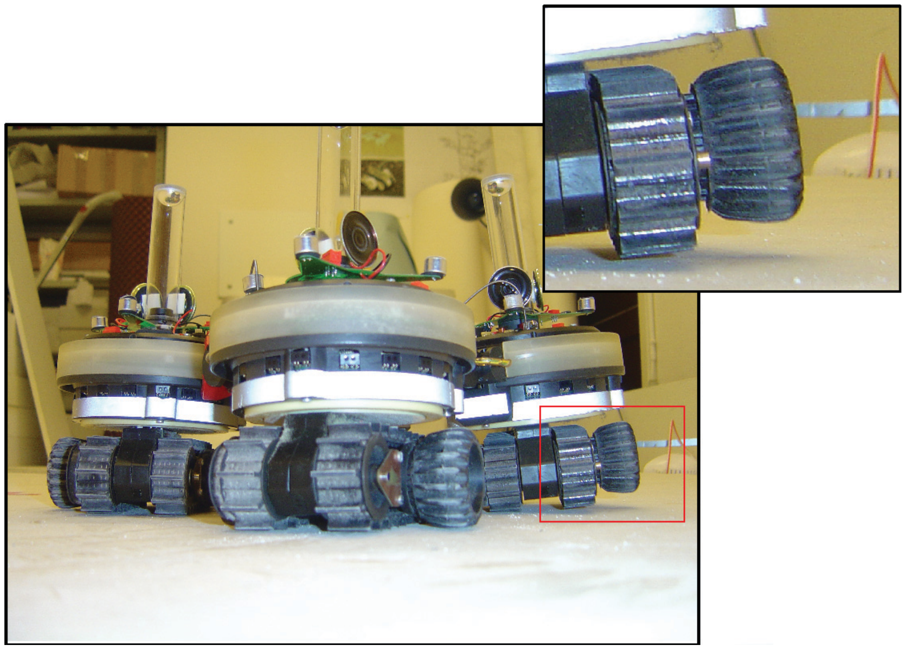


Figure 4-10: An example of a difference between simulation and reality. In simulation s-bots are assumed to always have the treels on the ground, whereas in reality it can happen that an s-bot is tipped and one treel looses contact with the surface due to the forces of other s-bots.

There are two ways in which the issues resulting from the discrepancies between simulation and reality can be overcome, namely to build more accurate simulators or to ensure that the artificial evolution does not rely on features valid only in simulation. It is questionable if it would be feasible to build simulators so accurate that strategies like the one shown in Figure 4-9 could be directly and successfully transferred to real robots. First of all it would require a huge effort to model every part of an s-bot in high detail, implement the control logic, and finally verify and fine-tune the implementation. Secondly, if it is possible at all to build a simulator with a complexity sufficiently high, its performance would likely be so low, that it would be impractical to run artificial evolutions using it. In any case, none of the simulators developed during the life-time of the Swarm-bots projects have attained this accuracy.

The second solution, namely to steer evolution towards solutions not relying on inaccurate and/or unrealistic features, seems to be a more plausible solution. In order to avoid that evolution find solutions that rely on rules only valid in the virtual world, we extend the fitness function with a component that promotes coordinated-motion through minimization of traction. This approach to evolving coordinated-motion is similar to Trianni et al. [2004a]. The traction sensor outputs four values between 0 and 1, where 0 means no traction and 1 means that the maximum traction force perceivable is perceived. Thus, in order to minimize the traction between s-bots in a swarm-bot formation, we have to minimize the highest traction perceived by any s-bot for each control cycle. If we let τ_i^{max} denote the maximum traction detected by any s-bot at simulation step s_i, then the fitness function for minimizing traction is defined as follows:

$$f_{minimizetraction} = \frac{\sum 1 - \tau_i^{max}}{|S|} \tag{4-4}$$

Thus, the sum over 1 minus the highest traction perceived by any s-bot for every simulation step, divided by the total number of simulation steps, $|S|$.

When we combine this fitness function with our previous fitness function, we obtain our final fitness function, which allows artificial evolution to find transferable strategies:

$$f_{final} = f_{light} \cdot f_{stayalive} \cdot f_{minimizetraction} \tag{4-5}$$

Thus, our fitness function consists of three components: Individuals are rewarded for moving towards the target area marked by a light source, they are rewarded for not falling into holes, and finally for minimizing traction[4].

4.4 Neural Network Type and Structure

In this section we investigate the performance of various neural network types and structures with the intent of finding *the simplest network which can be evolved to successfully solve the task*. Whereas for some functions, like the binary exclusive-or function, a formal argument can be given that they cannot be represented by certain types of networks (perceptrons), in evolutionary robotics we often have little or no prior knowledge as to what strategies evolution might find - let alone what is necessary for solving a given task. Therefore, we cannot, in the general case, formally prove that a neural network of a certain complexity is required to solve a given task. As a result we are neither able to make any claims that we have found *the* simplest network capable of solving a task (unless a single neuron, the simplest neural network of all, is shown to be capable of solving the task of course). However, by prioritizing *evolvability* and *simplicity* we have two metrics by which the relative performance of different networks can

[4]It is important to take note of the fact that by introducing the last component, $f_{minimizetraction}$, we also limit evolution from exploring some strategies which do not necessarily require traction to be low. Some of these strategies not explored might have been better than the ones found with this fitness component introduced, but since evolving transferable controllers relying on such strategies requires very complex simulators, the component represents a practical measure, which allows us to evolve controllers that do solve the task and which are transferable.

be evaluated. In this way the metrics, evolvability and simplicity, should be understood in a practical context as opposed to a theoretical one.

In order to find a suitable neural network structure, we start with the structures from the baseline set (see Section 4.2.5 on page 59). That is, a single layer perceptron, and three multilayer perceptrons with 2, 10 and 19 hidden nodes, respectively. To determine their performances, we have, for each the four types of neural networks, run 20 evolutions of 1000 generation, with three different mutation rates as described on page 60. Thus, for each combination of neural network structure and mutation rate in our baseline set, we have results of 20 evolutions, and we are interested in comparing their performances. The performance of an evolutionary run is determined by the average of the fitness scores obtained by the *highest ranking individual of the last generation (1000th) during 100 post-evaluation runs*. In a post-evaluation run an individual is evaluated in the same manner as individuals are during evolution, however, by subjecting the individual to a higher number of trials (100 during the post-evaluation run versus 8 during evolution) a more accurate measure of its general performance is obtained.

We use Friedman's test to compute statistical significance of our results and to determine which neural network yields the best results [Conover, 1999]. The Friedman test is nonparametric and only requires observations to be mutually independent and rankable according to some criterion to be applicable. Our observations, that is fitness scores obtained during post-evaluation runs, are mutually independent and can be ranked according to the fitness scores obtained. One of the benefits of using the Friedman test as opposed to One-way ANOVA (ANalysis Of VAriance) and Tukey's test [Montgomery, 2001], is that no assumptions need to be made about the distributions underlying the phenomenon of interest. Moreover, the Friedman test only uses the relative ranks of the observations and therefore no interpretation or assumptions concerning the fitness scores' numerical values are necessary.

The ranking works as follows: For each evolutionary setup, E_i, we have 20 *blocks*. Each block corresponds to a starting condition (initial random seed). We can arrange the post-evaluation results in a matrix:

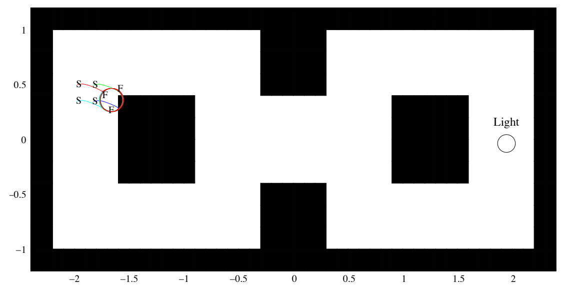
$$\begin{array}{ccccc}
\textit{block} & E_1 & E_2 & \dots & E_n \\
1 & f_{11} & f_{12} & \dots & f_{1n} \\
2 & f_{21} & f_{22} & \dots & f_{2n} \\
3 & f_{31} & f_{32} & \dots & f_{3n} \\
\dots & \dots & \dots & \dots & \dots \\
m & f_{m1} & f_{m2} & \dots & f_{mn}
\end{array}$$

where f_{xy} is the post-evaluation fitness obtained from evolutionary setup E_x with initial starting conditions y. The ranking is then done on a per-block basis (per-row basis in the matrix shown above). Hence, each evolutionary setup has a total of 20 ranks since we have performed 20 evolutionary runs for each setup. The Friedman test then performs a statistical test on these ranks. For each block rank 1 is assigned to the evolutionary setup with the highest performance for that block, while rank 2 is given to setup with the second highest performance, and so on. We have used the software package R to perform the statistical tests. For further information on R we refer to following literature: Ihaka and Gentleman [1996], Ripley [2001], Dalgaard

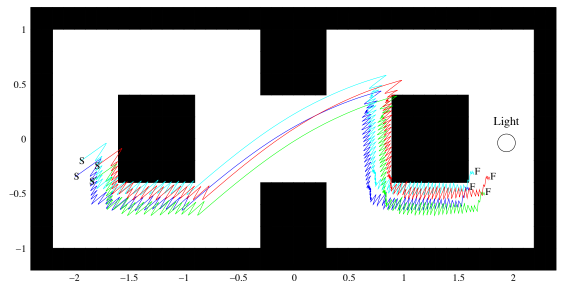
[2002], Maindonald and Braun [2003], and Verzani [2005].

4.4.1 Results

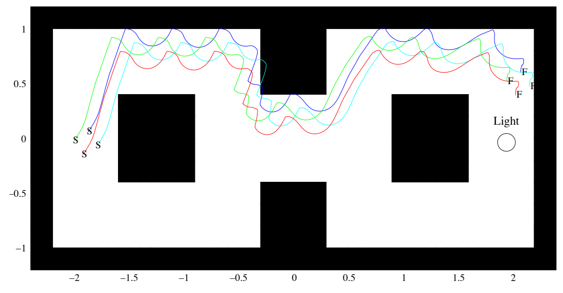
Before we analyze the fitness scores for the different network structures, we take a look at the strategies found by evolution. The highest performing controllers of all the evolutionary runs can be divided into three groups according to the strategy they employ. Examples of the three strategies are shown in Figure 4-11.



Fail: A strategy where the s-bots successfully coordinate and start moving, but as a hole is detected they start rotating about their center of mass and fail to coordinate their movements in a new direction. Thus, the swarm-bot gets stuck close to a hole and fails to reach to target area.



Reverse: A strategy in which motion is reversed completely when a hole is detected. Due to the reactive nature of the neural networks used this causes a “bouncing” behavior, since the s-bots move towards the light as long as none of them detect a hole. During the time the hole is detected, the swarm-bot reverses. This causes the path of the s-bots to resemble saw teeth in the regions where the swarm-bot is close to a hole.



Turn: A strategy that relies on changing direction, however, the direction is not completely reversed as it is the case with the strategy described above, but instead the all the s-bots turn away from the hole until it is no longer detected. This causes the trajectory to resemble semi-circles, when the swarm-bot is close to the edge of a hole.

Figure 4-11: Examples of the three different strategies found by evolution:*fail*, *reverse*, and *turn*.

The three strategies, which we denote *fail*, *reverse*, and *turn*, all represent local optima on the fitness landscapes for the network structures. The *fail* strategy is an undesirable local optimum that results in controllers, which do not solve the task. The fitness scores of such controllers reflect their inadequacy by being low. Controllers relying on one of the two other strategies, namely the *reserve* and the *turn* strategy, are capable of solving the task and the fitness score for a given controller does not alone reveal which of the two are used by the controller. Although this could indicate that the two strategies are *equally good*, since they result are comparable in terms fitness scores, their performances on physical robots are different. Controllers implementing the *reserve* strategy do not transfer as easily as controllers implementing the *turn* strategy. Recall that in simulation the control cycles are synchronous, which is not the case on real robots, moreover the speakers, microphones, and the frequency analysis code proved not to be as reliable as anticipated, and therefore the synchronous behavior, on which the *reverse* strategy relies, is non-existent in reality. On real robots the performance of controllers based on the *reverse* strategy is therefore not as good as controllers based on the *turn* strategy. Had this issue been known during the development of TwoDee or during the engineering of the fitness function, we might have taken measures to prevent such strategies from being explored by evolution. However, we can never guarantee a total correspondence between simulation and reality, and therefore we cannot prevent that evolution sometimes find solutions that are based on features existing in simulation only.

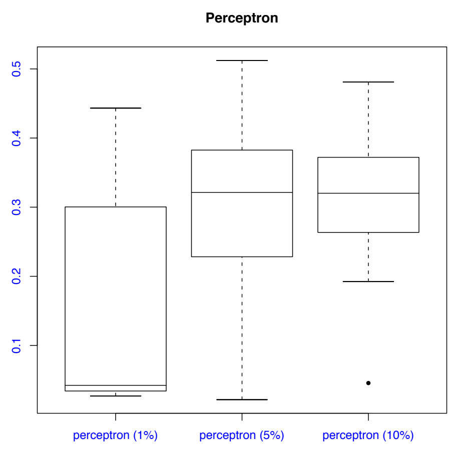
Analysis of Fitness

Box-plots for each of the neural network structures are shown in Figure 4-12. The figure contains four box-plots, one for each neural network structure, and each of the box-plots contains three populations, one for each of the different mutation rates tested.

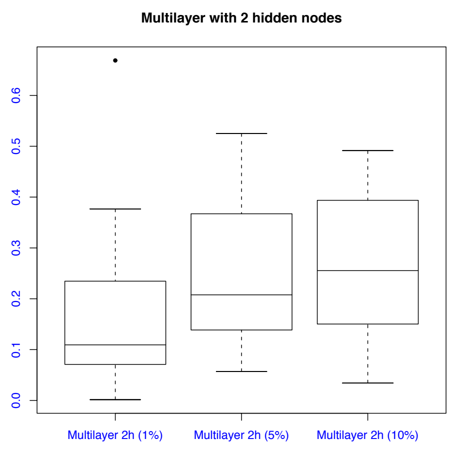
Figure 4-13 is a Friedman box-plot of the results for all of the neural network structures and mutation rates. The results have been ordered according to the average ranking of the evolutionary setups and statistical insignificance between the them is indicated by the solid vertical lines on the left-hand side of the figure (those evolutionary setups covered by the same vertical line are not significantly different). The results shown in the figure suggest that the simpler networks perform quite well. More specifically, the multilayer network with 2 hidden nodes and the perceptron appear to be adequate in solving the task, and clearly outperform the largest network structure in our baseline set, namely the multilayer network with 19 hidden nodes. Figure 4-13 also suggests that a mutation rate of 1% is too conservative to move the standard genetic algorithm with crossover and rank-proportional selection to regions of the fitness landscapes containing successful strategies as successfully genetic algorithms with higher mutation rates.

Based on the results shown Figure 4-13 a perceptron seem to suffice for solving the task while being the simplest network structure that we have tested. We might be conservative and pick the multi-layered neural network with 2 hidden neurons, whose average performance was measured to be slightly higher than that of perceptrons, although there is no strong statistical evidence that it does in fact perform better than perceptrons in general.

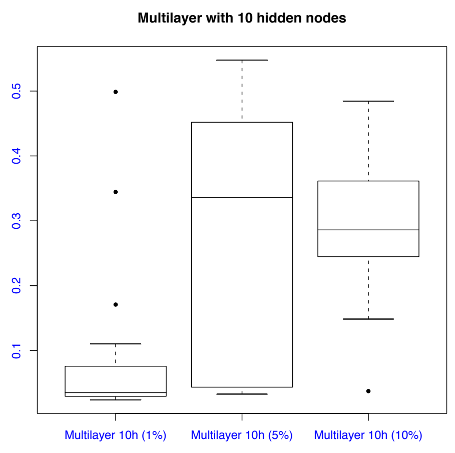
As mentioned earlier, the strategies found by evolution can be divided into three groups (see Figure 4-11). Now if we consider the number of strategies found the different evolutionary setups belonging each group, a different picture emerges. Table 4-1 lists the number of strategies from



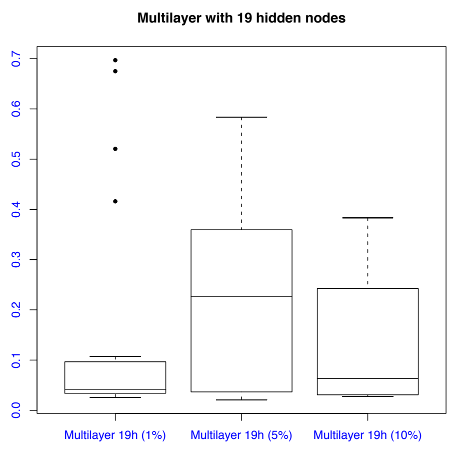
(a)



(b)



(c)



(d)

Figure 4-12: One box-plot for each of the four network structures, where each box-plot contains three populations, one for each set of evolutionary run with a given mutation rate.

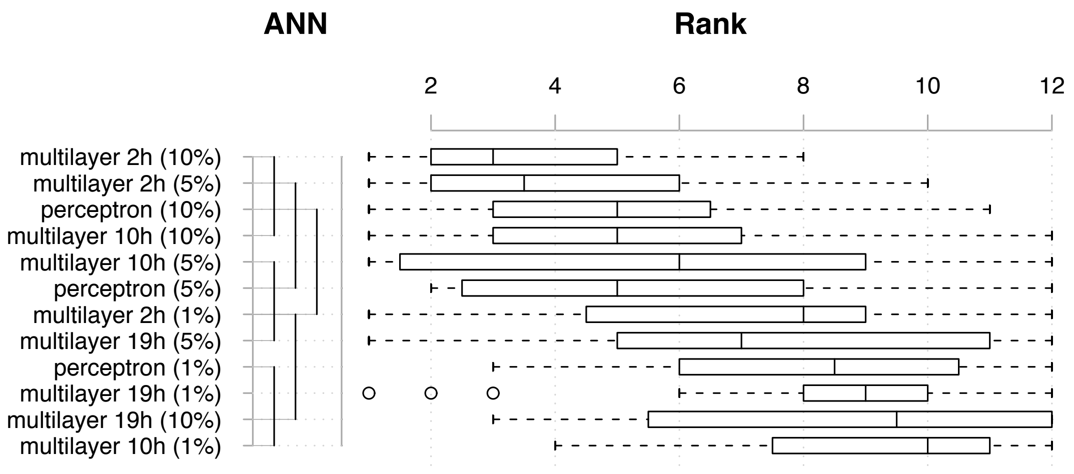


Figure 4-13: Friedman box-plot of all neural network structures evolved with different mutation rates ordered according to their average ranking in 20 evolutionary runs done for each of the neural network structures. The solid vertical lines on the left-hand side indicated statistical insignificance with a confidence level of 95%. The results show that the multilayer network with 2 hidden nodes and weights evolved with a genetic algorithm and a mutation rate of 10% received the highest average ranking, but statistically the results for this setup are not different from the next three evolutionary setups listed, including a perceptron whose weights were evolved with a mutation rate of 10%.

	1%		5%		10%		**total**	
	turn	*rev.*	*turn*	*rev.*	*turn*	*rev.*	*turn*	*rev.*
Perceptron	0	4	0	14	0	16	0	34
Multilayer 2h	7	2	12	3	13	1	32	6
Multilayer 10h	1	1	7	6	11	6	19	13
Multilayer 19h	3	1	7	4	1	6	11	11

Table 4-1: The number of *turn* and *reverse* strategies found for the evolutionary setups out of 20 evolutions. Notice, that for the perceptron controller, not a single evolution found the *turn* strategy. This could be an indication that a perceptron cannot encode such a strategy efficiently or, alternatively, that the strategy is (very) hard to find in the fitness landscape for the perceptron.

each group found by the evolutionary setups run 20 times with different initial conditions. In Figure 4-14 the evolutionary setups have been arranged according to there average rank going from lowest to highest. Notice, that for the perceptron controller, not a single evolutionary run has resulted in a controller relying on the *turn* strategy. This could indicate that a perceptron cannot encode such a strategy efficiently, or that the strategy is (very) hard to find in the fitness landscape for the perceptron.

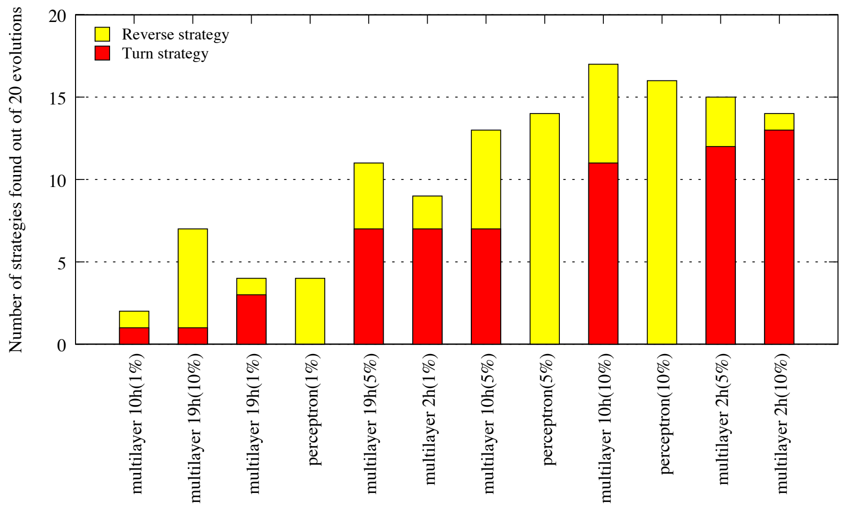


Figure 4-14: Number of *turn* and *reverse* found by 20 evolutions with different initial conditions for each setup. The evolutionary setups have been arranged according to the average rank going from lowest to highest. Notice that none of the evolutionary setups involving perceptrons found *turn* strategies.

Conclusion

The simplest network structure that we have tested, the perceptron, appears to be incapable of representing the *turn* strategy, or at least represent it in an efficient manner easily found by artificial evolution. The second simplest network structure, which we have tested, a multilayer network with 2 hidden nodes was ranked 1st and 2nd in terms of average performance for all evolutionary runs (see Figure 4-13). A multilayer network with 2 hidden nodes is capable of representing both the *turn* and the *reverse* strategy and it can be successfully evolved. Therefore, we choose this network structure and move our focus to the neuro-evolution method and evolutionary parameters.

4.5 Neuro-Evolution Method

In this section we shift our focus to the neuro-evolution method. In the previous section we have found an adequate fitness function and a simple neural network for which it is possible to evolve successful strategies that are transferable to real robots. So far we have only considered genetic algorithms with one-point crossover, rank-proportional selection, elitism, and three different mutation rates. We call this a *standard GA*. In the following we compare the performance of a standard GA with that of a $[\mu, \lambda]$ evolutionary algorithm[5], and the neuro-evolution method known as *cooperative coevolutionary genetic algorithm* (CCGA)[6]. Moreover, we test various approaches to incremental evolution and a modular method known as *evolving neural arrays*[7].

We first compare the performance of neuro-evolution methods in which the whole task is learnt by one network and through a single evolutionary run. More specifically, we compare the performances of standard GAs, $[\mu, \lambda]$ algorithms, and CCGAs on our task. The highest performing approach will then be used when we evaluate the performance of incremental evolution and evolving neural arrays. The controllers evolved consist of multilayer networks with two hidden nodes in all our tests, except from those concerning evolving neural arrays for which we also tested the performance of controllers consisting of multiple perceptrons.

4.5.1 Standard GA vs. $[\mu, \lambda]$ vs. CCGA

In order to compare the performance of standard GAs, $[\mu, \lambda]$ evolutionary algorithms, and CCGAs on our task, we have chosen a subset of evolutionary parameters for each method and conducted 20 evolutionary runs for every one of the resulting setups. The approach is similar to the approach taken for finding an appropriate neural network structure described in Section 4.4.

Standard GA: The standard GA is the evolutionary algorithm that we have used so far to find a suitable fitness function and to find a simple neural network structure. In our previous test runs with the standard GA, we found that a mutation rate of 1% in all cases was too conservative, while good results were obtained for mutation rates of both 5% and 10%. The box-plot shown in Figure 4-12(b) on page 72 actually indicates that a mutation rate higher than 10% could perform well (the average fitness of individuals evolved with a mutation rate of 10% is higher than the average fitness for individuals evolved with a mutation rate of 5%). Therefore, we have included evolutionary runs with a mutation rate of 15%. For all the standard GAs, the $[\mu, \lambda]$ algorithms, and for the CCGAs, we have run test with mutation rates 5%, 10%, and 15%.

$[\mu, \lambda]$ **evolutionary algorithm:** Recall that the $[\mu, \lambda]$ algorithm is an evolutionary algorithm for which the best μ individuals from a generation are selected for reproduction of the next generation. A new individual is created from a single parent, and hence search in the space of solutions is done solely through mutation and without crossover. λ denotes the population size, which we have fixed at 100 individuals. We have chosen 3 different values for μ, since the performance of $[\mu, \lambda]$ evolutionary algorithm is likely very dependent on the number of parents selected. More specifically, we have run tests for which the 5, 10 and 20 highest performing individuals are selected for rearing the next generation. If only a few parents are used (that is, low values of μ) it means that the search is performed

[5]See Section 3.2.2 on page 45 for more on $[\mu, \lambda]$ evolutionary algorithms.
[6]See Section 2.3.1 on page 24 for more on CCGAs.
[7]See Section 2.3.1 on page 24 for more evolving neural arrays.

aggressively because it is concentrated around fewer points in the solution space. If more parents are selected for reproduction, a higher level of population diversity is maintained, but convergence to good solutions could be slower since less exploration is done around the highest scoring solutions for a given generation. For each of the three selected values of μ, we have run 20 evolutionary runs for each of the three mutation rates. Thus, for the $[\mu, \lambda]$ evolutionary algorithm we have a total of 9×20 evolutionary runs, while for the standard GA and for the CCGA we have 3×20 evolutionary runs for each.

CCGA: For the CCGA we have a sub-population for each non-input neuron, instead of a single population as it is the case for both the standard GA and the $[\mu, \lambda]$ evolutionary algorithm. Therefore, we have a total of 4 sub-populations; one for each of the two hidden nodes, and one for each of the two output neurons. Each of the sub-populations has a size of 25 individuals. Thus, in each generation the performances of $4 \times 25 = 100$ neural networks are evaluated. In the original publication on the subject of combining CCGAs and neural networks [Potter and De Jong, 2000], the authors used sub-populations of 50 individuals each. However, in order for the method to be scalable, the number of individuals in a sub-population has to be kept reasonably low. With 25 individuals in each sub-population we achieve the same number of evaluations per generation as with the other neuro-evolution methods and thus we are able to directly compare the performance of the resulting individuals.

The main difference between the standard GA and the $[\mu, \lambda]$ evolutionary algorithm is that in the former crossover is used, while in the latter mutation is the only genetic operator used to perform the search. The CCGA differs from both the standard GA and the $[\mu, \lambda]$ algorithm, because multiple sub-populations are used. By using sub-populations and thereby keeping neurons for different positions in the neural network separate, the idea is to let the them specialize and adapt to each other's specializations. In other words, the idea is to let the neurons *co-evolve*. This should result in better solutions being obtained faster [Potter and De Jong, 2000], however our results presented below indicate that it is not always so.

Results

A box-plot of all evolutionary setups is shown in Figure 4-15. In Figure 4-16 the evolutionary setups have been arranged according to the average ranking achieved during the post-evaluation runs and statistical insignificance (with a confidence level of 95%) is indicated by the solid vertical lines on the left-hand side of the figure. The highest performing evolutionary setups are those involving the $[\mu, \lambda]$ evolutionary algorithm, followed by the standard GA with crossover and rank-proportional selection. Finally, the CCGA performs quite poorly for all three mutation rates tested.

The low performance of the CCGA can of course be due to the evolutionary parameters that we have selected for the algorithm, such as the size of each sub-population, the selection scheme, and number of elites copied from one generation to the next, and so on. As mentioned above, we have used a population size of 25 individuals for each sub-population. Furthermore, a rank-proportional selection was used, reproduction happened solely through mutation (*no crossover*),

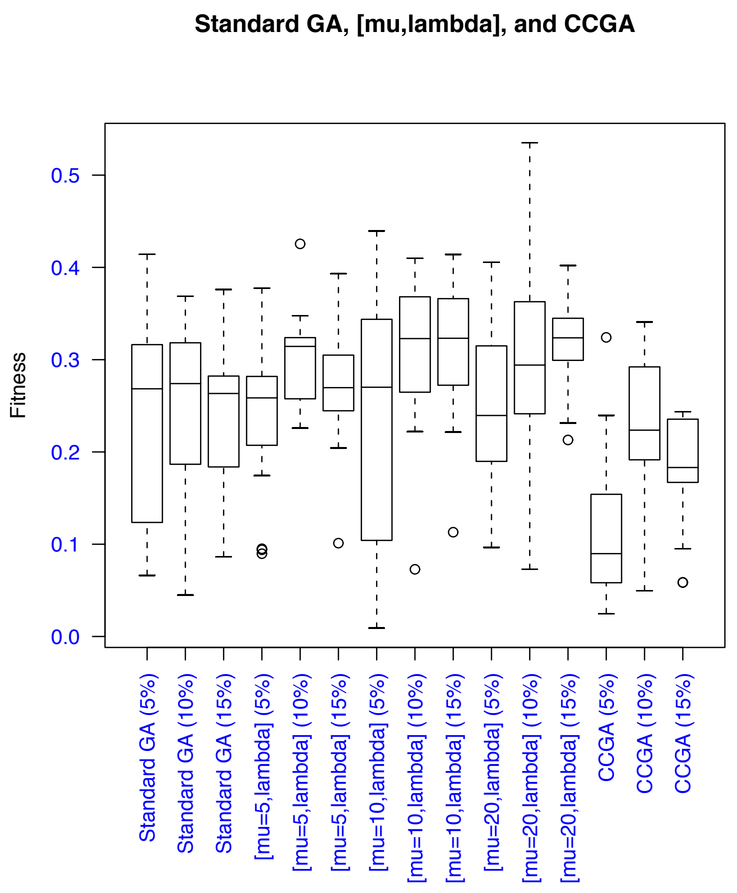


Figure 4-15: Box-plot of the performance for the standard GA, the $[\mu, \lambda]$ evolutionary algorithm ([mu,lambda]), and the CCGA. The results have been arranged according to neuro-evolution method. In Figure 4-16 on the next page the results have been arranged according to their average ranking.

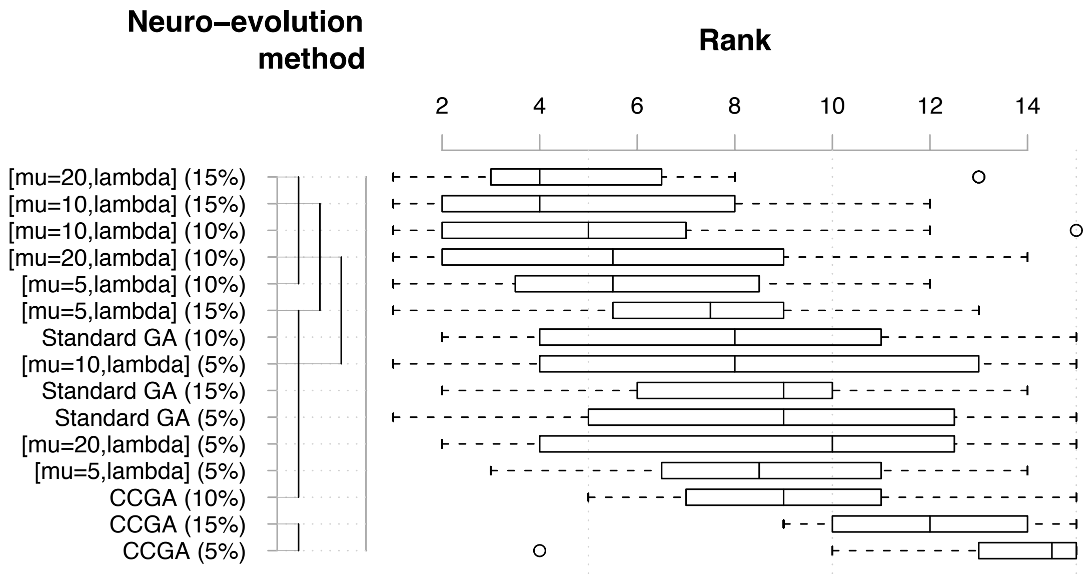


Figure 4-16: Friedman box-plot of the ranking for the standard GA, the $[\mu, \lambda]$ evolutionary algorithm ([mu,lambda]), and the CCGA with various mutation rates and for 20 evolutionary runs. The solid vertical lines on the left-hand side indicate statistical insignificance with a confidence level of 95%. The results show that the $[\mu, \lambda]$ evolutionary algorithm achieves the highest average ranking, and that several evolutionary setups involving the $[\mu, \lambda]$ algorithm perform better (with statistical significance) than the standard GA and the CCGA. Actually, the CCGA performs very poorly obtaining the three lowest average rankings.

and we copy the best two individuals from one generation to the next. Thus, the evolutionary algorithm used for the sub-populations is close to a $[\mu, \lambda]$ algorithm except that the selection scheme of letting the μ best individuals has been replaced by a rank-proportional selection scheme.

Therefore, we believe that the poor performance of the CCGA is due to the fundamental way in which the it operates combined with the properties of our fitness function: When an individual (a neuron) from a sub-population is evaluated, a neural network is constructed from the *highest performing neurons*[8] from the other sub-populations. Since our fitness function is rather noisy[9], the highest performing neuron in some generation is not necessarily the best in a more general sense. However, it could be that conditions were such that it received the highest fitness partly by chance because of the noisy fitness function. This means that the performance of the other neurons is measured with respect to this sub-optimal neuron and they could therefore perform sub-optimally themselves, possibly to a degree where the ranking within other sub-populations becomes screwed towards a different specialization leading to an overall regression in performance. In other words: A noisy fitness function could lead to the CCGA becoming unstable due to the way in which neurons are evaluated. Although, we have not analyzed this phenomenon in detail, we observed oscillations in terms of fitness scores in several of the evolutionary runs we conducted. An example is shown in Figure 4-17. This could indicate that CCGAs indeed suffer from the issue described above.

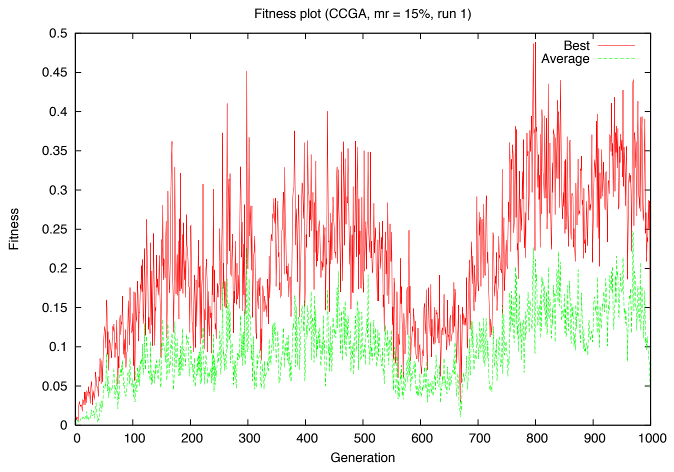


Figure 4-17: An example of a fitness plot for a CCGA. Notice how the fitness score oscillates and drops around, for instance, the 220th , 320th, and especially the 600th generation.

The standard GA and the $[\mu, \lambda]$ algorithm differ only with respect to the selection scheme and to the use of crossover. Crossover is used only in the standard GA and not in the $[\mu, \lambda]$ algorithm.

[8]"Highest performing neuron" refers to the neuron that was part of the neural network, which obtained the highest fitness score, of all the neurons in the sub-population, during the latest generation.

[9]For an example of a plot of the fitness evolution under a CCGA see for instance Figure 4-17. Examples of the fitness evolution under a standard GA are shown in Figure 4-19 on page 83.

The selection scheme in the standard GA is rank-proportional, which in the $[\mu, \lambda]$ algorithm the μ best individuals reproduce. If we consider that there are evolutionary setups for all the tested values of μ (5, 10 and 20), which perform better than the standard GA (see Figure 4-16), it seems that the selection scheme plays a minor role compared with the role of crossover. The idea behind crossover is to combine good partial solutions in order to create better complete solutions. However, our results indicate that crossover inhibits evolutionary search by frequently producing poorly performing individuals. Moreover, it is questionable if talking about "partial solutions" *within a single neural network* makes much sense. Further studies might uncover the dynamics within each of the neuro-evolution methods. However, for our purposes it suffices to finding the best method, which turned out to be the $[\mu, \lambda]$ algorithm with a mutation rate of 15% and $\mu = 20$.

4.5.2 Incremental Evolution

Incremental evolution refers to methods in which evolution begins with a population that has been trained for a simpler but somehow related task [Harvey et al., 1994]. By progressively making the task more and more complex, bootstrapping problems can possibly be overcome and evolution can be sped up. However, the use of incremental evolution can in many cases require a substantial engineering effort, because the goal-task has to be divided down into sub-tasks, which again has to be ordered going from simple to complex sub-tasks. It is not given that such an order can be found or even that the goal-task can be split into sub-tasks. Nevertheless, we have experimented with various approaches for incremental evolutions with respect to our task. Below we discuss some of the related research and previous experiments with incremental evolution, followed by a presentation of our experimental setup and the results we have obtained.

Harvey et al. [1994] evolved controllers incrementally to distinguish between white triangles and rectangles on a dark background. The goal was to evolve controllers that would only move robots towards triangles. The task was divided into sub-tasks where the robots would first learn to orient themselves to face a large rectangle easily detectable by their sensors, then to face and move towards a smaller, moving rectangle, and finally to distinguish between rectangles and triangles, and only move towards triangles. Thus, controllers where first trained to follow white rectangles and then later trained not to follow them, but instead to follow triangles only. The authors divided the goal-task into sub-tasks in which recognition and pursuit were prioritized over discrimination between the two geometric shapes in the first phases. The controllers obtained with incremental evolution were shown to be more robust than controllers trained on the complete task from an initial random population.

Nolfi et al. [1994] used incremental evolution to overcome some of the discrepancies between simulation and the real world. Controllers evolved in simulation where transferred to real robots and evolution was commenced on real robots. After a few generations the performance of the controllers on real robots reached the same level as achieved in simulation. Thus, incremental evolution was used to adapt controllers, trained in simulation, to the noise signature and behavior of the physical robot hardware.

Gomez and Miikkulainen [1997] used incremental evolution, combined with enforced sub-population[10] and delta-coding, to evolve obstacle avoidance and predator evasion. Incremental evolution was performed by first evolving populations of neurons capable of avoiding a single enemy moving at low speed on a discrete 10x10 grid. The size of the grid was then increased to a 13x13 grid and another enemy was added. Two increments followed in which the speed of the two enemies was increased. The authors showed that evolving controllers for the complete task directly was infeasible, while incremental evolution yielded satisfactory results.

In the next section we describe two different ways of sub-dividing our goal-task and the results obtained for incremental evolution based on these divisions.

Experimental Setup

We sub-divide our task in two ways:

Environmental complexity: Evolution is started in a simple arena with a light source and no holes. Holes are added incrementally when the task can be solved in the previous area. The purpose of applying incremental evolution in this manner is to shape the initial fitness landscapes in such a way that solutions are easier to find because there are none or only few holes. When the complexity of the arena is increased, we expect the evolutionary search will start in region(s) of the fitness landscapes closer to a good solution than a random population would. For instance, this way of performing incremental evolution could prevent evolutionary runs in which the s-bots fail to coordinate and move, because in the first increment there are no holes into which swarm-bots can fall and individuals who move will therefore be rewarded. The arenas used for the environmental complexity incremental evolution are shown in Figure 4-18. We use the f_{final} fitness function.

Behavioral decomposition: If it is assumed that a successful behavior can be decomposed into sub-behaviors: Coordinated-motion, hole-avoidance, and phototaxis, it is possible that these behaviors could be learnt in an incremental fashion. That is, the first learning task is concerned with coordinated-motion in an arena without holes under a fitness function that rewards coordinated-motion[11]. Once a satisfactory solution has been found, holes are added and the fitness function is extended with a component which rewards controllers that stay clear of holes ($f_{stayalive}$). Finally, phototaxis is rewarded and the evolved controllers should be able to solve the goal-task (f_{final}).

Behavioral decomposition requires more of an engineering effort than incremental evolution through environmental complexity, because we have to make certain assumption about how

[10] Enforced sub-population is another name for cooperative coevolutionary genetic algorithms in which each neuron (and its incoming weights) is represented by one individual in its own sub-population - but evaluated in neural networks constructed by neurons from all sub-populations. See Section 2.3.1 on page 24.

[11] The fitness function rewards coordinated-motion by measuring the distance moved, in a straight line, during 3 different time intervals. Every 7, 13 and 29 seconds, the position of the swarm-bot is compared to its position, 7, 13, and 29 seconds before, and the controller achieves an accumulated fitness based on the distances covered during these intervals. Initial tests showed that multiple time intervals are necessary in order to prevent circular paths.

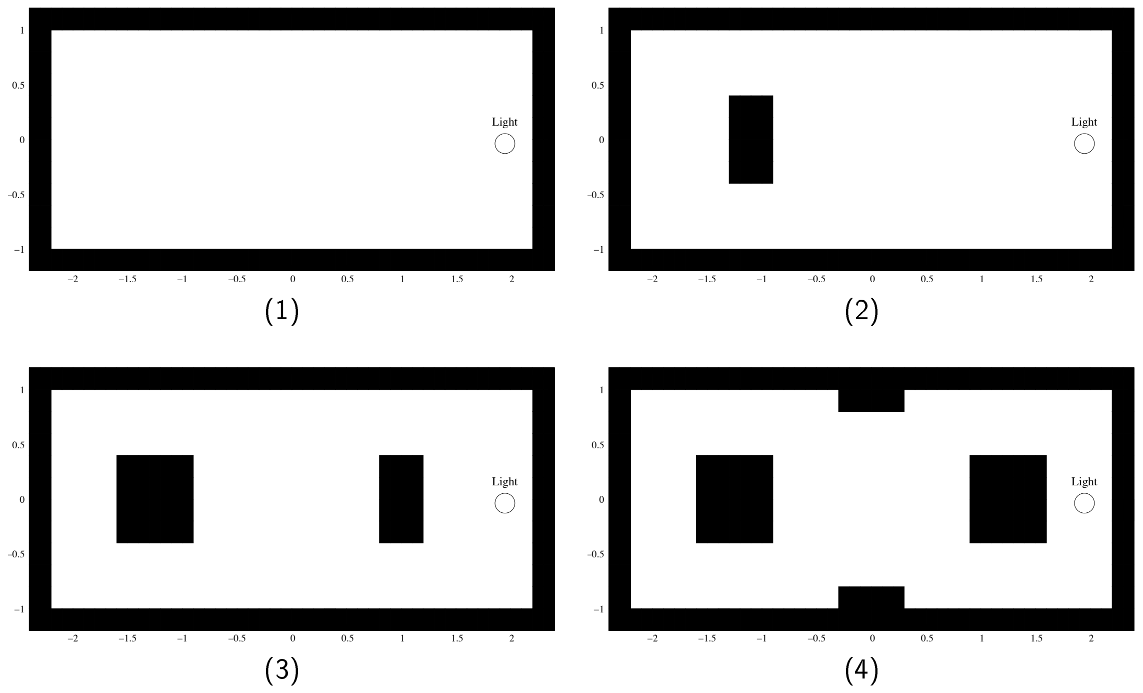


Figure 4-18: The arenas used for incremental evolution based on environmental complexity. A population is first trained in (1) until an acceptable performance is achieved, then in (2) and so on. After an acceptable performance has been achieved in (4), the arenas (a) and (b) shown in Figure 4-7 on page 64 are used. The final incremental step consists of all the arenas shown in Figure 4-7 are used in different trials in order to obtain controllers based on general strategies (see Section 4.3.3 on page 62).

behavior can be decomposed. In order to do so successfully we need thorough understanding of the task and possible strategies, which, to a certain degree, contradicts the purpose of using evolutionary robotics, namely to let evolution find effective strategies by taking advantage of novel environmental features as they are sensed by the robots. However, *if* behavioral decomposition is feasible it could prove a powerful tool for developing complex behaviors since new, more advanced behaviors could be added incrementally.

In both types of incremental evolution we have to decide on a termination criterion for each increment. That is, we have to decide when to move from one increment to the next. We can do so based either on a certain number of generations or on the performance of the population. If we simply allot a number of generations to each increment, we cannot be sure that a task has been learnt before evolution commences with a more complex one. In some cases, however, it does make sense to base the time spent on each increment on a number generations and not on the performance of a population, namely when the evolutionary computation is robust and predictable in the sense that evolutionary runs in general follow similar patterns. Our evolutionary runs are far from robust in this way: Two fitness plots from the previous phase concerning the neural network type and structure are shown in Figure 4-19. The fitness plot are both for a multilayer network with 2 hidden nodes and a standard GA with a mutation rate of 10%, but the plots are for evolutionary runs started with different initial populations. As the results show, one evolutionary run starts climbing steeply around the 140th generation, while the other only starts around the 550th generation. These two events indicate the crucial points in the evolutionary runs where the evolutionary search has found a hill of the fitness landscape containing a successful strategy.

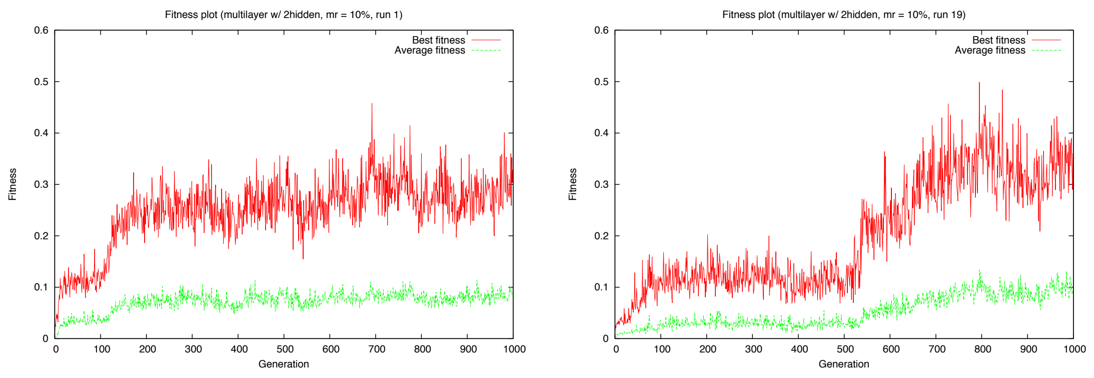


Figure 4-19: Two plots of fitness scores for the same evolutionary setup (multilayer network with two hidden nodes, a standard GA with a 10% mutation rate), but with different initial random seeds. The plots show that the evolutionary runs can progress rather differently in terms of the point at which a region of the solution space containing successful strategies is reached. The evolutionary run shown on the left-hand side reached such a region around generation 140, while the run on the right-hand side only managed to do so around generation 550.

Given that our evolutionary runs can be as unpredictable as the example in Figure 4-19 shows, we have chosen to use performance of the individuals in a generation to determine when an incremental step should be taken as opposed to allotting a fixed number of generations to each increment. The functions we use are all relative noisy and we have to avoid that "spikes" force us to take the next incremental step before the task for the current step has been truly learnt[12]. Therefore, we require the fitness score of the best performing individual to be above a certain threshold for 10 consecutive generations before an increment is performed[13]. The thresholds are different for each increment and they were determined based on the fitness function, stagnation of fitness scores during trial runs, and visual inspection of strategies found during the trial runs. Thus, the task of finding these thresholds is, like finding a suitable sub-division of the goal-task, an engineering effort.

In all cases we have used a $[\mu, \lambda]$ with $\mu = 20$ and a mutation rate of 15%, since this setup was found to be the highest performing in Section 4.5.1.

Results

An example of the evolution of fitness scores for an incremental evolution based on incremental complexity is shown in Figure 4-20. In the figure, we have indicated where increments take place. Since the same fitness function is used in all increments it is natural that the fitness scores obtained by individuals during the first increments are higher than those in later increments.

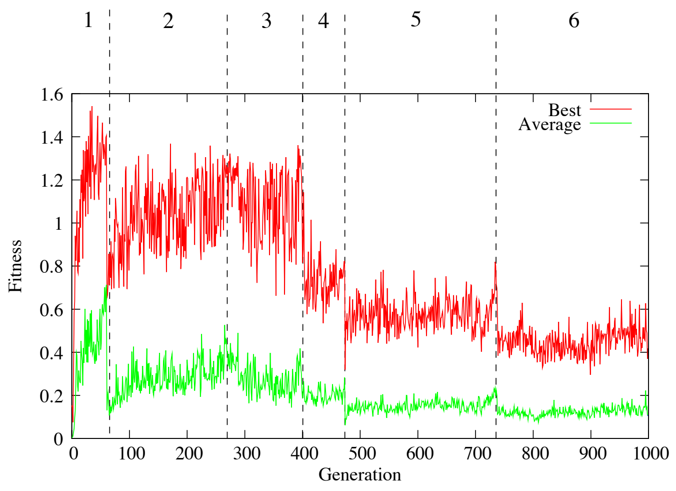


Figure 4-20: A fitness plot for an evolutionary run based on incremental evolution performed by progressively increasing the environmental complexity. There are a total of 6 increments and the switches from one increment to the next take place at generation 60, 288, 399, 473, and 737 as indicated in the figure.

Although the example shown in Figure 4-20 could give the impression that incremental evolution through increased environmental complexity works relatively well, a large number of evolutionary

[12]Spikes occur when one or more individuals receive a high fitness score due to a series of coincidences and not because the controller in question is capable of solving the task in general. This can be the case if an individual has a favorable starting position, by chance moves in the right direction, and so on.

[13]This functionality is part of our simulator, TwoDee.

runs did not go through all 6 increments before generation 1000. Only in 11 out of the 20 evolutionary runs did evolution reach the final increment. This issue influences the post-evaluation results, because the controllers evolved during the 9 evolution runs, which finished at an earlier increment, have not had the chance to "learn" the environments in which post-evaluation takes place.

In an attempt to overcome this issue we reduced the number of increments from 6 to 3, starting with an arena containing a single hole, then an arena with multiple holes, and finally four arenas[14]. Using 3 increments instead of 6, 17 of the 20 evolutionary runs reached the final increment before generation 1000 and thus had the opportunity to learn the arenas in which post-evaluation was performed.

The results for incremental evolution through behavioral decomposition and through environmental complexity are shown on the box-plot in Figure 4-21. We have included the results for the $[\mu, \lambda]$ evolutionary algorithm with $\mu = 20$, obtained without the use of incremental evolution. These results have been included so that results can be compared easily.

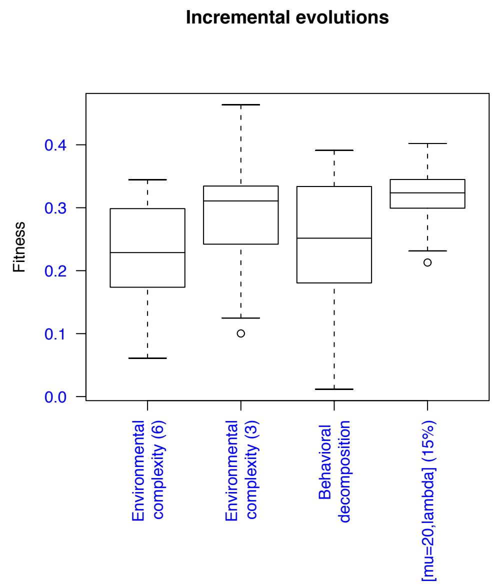


Figure 4-21: Box-plot for incremental evolution through increasing environmental complexity (results for both 6 and 3 increments, see text) and behavioral decomposition. We have included the results for the $[\mu, \lambda]$ evolutionary algorithm with $\mu = 20$, obtained without the use of incremental evolution, to allow for easy comparison of results.

As it can be seen in Figure 4-21 none of the incremental evolution approaches in general lead to better controllers than the simpler evolutionary setup without increments. It is possibly that

[14] The arena used in the first increment is shown in Figure 4-18(2), for the second increment the arena is in Figure 4-18(4) was used, and all four arenas shown in Figure 4-7 on page 64 were used for the final increment.

we could tune a number of the evolutionary parameters related to the incremental evolution and the evolutionary algorithm so that one the incremental approaches would do at least as well as an increment-free approach, however, the additional effort required to design and test the setup would only be justified in case incremental evolution proved to be significantly better than the simple approach. We discuss these results further and why they do not perform better than a non-incremental approach after Section 4.5.3 below on evolving neural arrays.

4.5.3 Evolving Neural Arrays

Evolving neural arrays (see Section 2.3 on page 23) should not be confused with incremental evolution, even though evolution takes place in stages. In incremental evolution the last generation of the previous increment is used as the first generation in the new increment, while for evolving neural arrays a new neural network is added to the array at each “increment”. The new neural network is evolved from a random population while the networks evolved during previous stages are held constant. Thus, the idea is to have an array of specialized neural networks, where each of them is trained for a specific sub-task and the learning for each sub-task takes place in isolated stages.

Beer and Gallagher [1995] approached the evolution of locomotion of a six-legged simulated robot in a similar fashion. A controller was first trained to move a single leg followed by another training phase for the locomotion of the entire robot, but with the single-leg parameters were held constant. The performance of the two-phase approach was found to be inferior to a controller for which all parameters were evolved simultaneously. However, the results obtain in a study concerning the use of evolving neural arrays for a robot navigation task [Corbalán and Lanzarini, 2003], showed that it was superior to SANE, which in turn has been shown to perform better than a number of other neuro-evolution methods popular at the time [Moriarty and Miikkulainen, 1996].

The idea behind evolving neural arrays is to let networks in a later phase learn what situations they are able to handle and when to delegate the control to previously evolved network. Now, if such a neuro-evolution method could be applied to complex tasks it would solve a number of issues that we are faced with if we try to evolve complex behavior. More complex tasks require more complex controllers and therefore more complex neural networks. The search space for complex neural networks is larger than for simpler ones, which is likely to have an impact on the performance of neuro-evolution[15]. If we instead can keep each component of the solution (each neural network in our evolving neural array) simple and evolve the networks one at a time, it would be a promising method for extending and refining functionality. Ultimately, this modularized approach could lead to the neuro-evolution of complex behaviors.

In order to test evolving neural arrays for our task, we reuse the setups from the incremental evolution approach described in Section 4.5.2. However, instead of starting from a previously evolved population, we add another neural network to our array at each “increment”. For each

[15]The multilayer network with 19 hidden nodes, the most complex network that we tested in Section 4.4 on page 68, had a performance significantly lower than the simpler multilayer network with 2 hidden nodes. Given that we have obtained this result for our relatively simple task, it would be very surprising if it would be *easier* to evolve more complex behaviors in more complex networks.

evolutionary phase, we use a $[\mu, \lambda]$ algorithm with $\mu = 20$ and a mutation rate of 15% as this setup was found to be the one with the best performance of those tested in Section 4.5.1.

Results

The results obtained in our test of evolving neural arrays are shown in Figure 4-22. From the results obtained, it is obvious that evolving neural arrays do not perform nearly as well as a simpler evolutionary setup in which the weights for a single network are evolved from a random population.

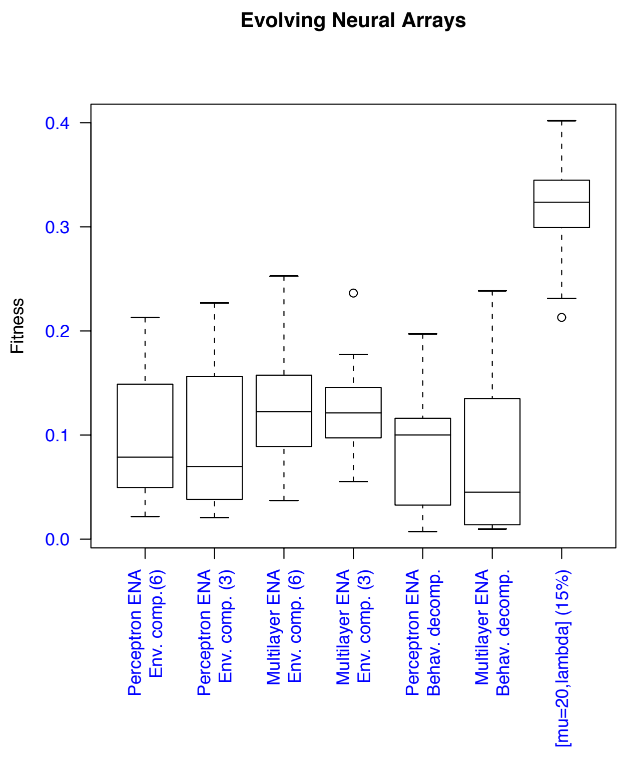


Figure 4-22: Box-plot of the performance on evolving neural arrays. The task has been sub-divided in two ways: One concerning environmental complexity (3 and 6 steps) and one concerning behavioral decomposition. Results are shown for evolving neural arrays consisting of perceptrons and for evolving neural arrays consisting of multilayer networks with two hidden nodes. We have included the results for the $[\mu, \lambda]$ evolutionary algorithm with $\mu = 20$, obtained without the use of evolving neural arrays, to allow for easy comparison of results.

We would have liked to have been able to compare the performances of evolving neural arrays consisting of multilayer networks with arrays consisting of perceptrons. However, the resulting controllers for both sets of tests were of such a low performance, that such a comparison would not be meaningful. In the following section we attempt to answer the question as to why evolving neural arrays (and to a lesser degree incremental evolution) fail.

4.5.4 Conclusion

We have tested a number of different neuro-evolution methods and found that of those tested one of the simpler approaches, namely a $[\mu, \lambda]$ algorithm, yielded the best results. Neither more advanced approaches like CCGA in which individual neurons are treated as individuals in their own sub-population, nor incremental or modular approaches lead to better controllers. Does that mean that we can dismiss these more advanced methods in evolutionary robotics? Of course not, but we can conclude that the more advanced neuro-evolution methods tested did not result in better controllers than a simple evolutionary setup, at least for our task and for the sets of evolutionary parameters, which we have tested.

Before we draw any further conclusions it is important to put our results into perspective with respect the task that we have evolved controllers to solve, its complexity, and the evolvability of solutions: One way of looking at the task of performing phototaxis and hole-avoidance in a swarm-bot is by decomposing a successful behavior into coordinated-motion, hole-avoidance, and phototaxis, as we did in Section 4.5.2 on incremental evolution. If we take a closer look at the solutions found by artificial evolution, it is questionable if a final successful behavior, which solves the task, can be separated into such sub-behaviors. An example of a strategy found by evolution is shown in Figure 4-23. Regardless of the initial position of the swarm-bot, the initial orientation of the s-bots and the arena, the result is always that the swarm-bot starts moving left with respect to the light (left corresponds to up in the figure). This could indicate that the s-bots mainly coordinate based on the sensed direction of the light, and not based on the readings of the traction sensor like it has been the case for previous studies on coordinated-motion and hole-avoidance for swarm-bots [Trianni et al., 2004b], [Trianni, 2003]. Thus, this suggests that coordinated-motion is partly a by-product of phototaxis and it is therefore not beneficial to evolve coordinated-motion and phototaxis independently.

Another interesting observation regarding the solutions found by evolution is that the controllers do not steer the s-bots directly towards the light. As mentioned above, the controller which resulted in the path shown in Figure 4-23 always steer the s-bots left when a simulation is started. Some successful strategies found by other evolutionary runs steer s-bots right, but otherwise show similar behaviors. None of the controllers, which solve the task, move directly towards the light at any point[16]. This could explain why incremental evolution through increasing environmental complexity does not perform better than a non-incremental setup, namely because in arenas with fewer and/or smaller holes, other strategies are found, which do rely on the s-bots moving directly towards the light. In this way, incremental evolution through increasing environmental complexity makes evolution take a detour.

Incremental evolution and modular evolution (through the use of evolving neural arrays) do not perform well on our task because the successful solutions are both indivisible and specialized for the complexity of the arenas in which the task is solved. Moreover, the final solutions were easy

[16]Of course some s-bots face the light at some points, all s-bots in a swarm-bot might even face the light at the same time, but this only happen while they are turning to avoid holes.

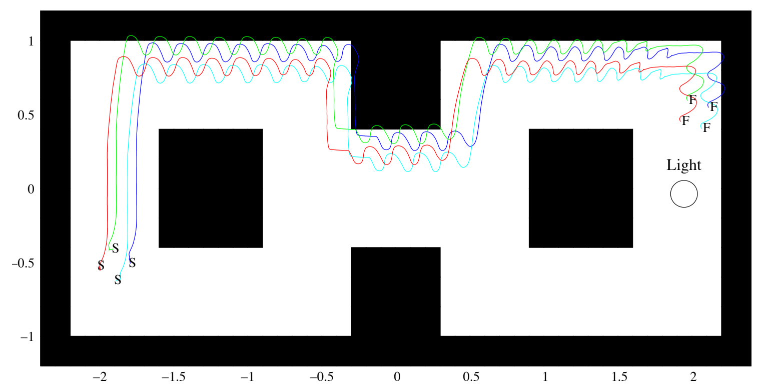


Figure 4-23: An example of a successful strategy. Notice that the swarm-bot does not move directly towards the light but moves left with respect to the light (or up in the figure). Since successful strategies do not perform phototaxis by moving directly towards the light and since coordinated-motion likely is a by-product of coordination with respect to the light, incremental and modular evolution does not perform better than a simpler evolutionary setup.

enough to find from a random population using a simple evolutionary algorithm. However, had the task been divisible, for instance if the s-bots had to assemble into a swarm-bot before moving towards the light source, modular evolution might have out-performed a simpler approach. Likewise, if we had not been able to evolve a solution of sufficient quality from a random population, we could have been forced to use an incremental approach. We can therefore conclude that given the nature of our task, with respect to its divisibility and complexity, incremental and modular evolution do not perform better than a simpler and traditional approach. However, further research should go into applying these more advanced neuro-evolution methodologies to more complex tasks.

4.6 Test on Physical Robots

We have conducted a number of tests on physical robots in an arena shown in Figure 4-24(a). Instead of holes we used black duck tape. A black surface and holes are perceived in the same manner by the s-bots' ground sensors, while using duck tape instead of holes prevents physical damage to the robots in case they should “fall in”. The arena we have built is a replica of the one used in simulation, see Figure 4-24(b). The measures of the arena are shown in Figure 4-3 on page 56.

The s-bots are proto-types built for the Swarm-bots project. During the life-time of the project the hardware and software have evolved. However, the s-bots must still be considered proto-type hardware given the fragility and individual differences between the robots. The project has officially ended and therefore support on the hardware is very limited. These issues meant that before tests could be run on the physical robots they had to be thoroughly tested and sensors and actuators had to be re-calibrated. Various settings, such as the scaling of the inputs from the traction sensors, had to be adjusted individually for each robot.

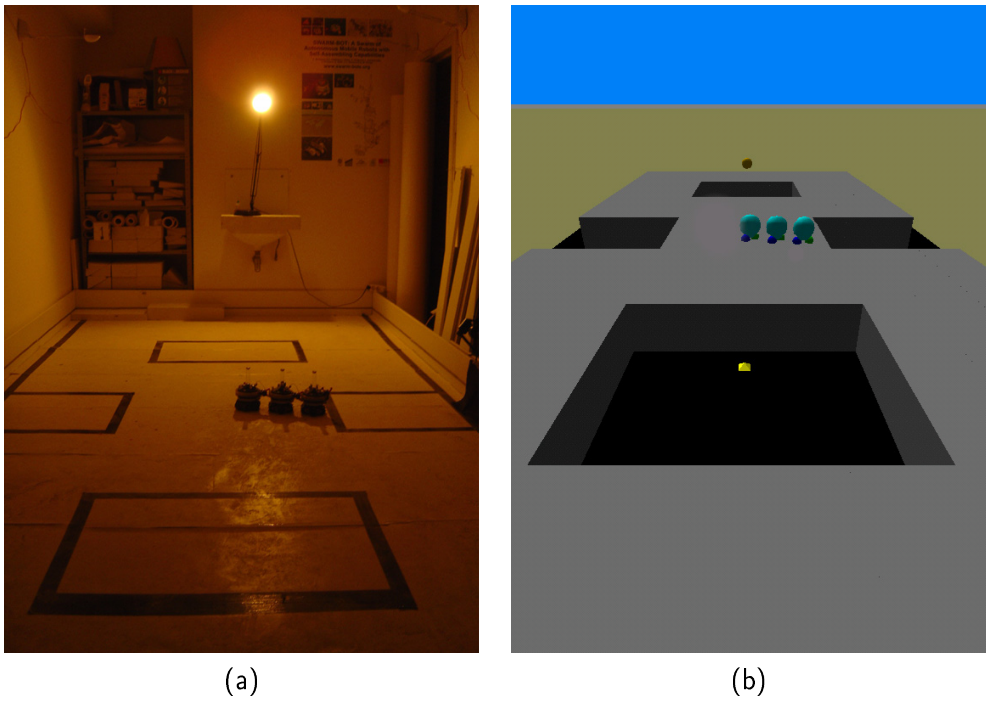


Figure 4-24: A photo of the arena used for the experiments on real s-bots is shown in (a). The corresponding simulated arena is shown in (b).

Although we reused parts of control programs that have already been developed and tested, we still had to design, implement, and test a non-trivial amount of code to be able to conduct our experiments on the s-bots. Developing software for embedded systems is often quite challenging, especially during the testing and debugging phase, where the code is run on the embedded platform itself. To ease this task, we found that using the colored leds on the robot worked quite well, e.g. to visually validate that robots detected the light correctly, that they sensed signals from other robots when holes were perceived, and that the robots had not crashed. This scheme proved efficient and intuitive compared to dumping debugging information to a terminal or a log file.

As a proof-of-concept, we took an evolved artificial neural network controller - a multilayer network with 2 hidden nodes based on the *turn* strategy - and ran tests with one, two, and three robots[17].

4.6.1 Results

The physical response of an arena with holes differs from that of an arena in which holes are replaced by black duck tape. In an arena with real holes, an s-bot, which is over a hole, cannot directly influence the rest of the swarm-bot by moving its treels, whereas if duck tape is used

[17]We would have preferred to have run tests with a higher number of s-bots, but at the time of writing we only have access to three s-bots on which the sensors and actuators, which we use, function properly.

instead of holes, s-bots still have surface contact while moving on the duck tape. The motion of their treels, therefore, influences the swarm-bot just like s-bots that are not over a "hole". Despite this fundamental difference, our initial tests on real robots showed that the evolved controllers are indeed transferable and that the strategies observed in simulation match those observed in simulation.

We had to reduce the maximum speed of the real robots slightly for the evolved controllers to work on real hardware. By reducing speed noise was smoothened and the impact of each action (or control cycle) was reduced. This seemed to compensate for the differences in noise signatures, sensor delays, and so on, between simulation and reality.

An example of a test run on the s-bots is shown in Figure 4-25. In the figure we also show an example of a similar behavior in simulation. Notice, that in simulation the swarm-bot does not turn as far away from the edge of the arena as it is the case on the real robots. It can be hard to tell from the path completed by the robots in simulation drawn in the upper left corner on Figure 4-25, that the controller actually relies on the *turn* strategy. The swarm-bot simply seems to be following the edge of the arena. However, the controller does rely on the *turn* strategy, although the turns are very small with the s-bot closest to the edge detecting the hole frequently, that is every 1-2 seconds. On the real robots the turns away from edges are larger. The reason behind this difference is likely that the reaction time on real robots is longer than anticipated during the development of our simulator. Moreover, a number of control cycles pass from the time at which a hole is detected until the other robots detect the sound signal emitted by the s-bot detecting the hole. This means that an s-bot is pushed a bit further over the edge of a hole than in simulation and it therefore takes longer before the other s-bots manage to pull the s-bot detecting the hole onto the area again. Parts of the strategy employed by the evolved controller works such that when s-bots detect a sound signal they turn away from the edge[18]. Since a hole is detected for a longer period of time on real robots than in simulation, the other s-bots turn further away from the hole, and the turns therefore become larger.

Although there are differences between the performance in simulation and on real robots, the fundamental strategy is the same and the evolved controllers do perform phototaxis and holes-avoidance when run on real robots. Our tests with one and two robots showed similar results: We saw correspondence between the behavior in simulation and in reality. Given that the speed was reduced the real robots in general took longer to reach the target area. For example the paths shown in Figure 4-25 took 5m04s for the real s-bots, while only 2h19s in simulation. However, it is our *impression* that hole-avoidance actually is more efficient on the real robots than in simulation, because of the reduced speed. Due to the lower speed, less momentum is built up on the real robots and they have more time to react than their simulated cousins. Hence, quantitatively the performance on real robots is not on par with the performance in simulation (which is not uncommon in evolutionary robots), while qualitatively the performance appears to be at least as good as in simulation.

In order to draw more general conclusions on the performance of our evolved controllers on physical robots and the differences between performance in simulation and reality, we would

[18]Strictly speaking the s-bots do not explicitly turn away from the edge, but rather turn counter-clockwise when a sound signal is sensed.

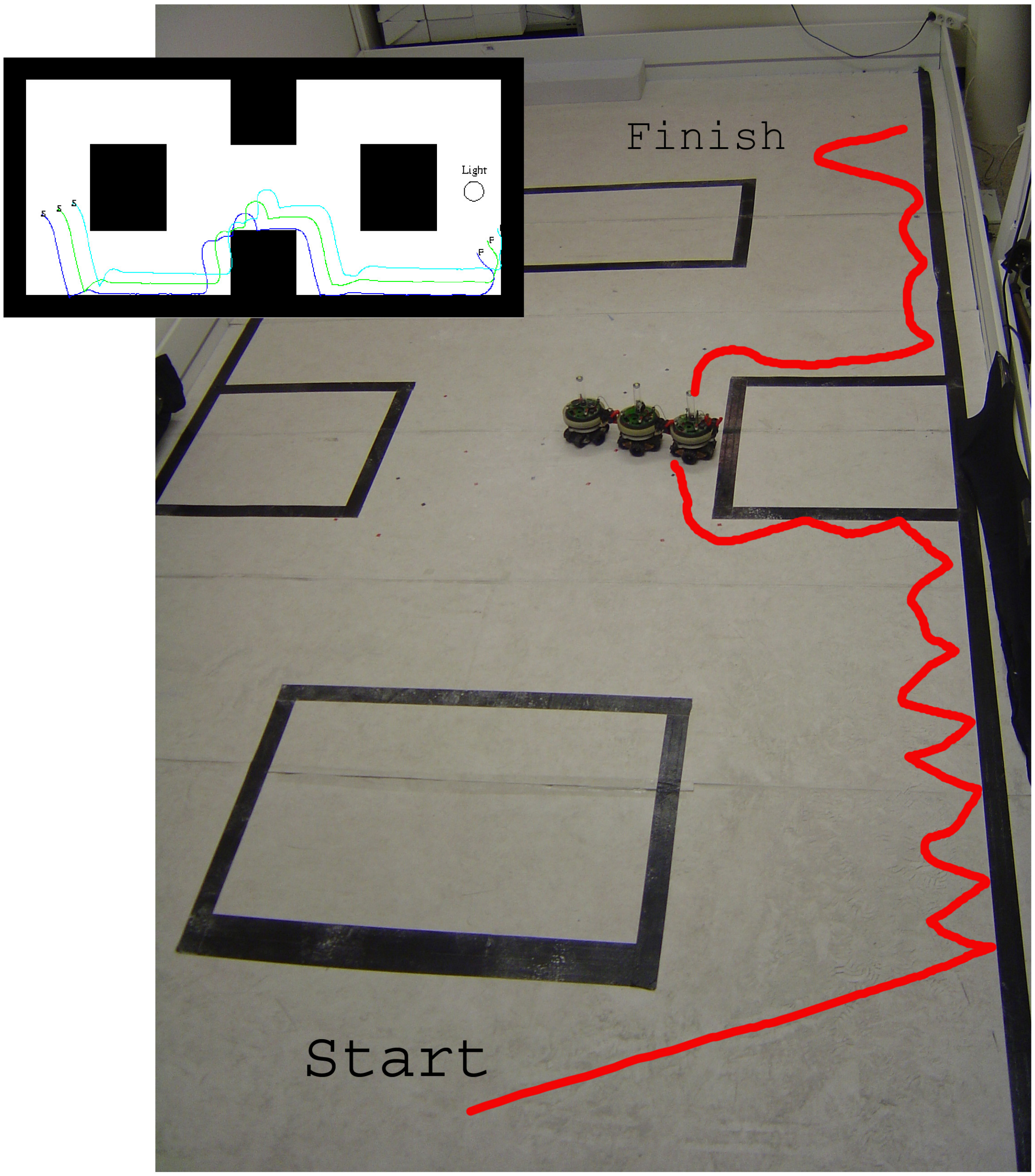


Figure 4-25: An example of a test run on real robots. A similar simulation run is shown in the upper left corner.

need to build an arena with real holes and to have access to additional and fully operational s-bots. If such facilities should become available, further studies will be conducted.

Based on the results of our initial tests, we can conclude that controllers evolved in our simulation can be transferred successfully to physical robots. That is, we have shown physical s-bots steered by controllers evolved in TwoDee perform phototaxis and hole-avoidance.

4.7 Discussion

In the chapter we motivated the use of a phased method for determining a suitable evolutionary setup. An interesting question to ask is whether following this method benefited the process and the results compared to following a more traditional, ad-hoc approach. Of course this question cannot be answered easily, but one of the advantages of having approached the evolution of controllers in a structured fashion is that we have a basis for comparison and discussions of alternatives. Furthermore, by treating each component of the evolutionary setup separately, we have been able to compare a number of different neuro-evolution methods: Different flavors of classic evolutionary setups, where one individual corresponds to the weights in an entire neural network, cooperative coevolutionary genetic algorithms, incremental evolution, and evolving neural arrays. The three latter methods have all been praised by their inventors as being superior to more classical approaches. Their conclusions were based on a single or very few experiments. However, in none of our tests we obtained results that showed the more advanced neuro-evolution methods to be better than a simpler approach. That could very well be due to the nature of our task as we have discussed earlier in this chapter. However, following the method suggested, where the different aspects of the neuro-evolution setup are treated systematically one at a time, we build are stronger basis for drawing conclusions, if not for other reasons then simply due to the amount of empirical data collected and presented.

With respect to our experimental results, we managed to evolve controllers in our simulator TwoDee, which were transferred to s-bots and allowed a swarm-bot to perform phototaxis and hole-avoidance. Although more experiments on the robots would be needed in order to draw further conclusions on the performance obtained, initial results indicate that quantitatively there is a performance drop when going from simulation to reality, but qualitatively the performance and strategies appear to match those in simulation.

Conclusion

Chapter 5

In this DEA thesis we have focused on evolving a controller capable of performing hole-avoidance and phototaxis in a group of connected robots. We have taken a broad perspective on the subject and discussed parts of the theoretical foundations in Chapter 2, we developed our own software platform, TwoDee, for evolving controllers as described in Chapter 3, and in Chapter 4 we suggested a methodology for finding a suitable evolutionary setup. We applied this methodology on our task, namely to evolve a controller that could steer a swarm-bot to a target without falling into holes on the way. In the process we tested several different neuro-evolution methods and we showed that a controller evolved in TwoDee could indeed move a group of real s-bots towards a light source while avoiding holes.

Besides from evolving controllers capable of solving the task, we also wanted to take a close look at the process of evolving controllers with the aim of optimizing it. In order to do so we analyzed a number of existing software platforms, but found that they were either too complex (those relying on third-party software dynamics engines) or too specialized to be useful for our purposes. We therefore developed our own simulator, TwoDee, which relies on a customized dynamics engine. Basically, we sacrificed the accuracy and generality of the dynamics available in the complex simulators and gained performance instead. At the time of writing, TwoDee does not simulate rough terrain, but only flat terrain with holes. On the other hand, we are able to simulate thousands of robots (connected or not), because the time complexity of our custom-built dynamics engine is linear in the number of simulated objects. This is a dramatic improvement over simulators attempting to simulate dynamics accurately as their time and space complexity usually are cubic and quadratic, respectively, in the number of objects simulated. Furthermore, it is questionable if virtual models of the robots and the arenas can be built in such high detail that one can really benefit of the complexity of general software dynamics engines. Aside from the benefits of scalability, with TwoDee we have also obtained a 50-fold performance increase simulating 4 connected s-bots compared to NS, which is another, newly developed, third-party dynamics engine-based, s-bot simulator. To put things into perspective: Had we relied on one of the more complex simulators like NS and if we assume that we had had exclusive, uninterrupted access to our current 32 CPU Opteron cluster, the evolutionary runs presented in Chapter 4 would have taken 1.042 days[1], or almost the nominated time for doctoral studies at our university, to complete.

[1] Each trial lasts 240 virtual seconds. According to the benchmark results shown in Table 3-1 on page 49 simulating 4 robots for 240 seconds of simulation time would take at least 5 seconds using the NS simulator. We have used 8 samples per individual, 100 individuals per generation and 1000 generations in each evolutionary run. If we omit the evolutionary runs conducted during the engineering of the fitness function, we have a total of 720 evolutionary runs. This equals 800.000 CPU hours, or 1.042 days if we have access to 32 CPUs. This does not include a large number of test runs, the CPU time needed to compute the outputs of the neural networks, perform the post-evaluation analyzes, hence, 1.042 days is a rather optimistic estimate.

Software tools that allow for short turnaround times do not only speed up evolution, they also allow for broader comparisons of fitness functions, neural network structures, and neuro-evolution methods. The method that we described in Chapter 4 for finding a good evolutionary setup depends on the ability to conduct a large number of evolutionary runs. To our knowledge there currently exist no method or structured approach for performing artificial evolution of robot controllers. We motivated a phased methodology in which the evolutionary setup becomes progressively more specific. First we developed, tested, and modified fitness functions until a suitable one was found, thereafter the focus was shifted to the neural network type and structure, and finally to the neuro-evolution method and its parameters. By following this method, we found that a multilayer network with only 2 hidden nodes was capable of solving our task and easier to evolve than more complex networks. Moreover, it was found that a $[\mu, \lambda]$ evolutionary algorithm performed better than the genetic algorithms relying on crossover. In our experiments involving more advanced neuro-evolution methods, we found that the performance of cooperative coevolutionary genetic algorithms sometimes regress during evolutionary runs when the fitness function is noisy. We also found that neither incremental nor modular evolution strategies performed better than a simpler approach on our task. We believe that the main reason for the poor performance of incremental and modular evolutionary setups is that the task we have focused on cannot be divided into sub-tasks (or sub-behaviors) in a beneficial way. This is because the successful solutions that evolution found were all simple, coherent behaviors. For more complex and divisible tasks and/or behaviors, however, incremental evolution, modular evolutions, or some other scheme, which is capable of solving the issues related to complexity and evolvability, will without doubt become necessary if we wish to apply evolutionary robots to more complex problems than those tackled today.

5.1 Future Work

We strongly believe that research into methodical approaches is necessary if evolutionary robotics is to be used for anything but toy problems. The reason is, that currently the field of evolutionary robotics resembles the Wild West, where a lot of work is being done in many different directions, but for which no methods or rules exist. Many researchers claim that their newly developed neural network structure/neuro-evolution method/mutation operator/incremental evolution scheme/etc. is superior. Of course they cannot all be superior and, as our results show, their applicability might be quite problem-dependent. As a first step, it is important that authors devote some time and space in publication to be explicit about their method for choosing a given evolutionary setup. This would allow for comparisons, replications, discussions, and reuse of ideas. As a result, our understanding of the underlying mechanisms could increase and one would be able to rely on best practices for finding and using suitable evolutionary setups.

Aside from the general engineering aspects of evolutionary robotics, there are a number of ideas and practical and theoretical issues related to the work presented in this thesis, which we would like to investigate further:

Perform tests with real holes: In our tests of the controllers on real robots, we used black duck tape to draw up the arena. Black duck tape and holes are perceived in the same manner by the ground sensors on an s-bot, however, the physical behavior of the swarm-bot would be different in the presence of real holes. Therefore, a future task is to build an arena with real holes and test the evolved controllers herein.

Perform tests with more robots: We only had access to three s-bot with the necessary sensors working. It would be interesting to test if the behavior scales with the number of robots as it does simulation, and to discover what the differences in terms of performance might be.

Apply our methodology on a more complex task: We tested a number of advanced neuro-evolution methods, but found that none of them worked well for our task, because solutions to the task we focused on could not be decomposed easily. In order to test the methodology and the advanced neuro-evolution approaches described in Chapter 4, we should study more complex problems. This could, for instance, involve functional self-assembly and multiple tasks being performed by a swarm of robots.

Evolve the neural network structure: The neuro-evolution methods that we have tested have all worked on a fixed neural network structure. In other methods, such as SAGA[2], the structure of the network itself is under evolutionary control. If such methods in general perform well, it could reduce the amount of human intervention necessary, and the phase in our method concerned with choosing a suitable neural network structure would become obsolete.

[2]See Section 2.3.1 on page 23 for more on SAGA.

Bibliography

G. Baldassarre, S. Nolfi, and D. Parisi. Evolving mobile robots able to display collective behaviours. In C. Hemelrijk and E. Bonabeau, editors, *Proceedings of the International Workshop on Self-Organisation and Evolution of Social Behaviour*, pages 11–22, Monte Verità, Ascona, Switzerland, Sept. 8-13, 2002. University of Zurich.

R. D. Beer. On the dynamics of small continuous-time recurrent neural networks. *Adaptive Behavior*, 3:471–511, 1995.

R. D. Beer and J. C. Gallagher. Evolving dynamical neural networks for adaptive behavior. *Adaptive Behavior*, 1:91–122, 1995.

D. A. Coley. *An Introduction to Genetic Algorithms for Scientists and Engineers*. World Scientific Publishing Company; Bk&Disk edition, 1997.

W. J. Conover. *Practical Nonparametric Statistics*. Wiley & Sons, New York, 3rd edition, 1999.

L. Corbalán and L. Lanzarini. Evolving neural arrays a new mechanism for learning complex action sequences. *CLEI (Latinamerican Center for Informatics Studies) Electronic Journal*, 6 (1), 2003.

P. Dalgaard. *Introductory Statistics with R*. Springer, 2002.

D. T.-D. Dung. Master thesis: Simulation 3d d'un groupe de robots. *Université Libre de Bruxelles, Faculté de Science Appliquées*, 2005.

J. L. Elman. Finding structure in time. *Cognitive Science*, 14(2):179–211, 1990.

L. V. Fausett. *Fundamentals of Neural Networks (1st Edition)*. Prentice Hall, 1994.

D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Publishing Company, Reading, MA, 1989.

D. E. Goldberg. *The Design of Innovation: Lessons from and for Competent Genetic Algorithms*. Kluwer Academic Publishers, Boston, MA, 2002.

F. Gomez and R. Miikkulainen. Incremental evolution of complex general behavior. *Adaptive Behavior*, 5:317–342, 1997.

R. Groß and M. Dorigo. Cooperative transport of objects of different shapes and sizes. In M. Dorigo, M. Birattari, C. Blum, L. M. Gambardella, F. Mondada, and T. Stützle, editors, *Ant Colony Optimization and Swarm Intelligence, 4th International Workshop, ANTS 2004*, volume 3172 of *Lecture Notes in Computer Science*, pages 107–118. Springer Verlag, Berlin, Germany, 2004a.

R. Groß and M. Dorigo. Evolving a cooperative transport behavior for two simple robots. In P. Liardet, P. Collet, C. Fonlupt, E. Lutton, and M. Schoenauer, editors, *Artificial Evolution – 6th International Conference, Evolution Artificielle (EA 2003)*, volume 2936 of *Lecture Notes in Computer Science*, pages 305–317. Springer Verlag, Berlin, Germany, 2004b.

R. Groß and M. Dorigo. Group transport of an object to a target that only some group members may sense. In X. Yao, E. Burke, J. A. Lozano, J. Smith, J. J. Merelo-Guervós, J. A. Bullinaria, J. Rowe, P. T. A. Kabán, and H.-P. Schwefel, editors, *Parallel Problem Solving from Nature – 8th International Conference (PPSN VIII)*, volume 3242 of *Lecture Notes in Computer Science*, pages 852–861. Springer Verlag, Berlin, Germany, 2004c.

I. Harvey. Species adaptation genetic algorithms: a basis for a continuing SAGA. In F. J. Varela and P. Bourgine, editors, *Proceedings of the First European Conference on Artificial Life. Toward a Practice of Autonomous Systems*, pages 346–354, Paris, France, 11-13 1992. MIT Press, Cambridge, MA.

I. Harvey, P. Husbands, and D. Cliff. Issues in evolutionary robotics. Technical Report Cognitive Science Research Paper CSRP219, Brighton BN1 9QH, England, UK, 1992.

I. Harvey, P. Husbands, and D. Cliff. Seeing the light: artificial evolution, real vision. In *Proceedings of the third international conference on Simulation of adaptive behavior: From animals to animats 3*, pages 392–401, Cambridge, MA, 1994. MIT Press.

S. Haykin. *Neural Networks: A Comprehensive Foundation (2nd Edition)*. Prentice Hall, 1998.

C. Hecker. Physics, part 1: The new frontier. *Game Developer Magazine, Oct/Nov*, 1996a.

C. Hecker. Physics, part 2: Angular effect. *Game Developer Magazine, Dec/Jan*, 1996b.

C. Hecker. Physics, part 3: Collision response. *Game Developer Magazine, Feb/Mar*, 1997a.

C. Hecker. Physics, part 4: The third dimension. *Game Developer Magazine, June*, 1997b.

S. Hochreiter and J. Schmidhuber. Bridging long time lags by weight guessing and "long short-term memory". pages 65–72, Amsterdam, Netherlands, 1996. IOS Press.

J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. In *Proceedings of the National Academy of Science*, Washington, DC, 1982. National Academy Press.

K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989. ISSN 0893-6080.

R. Ihaka and R. Gentleman. R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics*, 5(3):299–314, 1996.

N. Jakobi. Half-baked, ad-hoc and noisy: Minimal simulations for evolutionary robotics. In P. Husbands and I. Harvey, editors, *Proceedings of the Fourth European Conference on Artificial Life: ECAL97*, pages 348–357. MIT Press, Cambridge, MA, 1997.

M. I. Jordan. Attractor dynamics and parallelism in a connectionist sequential machine. In *Artificial neural networks: concept learning*, pages 112–127, Piscataway, NJ, 1990. IEEE Press.

T. Kohonen. *Self-Organizing Maps*, volume 30. Springer Series in Information Sciences, Springer, Berlin, Heidelberg, New York, third extended edition edition, 2001.

R. A. Krohling, Y. Zhou, and A. M. Tyrrell. Evolving fpga-based robot controllers using an evolutionary algorithm. In J. Timmis and P. J. Bentley, editors, *Proceedings of the 1st International Conference on Artificial Immune Systems (ICARIS)*, pages 41–46, University of Kent at Canterbury, September 2002. University of Kent at Canterbury Printing Unit.

T. Labella, M. Dorigo, and J.-L. Deneubourg. Efficiency and task allocation in prey retrieval. In A. Ijspeert, D. Mange, M. Murata, and S. Nishio, editors, *Proceedings of the First International Workshop on Biologically Inspired Approaches to Advanced Information Technology (Bio-ADIT2004)*, Lecture Notes in Computer Science, pages 32–47. Springer Verlag, Heidelberg, Germany, 2004a.

T. Labella, M. Dorigo, and J.-L. Deneubourg. Self-organised task allocation in a group of robots. In R. Alami, editor, *Proceedings of the 7th International Symposium on Distributed Autonomous Robotic Systems (DARS04)*, Toulouse, France, June23–25 2004b.

J. Maindonald and J. Braun. *Data Analysis and Graphics Using R*. Cambridge University Press, Cambridge, 2003.

W. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, pages 115–133, 1943.

Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer-Verlag, Berlin, Germany, 1999.

O. Miglino, D. Denaro, G. Tascini, and D. Parisi. Detour behaviour in evolving robots: Are internal representations necessary? In P. Husbands and J.-A. Meyer, editors, *EvoRobots*, volume 1468 of *Lecture Notes in Computer Science*, pages 59–70. Springer, 1998.

O. Miglino, H. H. Lund, and S. Nolfi. Evolving mobile robots in simulated and real environments. *Artificial Life*, 2(4):417–434, 1995. ISSN 1064-5462.

M. Minsky and S. Papert. *Perceptrons*. MIT Press, Cambridge, MA, 1969.

M. Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA, 1996.

F. Mondada, A. Guignard, M. Bonani, D. Bär, M. Lauria, and D. Floreano. Swarm-bot: From concept to implementation. In *Proceedings of the 2003 IEEE/RSJ International Conference on Intelligent Robot and Systems (IROS 2003)*, pages 1626–1631, Las Vegas, Nevada, US, October 27 - 31, 2003 2003. IEEE Press.

F. Mondada, G. C. Pettinaro, A. Guignard, I. Kwee, D. Floreano, J.-L. Deneubourg, S. Nolfi, L. Gambardella, and M. Dorigo. Swarm-bot: a new distributed robotic concept. *Autonomous Robots*, 17(2–3):193–221, 2004.

D. C. Montgomery. *Design and Analysis of Experiments*. John Wiley & Sons Inc., New York, US, 2001.

D. E. Moriarty and R. Miikkulainen. Efficient reinforcement learning through symbiotic evolution. *Machine Learning*, 22:11–32, 1996.

S. Nolfi and D. Floreano. *Evolutionary Robotics: The Biology, Intelligence, and Technology of Self-Organizing Machines*. MIT Press/Bradford Books, Cambridge, MA, 2000.

S. Nolfi, D. Floreano, O. Miglino, and F. Mondada. How to evolve autonomous robots: Different approaches in evolutionary robotics. *Artificial Life IV*, pages 190–197, 1994.

S. Nouyan and M. Dorigo. Chain formation in a swarm of robots. Technical Report TR/IRIDIA/2004-18, IRIDIA, Université Libre de Bruxelles, March 2004.

M. A. Potter and K. De Jong. A cooperative coevolutionary approach to function optimization. In Y. Davidor, H.-P. Schwefel, and R. Männer, editors, *Parallel Problem Solving from Nature – PPSN III*, pages 249–257, Berlin, 1994. Springer.

M. A. Potter and K. De Jong. Evolving neural networks with collaborative species. In *Proc. of the 1995 Summer Computer Simulation Conf.*, pages 340–345. The Society of Computer Simulation, 1995.

M. A. Potter and K. De Jong. Cooperative coevolution: An architecture for evolving coadapted subcomponents. *Evolutionary Computation*, pages 1–29, 2000.

B. D. Ripley. The R project in statistical computing. *MSOR Connections. The newsletter of the LTSN Maths, Stats & OR Network.*, 1(1):23–25, February 2001.

F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, pages 386–408, 1959.

F. Rosenblatt. *Principles of Neurodynamics*. Spartan Books, New York, 1962.

H.-P. Schwefel. *Evolution and Optimum Seeking*. Wiley & Sons, New York, 1995.

R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, Cambridge, MA, 1998.

A. Thompson. On the automatic design of robust electronics through artificial evolution. In *ICES '98: Proceedings of the Second International Conference on Evolvable Systems*, pages 13–24, London, UK, 1998. Springer-Verlag.

V. Trianni. Evolution of coordinated motion behaviors in a group of self-assembled robots. Technical Report TR/IRIDIA/2003-25, IRIDIA - Université Libre de Bruxelles, Belgium, May 2003. DEA Thesis.

V. Trianni, R. Groß, T. Labella, E. Şahin, and M. Dorigo. Evolving Aggregation Behaviors in a Swarm of Robots. In W. Banzhaf, T. Christaller, P. Dittrich, J. T. Kim, and J. Ziegler, editors, *Advances in Artificial Life - Proceedings of the 7th European Conference on Artificial*

Life (ECAL), volume 2801 of *Lecture Notes in Artificial Intelligence*, pages 865–874. Springer Verlag, Heidelberg, Germany, 2003.

V. Trianni, T. H. Labella, and M. Dorigo. Evolution of direct communication for a swarm-bot performing hole avoidance. In M. Dorigo, M. Birattari, C. Blum, L. M. Gambardella, F. Mondada, and T. Stützle, editors, *Ant Colony Optimization and Swarm Intelligence – Proceedings of ANTS 2004 – Fourth International Workshop*, volume 3172 of *Lecture Notes in Computer Science*, pages 131–142. Springer Verlag, Berlin, Germany, 2004a.

V. Trianni, S. Nolfi, and M. Dorigo. Hole avoidance: Experiments in coordinated motion on rough terrain. In F. Groen, N. Amato, A. Bonarini, E. Yoshida, and B. Kröse, editors, *Intelligent Autonomous Systems 8*, pages 29–36. IOS Press, Amsterdam, The Netherlands, 2004b.

V. Trianni, S. Nolfi, and M. Dorigo. Cooperative hole avoidance in a *swarm-bot*. *Robotics and Autonomous Systems*, 2005. to appear.

V. Trianni, E. Tuci, and M. Dorigo. Evolving functional self-assembling in a swarm of autonomous robots. In S. Schaal, A. Ijspeert, A. Billard, S. Vijayakumar, J. Hallam, and J.-A. Meyer, editors, *From Animals to Animats $VIII$. Proceedings of the 8^{th} International Conference on Simulation of Adaptive Behavior*, pages 405–414. MIT Press, Cambridge, MA, 2004c.

J. Verzani. *Using R for Introductory Statistics*. Chapman & Hall/CRC, Boca Raton, FL, 2005.

M. D. Vose. *The Simple Genetic Algorithm: Foundations and Theory*. MIT Press, Cambridge, MA, 1999.

B. Yamauchi and R. Beer. Integrating reactive, sequential and learning behavior using dynamical neural networks. In *From Animals to Animats 3: Proceedings of the Third International Conference on Simulation of Adaptive Behavior (SAB 94)*, pages 382–391, Cambridge, MA, 1994. MIT Press.