Adaptive Behaviour Based Robotics using On-Board Genetic Programming

Anders Kofod-Petersen

For the degree of Cand. Scient.
Norwegian University of Science and Technology
2002

Abstract

This thesis investigates the use of Genetic Programming (GP) to evolve controllers for an autonomous robot.

GP is a type of Genetic Algorithm (GA) using the Darwinian idea of natural selection and genetic recombination, where the individuals most often is represented as a tree-structure. The GP is used to evolve a population of possible solutions over many generations to solve problems.

The most common approach used today, to develop controllers for autonomous robots, is to employ a GA to evolve an Artificial Neural Network (ANN). This approach is most often used in simulation only or in conjunction with online evolution; where simulation still covers the largest part of the process.

The GP has been largely neglected in Behaviour Based Robotics (BBR). The is primarily due to the problem of speed, which is the biggest curse of any standard GP.

The main contribution of this thesis is the approach of using a linear representation of the GP in online evolution, and to establish whether or not the GP is feasible in this situation. Since this is not a comparison with other methods, only a demonstration of the possibilities with GP, there is no need for testing the particular test cases with other methods.

The work in this thesis builds upon the work by Wolfgang Banzhaf and Peter Nordin, and therefor a comparison with their work will be done.

Acknowledgements

I would like to thank Asbjørn Thomasen and Trond Kandal for their invaluable help with the C programming. Wolfgang Banzhaf for pointers and good suggestions for improvement. Gorm Andersen for his assistance with the experiments. And, last but not least, my supervisor Keith Downing for his patience with me.

Contents

I Introduction **1**

1 Introduction **3**
1.1 Motivation 4
1.2 Layout of the thesis 4

II Theory and Approach **5**

2 Theory **7**
2.1 Introduction 7
2.2 Behaviour Based Robotics 7
2.2.1 Introduction 7
Behaviour based on neuroscience 8
Behaviour based on psychology 8
Behaviour based on ethology 9
2.2.2 Behaviour in robot 9
2.2.3 Behaviour based architectures 10
2.2.4 Adaptive behaviour 10
Adaptive individuals – learning 11
Reinforced learning 11
Fuzzy control 11
Artificial Neural Networks 11
Adaptive populations – evolution 13
2.2.5 Summary 14
2.3 Genetic Programming 14
2.3.1 Introduction 14
2.3.2 Definition 14
2.3.3 Genetic programming basics 15
Terminals and functions 15
Fitness and selection 15
Mutations 17
Crossover 17
Asexual reproduction 18
2.3.4 Summary 18

3 Related Work **19**
3.1 Introduction . . . 19
3.2 Work with genetic algorithms . . . 19
3.2.1 Introduction . . . 19
3.2.2 Simulation . . . 20
3.2.3 On-board . . . 21
3.2.4 Hybrid . . . 22
3.3 Work with genetic programming . . . 22
3.3.1 Introduction . . . 22
3.3.2 Simulation . . . 23
3.3.3 On-Board . . . 24
Introduction . . . 24
Basic model . . . 25
ADF model . . . 26
3.3.4 Hybrid . . . 27
3.4 Summary . . . 27

4 Approach **29**
4.1 Introduction . . . 29
4.2 The Khepera . . . 29
4.3 Genetic programming structure . . . 30
4.3.1 Individuals . . . 32
4.3.2 Reproduction . . . 33
Selection . . . 34
Crossover . . . 34
Mutation . . . 35

III Results and Evaluation **37**

5 Results **39**
5.1 Introduction . . . 39
5.2 General settings . . . 39
5.2.1 Function set . . . 40
5.2.2 Terminal set . . . 41
5.2.3 Selection, crossover, and mutation . . . 41
5.3 Test run . . . 41
5.3.1 Upside down . . . 42
5.3.2 Correct orientation . . . 43
5.4 Obstacle avoidance . . . 43
5.4.1 Test case 1 - The population way . . . 44
5.4.2 Result . . . 44
5.4.3 Test case 2 - The individual way . . . 45
5.4.4 Result . . . 46
5.5 Comparison . . . 48
5.5.1 Introduction . . . 48
5.5.2 Comparison between the two experiments . . . 48
Verification . . . 49
Individual vs. population solution . . . 49
Performance . . . 49

5.5.3 Comparison between this work and others 50

6 Discussion **51**
6.1 Introduction 51
6.2 Summary 51
6.3 The research area 52
6.4 Robots 52
6.5 Future work 53

Bibliography **53**

IV Appendices **61**

A Code **63**
A.1 Header file 63
A.2 Main code 71

List of Figures

2.1 Overview of a GP tree structure 16

4.1 The Khepera robot 29
4.2 Overview of the framework 30
4.3 Systems execution cycle 31
4.4 Overview of the individuals structure 32
4.5 Example of crossover 34

5.1 Fitness graph for first test run 42
5.2 Fitness and collisions graphs for second test run 43
5.3 Population first run 45
5.4 Population second run 45
5.5 Population third run 45
5.6 Population average 45
5.7 Three stages of behaviour 46
5.8 Individual first run 47
5.9 Individual second run 47
5.10 Individual third run 47
5.11 Individual average 47
5.12 Average number of collisions for both experiments 48

Part I

Introduction

Chapter 1

Introduction

The most exciting phrase to hear in science, the one that heralds the most discoveries, is not "Eureka!", but "That's funny..."

– Isaac Asimov

Within the last decade a huge amount of work has been made in the field of Behaviour Based Robotics (BBR).

Up until the mid-1980s the majority of work was based on the *sense-plan-act* paradigm. At this time Rodney Brooks developed his *subsumption* architecture, which is a purely reactive behaviour based method. As a result of Brooks work on BBR, the research in the later years have been mostly concentrated on the reactive behaviour, and less on the *sense-plan-act* paradigm; the later haven't completely vanished, since a lot of work is still done in the field of *Reinforced learning*.

For the last decade most of the research on BBR has been concentrated on using some kind of Artificial Neural Network (ANN) to control a robot. This has either been pure ANNs or ANNs in conjunction with some kind of Evolutionary Method (EM).

Most of the work on BBR have been directed towards the use of simulation, even though lately some focusing have been done on the use of simulation and on-board evolution simultaneous, and some work have been done on on-board evolution only.

One major approach within the field of EM has been largely neglected. Namely the field of Genetic Programming (GP). This is largely because of the time this method consumes when running, and some have argued that level at which a GP operates is not low enough. Some work has been done in trying to use GP to generate ANNs for controlling a robot, most of this work, however, has been done in simulation and then tested on the real robot.

Recently some work from Peter Nordin and Wolfgang Banzhaf has explored the possibilities of using a variation of the GP to evolve controllers for the *Khepera* robot on-board [NB95].

1.1 Motivation

Robots, or agents, working in the real world is faced with the problem of a very open domain; it is often hard to know any thing about the dynamics of the real world. This uncertainty leads us, according to Brooks [Bro87] and others, for an example see [BG92], [HHC+96], to the pragmatic view that reactive behaviour is more suited to "survival" than the more conservative *sense-plan-act* paradigm; as used by Nilson [Nil84].

The subsumption theory is not the only way of achieving a reactive behaviour. Several interesting approaches have emerged in the last decade. One of the most promising "bottom-up" approaches today is the use of an EM, often in conjunction with ANNs, to generate the controller for a particular robot and its task.

The motivation for this work can be divided into two major parts: first, the lack of GP in todays BBR, and secondly the problem of simulation, where it is very hard to construct a simulator which somehow mimic the world in any useful way.

This work will try to show that it is possible, and feasible, to use a GP running on-board on the robot.

1.2 Layout of the thesis

This thesis is organised into tree pieces. First the theory and background to support this work will be covered. The design choices and implementation will be described. Secondly, the implementation will be tested on *obstacle avoidance*. Last, but not least, it will try to draw some conclusions based on the work presented here.

Part II

Theory and Approach

Chapter 2

Theory

The science of today is the technology of tomorrow.

– Edward Teller

2.1 Introduction

This chapter will first cover the theory of behaviour based robotics, it will then describe an overview of the Genetic programming theory, and finally it will describe and discuss the pros and cons of on-board evolution.

2.2 Behaviour Based Robotics

This part will briefly cover a definition of intelligence; it will then describe what behaviour in animals is, and how it can inspire the use of behaviour based (artificial) systems, covering the more theoretical part first, and then the engineering view. Then a few behaviour based architectures will be covered; with a discussion of adaptive behaviour to follow.

2.2.1 Introduction

Before the topic can be covered in any meaningful way, some definition of robots and intelligence must be agreed upon. This work will adapt the definition stated by Arkin ([Ark98]).

> An intelligent robot is a machine able to extract information from its environment and use knowledge about its world to move safely in a meaningful and purposive manner.

Furthermore some kind of framework for the classification of behaviour is required. The science of behaviour can be classified into three major fields[†]; neuroscience, psychology, and ethology.

[†]Not exclusive in any way, and definitely overlapping

Behaviour based on neuroscience

From the field of neuroscience the study of the *central nervous system* (CNS), and the brains of humans and other animals has resulted in a number of theories concerning behaviour and the brain.

The brain of any mammal consists of three major parts: the fore brain, the brain stem, and the spinal cord.

The *fore brain* includes, among other things, the limbic system, which provides the basic behavioural survival responses; Thalamus, which processes incoming sensory information and outgoing motor activity; and Hypothalamus, which controls homeostasis. The later is the place which is most often associated with what is called *behaviour based systems*.

The *brain stem* includes, the mid brain, which processes incoming sensory information, and controls primitive motor response systems; the hind brain, which consists of the pons; the cerebellum; and the medulla oblongata, which connects the brain to the spinal cord.

The *spinal cord* is the reflexive pathway which is use for controlling the different motor systems. For a more thorough discussion of the brain see for an example: [GIM98].

Behaviour based on psychology

Psychology has historically focused more on the functions of the *mind* as opposed to the functions of the *brain*. The two most interesting fields within psychology, at least from the perspective of behaviour based robotics, is probably *behaviourism* and *ecological psychology*.

Behaviourism was founded by John B. Watson in 1913, as a reaction to the psychology of his time, which was primarily based on *introspection*. Watson argued that almost all behaviour is a result of conditioning, and that the environment shapes our behaviours by reinforcing habits[†]. The behaviourists based their theories on the *stimuli-response* paradigm, which states that to any stimulus some kind of response will be the result.

Ecological psychology owes a great deal to the American psychologist J. J. Gibson. He suggests a high correlation between the organism, it's environment, and how evolution affects it's development. In Gibson's own words:

> The word "animal" and "environment" make an inseparable pair. Each term implies the other. No animal could exist without an environment surrounding it. Equally, though not so obvious, an environment implies an animal (or at least an organism) to be surrounded.
> [Gib79]

Gibson was mostly concerned with the theories of visual perception, both in animals and humans. From his theories spring forth the two concepts of (in the terms of Brooks [Bro91]) *situatedness*, the robot (organism) is situated and surrounded by the real world; and *embodiment*, a robot (organism) has a real physical form.

[†]Perhaps one of the most known examples of this approach is Pavlov's dogs.

Behaviour based on ethology

Ethology is the study of animals behaviour in their natural environment, or as Niko Tinbergen puts is "the biological study of behaviour". Tinberg focused on four primary behaviour areas: *i)* causation, which is the study of those outside influences and internal states that lead animals to specific behaviour; *ii)* survival value, which is pretty much self explanatory; *iii)* development, which springs from the the controversy between comparative psychologist and ethologists, where the psychologists concentrated on *nurture* as opposed to the ethologists who stressed the importance of *nature*; *iv)* evolution, where ethology borders genetics, evolutionary biology, and ecology. For a more thorough introduction to ethology see [Sla85].

2.2.2 Behaviour in robot

Behaviour in robots comes in a broad spectrum. From the *purely reactive*, or reflexive; to the very *deliberate*. This work will lean heavily on the part of the spectrum concerning reactive behaviour.

The simplest way to view behaviour in robotics is to adopt the position advocated by behaviourists, that to each stimulus some reaction exists. In this view behaviour is seen as basic building blocks for actions. Probably the most famous example of this approach is *Braitenberg's vehicles* [Bra84].

The behaviouristic approach to robotics grows from the recognition that planning is often a waste of time; especial in a highly dynamical world, i.e. the real world [Bro87].

The reactive system, or reflexes, avoids the use of *explicit abstract knowledge* about the world which the robot inhabits. Since the construction of abstract world models can be very time consuming, this approach is very valuable in very dynamic and/or hazardous worlds. Three key concepts of the reactive view on behaviour emerges. Namely, *situatedness* and *embodiment*, which already has been discussed i section 2.2.1, and the third *emergence*, which states that intelligence arises from the interaction between the robot and the environment [Bro91].

The major issue in behaviour base robotics is where do behaviour come from? what are the right building blocks? and how do we define primitive behaviours? Several approaches has been tested throughout the short history of behaviour based robotics. Three major directions stands out: ethologically guided design, situated activity based design, and experimental driven design.

In Ethologically guided design the basis is the study of animals. A model of a particular animals behaviour is found in biology. The model is modified to suit any computational, or hardware, restrictions. The experiment is then conducted, preferably on a physical robot, and any result is compared with the original biological experiment.

One nice example of this approach can be found in [LWH98] where the phonotaxis of the female cricket *Gryllus Bimaculatus* was produced on a robot with a simple artificial neural network. The experiment was conducted under the same conditions as the biological experiment. The robot responded with phonotaxis to the calling song of real male Gryllus Bimaculatus. This work showed that phonotaxis could be achieved with a ANN much smaller than the model used when studying the real cricket. Furthermore, it showed that the close coupling between the morphological auditory matched filtering of the robot cricket, and it's simple neural control mechanism could explain the behaviour seen in nature.

The major point with this work was, not that the model of Lund et. al. is the way a real cricket works, but that this *very* simple model *could* explain the patterns seen in

nature.

Other work of the same kind has been done by Miglino and Lund, where they reconstructed an well known experiment on rats, where the rats are supposed to navigate through a labyrinth. Here they also showed that there was a much more simple model than the one used by most biologists, which is based on the rat making cognitive maps of the labyrinth [LM98],[ML99].

Situated activity based design is based on the approach that whenever a robot finds itself in a new situation it chooses one appropriate action to take. This means that any problem becomes one of perception. The robot have to recognise which situation it is in to select the proper response.

Experimental driven design is the "true" human bottom up approach. The principle is that one starts of with some limited amount of capabilities, test whether or not they work, debug the behaviour which does not work, add new features, and go through the loop again.

One very nice example can be found in [Bro89], where a six legged robot's ability to walk was developed. Several iterations were used, where different capabilities were added, such as force balancing, whiskers, and pitch stabilisation.

2.2.3 Behaviour based architectures

The design and building of behaviour based robotic systems has lead to a great variety of types. Each with their own idiosyncrasies; yet they exhibit a lot of common features.

First of all, the emphasis is on the coupling of sensory inputs to responses, as dictated by behaviourists (see section 2.2.1).

Secondly, the use of sub symbolic methods is prevalent, as opposed to representational symbolic knowledge.

Third, the use of decomposition is common, i.e. use of meaningful behaviour units.

For a very thorough discussion of different architectures see [Ark98], and for a overview of current work in the field see [GG96].

2.2.4 Adaptive behaviour

Up until now this part has covered the purely reactive robotic systems. Since reactive behaviour, or reflexes, only can help in a narrow group of problems, some kind of adaptation is needed to function in an ever changing environment.

Before continuing it would be fruitful to define adaptation. According to Merriam-Webster's Collegiate Dictionary, adaptation is:

> Modification of an organism or its parts that makes it more fit for existence under the conditions of its environment

Adaptation comes in two general flavours, *a)* adaptation in an individual within its own life time – learning, or *b)* adaptation within a population over time – evolution.

This part will first cover some different methods for introducing adaptation in an individual: *reinforced learning*, the use of *fuzzy* system, and *artificial neural networks*.

It will then describe the different approaches to applying evolution to a population of possible solutions (individuals).

Adaptive individuals – learning

Adaptation in an individual throughout its lifetime is learning. According to Merriam-Webster's Collegiate Dictionary, learning is:

> Modification of a behavioral tendency by experience (as exposure to conditioning)

Learning in robotic systems comes in a variety of flavours, here reinforced learning and fuzzy control will be covered briefly. Since the trend in the behaviour based robotics community has been to use ANNs, this will be cover more throughly.

Reinforced learning has its roots in behaviourism and the likes of Watson and Pavlov (see section 2.2.1). For reinforced learning to work an agent (robot) has to build on a certain model. In this model an agent has a set of sensors to observe the *state* of its environment, and a set of *actions* it can perform to change the state in which the agent is. Last, not not least, the agent needs a goal.

To learn the agent how to achieve it's goal reinforcement is used. A reward is given to each distinct action the agent can perform in each distinct state. So, a reward will be given to each state-action transition which achieves the goal, and no reward to all other transitions. One of the most popular learning algorithms used today is Q learning.

Since this is beyond the scope of this work, a more through description of reinforced learning can be found here [Mit97]. Examples of work done with robots and reinforced learning includes: [MB90], [Tan91], [Lin91], [MC91], and [Kae92].

Fuzzy control is based on fuzzy logic, which set itself apart from predicate logic by allowing values to be, more or less, members of different set, and not just the conventional true or false based membership. Fuzzy systems has been one of the AI worlds biggest success stories, largely due to the fact that "fuzzynes" is an integrated part of many things connected with humans; especial our language; almost everyone has used a sentence like: "This car is a bit more green than this one".

Fuzzy systems are excellent for a large group of problems, such as: maintaining an airplane in straight-and-level flight, hold the temperature of a room at 20 degrees, and dispatching elevators in a building [MF00]. Even though this approach should be well suited to solve behaviour based problems, only a few experiments has been conducted. The best known are properly the robot "Flakey" from SRI, and "MARGE" from North Carolina State University.

For a more through description of fuzzy systems, see for example: [MF00].

Artificial Neural Networks was first described in 1943, where McCulloch-Pitts, inspired by neurobiology, proposed a model of neurons [MP47]. This model was based on the assumption that neuron was either on or off. This led to a network structure where the network was composed of directed unweighted edges of *excitatory* or *inhibitory* type. Each unit is also provided with a certain threshold value.

Even though this kind of neuron is a very simple structure, it can been shown that all logical functions can be implemented with a McCulloch-Pitts network. Weighted networks and McCulloch-Pitts Networks are equivalent, and any *finite automaton* can be simulated (see [Roj96] for examples and references).

The major problem with the McCulloch-Pitts units is their lack of free parameters. Learning can only be implemented by modifying the way units are connected, and their threshold values.

This lack of flexibility leads to a more complex version of ANNs. In 1958 Frank Rosenblatt proposed the *Perceptron*, a more general computational model that McCulloch-Pitts units. In the 1960s The model was refined by Minsky and Papert [MP69]. The model proposed by Rosenblatt consisted of a whole network for the solution of pattern recognition problems, where the only significant difference from the McCulloch-Pitt unit was the presses of weights in the network. The learning algorithm proposed by Rosenblatt was the *Perception Learning Algorithm* (see equation: 2.1), where A is the set on input vectors where a perceptron computes the binary function $f_w(x) = 1; \quad \forall x \in A$ and $f_w(x) = 0; \quad \forall x \in B$

$$E(w) = \sum_{x \in A} (1 - f_w(x)) + \sum_{x \in B} f_w(x) \tag{2.1}$$

This function is defined over all of weight space and the purpose is to minimise it. Since $E(w)$ is positive or zero, we want to reach the global minimum where $E(w) = 0$. This is done by selecting a weight, by random, and then search the weight space for a better solution, in an attempt to reduce the error function $E(w)$ in each step.

Minsky and Papert modified Rosenblatt's model. By adding the possibility of predicates in the input layer, where the value of a single input bit can be calculated. These predicates can be of any format, e.g. a simple logical function, or a large program running on a super computer.

McCulloch-Pitts and Minsky-Papert was not the only ones conducting research on ANNs; But they were some of the more influential. Several others deserves to be mentioned, here I will only name two of the best known: Hebb, who described a learning paradigm. Where one of the core ideas was that the strength of synaptic connection was proportional to the correlation of pre- and post-synaptic potentials. Uttley, who in the late 50's developed a paradigm based on Shannon's theorem. From all the contributions these researchers have made, spring forth a great variety of ANN theories today. One of the most common topologies of ANNs today is the *feed-forward* network.

In general, the feed-forward network is defined as a graph whose notes are computing units, and whose directed edges transmit numerical information from node to node. Each node can compute a single primitive function, which transform input to output. The learning algorithm should find the optimal combination of weights. It is well known that a multi-layer network can compute a wider range of Boolean functions than a single-layer network. However the effort needed to find the correct combination of weight increases dramatically when the number of parameters grows. One of the most common learning algorithms today is the *Back-propagation algorithm* (BPA).

The *Back-propagation algorithm* minimises the error function in weight space using the gradient decent method. Since this method requires that the gradient function is calculated at each iteration it must be continuous and differential. One of the most popular functions for a back-propagation network is the *Sigmoid* (see equation 2.2).

$$f_c(x) = \frac{1}{1 + e^{-cx}} \tag{2.2}$$

Since the use of back-propagation is beyond the scope of this work, I will not go into details about the mathematics of this algorithm. Most of the work in BBR which uses ANNs, has a model where some kind of evolutionary method is used to evolve

ANNs (see section 3.2.1). One work which is based on the use of back-propagation for making controllers to a Khepera robot can be found here: [MWO+98].

Adaptive populations – evolution

Adaptation in a population over time is evolution. Again Merriam-Webster's Collegiate Dictionary can give a good definition before continuing, evolutions is:

> A process of continuous change from a lower, simpler, or worse to a higher, more complex, or better state

Artificial evolution used in BBR, and many other areas in computer science, is based on Darwin's work in the 19th century. Darwin was not the first to speculate about the evolutionary way of nature: Thomas Malthus had already written *Essay on the Principle of Population* in 1798 [Mal98].

Malthus had three major points: *i*) If a population is unchecked, it will grow exponentially. *ii*) The environment has limited capacity to support life (when a competition for resources arises it is known as the *Malthusian Crunch*), *iii*) More children are produced than can possibly survive to the age of reproduction.

Darwin extended the work of Malthus, and in his now so famous book *On the Origin of Species* [Dar28], he made the following points: *i*) When the Malthusian crunch comes, some phenotypes will be better equipped to deal with the environment, hence have a better chance of survival. *ii*) These individuals will have a higher chance of survival and reproduction. *iii*) If these individuals can pass on their advantages to their offsprings, this will increase the chance of survival for future generations. So the essence of Darwin's work on evolution can be, rather crudely, summarised as:

> Competition for resources + heritable variation => evolution by natural selection.

A very complete discussion of evolution, and genetics in general can be found in [Rou96].

Several people was inspired by the work of Darwin, and the potential they could see for using evolution as computer algorithms. The one who came to be one of the most influential was John H. Holland with his book *Adaptation in Natural and Artificial Systems* from 1975 [Hol92], where *Genetic algorithms* (GA) was, more or less, born.

Genetic algorithms, and most other evolutionary algorithms, basically works in the follow way:

1. Generate initial random population of genotypes.
2. Translate the genotypes to phenotypes.
3. Test each member against your fitness function.
4. Select the best fit individuals.
5. Perform crossover on the parents' genotypes to obtain children.
6. Mutate the children.
7. Perform step 2 through to 6 until you are satisfied.

In the standard GA the population typically consists of bit-strings (the genotype), which maps to some phenotype which fits a particular problem. Beside a random initial population, some kind of fitness function has to be supplied to test the individuals.

Evolutionary methods has lately become very popular in many fields within computer science, and extremely popular in behaviour based robotics. When behaviour based robotics uses a evolutionary method it is often referred to as *evolutionary robotics* (ER). The popularity of evolution in BBR is partly based on the notion that when evolution is used, the biased view a programmed should not explicitly hamper the robots possibilities; hence, the system can possibly construct solutions an engineer couldn't thing off.

This view is not blindly shared by all, Stone [Sto94] argues that trusting the objectivity of a evolutionary method only goes so far, and Harvey describes the limitation of standard GA methods in ER [Har97]. Nolfi, on the other hand, describes many positive sides with using an evolutionary method in robotics [Nol98]. Several examples of work done with different evolutionary methods can be found in chapter 3.

2.2.5 Summary

As it has been shown behaviour based robotics has threads back to the real world; in particular the world of psychology and biology. BBR draws upon theories from as diverse fields as neuroscience, psychology, ethology, and evolution.

Even though there is many different views on how BBR should be done something common stands out.

First of all, the concepts of *situatedness* and *embodiment* is central to most, if not all, work conducted in this field.

Secondly, learning and, more often than not, a genetic algorithm plays a big role in BBR.

One problem with the GA is its fixed representation; the chromosome it most often a bit string. When using a GA one must find a suitable way to encode the solution to the particular problem; this is still an art, and successful GAs often depends on the encoding of the problem. This problem is avoided in the extension to GA called genetic programming, where the population consists of computer programs.

2.3 Genetic Programming

This part will briefly cover the history of genetic programming (GP). It will then cover the basics of GP.

2.3.1 Introduction

Genetic Programming is a part of a major field within artificial intelligence called *Evolutionary Algorithms* (EA). Several importen contributions have been made to this field since it's beginning some forty years ago. (For a good summary of the history of EA, see [BNKF98]).

2.3.2 Definition

Since the work of Koza [Koz92] the most common definition of Genetic Programming has been the evolution of *tree-structures*. This is a very strict and confining definition,

so this work will adopt the definition given in [BNKF98]. Which states the following: (paraphrased)

1. The term *genetic programming* includes all systems that constitutes or contains explicit references to programs or to programming language expressions.

2. The definition of GP includes all means of representing programs.

3. Algorithms which are not primarily programs, such as artificial neural networks, should not be excluded from this definition.

4. This definition is not limited to the use of crossover; all systems which use a population of programs or algorithms for the benefit of search are included.

2.3.3 Genetic programming basics

This part will cover the basic terminology of GP. First the primitives used in a GP, and the importance of choosing them carefully. Secondly, the use of fitness and how selection in executed will be discussed. Then it will cover the two most important operators in any evolutionary system: mutation and crossover. Finally, the rules of reproduction will be covered.

Terminals and functions

The terminals and functions are the two primitives with which any GP will build it's structure. Both terminals and functions are referred to as nodes in the GP structure.

The terminals and functions for any GP fill two different roles. The terminals will provide the system with values. This means that any constants, sensor values from sensors attached to a robot, or functions which takes no arguments constitute terminals.

Using the tree analogy, any node which is a *leaf* is a terminal. Functions, on the other hand, are composed of any statements, operators, or functions available to the GP. Examples could be: boolean functions, arithmetic functions, conditional statements, loop statements, or subroutines. Using the tree analogy, any node which is a *branch* is a function (see figure: 2.1).

When it comes to choosing the terminals and functions for any GP run, some care must be taken. It is important not to choose the function set to strictly. For example, using only AND will probably not solve any very interesting problems. But, on the other hand, it is important not to choose a set which is too large. Often a parsimonious approach, or the use of *Occam's razor*, is recommended.

The same carefulness is also importen in choosing the constants in any GP. The ability of the GP to combine chosen constants via arithmetic functions, will often compensate for a small number of constants.

Fitness and selection

For GP to work it must somehow choose which subset of the population is to undergo the genetic operators such as mutation, crossover, and reproduction. To do that GP utilises one of the most importen cornerstones in biologically evolutionary theory, namely *fitness based selection*. Where the best suited individual for a specific task (in biology often survival) will be allowed to reproduce.

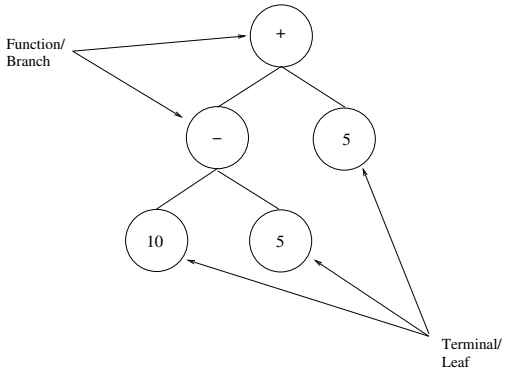

Figure 2.1: Overview of a GP tree structure

This theory which was first proposed by Charles Darwin [Dar28], inspired by the 1798 essay by Thomas R. Malthus, *On the Principle of Population*, where the *Malthusian Crunch* was introduced (see section 2.2.4). Two other well know theories deserves to be mentioned her: First, Wright's theorem which states that *natural selection increases the adaptation of individuals to their environment*; and Fisher's fundamental theorem of natural selection which states that *evolution requires fitness variance, and the more variance, the faster the population evolves*. See [Rou96] for a more through discussion of both theorems.

Since GP, as natural evolution, utilises the variation in fitness and selects the most fit, some sort of fitness function must be provided. Fitness functions can be very different depending on the problem which is to be solved. A fitness function must be able to measure how well a given program is to predict the output from the programs input.

Examples of factors used in fitness functions could include:

- The number of walls hit by a robot.
- Time spent travelling in a straight line.
- Area cover in exploration.
- A combination of any above.

Once the GP has assigned fitness to all of the tested individuals selection must be initiated. Selection can be viewed as the competition between individuals within the subset[†] of the population chosen for testing.

Several standard selection mechanisms exists, including *fitness-proportional*, *rank*, *tournament*, *sigma scaling*, and *elitism*. Here the first three will briefly be described.

Fitness-proportional selection is perhaps the most common, and has definitely been the primary choice for many since it's introduction by Holland [Hol92]. An individuals

[†]The subset can of course include all individuals in the population

chance to form offspring in the next generation is given by equation 2.3; where the probability of a particular individual's selection, is the fitness of the individual divided by the fitness of the whole population.

$$p_i = \frac{f_i}{\sum_j f_j} \tag{2.3}$$

The fitness-proportional selection has been criticised for it's tendency to fuel *premature convergence*. Premature convergence occurs early in an evolutionary run, when the fitness variance is high, and a limited number of individuals will multiply quickly, thereby preventing the GP from doing any more exploration.

The fitness-proportional selection has also been criticised for attaching differential probabilities to the absolute fitness value.

Rank selection is another often used mechanism. The fitness value assigned to a particular individual is a function of the rank within the population. For linear ranking, which is often used, the probability is a linear function of the rank (2.4).

$$p_i = \frac{1}{N}\left[p^- + (p^+ - p^-)\frac{i-1}{N-1}\right] \tag{2.4}$$

where $\frac{p^-}{N}$ is the probability of the worst individual being selected, and $\frac{p^+}{N}$ that of the best being selected, and (2.5)

$$p^- + p^+ = 2 \tag{2.5}$$

should hold if the population size is to stay constant.

Last, but not least, is tournament selection. This mechanism separates itself from the rest by not using the whole population, but only a true subset. Some group of the population, the tournament size, is randomly selected, and a selective competition is initiated. The traits of the fitter individuals are then allowed to replace the less fit.

Tournament selection has one major advantages over, for an example, fitness-proportional selection. It does not require any centralised comparison module. This allows more parallelising and a general speed up of a GP.

Mutations

Mutation in GP and other Darwinian based systems, serves two beneficial purposes: *i)* Mutation sets the stage for evolution. Without mutation, there can be no variation in genotypes, hence no basis for evolutionary change (The Hardy-Weinberg law and Fisher's Theorem). *ii)* Mutation works as a rescue operator; given that the GP is stuck in one, of potentially several local maxima, mutation can push the population off the peak, and hopefully towards the global maximum.

Mutation, on the other hand, can also have a negative effect. If, for instance, the mutation rate of a given system, natural or artificial, is to high, all the beneficial effects of mutation will drown in its destructive forces.

Crossover

The crossover operator is, by far, the most importen operator of the GP. Given a tree based GP the crossover operator works in the following manner:

1. Select two individuals, based on the selection mechanism.
2. Select a random subtree in each parent.
3. Swap the two subtrees or sequences of instructions.

The crossover operator closely mimics the process of biological sexual reproduction. Based on this notion, the crossover operator has been used to claim that GP search is more efficient than methods based solely on mutation. Koza [Koz92] argued that a population in a GP system contains *building blocks*; a building block can be any tree or subtree which is present within a fraction of the population. This hypothesis follows the same line of argument as the *Building block hypothesis* from genetic algorithms [Hol92].

Given the building blocks and crossover the systems should be able to combine "good" building blocks to more fit individuals, and small building block should be able to combine into larger building blocks, hence, making the GP search more efficient than other methods. A more thorough discussion of the viability of this hypothesis it outside the scope of this work, and can be found in chapter 6 of [BNKF98].

Asexual reproduction

Asexual reproduction is, by far, the easiest operator to handle. When an individual is selected, it is copied, and there are now two of the same individual in the population.

2.3.4 Summary

Genetic programming is a method which has shown great promise in a lot of areas. However, it still possesses some problems or challenges. First of all, the choice of functions and terminals is a major issue. Secondly, the mutation rate of a system must be carefully chosen, since mutation can go both ways. Last, but not least, lack of speed has long been the major problem with GP.

Chapter 3

Related Work

I can see as far as I do because I stand on the shoulders of giants

– Issac Newton

3.1 Introduction

This section will first cover some of the work done in the fields of GAs and ANNs, then it will describe the work done with GP and robotics, most notable the work by Nordin and Banzhaf.

3.2 Work with genetic algorithms

3.2.1 Introduction

Within the last couple of years a new approach in BBR has evolved; this involves some form, of simulated evolution. What is to be evolved has been heavily debated. One approach which seems to be growing is the use of a GA to evolve an ANN.

The research within the field of GAs and ANNs can be divided into three groups: *i)* Those who use simulation only. *ii)* Those who use on-board evolution only. *iii)* Those who use some kind of hybrid approach, i.e. simulation and on-board. Nolfi et. al. defends the approach with GAs and ANNs by stating the following reasons. [NFMM94] *i)* "Neural networks can easily exploit various forms of learning during a life-time", and like *the Baldwin effect* [Bal96] in GA shows, the "learning process may help and speedup the evolutionary process" [HN87], [FM90], [FP96]. *ii)* Neural networks are notable for there resistance to noise which is highly present in robot interaction with the real world. *iii)* The common agreement today is that primitives manipulated by evolution should be on the lowest level possible, to avoid the possibility of undesirable choices made by a human programmer.

To support this approach Nolfi et. al. conducts three experiments with a Khepera robot, unfortunately all of these experiments were only supposed to roughly evolve the same ability, all of which were some kind of exploration. The experiments were

divided into: evolution of controller in simulation only, and then testing on the physical robot; evolution on-board; and a hybrid solution, where the first 300 generations were in simulation and the last 30 on-board. Furthermore they used fixed ANNs, a single layer feed forward net with eight input- and two output nodes. As Stone [Sto94], and Harvey [Har92] notes, there is always some degree of human meddling no matter how low a level we evolve from. The use of fixed nets is not a perfect fit with the point of the risk of undesirable choices made by a human programmer.

They did, however, show that it is possible and feasible to utilise a GA in constructing controllers for autonomous robots. They also concluded that it is better to employ the physical robot in the process, not in an on-board situation only, but in a hybrid way. Several other examples of use of GAs and ANNs has been described. Here I will divide the examples into three groups: simulation only, on-board only, and hybrid.

3.2.2 Simulation

Nolfi demonstrated the possibilities of evolving non trivial behaviour, specifically a garbage collection robot, based on the Khepera robot with a gripper attached [Nol96]. Here several basic behaviour were pre-generated, such as: *leave-nest, get-food, avoid-obstacles*. Then an ANN was constructed consisting of a feed-forward net with 7 sensory nodes, 16 motor nodes, and no internal nodes. The controller was evolved in simulation and then downloaded on the real robot for testing. He was able to show that the best fit individual was able to clean the arena of garbage.

Animals and robots needs to function in a dynamic environment, for this they need a palette of different types of behaviours. In an example given by Yamauchi and Beer [YB94] foraging for food requires the following behaviours while exploring the environment. It needs to react to any threats or obstacles, remember the way back to it's nest, and the location of any food sources. This means that if the animal is to forage in a efficient way, it requires the capabilities of "reactive and sequential behavior, as well as the ability to learn from experience".

To show that it was possible to evolve these behaviours Yamauchi and Beer implemented a 8-node continuous-time recurrent neural network, where the network parameters were encoded as a vector of real numbers in the GA. They then set up the task of *landmark recognition* to test this. The ANN was evolved in simulation and then tested on the physical robot, which was a Nomad 200.

After 15 generations, the network was capable to recognise the landmarks in simulation. It was then transfered to the robot where is was able to correctly classify the landmarks in 85 % of the test cases.

Harvey et. al. advocates the use of GA to evolve dynamical neural networks in simulation [HHC93]. They do, however, take the ANNs a step further. They describe a continuous real-valued networks with unrestricted connections and time delays between units. For their implementation the use a network with a fixed number of input- and output nodes, and no specified number of hidden nodes.

They also stress the importance of the simulator being kept as close to reality as possible. They put forth the following techniques to do this: *i)* The simulator should be calibrated with the robot hardware involved. *ii)* Empirical data should be the base of sensor simulation. *iii)* Noise must be taken into account. These points are also elaborated in [JHH95]. The last point which was stressed by Harvey et. al. was that every robot which is going to do any thing useful; must have vision. To defend these view they implemented a simulator where the ANN was evolved, which should develop

wandering behaviour in an office like environment. Within the 50 generations available to the simulator wandering behaviour was evolved.

To sum up, it is clear that this approach is useful for developing controllers for robots. Several prerequisites most hold for this approach to be successful. Firstly, the simulator must be as close to the robots reality as possible, meaning, for an example, that simulated sensors should be sampled from the real robot, or that noise should be generated to compensate for uncertainties in the real world. Secondly, the tasks seems to have to be somewhat limited, for this approach to work; which perhaps, according to Harvey et. al., could be remedied by vision. Lastly, perhaps the solutions where network parameters have been pre-defined, should be extended to open-ended evolution; as argued in [Har92].

3.2.3 On-board

As described above it is better to employ the physical robot in any work done with a simulator. This enhances the performance once the real world is tackled.

The extreme approach is to run everything on the physical robot, i.e. the GA/ANN framework and the testing. This *on-board* approach ensures that any physical limitations, or noise, will be handled by the system.

Some of this work has already been described (see section 3.2.1), by Nolfi et. al. [NFMM94]. More thorough investigations has been done by Floreano et. al. [FM96a], where they describe a pseudo on-board system which consists of a Khepera robot connected to a work station. The work station generated the simple ANNs which only had three neurons; one input, one hidden, and one output. All of which received synaptic connections from all eight infra red sensors, and the hidden node. These were all downloaded onto the Khepera where each node could change its strength according to one of four *Hebbian* rules: *i)* pure Hebbian. *ii)* post-synaptic. *iii)* pre-synaptic. *iv)* covariance.

The task were to follow the walls around in circular environment with an obstacle in the middle.

After around 50 generations the best fit individuals were able to keep a straight trajectory around the area, without hitting the obstacle. Similar results were achieved in [FM94] were a similar problem were solved by only evolving synaptic strength. The results in the later work were, however, better [FM96b].

Steels argues for four major point in the use of evolution for development of robot controllers [Ste94]: *i)* the use of a sub- or pre-symbolic level in evolution. *ii)* The use of a "level playing field" where modules can not exhibit each other. *iii)* The use of on-board (on-line) evolution. *iv)* open functionality, as in [Har92].

To test these statements he constructed the *PDL robot architecture*. This architecture consists of three basic units. *i)* Quantities, which can be external, like sensor values, or internal, like world states. *ii)* Processes, differential equations which are dynamical relations between quantities. *iii)* Behaviour units, a regularity within the interaction dynamics between the robot and it's environment.

Furthermore, this architecture is guided by these two principles. All behaviour systems are active at the same time, and the influence of the different behaviour systems are summed to select the appropriate behaviour; this is also known as Arbitration.

Steels use two experiments, one where he was suppose to evolve the *behaviour system* which enables the robot to move forward and backward, and another where the ability to come to a hold was sought after. Both problems were easily solved within an acceptable time frame.

3.2.4 Hybrid

As it has become clear there are problems both with only using simulation or on-board. Several people has tried to improve the quality of robot controllers by using a hybrid method.

One interesting work by Miglino et. al. addresses several of the problems both with simulation and on-line running [MLN95]. Their work was made on a Khepera robot. Where the weights of an ANN was evolved in simulation, and then tested on the real robot.

One of the major problems with simulation is the lack of perfect hardware. When a robot has several sensors they should return the same value under the same circumstances; alas, no hardware does that. So the simulator used by Miglino et. al. was build on a sample of the real world, through the physical sensors on the Khepera. This approach greatly improved the simulator.

Miglino et. al. also investigated the introduction of "noise" into the world model. This was based on the hypothesis that the world is dynamic, hence noisy. The introduction of noise also improved the performance of the robot, once it was tested in the real world. The noise issue has been investigated by many people, such as Jakobi [Jak97], Reynolds [Rey94b], and Jakobi et. al. [JHH95].

One thing which was observed by Miglino et. al., was that no matter how well they constructed their simulator a decrees in performance was often observed when the controller was introduced to the real environment. This problem was nicely solved by running the last few generations on the real robot.

This work demonstrated that it can be feasible to run a simulation of a particular robot; as long as the problems of imperfect hardware, and the "noisiness" of the real world is addressed properly. The major benefit of simulation is the speed of the entire process. Simulation speeds up the otherwise tiresome process of running a *full generational replacement* on a real robot.

Other work by Miglino and others also stresses the point of the simulation/embodiment solution. One work [MNT95], where wandering behaviour was evolved by an GA evolving the weight for an ANN; this time using a Lego^{TM} robot. They used one population which ran in simulation only. One population where the evolutionary part was made on a workstation, and then tested on the real robot.

As it could be expected the real-world population outperformed the simulated population, even though not by much. And the introduction of noise improved the simulated population.

The conclusion in this work argued pretty much the same as in [MLN95], that the simulation/embodiment solution is feasible, and the solutions which is produced by this hybrid approach can perform acceptable.

Quite a lot of work has been done with hybrid solutions, several of which has already been mentioned, others include Lewis and Fagg [LF92], Nolfi et. al. [NFMM94], Grefenstette and Schults [GS], Yamauchi and Beer [YB94], and Lund and Miglino [LM96].

3.3 Work with genetic programming

3.3.1 Introduction

Historically most of the work done with adaptive BBR has been done with GAs and ANNs. More recently, with few exceptions, some work has begun to explore the pos-

sibilities of using GP as the evolutionary method.

3.3.2 Simulation

Properly the first experiment with autonomous robots and genetic programming is the work of Koza and Rise [KR92], where they simulate a box pushing robot. The robot's controller was evolved using a standard full generational replacement GP. The GP successfully found a solution in 45 generations.

Craig W. Reynolds expanded the work by Koza and Rise. Both used high level functions, such as *move-forward*, *turn-left*, and *turn-right* [KR92]; *turn*, and *look-for-obstacle* [Rey94a] (They both used arithmetic functions beside these high-level functions). The major difference between the work of Koza and Rise, and this is the use of a *steady-state* GP instead of full generational replacement. *Steady-state* is the opposite of *full generational replacement*, here only a sub-set of the population, instead of the whole population, is tested in each iteration.

Even though, both of these experiments clearly shows that it is possible to evolve controllers using a GP, they also, [Rey94a] most prominent, show that *robustness* is quite another matter.

The robustness of GP evolved controllers and a solutions to the problem is described in [Rey94b]. Reynolds repeated his experiments from [Rey94a], except that he added noise to the simulation; here noise is defined by taking the "worst of four noisy trails", as in [HHC93].

As discussed in section 3.2.4, noise plays a very beneficial role in evolving robust controllers. This work by Reynolds shows that the simulation-noise connection also holds for GP. And that it is possible to evolve robust controllers using GP.

Bennett expanded on the work by Koza and Rise. Inspired by parts of psychology, which argues that the mind works in a distributed way, and that there is no central unit which decides everything, he was interested in evolving a controller that could cope with several different problems at once: *lawn mower*, *wall following*, *gold collector*, and *mine sweeper* [III97].

The simulated world which this robot inhabits is made up of four squares – one for each problem – which are connected in the middle. The GP structure consisted of a "top tree" which, over time, contains sub trees for each problem. The evolutionary system worked on two levels: *i*) The meta-level, where the top tree was evolved, *ii*) The sub-level, where an ordinary evolution of the GP's *automatically defined functions* (ADF) was evolved.

With this work Bennett showed that a GP could discover good programs, which could solve different problems by organising the control program so that it could select the appropriate behaviour when needed.

Even more advanced work has been done by Lee et. al., who presented some work where they used an *evolutionary island* model to evolve behaviour for a Khepera robot [LHL97] The controllers which were evolved were defined at the logic level. The controller works almost as a *boolean network*, where *conditional structures* are used to select appropriate motor commands and activating behaviour primitives. The GP system was used to evolve these boolean controllers.

They evolved behaviour for *box-pushing* and *box-side-following*, each was controlled by a separate behaviour primitive. The GP consisted of three non-terminals: *dummy root nodes*, which was used to connect some arbitrary number of sub-trees,

here a PROG[†] which allows several instructions to be executed in a sequence; *logic components*, which were AND, OR, NOR, XOR; and *comparators*, which were $>=$. The terminals were sensor values and thresholds, all were between 0 and 1 inclusive. The controllers were evolved in simulation with sampling of the robots sensors, as described in [Nol96]. When a suitable controller was evolved, in 50 generations, it was transfered to the robot for testing.

When the two primitives were evolved, box-pushing and box-side-following, a *arbitrator* was evolved to decide when the two primitives should be used.

In line with the work of Bennett [III97], this work by Lee et. al. showed that it is possible and feasible for a GP to evolve the special structure of behaviour primitives, and to solve more complex problems than the typical *wall-following* and the like.

Another work by Koza et. al. [KBKA97], investigated the use of GP for evolving controllers to navigate a robot safely to any arbitrary point in minimal time. This work is more or less an extension to the work of Bennett [III97], with the little twist, that they also use a GP to evolve an analog electrical circuit to use with their evolved controller in a robot. Other work done in this field includes [Han94].

3.3.3 On-Board

Since GP tends to be very time consuming there hasn't been much work done in real time on-board GP in robots. However, lately some work has emerged from Department of Computer Science, University of Dortmund led by Wolfgang Banzhaf and Peter Nordin [NB97b]. This part will cover this work, which is the base of the work in this thesis.

Introduction

Banzhaf et. al. has been working on a genetic program running on-board a Khepera robot. The work has been done in two major parts: *i*) evolving controllers for simple problems, e.g. *obstacle avoidance* and *wall following* [NB95]. *ii*) more advanced hierarchical structures, where they utilise the basic behaviour evolved in the first part [ONB96].

All of the experiments is conducted using a GP where the individuals consists of variable length strings of 32 bit instructions for a register machine. The GP individual is represented in a linear fashion, where each node is an instruction for the register machine. The 32 bit instructions represent basic arithmetic operations such as $a = b + c$ or $d = b + 9$. The actual format of the instructions is the machine code format of a SUN-4 (Sparc). The population is typically quite small, i.e. under 50 individual.

The GP itself is a *steady-state* system using *tournament* selection with the size of 4, where candidate one and two returns one parent, and candidate three and four supplies the second parent. Both "losers" are deleted and the children from the crossover are added to the population instead of the two "losers".

The crossover is a *two-point string crossover*, where each node is the crossover unit. Crossover can occur on either side of the node, but because of the integrity of the program, never inside a program. Mutation flips bits within a 32-bit block, and makes sure that only legal functions, variables, or constants occurs [NB97a].

The design of this system are based of the following assumptions [BNO97] (paraphrased):

[†] A construction very familiar to any LISP programmer

- Any behaviour can be produce with a general purpose language.
- GP produces symbolic output, in contrast to ANNs.
- Goal definition is only done by deciding on a fitness function.
- GP has a built-in tendency to generalise from presented situations.
- This particular type of GP is fast, memory efficient, and can run on a robot with very limited computational power.

Another problem which is removed in this GP, is the need for the same starting point for all test cases. This is handled implicit by the way that each individual is tested against a real-time fitness case, thereby making a probabilistic sampling of the environment [NB95]. This approach could give rise to "unfairness" in testing individual, as some individual could have more "luck" with their starting point. But this problem is removed over time.

The randomness in the starting point for each individual results in two types of survival strategies: *i*) *cooperative* strategy where the individual manages to fulfil the fitness criteria, even under poor conditions. *ii*) *competitive* strategy where the individual tries to *minimise* the fitness of the other individuals by placing the robot in situations which are very hard to get out of [BNO97].

The two training environments were: one small rectangular shaped world 30×40 cm., and one larger and irregular shaped rectangular world 70×90 cm., where object could be placed at arbitrary locations [NB97a].

Basic model

In the basic model the interest was on evolving *obstacle avoidance* in a *sense-think-act* context. The GP systems evolved controllers in real time, using real noisy sensorial data [NB95].

This experiment was not compiled to run autonomously on the Khepera, it ran on a workstation were it communicated with the Khepera for sensory information, and for sending information to the motors. Hence, this experiment was not *on-board* but *on-line*.

The small population, typically less that 50 individuals, of genetic programs each used six values from the sensors, and produced two output values for the motors. Each individual does this autonomously. The fitness function was divided into two parts; a pain and a pleasure part. The pain part was just the sum of all sensory input, and the pleasure part was assigned by measuring how log the robot ran straight and fast. See equation (3.1), where m_1 and m_2 are motor 1 and motor 2, and p_i is the sum of the sensors. The goal is to minimise the function.

$$f = \sum p_i + |15 - m_1| + |15 - m_2| + |m_1 - m_2| \tag{3.1}$$

The function set for the GP in this experiment was: *addition*, *subtraction*, *multiplication*, *shift left*, *shift right*, *and*, *exclusive or*, and *or*. The population size was 30, crossover probability was 90% and mutation probability was 5%. The system also had 256 nodes as maximum length of any given individual.

This experiment showed that *exploratory behaviour* came right away. This was due to the diversity in the initial randomly generated programs. During the first minutes, the robot bumped in to the wall, but over time the bumping became less and less frequent.

After about 20 minutes, which is approximately 100-150 generations, the robot had learned obstacle avoidance. In the more complex environment it took around a hour to evolve obstacle avoidance.

The obstacle avoidance experiment was expanded to include *object following* [NB97b]. In this work the set up was almost identical to the first experiment. There was a few differences: *i*) To make sure that the algorithm was not brittle, it was evolved in the complex environment, and then tested in an even larger environment (100×100) cm. *ii*) In the object following experiment a new fitness function was used (see equation 3.2), where s_1 to s_4 is the four forward looking sensors. *iii*) The algorithm was cross-compiled and tested on-board; giving much the same result.

$$f = (s_1 + s_2 + s_3 + 2_4 - 1000)^2 \tag{3.2}$$

For the object avoidance experiment the results were identical to the ones in [NB95]. It took around one hour to learn good object avoidance, and i 90% of the experiments the robot learned to reduce collisions to fewer than two per minute.

The object tracking experiment the robot actual learned the appropriate behaviour faster than the case of object avoidance; it took around 30 minutes. The reason could most properly be the relative easier fitness function.

The most interesting part of this work is the use of on-board evolution. Both tests were executed on-board with the same framework as the on-line version. Both performed identically. The only problem were that the battery time on a Khepera is approximately 40 to 60 minutes, which is very close to the minimum time required for training.

Both papers showed that it is possible to evolve robust controllers for a robot using GP. The most interesting part is that it is also possible to evolve in total seclusion.

ADF model

As an extension to the work described in the last part, Banzhaf et. al. investigated the possibility of evolving high-level action selection based on the basic behaviour mentioned above [ONB96].

This work uses the on-board version of their GP. This GP consists of five populations, one for each action primitive and the control structure:

go ahead Moving straight at maximum speed.

avoid obstacle Avoid obstacle at maximum speed.

seek obstacle The inverse of avoid obstacle.

find dark Searches for darkness to move to.

select action The populations of selection mechanisms.

When the system is running it feeds data into the data registers of each action selection mechanism. Four are selected for tournament selection, they each select one of the four action primitives, through tournament selection. The winners replaces the losers and genetic operators are used.

The performance of the system was tested in three different experiments: *collision avoidance*, *wall following*, and *hiding in the dark*. As an example, the robot showed effective collision avoidance after 800 cycles.

After a period of four to seven minutes, the robot shows robust behaviour with it's action primitives.

This work was even further extended in [BNO97], where the action primitives and action selectors were evolved. Further more, a *memory-based* GP was introduced. This splits the GP into two separate processes. One which communicates with sensors and motors as well as storing events, 50 event vectors are recorded. The other process tries to learn an induce a model of the world.

This memory addition speeds up the learning process with about a factor 40. Further more, an interesting thing which occurred was that a kind of "childhood" had to be introduced to make the system work in the best possible way. When using the standard memory model, a FIFO model, the robot forgot important early experience. The solution was to introduce a childhood, where the robot learns quickly, and the memories are hard to forget. The robot performed better if it became harder and harder to memorise events the older it got.

To achieve a robust system with learning, the population had to be increased from 50 to 10.000 [NB97a]. This removes the possibility of a true autonomous robot, since only 256KB of memory is available on the Khepera.

All in all, this work shows that primitive behaviour as more than possible to evolve using GP, both in an on-line and on-board fashion. Even more intelligent, or high level behaviours, can also be evolved by the same technique. Yet, the robot's limited hardware removes the possibility of true autonomous running when learning is included; however, this is a practical problem, which can be solved cheap and easily.

3.3.4 Hybrid

Since the use of on-board GP still is very limited, there is not much literature available beside the work done by Banzhaf et. al. Hence, not very much has been done on hybrid solutions either.

Marc Ebner has done some work on the use of GP both in simulation and on-board [Ebn98]. He investigated if a GP could evolve a controller for a large mobile robot (a Real World Interface B21); Not only a program for avoiding walls, but also a hierarchical structure á la ADF.

First an experiment in simulation of the world and the robot was conducted. Here several fitness cases were tested, and the most successful were then used on-board. the on-board evolution followed the same guidelines as the simulated. The experiment had a population size of 75, tournament selection with size 7 was used, the maximum time per fitness case were 300 s., and the whole system was run for 50 generations. The simulated solution and the on-board solution came out almost equal.

Due to the time problem (evolution took 197 hours, the whole experiment 2 months) only one run was performed. The experiment still showed that it is possible to evolve a hierarchical control architecture with GP. Ebner also concluded that it should be feasible to use computer simulations, and especial to find the main parameters for the run on the real robot.

3.4 Summary

This chapter has describe a lot of the work done with autonomous behaviour based robotics today. Most of this work includes some kind of evolutionary method; most often a genetic algorithm which evolves artificial neural networks. The technique has

proven most successful when simulation is used, preferable in conjunction with on-line evaluation. This goes nicely hand in hand with Brooks [Bro91] *situatedness* and *embodiment*. ANNs and GAs in particular are sensitive to certain problems. According to Banzhaf et. al. [BNO97] GP can solve many of these problems.

The major problem with GP has for a long time been the slowness of the system, but this problem has also been solved – it can even be faster than related evolutionary systems evolving ANNs ([NB97a] p. 17).

All in all on-board, or at least on-line, GP seems to be a promising approach in behaviour based robotics.

Chapter 4

Approach

Beware of bugs in the above code; I have only proved it correct, not tried it

– Donald E. Knuth

4.1 Introduction

This chapter will cover the work done with the robot, the design and the implementation of the GP. First an introduction to the Khepera robot will be presented; secondly, the design of the GP with details from the implementation will be covered.

4.2 The Khepera

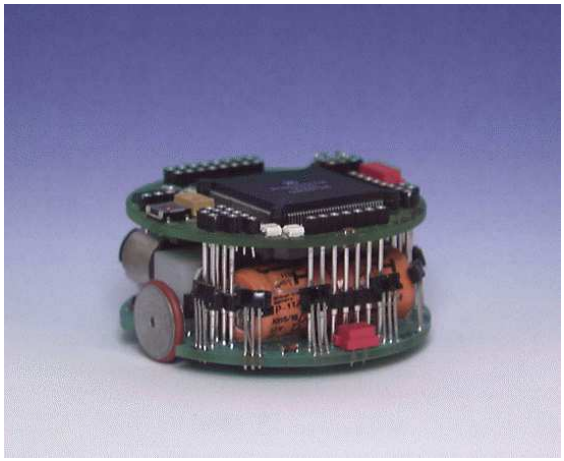

Figure 4.1: The Khepera robot

The experiments in this work has been carried out on a standard autonomous mobile robot (see figure: 4.1), the Swiss made mobile robot platform Khepera [MFI93].

The Khepera has eight infrared proximity, or light intensity, sensors; all of which are distributed around the robot, in a circular pattern. The robot has a diameter of six cm. and a height of three cm. It is also equipped with two separately controllable motors; a Motorola^{TM} 68331 processor with 256k of memory; a ROM containing the operating system which has simple multitasking; and the possibility of connecting it to a workstation with a serial cable.

There are two possible ways of controlling the robot. Either the algorithm could be run on a workstation, communicating with the robot through a serial line; or the algorithm could be complied for the Motorola^{TM}, and down-loaded to the robot, which then runs the entire program.

4.3 Genetic programming structure

Here a short introduction to the framework will be given. Then *individuals*, *initialisation*, *selection*, *crossover*, and *mutation* will be covered more throughly.

The structure chosen for this work is loosely based on a synthesis of the work done by Banzhaf et. al. (covered in section 3.3.3), and the work by Keith and Martin [KM94].

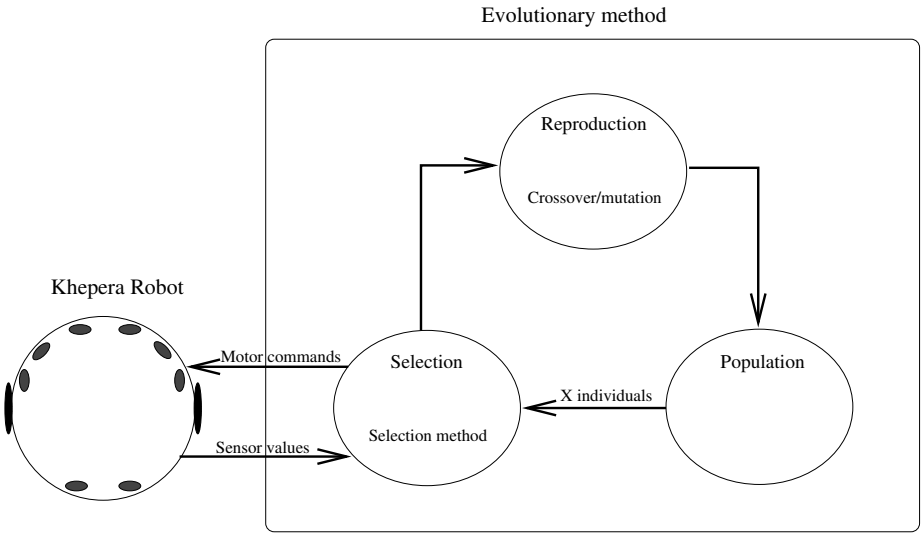

Figure 4.2: Overview of the framework

Since robots should behave in an autonomous fashion ([Bro91] [NB97b]). The system run entirely on the physical robot, as can be seen in figure 4.2 (Adopted from [BNO97]). The robot is connected to the workstation, solely for dumping statistical information and for receiving power.

This setup contains a population of C programs, which are exposed to selection. All input from the world arrives through the eight infrared sensors on the Khepera. Outputs are written to the two motors, and values from both sensors and motors are used for selection (fitness calculation).

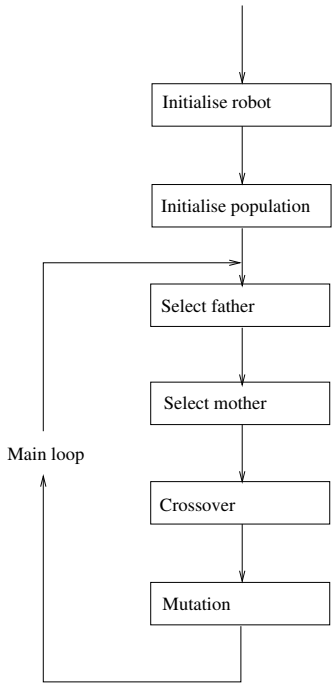

Figure 4.3: Systems execution cycle

The choice of ANSI C as the language is based on the work of Banzhaf et. al. They, however, used an assembler language for their evolutionary generated programs. I argue that even though the speed of the system properly will diminish when using a general purpose language, as opposed to assembler, the usability of evolved programs are higher with this setup. Not only will the code generated be portable, but it will be much more readable; and the long term goal here is not to make very efficient and commercial successful robots, but to understand what is going on.

The overview of the GP made for this thesis is:

- Implemented in ANSI C.
- The GP uses a *steady state* method, and *tournament selection* with two times two individuals for the tournament.
- The crossover is one point random for each parent.
- The two winning parents are kept in the population.
- Mutation is either changing one terminal for another, or changing one function.
- Individuals are a pointer-based dynamic structure. i.e. no constrains on the length of the individual.
- Compiled on a Solaris workstation using GCC 2.8.1 cross-compiler for the Motorola^{TM} 68k processor.
- The system is down-loaded and executed on-board the micro controller.

The systems goes through two preliminary steps *initialise robot* and *initialise population*. It then goes through a loop consisting of *selection*, *crossover*, and *mutation* (see figure 4.3).

4.3.1 Individuals

The population consists of a fixed number of individuals, typically 50 or less. Since the system should be as dynamic as possible, the individuals are represented as a pointer structure with no maximum length (see figure 4.4). This could potential lead to a population with very long individuals, and the size of these individuals could surpasses the memory available on the robot. But since a minimum- and maximum length is given when initialising the population, this hasn't been a problem.

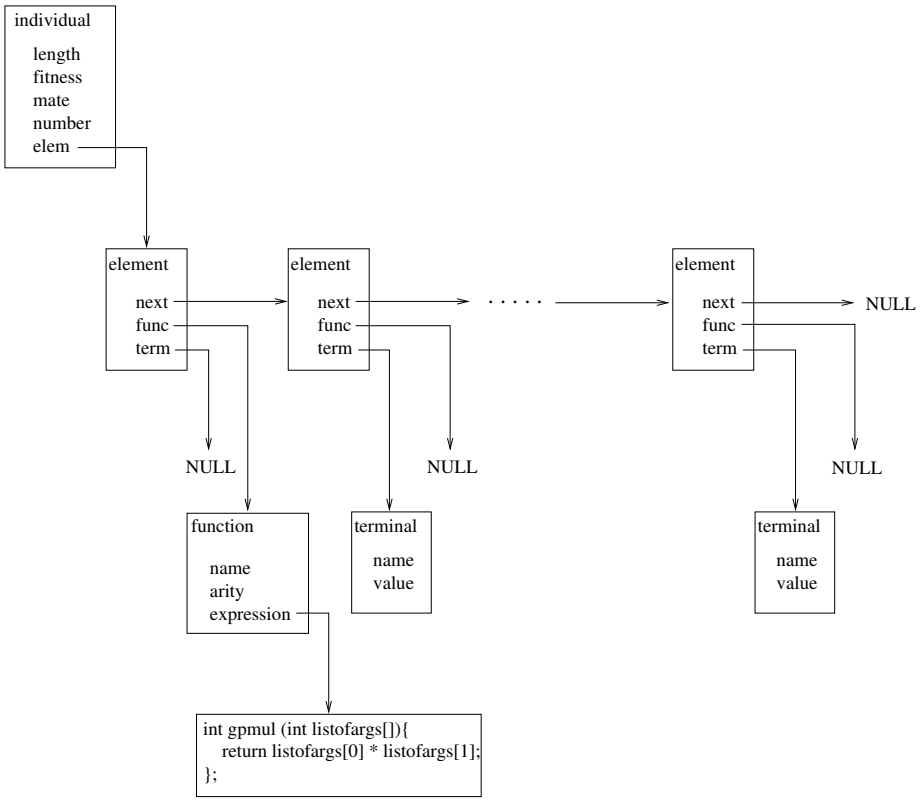

Figure 4.4: Overview of the individuals structure

The individuals are represented in a *linear* fashion. This should not be mistaken for a *linear GP*. The GP used in this work is actual a tree-based GP, it is just represented in a linear structure, and all manipulation is done on the linear structure. This is, with some modifications, based on the work by Keith and Martin [KM94]. The use of a tree-based GP in a linear form is different from the work by Banzhaf et. al. where a *true* linear GP was used (see section 3.3.3).

All individuals consists of two parts. *i*) Information about the individual. *ii*) The C program. Any individual has the following elements:

- **length** A integer containing the length of the individual.

  fitness The integer value of the individuals fitness.

  mate Boolean value to keep track of whether or not this individual has been selected as a parent of child.

  number The number of the individual, used for debugging purpose.

  elem A pointer to the first node in the individual's program.

The *elem* is the first node in the pointer array containing the C program. All nodes has the following in common:

- **next** The pointer to the next node in the array. Last one point to *NULL*.

  func A pointer to a function. Contains *NULL* if the node is a terminal.

  term A pointer to a terminal. Contains *NULL* if the node is a function.

Terminals and function has one thing in common; they both have names. These names are only for readability. A function contains the following variables:

- **name** The name of the function, e.g. *, -, +, /.

  arity The number of arguments handled by the function.

  expression A pointer to a function, of the form:

```
int <name> (int listofargs[]){
  return <calculation>;
  };
```

The terminals are short structures, they only contain the name of the terminal, e.g. *sensor0*, *constant1*. They also contains the value of the particular terminal.

The initialisation is done with two parameters *minimum-length* and *maximum-length*. To insure the integrity of the random generated programs the arity of the functions selected minus one was added to a checksum, and in case of terminals one was subtracted from the checksum.

When the expression is build the checksum must be minus one to guaranty a legal expression. This can be done purely random, but to speed up the process a trick from [KM94] was used:

```
if((openbranch + currentlength) >= indiv->length){
 force terminal selection
}
```

Where `openbranch` is the sum of the selected functions arity, minus one for each terminal selected. So when the expression is filed with functions and the rest of the available node has to be used for terminals to ensure the integrity; terminals are selected.

4.3.2 Reproduction

To have a reproduction system in evolutionary computation selection is necessary. And either crossover or mutation must be used, more often than not, both crossover and mutation is used, as is the case in this work.

Selection

The selection used in this system is based on the probabilistic sampling of the environment introduced by Banzhaf and Nordin in [NB95]. Here each individual is tested on a new fitness case, i.e. a new place in the environment. A discussion of this method can be found in ([NB95], [NB97b]) and others.

The selection itself is straight forward. It is *tournament* selection with the size of four within a *steady-state* GP. Two parents are selected based on their fitness returned by the fitness function, tested on-line in the real environment. For an example, if *obstacle avoidance* is sought the function (4.1) could be used.

$$f = \alpha \times ((m_1 + m_2) - |m_1 - m_2|) - \beta \times \sum_{0}^{7} s_i \tag{4.1}$$

Where α and β are constants used for scaling, m_1 and m_2 are the values of motor one and two, and s_i are the sensors. The two individuals who "lost" the tournament selection would be deleted and used for the children returned from the crossover function.

Crossover

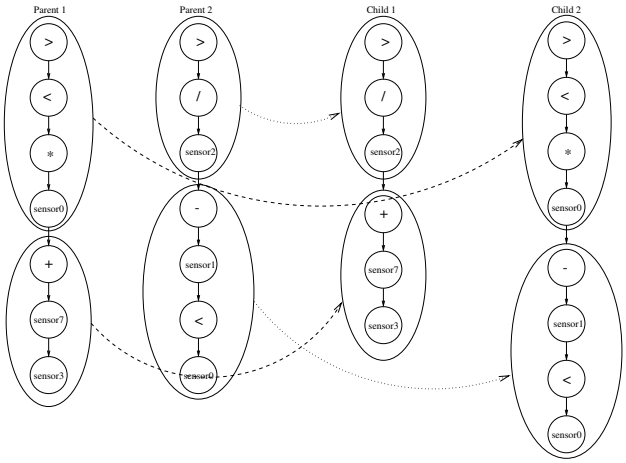

Figure 4.5: Example of crossover

To insure the integrity of any program constructed by crossover, the function has to be constructed in a particular fashion. This is very much like the way individuals are initialised, except the parts are bigger.

To generate a legal child the following can be done, here the two parents are *mother* and *father*, the children are *child 1* and *child 2*, and *checksum* is 0 (null) (see figure: 4.5 for an example).

- Select a random crossover point in each of the parents.
- Copy the parts before the crossover point from each of the parents to each of the children. e.g. copy the first part of *mother* to *child 1*.

- Start from the crossover point and copy each node to the child until *checksum* is -1; doing just the same as in initialising. e.g. copy the crossover segment from the *father* to *child 1*.
- Copy the rest of the parents to the children. e.g. copy the rest of *mother* to *child 1*.

It is worth noticing that if an individual is dividend in to three parts: *pre-part*, *crossover-part*, and *post-part*. The *pre-part* and *post-part* can have the length of zero, but the *crossover-part* must have a length of at least one.

Mutation

The mutation in this work is implemented as a simple version of sub-tree swapping. No tree are swapped but just nodes. To ensure the correctness of the mutation operator, functions are swapped with function of the same arity, and terminals are only swapped for other terminals.

Part III

Results and Evaluation

Chapter 5

Results

Results! Why, man, I have gotten a lot of results. I know several thousand thing s that won't work

– Thomas A. Edison

5.1 Introduction

The experiments in this work are based on the experiments done by Banzhaf et. al. [BNO97] First of all, this work will try to reproduce the results archived by Banzhaf et. al. This will be done by running one of their experiments, the *obstacle avoidance problem,* only with my own implementation.

After the reproduction of this results, this work will try to improve the implementation used. The goal here is to evolve an individual who can solve the problem, instead of the approach by Banzhaf et. al. where the population in cooperation solves the problem.

Before doing the experiments first conducted by Banzhaf et. al. Two test cases will be executed. The first test will be run to see if evolution is working, and to check several parameters. It will be run with the robot on it's back, i.e. with it's wheels spinning in the air; insuring a pure test environment.

The second test will be conducted with the robot running correctly. This will be done for demonstration the presentation form for the statistical material.

The rest of the experiment part will cover obstacle avoidance, first in the Banzhaf et. al. way, and then in the way specific for this work.

All experiments will be conducted in an autonomous way. The robot will be connected to a workstation, but only for receiving power, and dumping statistical information. All computation is done on the robot.

5.2 General settings

Some issues are general for all the experiments conducted here. All the general settings will be describe here.

For each experiment a table will describe the specific settings for that particular trial.

5.2.1 Function set

The following functions are a member of the set which can be used by the GP (see table 5.1). The function set is divided into three groups: The six first functions are standard operations (Add, SUB, DIV, MUL, AND, and XOR); the next two are bit shifting functions (SLL and SLR); and the two last ones are motor specific functions.

Name	Symbol	arity	Description
ADD	$+$	2	Addition
SUB	$-$	2	Subtraction
DIV	$/$	2	Protected division
MUL	$*$	2	Multiplication
AND	$\|$	2	Logical and
XOR	&	2	Exclusive or
SLL	L	1	Shifts bits left
SLR	R	1	Shifts bits right
SMO	$<$	1	Set motor one
SMT	$>$	1	Set motor two

Table 5.1: Function set

The two bit shifting functions (SLL and SLR) shifts bits either left or right. They are defined in the following way:

```
int gpsll (int listofargs[]){
  int tmp;
  tmp =  listofargs[0] & (0x0fff);
  return tmp<<4;
};

int gpslr (int listofargs[]){
  int tmp;
  tmp =  listofargs[0] & (0xfff0);
  return tmp>>4;
};
```

The way output are written to the motors are done differently here than in the work by Banzhaf et. al.

In this work writing output to the motors are an integrated part of the function set. The two functions *gpmotorleft* and *gpmotorright* can be called from any point within the GP structure.

This design choice has been done for the reason of making the GP even less suspect to meddling by the programmer; the controllers should not be forced to give output to the motors. Again Stone has been the inspiration [Sto94].

The two functions are defined in the following manner:

```
int gpmotorleft (int listofargs[]){
  mot_new_speed_1m(0,listofargs[0]);
  return listofargs[0];
};

int gpmotorright (int listofargs[]){
  mot_new_speed_1m(1,listofargs[0]);
  return listofargs[0];
};
```

The two functions follow the standard definition of functions within this GP (see section 4.3.1). They return the list of argument for further calculation.

The `mot_new_speed_1m(int motor, int value)` is a function defined in the C library, which accompanied the Khepera robot. The function takes the motor-number as the first argument, and the speed as the second argument.

5.2.2 Terminal set

The variables in this work are only the eight motors. There are an equal number of constants (see table: 5.2).

Name	Initial value	Description
Sensor0 - Sensor7	0	The infrared sensors on the Khepera measuring distance
Constant 0	-5	Predefined constant
Constant 1	-3	Predefined constant
Constant 2	-1	Predefined constant
Constant 3	0	Predefined constant
Constant 4	1	Predefined constant
Constant 5	2	Predefined constant
Constant 6	3	Predefined constant
Constant 7	6	Predefined constant

Table 5.2: Variable and constant set

5.2.3 Selection, crossover, and mutation

The selection are done by by tournament selection with two times two individuals, with no possibility of selecting the same individual twice in the same tournament.

Crossover is one point random for each parent chosen. For more information on the crossover mechanism see section: 4.3.2.

The mutation rate is per gene; only one mutation can occur in any give individual.

5.3 Test run

This experiment is run with the robot turned upside down for two reasons: *i*) to check if any evolution is going on, and *ii*) to find the good α and β parameters in function 5.1.

$$f = \alpha \times ((m_1 + m_2) - |m_1 - m_2|) - \beta \sum_{i=0}^{7} s_i \tag{5.1}$$

The fitness function for obstacle avoidance, where m_1 and m_2 are the two motors, and s_0 to s_7 are the eight infrared sensors. The function will be more thoroughly described in section 5.4.

5.3.1 Upside down

When the robot is turn upside down, the second half of the fitness function will have no effect on the fitness. The goal for this run will then be to get both motors to run forward as fast as possible. The parameters were as follow:

Parameter	**Value**
Population size	50
Crossover probability	100%
Mutation probability	10%
Maximum number of generations	200
Minimum initial individual length	10
Maximum initial individual length	30
Number of iterations pr. individual	1
Function set	ADD, SUB, DIV, MUL, AND, XOR, SLL, SLR
Terminal set	Integers in the range *int*

The run took 28 minutes for a total of 200 generations.

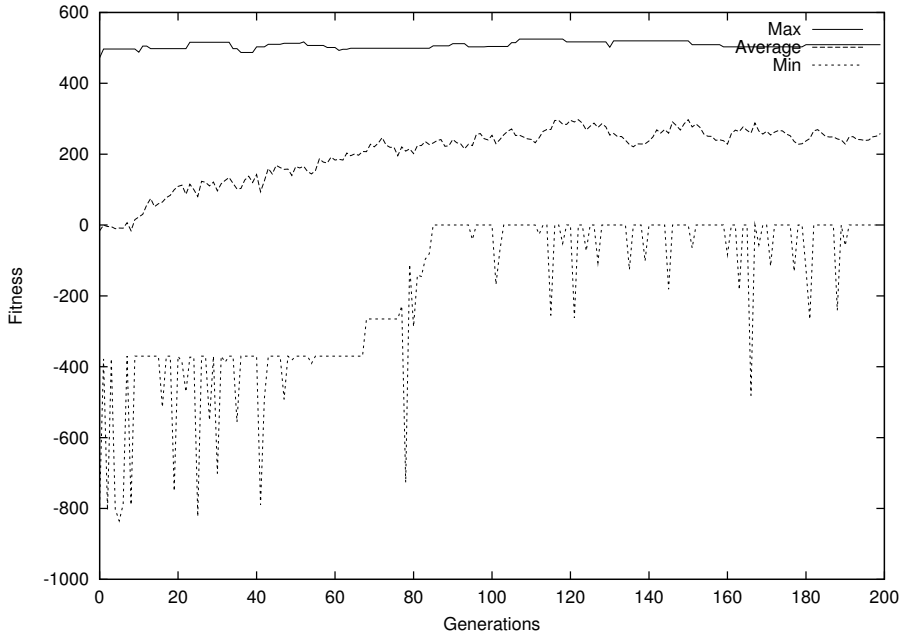

Figure 5.1: Fitness graph for first test run

The fitness graph clearly shows evolution occurring. A few issues needs to be covered though. It should come as no surprise that the problem is very easy. According to the fitness graph, the program seems to hit a very good solution already in the random population in generation one. But the average fitness improves nicely over time, giving the indication that evolution is occurring.

5.3.2 Correct orientation

The second part of this test run is done with the robot turned the right way. This has been done to demonstrate the way information will be presented in the following real experiments.

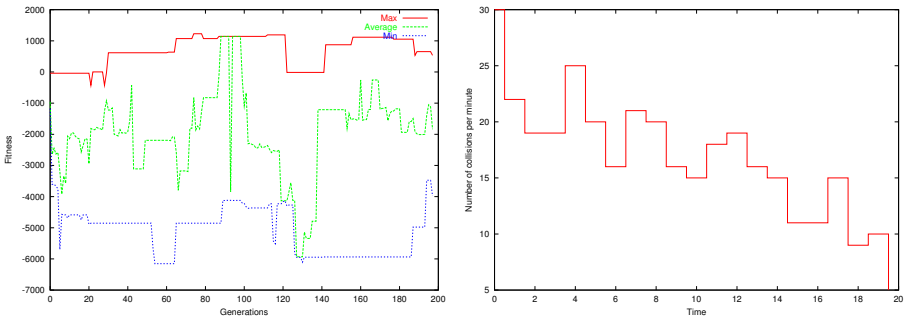

Figure 5.2: Fitness and collisions graphs for second test run

This run is based on the *obstacle avoidance* problem, the fitness function from section 5.4 is used. It is evident that the fitness graph (the one to the left in figure: 5.2) is next to useless for showing any progress in the evolution. This is due to the many parameters which are measured. There are eight infrared sensors and two motors. The problem is that any number of combinations of these ten values could give the same plotted fitness value, hence another way of visualising the progress must be used.

The same graph as used by Banzhaf et. al. can be used, namely the *number of collisions pr. minute* (the one to the right in figure 5.2). This graph clearly shows that the robot is getting better and better to avoid collision over time. Hence, this is the type of graph which will be used through out the rest of this chapter.

After extensive testing the behaviour of the robot was at its best when α was 16 and β was 2.

5.4 Obstacle avoidance

The first experiment is evolving *obstacle avoidance*. This experiment is based on the obstacle avoidance experiment Banzhaf et. al. [NB97a] and [BNO97].

Given that high value on the sensors is equal to closeness to an object, minimising the sum of sensory input is the main goal of the fitness function (5.2). To avoid a robot standing still far away from any objects, the first half of the fitness function gives credit to straight and fast moving robots.

$$f = \alpha \times ((m_1 + m_2) - |m_1 - m_2|) - \beta \sum_{i=0}^{7} s_i \tag{5.2}$$

The arena where this system was tested is a 60×80 centimetre regular shaped world with no internal obstacles. The Robot was run with the cable connected to the workstation, purely for power and data receiving purposes.

A connection to the workstation introduces the problem of tangled cables. When the robot has been turning around for some time, it needs to be lifted and to have the cable straightened. This, however, is not a problem for the result, since it seldom takes more time than it takes to evaluate one individual, hence it should not interfere with the general result.

5.4.1 Test case 1 - The population way

The approach by Banzhaf et. al. is to test a large number of individuals for a very short time, thereby building implicit cooperation between the individuals solving the problem [NB97b]. This will result in a population were no single individual can solve *obstacle avoidance*, but the population as a whole can (see section 3.3.3 for a more thorough description).

The following specific parameters were used:

Parameter	**Value**
Population size	50
Crossover probability	100%
Mutation probability	10%
Maximum number of generations	100
Minimum initial individual length	10
Maximum initial individual length	30
Number of iterations pr. individual	1
Sleep time between execution and testing	400 ms
Function set	ADD, SUB, DIV, MUL, AND, XOR, SLL, SLR, SMO, SMT
Terminal set	Integers in the range *int*

This experiment follows the format of the experiment by Banzhaf et. al. The major difference between those experiments and the one done here is that Banzhaf et. al. uses a register variable to hold the value of the two motors, and only when the program is finished are the two variables written to the motors. This work includes the motor function as an integrated part of the function set, see section 5.2.1 for an explanation.

Each run took around half an hour to conduct. In this experiment three runs were made.

5.4.2 Result

Figure 5.3 to 5.5 is the *number of collisions per minute* for each run. As can clearly be seen there are no perfect solutions for this problem. There is, however, a nice trend where the number of collisions definitely are decreasing over time. This trend is also visible in the graph showing the average for all three runs (Figure 5.6).

One thing which is worth noticing is the drop in number of collisions at around five minutes. This is a recurrent problem in all three runs, and that makes it very visible in the average graph. This drop is due to the problem of the cable. At around that time it needed to be straightened. The same problem is visible at around twenty minutes.

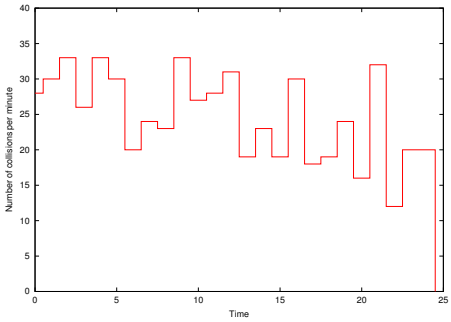

Figure 5.3: Population first run

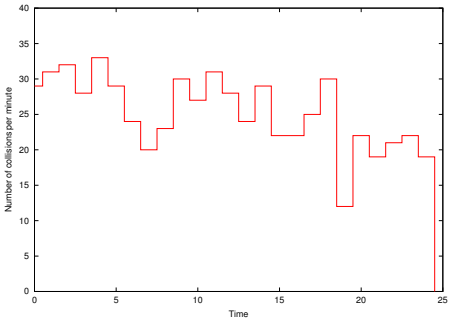

Figure 5.4: Population second run

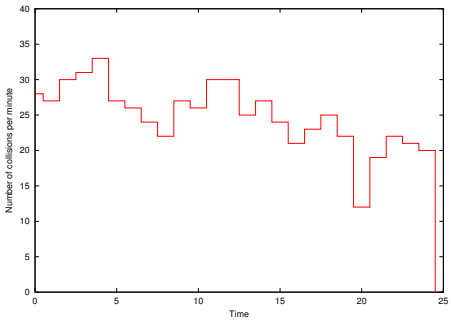

Figure 5.5: Population third run

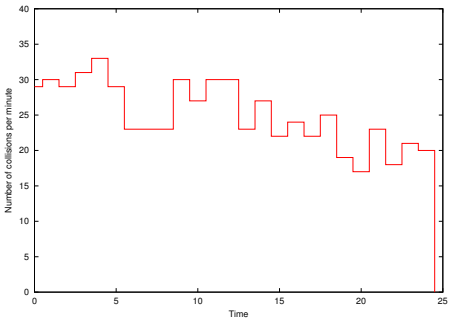

Figure 5.6: Population average

It should be noted that due to some limitations in my plotting software, the graph starts at 0.5 and ends 0.5 before it should. This means that the first small part of the graph should be ignored, and the very large fall in the end is not part of the data.

Here is it of a smaller scale. This diminishing in scale is due to a more varied behaviour from five minutes to twenty.

However, there is a clear improvement in the behaviour of the population. This improvement is also visible when conducting the experiments. When the population starts there is a lot of erratic behaviour in the population. A lot of spinning around, running in to walls and keep on running.

As time goes by the robot generally slows down a bit, and the spinning behaviour diminishes. At some point the robot tries to run in a straight line, until it crashes. And in the last five minutes there is clear evidence of a turning behaviour; that is the robot tries to run in a large circle (see figure 5.7).

5.4.3 Test case 2 - The individual way

This experiment is an extension to the original work done by Banzhaf et. al. The goal here is not to solve the obstacle avoidance problem by training the whole population in implicit cooperation, but rather to train the population in such a way that one individual can solve the problem alone.

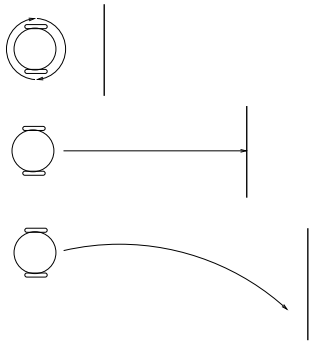

Figure 5.7: Three stages of behaviour

This twist is accomplished by running each individual through 10 iterations instead of just one. Hence, each individual will be given 10 shots at solving the problem, at (potential) 10 different locations.

Further more, the time which an individual "sleeps", that is the time frame from execution of the individual until measurement are done is only 40 milliseconds instead of 400 as in the first experiment. This has been done to minimise the time spent on running the experiments.

The following specific parameters were used:

Parameter	**Value**
Population size	50
Crossover probability	100%
Mutation probability	10%
Maximum number of generations	100
Minimum initial individual length	10
Maximum initial individual length	30
Number of iterations pr. individual	10
Sleep time between execution and testing	40 ms
Function set	ADD, SUB, DIV, MUL, AND, XOR, SLL, SLR, SMO, SMT
Terminal set	Integers in the range *int*

5.4.4 Result

Figure 5.8 to 5.10 is the *number of collisions per minute* for each run. As in the first experiment, there are no perfect solutions for this problem. But again there is a very good trend.

Except for the first run, the rest show a very clear improvement. The population starts out at around the same number of collisions as the first experiment (around 20), but has a far better rate of improvement. The final number of collisions is around seven.

The behaviour of the population mimics the first experiment. The population starts off with the same erratic behaviour, which primarily consists of spinning around the robot's own axis.

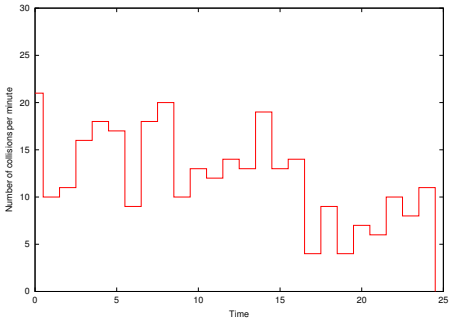

Figure 5.8: Individual first run

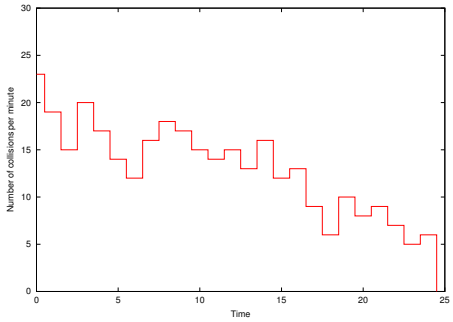

Figure 5.9: Individual second run

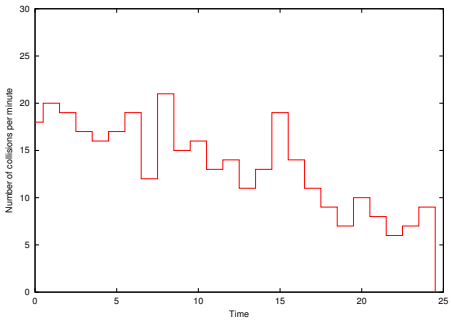

Figure 5.10: Individual third run

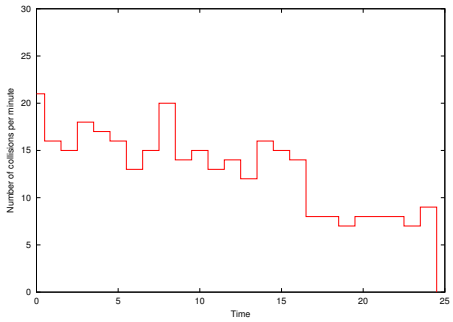

Figure 5.11: Individual average

It should be noted that due to some limitations in my plotting software, the graph starts at 0.5 and ends 0.5 before it should. This means that the first small part of the graph should be ignored, and the very large fall in the end is not part of the data.

As time goes by the same slow down, as in the first experiment, is apparent. The robot still seems to go through the same stages, where first it will try to run in a straight line, disregarding any obstacles, but at some point within the last ten minutes the turning behaviour occurs.

It seems that evolutions is occurring faster here than in the first experiment. On the other hand, spinning behaviour occurs at a far later point in time.

This is probably due to more thorough testing of each individual, ten times instead of one, where it should be harder for less flexible individuals to survive.

This can also explain why spinning behaviour is occurring at a later time. Since individuals are more thoroughly tested, fewer individuals will be tested within the same time frame. And since the random generated population seems to have a preference for spinning behaviour, it should occur at a later time.

5.5 Comparison

5.5.1 Introduction

Before any comparison between the two different experiments, and between this work and the work by Banzhaf et. al, can be done, a description of the way collisions are counted would be in place.

As a rule of thumb each time that the robot hits the wall one collision is counted. Furthermore, since the LEDs on the robot blinks each time a new individual is tested, it is possible to count each time a new individual "hits" the wall.

This means that if the first individual hits the wall, one collision is counted. If the next individual do not go away from the wall one more collision is counted, and so forth. This might not be the way it was done in the original work by Banzhaf et. al. It has not been possible to verify how counting was done by Banzhaf et. al., so it might not be possible to directly compare the numbers in the different experiments, but the trends and conclusions should be comparable.

5.5.2 Comparison between the two experiments

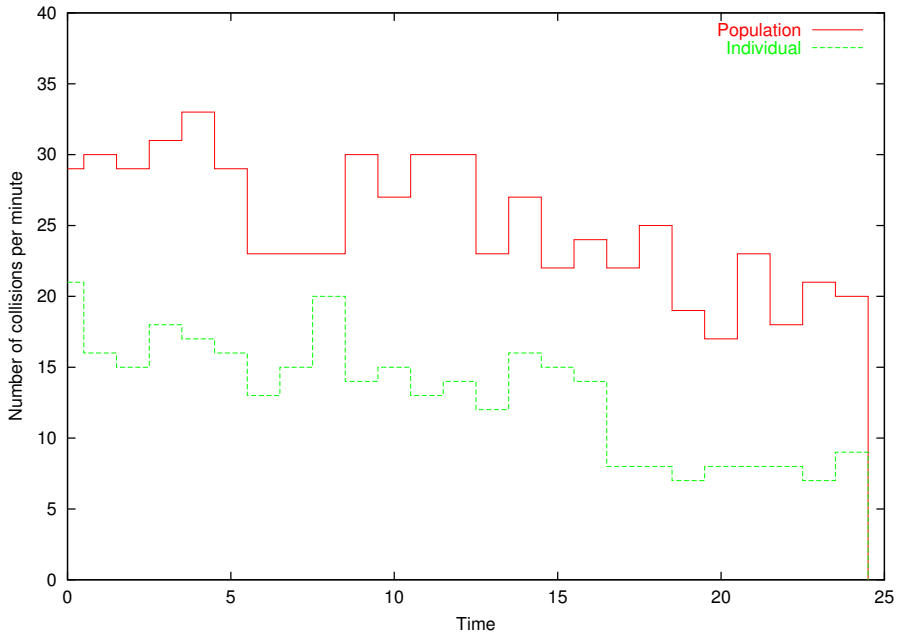

Figure 5.12: Average number of collisions for both experiments

As already mentioned in section 5.4.4 it looks like the individual way is better than the population way. Looking at the comparison graph, it suggests that the superiority of the individual way could be written off as pure coincidence, since it starts of better than the population way.

I would argue that it is not solely due to coincidence. The difference could be ascribed to the difference in the time each individual is allowed for testing. In the first

experiment 400 milliseconds are used for each individual. In the second only 40 milliseconds. This would allow for the robot to go much further in the first experiment; hence, getting the population closer to the wall. And since no meaningful behaviour occurs within the first 10 minutes, the spinning behaviour would punish the first experiment more than the second one.

The possible superiority of the individual way as opposed to the population way is not the most significant result from this work. The result can be divided into three points: *i*) Verification of the work done by Banzhaf et. al. *ii*) Evolution of a solution based on individuals instead of cooperation. *iii*) Comparable performance between this work and work by others.

Verification

For scientific work to be acceptable it must be possible for a 3rd party to verify or falsify any experiment[†]. In general this work shows that the use of Genetic Programming is both possible and feasible when working with Adaptive Behaviour Based Robotics. Specifically it shows that the approach introduced by Banzhaf et. al. is a useful and functional way to utilise the exiting field of Genetic Programming in conjunction with autonomous robots; something which is at the cutting edge of this research area.

Individual vs. population solution

The approach of Banzhaf et. al. is to let the *population* solve the problem of obstacle avoidance. This is done by the implicit cooperation between individuals, i.e. each individual specialises in some area such as getting out of corners or going in a straight line [NB97a]. The problem is then solved by individuals appearing at (hopefully) the right time in the experiment.

As stated in section 5.1 the solution chosen in this work is somewhat different from that approach. It is more of a standard way of using evolutionary methods. Each individual is evolved to solve the whole problem, and not just parts of it. This is accomplished through a more thorough testing of each individual. As described in section 5.4.3, each individual is tested ten times in a row, at potential ten different location. This should lead to a more flexible individual than the ones tested only ones.

This should allow for evolution of (at least) *one* individual who can solve the obstacle avoidance problem, hence at the end of an experiment picking the best individual should give us the solution.

Together with the choice of ANSI C as the language of the individuals genetic material, instead of assembler as in the work by Banzhaf et. al. I would argue that an individual who solves this problem is much more interesting from the perspective of understanding the process, than a population of individuals represented in assembler.

Performance

The choice of ANSI C and not assembler was, as mentioned above, done to help understand how the robot solves the problem (see section 4.3). It was suggested that the speed of the system would probably decrease compared to a system written in assembler.

This does not seem to be the case. When looking at figure 5.12, the individual approach easily compares to the populations approach. However, when comparing

[†]Even though Popper would probably not agree on the first part

with the original work by Banzhaf et. al. [NB97a], where they find an almost perfect solution. This approach still lacks some performance.

A real verification of the performance between this work and the original has unfortunately not been possible. The original work used approximately one hour to evolve an almost perfect solution, something which is almost three times as long as the run time that was possible in the experiments conducted in this work.

5.5.3 Comparison between this work and others

The implementation and testing of this work, has so far shown that the original idea by Banzhaf et. al., that fully autonomous use of GP is possible and feasible in behaviour based robotics.

This work also shows that to obtain reasonable results using a evolutionary method, other methods than genetic algorithms evolving artificial neural networks can be used. Looking at the on-board work presented in section 3.2.3, where the work by Nolfi et. al. [NFMM94] and Floreano et. al. [FM96a] was described, it is obvious that this solution is comparable to that work.

This solution does not perform as well as the work by Floreano et. al., but it is definitely not far off. It is worth noticing that these two experiments are not directly comparable, not just because of the two different methods, but also because of the non-autonomous way the Floreano experiment was done. It should also be obvious that running a GA/ANN solution, consumes far less time and resources than a full GP solution. Given these differences, I would argue that these solutions perform in comparable ways.

When comparing to work using the same method, the work by Banzhaf et. al. has already been discussed. Since that work is the only known work on purely autonomous on-board GP, it is only possible to compare this work with the work done on hybrid solutions, see section 3.3.4.

Comparing to the work done by Marc Ebner [Ebn98], where a part simulation and part on-board system was constructed, this work shows that speed is not a big a problem as it might look. The original work done by Ebner took 197 hours to evolve a controller for navigating a large physical robot. Even though his work included an ADF like structure, the difference in speed is very noticeable. Where Ebner's system used 197 hours, this work used around half an hour to evolve a reasonable solution to obstacle avoidance.

All in all it is reasonably to say that this system in certain ways, is comparable to systems utilising GAs and ANNs, and that is performs better than other work using GP.

Chapter 6

Discussion

Get your facts first, and then you can distort them as much as you please

– Mark Twain

6.1 Introduction

This chapter will evaluate the project as a whole. It will first present a summary of the thesis and the results gained from this work. It will then move on to discuss my views on the research area, and the experiences received from working with real world robots. Finally it will describe the future work which may yield worthwhile results.

6.2 Summary

In chapter 1 the main goals and motivations for this thesis was described. The main goal was to show the feasibility of using Genetic Programming for evolving controllers in a real world robot, and to it in a totally autonomous way.

The underlying reason for this is my strong belief that it is only through experiences in the real physical world that a concrete understanding of the behaviour of organisms (or robots) can come.

The choice of Genetic Programming springs forth from the usefulness it has shown in a wide variety of other areas, and the possibilities of evolving robot controllers that are not just useful, but also understandable for a person with the ability to read and understand a program written in a general purpose language; something which Artificial Neural Networks, and to a large degree Genetic Algorithms can not do.

To design and implement the necessary experiments and software for demonstrating the usefulness of Genetic programming, a thorough understanding of, not only Genetic Programming but also other Evolutionary Methods must be gained. But before such an understanding can be build, an understanding of the theories and research from the fields which inspired the use of artificial evolution must be gained.

Section 2.2 in chapter 2 describes the real world foundation for Behaviour Based Robotics, and some different approaches to adaptation in individuals and populations

were presented. This part builds the understanding that artificial evolution it not just some idea coming out of no where; it has been getting ripe since Malthus in 1798. Together with section 2.3 the foundation for Artificial Evolution of Behaviour using Genetic Programming has been constructed.

Before the specifics of the approach in this thesis can be decided upon, some research must be done to how others have tackled this area. Chapter 3 describes a major part of the relevant research done by others. The material discussed here is obviously only a subset of all the research done in this area, but an important part of the research is to decide what to include and what to exclude.

Since the work of Banzhaf et. al. is the first real extensive work on Genetic Programming in physical robots, their work would naturally take a more prominent role in this thesis than a lot of other interesting research.

In chapter 4 the specific design and implementation chosen for this thesis was described, and the motivation for the choices made.

Chapter 5 discussed the results obtain through the experiments conducted in the thesis. It also supported the goals and motivation from chapter 1, by showing that not only was the use of Genetic Programming useful and feasible, but the results were also level with the results by Banzhaf et. al., and comparable to other methodologies in Behaviour Based Robotics.

6.3 The research area

As describe in chapter 2 there are several ways to approach the problems of Behaviour Based Robotics. The most common way is to use a Genetic Algorithms and/or an Artificial Neural Network. The choice of Genetic Programming has had it up-sides and down-sides.

Looking at the positive side first, the use of Genetic Programming and real robots has allowed this work to explore areas where, when comparing with areas such as Artificial Neural Networks and Genetic Algorithms, very little work has been done. This has given me the freedom of, in the words of Star Trek: “to boldly go where [almost] no one has gone before”.

The novelty and somewhat cutting edge of this particular combination of Genetic Programming and Robotics, has sometimes been a blessing in disguise. Since very little work has been done, it has been hard to find any literature to support this work.

On a more general level the whole field of Evolutionary Methods appeals very much to me. The bottom-up approach and “non-determinism” easily leads to the: “That’s funny....”.

6.4 Robots

It is always a hassle to work in the physical world, simulations and assumptions are so much more pleasant to work with.

Even though I do not regret ever starting with robots, I must say that there are so many more problems when working with robots, than with nice manageable models. There are a lot of extra challenges when working with robots: mechanical problems, power issues, time consumption, etc.

These problems which are a common challenge when using any robot, has really been much more common with the Khepera. This robot demonstrates all the problems

with computers and new technology. First of all, is the documentation poor at best. Secondly, given the price on hardware today you do not get much for your money (256KB of memory, around 45 minutes of battery power). And finally, it do not look like a reliable platform.

It might sound like I dislike robots in general and the Khepera in particular. But even though there has been a lot of nerve wrecking times, it has actually been fun working with real physical things. It gives a new dimension to the non-deterministic way I enjoy so much.

One learns a lot when working with robots, it can for an example be very frustrating to see that when you instruct a robot to turn 90 degree to the left, it will only turn 80 or perhaps 100 – but not the same amount each time. These kind of experiences is actually the best way to learn that *ecological psychology*, and Brooks and the like may be right.

6.5 Future work

Even though the work presented in this thesis reaches the goals stated in chapter 1, several areas have shown itself to be worthy of closer inspection and more work.

It is obvious that the results of this work is not a perfect solution to navigation in the real world. It might be useful to reexamine the code written for this thesis and modify or change different parts, thus improving the overall performance of the system.

One area which would be very interesting to work on, it to see how robust the solutions evolved in this way are, i.e. will a controller work in another world?

It would also be worthwhile to investigate is the approach would be successful at solving more complex problems, and work with several more simple problems, such as wall following. If this is successful it would be worthwhile to develop arbitrators, such as in the work by Harvey [Har92], and use Automatically Defined Functions (ADF) for solving more complex problems.

Bibliography

[Ark98] Ronald C. Arkin. *Behavior Based Robotics*. MIT Press, 1998.

[Bal96] J. M. Baldwin. A new factor in evolution. *American Naturalist.*, 1896.

[BG92] Randall D. Beer and John C. Gallagher. Evolving dynamical neural networks for adaptive behavior. *Adaptive Behavior*, 1(1), 1992.

[BNKF98] Wolfgang Banzhaf, Peter Nordin, Robert E. Keller, and Frank D. Francone. *Genetic Programming, An Introduction: On the Automatic Evolution of Computer Programs and Its Applications*. Morgan Kaufmann Publishers, Inc., 1998.

[BNO97] Wolfgang Banzhaf, Peter Nordin, and Markus Olmer. Generating adaptive behavior for a real robot using function regression within genetic programming. In *2nd International Conference on Genetic Programming*, 1997.

[Bra84] Valentino Braitenberg. *Vehicles: Experiments in Synthetic Psychology*. MIT Press, 1984.

[Bro87] Rodney A. Brooks. Planning is just a way of avoiding figuring out what to do next. Technical report, MIT, 1987.

[Bro89] Rodney Brooks. A robot that ealks: Emergent behaviors from a carefully evoled network. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 692 – 694, 1989.

[Bro91] Rodney Brooks. New approaches to robotics. *Science*, 253:1227 – 1232, 1991.

[CD93] Marco Colombetti and Marco Dorigo. Learning to control an autonomous robot by dristributed genetic algorithms. In *From Animals to Animats 2: Proceedings of the Second International Conference on Simulation of Adaptive Behavior*, pages 305 – 312, 1993.

[Dar28] Charles R. Darwin. *The Origin of Species*. J. M. Dent & Sons Ltd., 1859, 1928.

[Ebn98] Marc Ebner. Evolution of a control architecture for a mobile robot. In *Proceedings of the Second International Conference on Evolvable Systems: From Biology to Hardware (ICES 98)*, 1998.

[FM90] J. F. Fontanari and R. Meir. The effect of learning on the evolution of asexual populations. *Complex Systems 4*, 1990.

[FM94] Dario Floreano and Francesco Mondada. Automatic creation of an autonomous agent: Genetic evolution of a neural-network driven robot. In *From Animals to Animats 3: Proceedings of the Third International Conference on Simulation of Adaptive Behavior*, pages 421 – 430, 1994.

[FM96a] Dario Floeano and Francesco Mondada. Evolution of homing navigation in a real mobil robot. In *IEEE Transactions on Systems, Man, and Cybernetics*, 1996.

[FM96b] Dario Floreano and Francesco Mondada. Evolution of plastic neurocontrollers for situated agents. In *From Animals to Animats 4: Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior*, pages 402 – 410, 1996.

[FP96] Forrest and Perelson. The baldwin effect in the immune system: Learning by somatic hypermutation. *Adaptive Individuals in Evolving Populations, SFI Studies in the Sciences of Complexity*, 1996.

[GG96] Takashi Gomi and Ann Griffith. Evolutionary robotics - an overview. In *Proceedings of IEEE International Conference on Evolutionary Computation*, 1996.

[Gib79] J. J. Gibson. *The Ecological Approach to Visual Perception*. Houghton Mifflin, 1979.

[GIM98] Michael S. Gazzaniga, Richard B. Ivry, and George R. Mangun. *Cognitive Neuroscience: The Biology if the Mind*. W. W. Norton & Company Inc., 1998.

[GS] J. C. Grefenstette and A. Schults. An evolutionary approache to learning in robots. In *Proceedings, Machine Learning Workshop on Robot Learning*. New Brunswick, NJ.

[Han94] Simon C. Handley. The automatic generations of plans for a mobile robot via genetic programming with automatically defined functions. In Kenneth E. Kinnear Jr., editor, *Advances in genetic programming*, chapter 18. Massachusetts Institute og Technology, 1994.

[Har92] Inman Harvey. Species adaptation genetic algorithms: The basis for a continuing saga. In *Toward a Practice of Autonomous Systems, Proceedings of the First European Conference on Artificial Life*, 1992.

[Har97] Inman Harvey. Artificial evolution and real robots. *Artificial Life and Robotics*, 1:35 – 38, 1997.

[HHC93] Inman Harvey, Phillio Husbands, and Dave Cliff. Issues in evolutionary robotics. In *From Animals to Animats 2: Proceedings of the Second International Conference on Simulation of Adaptive Behavior*, pages 364 – 373, 1993.

[HHC+96] P. Husbands, I. Harvey, D. Cliff, A. Thompson, and N. Jakobi. Artificial evolution of control systems. In *Proc. of the 2nd Intl. Conf. on Adaptive Computing in Engineering Design and Control*, pages 41 – 49, 1996.

[HN87] Geoffrey E. Hinton and Steven J. Nowlan. How learning can guide evolution. *Complex System 1*, 1987.

[Hol92] John H. Holland. *Adaptation in Natural and Artificial Systems*. MIT press, 1975, 1992.

[III97] Forrest H. Bennett III. A multi-skilled robot that recognizes and responds to different problem environments. In *Second Annual Genetic Programming Conference (GP-97)*, 1997.

[Jak97] Nick Jakobi. Evolutionary robotics and the radical envelope of noise hypothesis. *Joural Of Adaptive Behaviour*, 6, 1997.

[JHH95] Nick Jakobi, Phil Husbands, and Inman Harvey. Noise and the reality gap: The use of simulation in evolutionary robotics. In *Advances in Artificial Life: Proc. 3rd European Conference on Artificial Life*, 1995.

[Kae92] Leslie Pack Kaelbling. An adaptable mobile robot. In *Toward a Practice of Autonomous Systems, Proceedings of the First European Conference on Artificial Life*, 1992.

[KBKA97] John R. Koza, Forrest H. Bennett, Martin A. Keane, and David Andre. Automatic programming of a time-optimal robot controller and an analog electrical circuit to implement the robot controller by means of genetic programming. In *Computational Intelligence in Robotics and Automation, Proceedings., 1997 IEEE International Symposium on Computational Intelligence in Robotics and Automation*, pages 340 – 346, 1997.

[KM94] Mike J. Keith and Martin C. Martin. Genetic programming in c++: Implementation issues. In Jr. Kenneth E. Kinnear, editor, *Advances in genetic programming*, chapter 13. Massachusetts Institute og Technology, 1994.

[Koz92] John R. Koza. *Genetic programming: On the Programming of Computers by Natural Selection*. MIT press, 1992.

[Koz94] John R. Koza. Introduction to genetic programming. In Kenneth E. Kinnear Jr., editor, *Advances in genetic programming*, chapter 2. Massachusetts Institute og Technology, 1994.

[KR92] John R. Koza and James P. Rice. Automatic programming of robots using genetic programming. In *AAAI-92, Proceedings Tenth National Conference on Artificial Intelligence*, 1992.

[LF92] M. Anthony Lewis and Andrew H. Fagg. Genetic programming approach to the construction of a neural network for control of a walking robot. In *Proceedings of the 1992 IEEE International Conference on Robotics and Automation*, pages 2618 – 2623, 1992.

[LHL97] Wei-Po Lee, John Hallam, and Henrik Hautop Lund. Applying genetic programming to evolve behavior primitives and arbitrators for mobile robots. In *IEEE Int. Conf. on Evolutionary Computation*, 1997.

[Lin91] Long-Ji Lin. Programming robots using reinforcement learning and teaching. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, 1991.

[LM96] Henrik Hautop Lund and Orazio Miglino. From simulated to real robots. In *Proceedings of IEEE 3rd International Conference on Evolutionary Computation*, 1996.

[LM98] Henrik Hautop Lund and Orazio Miglino. Evolving and breeding robots. In *Proceedings of First European Workshop on Evolutionary Robotics*, 1998.

[LWH98] Henrik Hautop Lund, Barbara Webb, and John Hallam. Physical and temporal scaling considerations in a robot model of cricket calling song preference. *Artificial Life*, 4:95 – 107, 1998.

[Mal98] Thomas Malthus. *An Essay on the Principle of Population*. J. Johnson, In St. Paul's Church-yard, 1798. Can be found on-line at: http://www.trmalthus.com/essay.htm.

[MB90] Pattie Maes and Rodney A. Brooks. Learning to coordinate behaviors. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, 1990.

[MC91] Sridhar Mahdevan and Jonathan Connell. Automatic programming of behavior-based robots using reinforcement learning. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, 1991.

[MF00] Z. Michalewicz and D. B. Fogel. *How to Solve It: Modern Heuristics*. Springer Verlag, 2000.

[MFI93] Francesco Mondada, Edoardo Franzi, and Paolo Ienne. Mobile robot miniaturisation: A tool for investigation in control algorithms. In *Experimental Robotics III, Proceedings of the 3rd International Symposium on Experimental Robotics*, 1993.

[Mit97] Tom M. Mitchell. *Machine Learning*. McGraw Hill, 1997.

[ML99] Orazio Miglino and Henrik Hautop Lund. Do rats need euclidean cognitive maps of the enviromental shape? *Animal Behavior*, Submitted 1999.

[MLN95] Orazio Miglino, Henrik Hautop Lund, and Stefano Nolfi. Evolving mobile robots in simulated and real environments. *Artificial Life*, 2(4):419 – 434, 1995.

[MNT95] Orazio Miglino, Kourosh Nafasi, and Charles E. Taylor. Selection for wandering behavior in a small robot. *Artificial Life*, pages 101 – 116, 1995.

[MP47] W. McCulloch and W. Pitts. How we know universals: the perception of auditory and visual forms. *Bulletin of Nathematical Biophysics*, 9:127 – 147, 1947.

[MP69] M. Misky and S. Papert. *Perceptrons: An Introduction to Computational Geometry*. MIT Press, 1969.

[MWO+98] Kazuyuki Murase, Takaharu Wakida, Ryoichi Odagiri, Wei Yu, Hirotaka Akita, and Tatsuya Asai. Back-propagation learning of autonomous behaviour: A mobile robot khepera took a lesson from the future consequences. In *Evolvable Systems: From Biology to Hardware, Second International Conference, ICES 98*, 1998.

[NB95] Peter Nordin and Wolfgang Banzhaf. Genetic programming controlling a miniature robot. In *Working Notes of the AAAI-95 Fall Symposium Series, Symposium on Genetic Programming*, pages 61 – 67, 1995.

[NB97a] Peter Nordin and Wolfgang Banzhaf. An on-line method to evolve behavior and to control a miniature robot in real time with genetic programming. *Adaptive Behaviour*, 5(2):107 – 140, 1997.

[NB97b] Peter Nordin and Wolfgang Banzhaf. Real time control of a khepera robot using genetic programming. In *Cybernetics and Control*, volume 26, pages 533 – 561. 1997.

[NFMM94] Stefano Nolfi, Dario Floreano, Orazio Miglino, and Francesco Mondada. How to evolve autonomous robots: Different approaches in evolutionary robotics. In Rodney A. Brooks and Pattie Maes, editors, *Artificial Life IV: proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems*, pages 190 – 197, 1994.

[Nil84] N. Nilson. Shakey the robot. Technical report, Artificial Intelligence Centre, SRI International, Menlo Park, CA, 1984.

[Nol96] Stefano Nolfi. Evolving non-trivial behaviors on real robots: a garbage collection robot. *Journal Robotics and Autonomous System, Special issue on "Robot learning: The new wave"*, 1996.

[Nol98] Stefano Nolfi. Evolutionary robotics: Exploiting the full power of selforganization. *Connection Sciemce*, 10, 1998.

[ONB96] Markus Olmer, Peter Nordin, and Wolfgang Banzhaf. Evolving real-time behavioral modules for a robot with gp. In *Proc. 6th International Symposium on Robotics And Manufacturing (ISRAM-96)*, 1996.

[Rey94a] Craig W. Reynolds. Evolution of corridor following behavior in a noisy world. In *Simulation of Adaptive Behaviour (SAB-94)*, 1994.

[Rey94b] Craig W. Reynolds. Evolution of obstacle avoidance behavior: Using noise to promote robust solutions. In Kenneth E. Kinnear Jr., editor, *Advances in genetic programming*, chapter 10. Massachusetts Institute og Technology, 1994.

[Roj96] Raul Rojas. *Neural Networks: A Systematic Introduction*. Springer Verlag, 1996.

[Rou96] Jonathan Roughgarden. *Theory of Population Genetics and Evolutionary Ecology an Introduction*. Prentice Hall, 1979, 1996.

[Sla85] P. J. B. Slater. *An introduction to Ethology*. Cambridge University Press, 1985.

[Ste94] Luc Steels. Emergent functionality in robotic agents through on-line evolution. In Rodney A. Brooks and Pattie Maes, editors, *Artificial Life IV: proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems*, pages 8 – 14, 1994.

[Sto94] James V. Stone. Evolutionary robots: Our hands in their brains? In Rodney A. Brooks and Pattie Maes, editors, *Artificial Life IV: proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems*, pages 400 – 405, 1994.

[Tan91] Ming Tan. Cost-sensitive reinforcement learning for adaptive classification and control. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, 1991.

[YB94] Brian Yamauchi and Randall 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*, pages 382 – 391, 1994.

Part IV

Appendices

Appendix A

Code

A.1 Header file

```
#include <bios.h>
#include <math.h>

/*****************************************************************************/
/* Defines:                                                                  */
/*****************************************************************************/
#define NULL 0
#define POPULATIONSIZE 50
#define DATATYPE short int
#define NUMBEROFFUCTIONS 10
#define NUMBEROFTERMINALS 8
```

```
#define NUMBEROFCONSTANTS 8
#define MAXINDIVIDUALLENGTH 30
#define MININDIVIDUALLENGTH 10
#define MUTATIONRATE 100                          /* in per mille */
#define NUMBEROFITERATIONS 1 /* set it to 1 (one) for ordinary running */
#define ONLYNEWONES 0  /* 1: Only evaluate new ones in the select_individual, 0: Evaluate all */
#define KENNNY 0 /* 1: kills off bad ones before numberofiterations are finished */
#define NUMBEROFGENERATIONS 1000
#define PERIODOFSLEEP 40 /* sleep period in miliseconds */
#define ZEROFITNESSVALUE 32000 /* this is the test value for the
                                  statistics, make sure it is well
                                  above the maximum fitness */

/*****************************************************************************/
/* Statestics:                                                               */
/*****************************************************************************/
/*short int bestofgeneration[NUMBEROFGENERATIONS];*/
/*short int worstofgeneration[NUMBEROFGENERATIONS];*/
/*float averageofgeneration[NUMBEROFGENERATIONS];*/

/*****************************************************************************/
/* Typedefs:                                                                 */
/*****************************************************************************/
/* Stack structures                                                          */
/* defines the stacknode, which contains the data of the predefined datatype,   */
/* and a pointer to the next node in the stack                               */
typedef struct _stacknode {
  DATATYPE data;
```

```
  struct _stacknode* next;
} stacknode;

/* defines the stack                                                    */
typedef struct {
  stacknode* head;
} stack;

/* GP structures:                                                       */
/* defines the terminal structure, which contains a name of type char, and the  */
/* value of the type DATATYPE                                           */
typedef struct {
  char *name;
  int *value;
} terminal;

/* defines the function terminal, which contains the name of type char, the     */
/* functions arity, and a pointer to the C function                     */
typedef struct {
  char *name;
  short int arity;
  int (*expression)();
} function;

/* defines the structure element, where the list of elements is the C "program" */
/* which consitutes the individual. element contains a pointer to the next      */
/* element in the C "program", either a pointer to it's function or a pointer   */
/* to it's terminal.                                                    */
```

```
typedef struct _element{
  struct _element* next;
  function *func;
  terminal *term;
} element;

/* defines the individual in the population. It contains a pointer to the first */
/* element in it's C "program", the length of the individual, and teh fitness   */
/* of the individual                                                             */
typedef struct {
  element* elem;
  short int length;
  int fitness;
  short int mate;
  short int number;
} individual;

/* Stack for elements used in eval_expression                                    */
typedef struct _elemstacknode {
  element* data;
  struct _elemstacknode* next;
} elemstacknode;

typedef struct {
  elemstacknode* head;
} elemstack;

/*******************************************************************************/
```

```
/* Variables:                                                                 */
/*****************************************************************************/
/* Structure, a "mirror" of strct terminal.                                  */
typedef struct {
  char *name;
  int value;
} variable;

/* defines the terminal set to be used by the GP                             */
/* the number of terminals are 8 (sensors) + x (number of constants)         */
/* the numbers are just random.                                              */
variable terminalset[NUMBEROFTERMINALS+NUMBEROFCONSTANTS] ={
  {"sensor0",0},
  {"sensor1",0},
  {"sensor2",0},
  {"sensor3",0},
  {"sensor4",0},
  {"sensor5",0},
  {"sensor6",0},
  {"sensor7",0},
  {"constant0",-5},
  {"constant1",-3},
  {"constant2",-1},
  {"constant3",0},
  {"constant4",1},
  {"constant5",2},
  {"constant6",3},
  {"constant7",5}
```

```
};

/* set the population to be an array of individuals, of the length defined by   */
/* POPULATIONSIZE                                                               */
individual* population[POPULATIONSIZE];

/******************************************************************************/
/* Function prototyping:                                                      */
/******************************************************************************/
/* Stack:                                                                     */
void initialisestack(stack* s);
void initialiseelemstack(elemstack* s);
DATATYPE isempty(stack* s);
DATATYPE isemptyelem(elemstack* s);
void push (stack* s, DATATYPE data);
DATATYPE pop(stack* s);
void elempush(elemstack* s, element* data);
element* elempop(elemstack* s);

/* GP:                                                                        */
int gpadd(int listofargs[]);
int gpsub(int listofargs[]);
int gpdiv(int listofargs[]);
int gpmul(int listofargs[]);
int gpand(int listofargs[]);
int gpxor(int listofargs[]);
int gpsll(int listofargs[]);
int gpslr(int listofargs[]);
```

```
int gpmotorleft(int listofargs[]);
int gpmotorright(int listofargs[]);

/* defines the function set to be used by the GP                    */
function functionset[NUMBEROFFUCTIONS] = {
  {"*",2,gpadd},
  {"-",2,gpsub},
  {"/",2,gpdiv},
  {"+",2,gpmul},
  {"^",2,gpand},
  {"&",2,gpxor},
  {"<",1,gpmotorleft},
  {">",1,gpmotorright},
  {"L",1,gpsll},
  {"R",1,gpslr}
};

/* Random:                                                          */
short int randomint(short int maxrange);
float randomfloat(short int maxrange);

/* Robot comm.:                                                     */
void read_sensors();
void init_motors();
void alive();
void flip();

/* GP mortor stuff:                                                 */
```

```
int calc_result(int (*expression)(),int* list);
stack* eval_expression(element* elem, stack* argumentstack);
void cleanup_element_list(element* elem);
individual* generate_random_individual(short int in);
void initialise_population();
int fitness_function();
individual* select_individual(stack* argumentstack);
void crossover(individual* father, individual* mother, individual* fathercopy, individual* mothercopy);
element* copy_func(function* orgfunc);
element* copy_term(terminal* orgterm);
element* copy_elem(element* orgelem);
void copy_parent(individual* parent, individual* child);
void mutation(individual* indiv);
void evolution(stack* argumentstack);
void dump_data();
```

A.2 Main code

```
#include "khep.h"

/*****************************************************************************/
/* Stack things                                                              */
/*****************************************************************************/
/* initialises the stack, i.e. set the first pointer to NULL                 */
void initialisestack(stack* s){ s->head = NULL;};

/* initialises the elementstack, i.e. set the first pointer to NULL          */
void initialiseelemstack(elemstack* s){ s->head = NULL;};

/* checks to see if the stack is empty, returns 1 if it is and 0 if not      */
DATATYPE isempty(stack* s){ if (s->head == NULL) return 1; else return 0; };

/* checks to see if the elementstack is empty, returns 1 if it is and 0 if not  */
DATATYPE isemptyelem(elemstack* s){ if (s->head == NULL) return 1; else return 0; };

/* push, put a new item of DATATYPE on the stack                             */
/* it is called by: push(&stack,value)                                       */
void push(stack* s, DATATYPE data){

  stacknode* newnode = (stacknode*) malloc(sizeof(stacknode));

  newnode->data = data;
  newnode->next = s->head;
  s->head = newnode;
};
```

```
/* pop, returns the top element in the stack. If the stack is empty, it returns */
/* NULL. It is called with pop(&stack)                                           */
DATATYPE pop(stack* s){

  if(isempty(s))
    return ((DATATYPE) NULL);
  else {

    stacknode* secondnode;
    DATATYPE firstnodedata;
    stacknode* firstnode = s->head;

    secondnode = s->head->next;
    s->head = secondnode;
    firstnodedata = firstnode->data;
    free(firstnode);
    return firstnodedata;
  }
};

/* elempush, put a new item of DATATYPE on the elementstack                      */
/* it is called by: elempush(&stack,value)                                       */
void elempush(elemstack* s, element* data){

  elemstacknode* newnode = (elemstacknode*) malloc(sizeof(elemstacknode));

  newnode->data = data;
```

```
  newnode->next = s->head;
  s->head = newnode;
};

/* elempop, returns the top element in the elementstack. If the elementstack is */
/* empty, it returns NULL. It is called with elempop(&stack)                     */
element* elempop(elemstack* s){

  if(isemptyelem(s))
    return NULL;
  else{

    elemstacknode* secondnode;
    element* firstnodedata;
    elemstacknode* firstnode = s->head;

    secondnode = s->head->next;
    s->head = secondnode;
    firstnodedata = firstnode->data;
    free(firstnode);
    return firstnodedata;
  }
};

/*******************************************************************************/
/* GP things                                                                   */
/*******************************************************************************/
/* defines the functions to be used by the GP. All have the following form:    */
```

```
/* DATATYPE "name" (DATATYPE "list of arguments"){                    */
/*   check any constraints;                                           */
/*   return the value of the expression;                              */
/* }                                                                  */
/* The must be in the functionset list to be used                     */

/* ordinary addition                                                  */
int gpadd (int listofargs[]){
  return listofargs[0]+listofargs[1];
};

/* ordinary subtraction                                               */
int gpsub (int listofargs[]){
  return listofargs[0]-listofargs[1];
};

/* Protected division                                                 */
int gpdiv (int listofargs[]){
  if (listofargs[1] == 0)
    return 1;
  else
    return listofargs[0]/listofargs[1];
};

/* ordinary multiplication                                            */
int gpmul (int listofargs[]){
  return listofargs[0]*listofargs[1];
};
```

```
/* ordinary and (between the datetype)                                    */
int gpand (int listofargs[]){
  if (listofargs[0] && listofargs[1])
    return (int) 1;
  else
    return (int) 0;
};

/* gpxor is just a not-and */
int gpxor (int listofargs[]){
  if (listofargs[0] && listofargs[1])
    return (int) 0;
  else
    return (int) 1;
};

/* shifts the three right bytes one places left. NOT GENERIC!             */
int gpsll (int listofargs[]){

  int tmp;

  tmp =  listofargs[0] & (0x0fff);
  return tmp<<4;
};

/* shifts the three left bytes one places right. NOT GENERIC!             */
int gpslr (int listofargs[]){
```

```
  int tmp;

    tmp =  listofargs[0] & (0xfff0);
    return tmp>>4;
  };

  /*the motor controlfunctions*/
  int gpmotorleft (int listofargs[]){

    mot_new_speed_1m(0,listofargs[0]);
    return listofargs[0];
  };

  int gpmotorright (int listofargs[]){

    mot_new_speed_1m(1,listofargs[0]);
    return listofargs[0];
  };

  /*************************************************************/
  /* Random functions                                          */
  /*************************************************************/
  /* returns a random integer between 1 and maxrange           */
  short int randomint (short int maxrange){
    return ((short int)(((float)(rand()))/(((float)32767)/((float)maxrange))));
  };
```

```
/* returns a random real between 1.0 and maxrange                              */
float randomfloat(short int maxrange){
  return ((float)rand()/(((float)32767)/((float)maxrange)));
};

/*****************************************************************************/
/* Functions for communicating with the robot                                */
/*****************************************************************************/
/* The function takes the reflective light value of the 8 sensors, and stores */
/* them in the terminalset.                                                  */
void read_sensors(){

  short int i;

  for(i=0;i<NUMBEROFTERMINALS;i++){
    terminalset[i].value = sens_get_reflected_value(i);
  }
};

/* Initialise the motors, the numbers are from example 2 in the khepera package */
void init_motors()
{
  mot_config_speed_1m(0,3500,800,100);  /* set the PID parameters motor 0 */
  mot_config_speed_1m(1,3500,800,100);  /* set the PID parameters motor 1 */
  exit(0);
};

/* This fuctions recives a result, splits it into two parts, and gives each    */
```

```
/* motor a new speed.                                                    */
void set_motors(int value){

  short int left,right;

  left = ((short int) value & (0xffff0000));
  right = ((short int) value & (0x0000ffff));
  mot_new_speed_1m(1,left);
  mot_new_speed_1m(0,right);
  return;
};

/* Just blink the leds to see if the robot is running.                   */
void alive(){
  for(;;)
    {
      tim_suspend_task(1000);     /* wait 1.43 seconds */
      var_change_led(0);          /* change state of led 0 */
    }
  exit(0);
};

/* Flip the leds                                                         */
void flip(){
  var_change_led(0);
  var_change_led(1);
};
```

```
/*****************************************************************************/
/* GP "motor" stuff                                                          */
/*****************************************************************************/
/* calculates a given expression. All expressions must use an array of       */
/* arguments, in the form of:                                                */
/* DATATYPE gpadd (DATATYPE* arguments){return arguments[1]+arguments[2];}   */
int calc_result(int (*expression)(),int* list){
  return (*expression)(list);
};

/* evaluates a C "program". If the give element is a terminal, it's value is  */
/* pushed onto the stack. If its a function, the number of arguments required */
/* for this function (arity) is poped from the stack, the calc_result is called */
/* and the result is pushed onto the stack.                                  */
stack* eval_expression(element* elem, stack* argumentstack){

  elemstack* tmpstack;
  element* tmpelem;
  short int counter,ar,result,i;

  /* arl[] -- NOT GENERIC, must have enough space to hold the maximum arity of  */
  /* the function set! */

  short int arl[2];

  /* We have to swap the list around, used to be recursion, but was to slow for */
  /* the m64k.                                                                 */
```

```
tmpstack = (elemstack *) malloc(sizeof(elemstack));
initialiseelemstack(tmpstack);
tmpelem = elem;
counter=0;

for (;tmpelem->next != NULL;){
  elempush(tmpstack,tmpelem);
  tmpelem = tmpelem->next;
  counter++;
}

/* Now is the time to calculate */

for(;counter>0;){
  tmpelem = elempop(tmpstack);
  if(tmpelem->func){
    ar=tmpelem->func->arity;
    for(i=0;i<ar;i++){
      arl[i]=pop(argumentstack);
    }
    result = (tmpelem->func->expression)(arl);
    push(argumentstack,result);
  }
  else{
    push(argumentstack, *(tmpelem->term->value));
  }
  counter--;
}
```

```
  free(tmpstack);
  return argumentstack;
};

/* Cleans up an elemet list                                    */
void cleanup_element_list(element* elem){

  element *current,*kenny;
  short int i=0;

  current = elem;
  kenny = current->next;
  if(current->func){
    free(current->func);
  }
  else{
    free(current->term);
  }
  free(current);
  for(;kenny->next != NULL;){
    current = kenny;
    kenny = current->next;
    if(current->func){
      free(current->func);
    }
    else{
      free(current->term);
    }
```

```
    free(current);
    i++;
  }
  i++;
  free(kenny);
};

/* generates one legal individual, where the max length is MAXINDIVIDUALLENGTH, */
/* and, the min length is MININDIVIDUALLENGTH.  To check wether an individual   */
/* is legal or not the arity of the functions and terminals -1 is summed, if    */
/* the sum is -1 the expression is legal.                                       */
individual* generate_random_individual(short int in){

  short int y,i,checksum,numberofnodes,openbranch,currentlength,chance,debug;
  individual* indiv;
  element* node;
  function* cfunc;
  terminal* cterm;

  checksum=0;
  numberofnodes = NUMBEROFFUCTIONS+NUMBEROFTERMINALS+NUMBEROFCONSTANTS;

  indiv = (individual*) malloc(sizeof(individual));
  indiv->elem = NULL;
  indiv->fitness = ZEROFITNESSVALUE;
  indiv->mate = 0;

  debug = 0;
```

```
for(;checksum != -1;){

  checksum = 0;
  currentlength = 0;
  openbranch = 0;

  if(NULL != indiv->elem){
    cleanup_element_list(indiv->elem);
    indiv->elem = NULL;
  }

  indiv->length = randomint(MAXINDIVIDUALLENGTH-MININDIVIDUALLENGTH)+MININDIVIDUALLENGTH;
  for(i=0;i<indiv->length;i++){

    if((openbranch + currentlength) >= indiv->length){
      y = randomint((NUMBEROFTERMINALS+NUMBEROFCONSTANTS)-1);
      node->next = (element*) malloc(sizeof(element));
      node = node->next;
      node->next = NULL;
      cterm = (terminal*) malloc(sizeof(terminal));
      cterm->name = terminalset[y].name;
      cterm->value = &terminalset[y].value;
      node->term = cterm;
      node->func = NULL;
      checksum--;
      openbranch--;
      if (checksum == 0){
```

```
    openbranch = 0;
  }
 }
 else{
  if(currentlength==0){
   y = randomint(NUMBEROFFUCTIONS-1);
   node = (element*) malloc(sizeof(element));
   indiv->elem = node;
   node->next = NULL;
   cfunc = (function*) malloc(sizeof(function));
   y = randomint(5); /*numberoffunctions with arity 2*/
   cfunc->arity = functionset[y].arity;
   cfunc->name = functionset[y].name;
   cfunc->expression = functionset[y].expression;
   node->func = cfunc;
   node->term = NULL;
   checksum = checksum + ((cfunc->arity) - 1);
   openbranch = cfunc->arity;
  }
  else{
   chance = randomint(numberofnodes);
   if(chance <= 23){
    y = randomint(NUMBEROFFUCTIONS-1);
    node->next = (element*) malloc(sizeof(element));
    node = node->next;
    node->next = NULL;
    cfunc = (function*) malloc(sizeof(function));
    cfunc->name = functionset[y].name;
```

```
      cfunc->arity = functionset[y].arity;
      cfunc->expression = functionset[y].expression;
      node->func = cfunc;
      node->term = NULL;
      checksum = checksum + (cfunc->arity - 1);
      if(openbranch == 0){
        openbranch = openbranch + cfunc->arity;
      }
      else{
        openbranch = openbranch + (cfunc->arity - 1);
      }
    }
    else{
      y = randomint((NUMBEROFTERMINALS+NUMBEROFCONSTANTS)-1);
      node->next = (element*) malloc(sizeof(element));
      node = node->next;
      node->next = NULL;
      cterm = (terminal*) malloc(sizeof(terminal));
      cterm->name = terminalset[y].name;
      cterm->value = &terminalset[y].value;
      node->term = cterm;
      node->func = NULL;
      checksum--;
      openbranch--;
    }/*end make terminal*/
  }/* end currentlength !=0*/
}/*end openbranch+currentlength != indiv->length*/
currentlength++;
```

```
    }/*end for(i=0;i<indiv->length;i++) */
    debug++;
  }/*end for(;checksum != -1;)*/
  indiv->number = in;
  return indiv;
};

/* This function initilises the whole population, and fills it with random      */
/* idndividuals.                                                                */
void initialise_population(){

  short int i;

  for(i=0;i<POPULATIONSIZE;i++){
    population[i] = generate_random_individual(i);
  }
  return;
};

/* This is a fitness function, should be changed for different experiments.     */
int fitness_function(){

  int fitness,m1,m2,s;
  int alpha,beta,i;
  /*  int alpha,beta;*/

  m1=mot_get_speed(0);
  m2=mot_get_speed(1);
```

```
  /*  s=0;*/

  /* TEST THE WEIGHTS ALPHA=1 IS TO SMALL, NOW TESTING ALPHA=5 */

  /* constants for weighting*/
  alpha=16;
  beta=2;

  for(i=0;i<NUMBEROFTERMINALS;i++){
    s = s+terminalset[i].value;
  }

  /*  s = terminalset[0].value;
  /*    for(i=1;i<8;i++){
  /*      if(s < terminalset[i].value){
  /*        s = terminalset[i].value;
  /*      }
  /*    }

  /*obstacle avoidance */
  fitness = (alpha * ((m1+m2)-abs(m1-m2))) -  (beta * s);
  /*WF*/
  return fitness;
};

/* Function for selecting an individual                                    */
/* Finds the sensor values, runs the robot, sleeps for 1/2 sek, reads the  */
```

```
/* sensors and finds the  fitness                                    */
/* The two parents can't be the same individual!                     */
/*  individual* select_individual(stack* argumentstack){ */
individual* select_individual(stack* argumentstack){

  short int testindiv,i,used;
  int fitness=0;
  individual *indiv;
  int value;

  used = 1;
  for(;used == 1;){
    testindiv = randomint(POPULATIONSIZE-1);
    indiv = population[testindiv];
    used = indiv->mate;
  }

  indiv->mate = 1;

    /*  uncomment this to run ONE time */

    read_sensors();
    value = pop(eval_expression(indiv->elem,argumentstack));
    tim_suspend_task(600);
    read_sensors();
    indiv->fitness = fitness_function();
    mot_stop();
    return indiv;
```

```
    /* do the testing for more than one iteration */

/*      for(i=0;i<NUMBEROFITERATIONS;i++){ */
/*      read_sensors(); */
/*      value = pop(eval_expression(indiv->elem,argumentstack)); */
/*      tim_suspend_task(PERIODOFSLEEP); */
/*      read_sensors(); */
/*      fitness = fitness + fitness_function(); */
/*    } */
/*    indiv->fitness = ((int) ((float) fitness)/((float) NUMBEROFITERATIONS)); */
/*    return indiv; */
};


/* Here is the crossower function. one point crossower, random.                */
/* crossower is only allowed where there is no posibility of making ilegal     */
/* individuals.                                                                */
/* The parents is exchanged with the children.                                 */
element* copy_func(function* orgfunc){

  element* tmpelem;
  function* tmpfunc;

  tmpelem = (element*) malloc(sizeof(element));
  tmpelem->next = NULL;
  tmpelem->term = NULL;
  tmpfunc = (function*) malloc(sizeof(function));
```

```
  tmpfunc->name = orgfunc->name;
  tmpfunc->arity = orgfunc->arity;
  tmpfunc->expression = orgfunc->expression;
  tmpelem->func = tmpfunc;

  return tmpelem;
};

element* copy_term(terminal* orgterm){

  element* tmpelem;
  terminal* tmpterm;

  tmpelem = (element*) malloc(sizeof(element));
  tmpelem->next = NULL;
  tmpelem->func = NULL;
  tmpterm = (terminal*) malloc(sizeof(terminal));
  tmpterm->name = orgterm->name;
  tmpterm->value = orgterm->value;
  tmpelem->term = tmpterm;

  return tmpelem;
};

element* copy_elem(element* orgelem){

  if(orgelem->func){
    return copy_func(orgelem->func);
```

```
  }else{
    return copy_term(orgelem->term);
  }
};

void copy_parent(individual* parent, individual* child){

  element *origelem,*copyelem;
  short int i;

  origelem = parent->elem;
  copyelem = copy_elem(origelem);
  child->elem = copyelem;
  if(parent->length != 1){
    for(i=1;i<parent->length;i++){
      copyelem->next = copy_elem(origelem->next);
      copyelem = copyelem->next;
      origelem = origelem->next;
    }
  }
  child->length = parent->length;
};

void crossover(individual* father, individual* mother, individual* fathercopy, individual* mothercopy){

  short int fathercross,mothercross,i,checksum,fathercopylength,mothercopylength;
  element *origelem,*copyelem,*elemorig,*elemcopy;
```

```
cleanup_element_list(fathercopy->elem);
cleanup_element_list(mothercopy->elem);
fathercopy->elem = NULL;
mothercopy->elem = NULL;
fathercopy->mate = 0;
mothercopy->mate = 0;
fathercopy->fitness = ZEROFITNESSVALUE;
mothercopy->fitness = ZEROFITNESSVALUE;

if((father->length == 1) && (mother->length == 1)){
  /* both parents has a length of 1 */
  fathercopy->elem = copy_elem(father->elem);
  fathercopy->length = 1;
  mothercopy->elem = copy_elem(mother->elem);
  mothercopy->length = 1;
}
else if(father->length == 1){
  /* only father has a length of 1 */
  mothercross = randomint(mother->length);
  if(mothercross == 0){
    copy_parent(father,fathercopy);
    copy_parent(mother,mothercopy);
  }
  else{
    /* mother has another crossoverpoint that 0 but father->length == 1 */
    /* copy mothers first part onto mothercopy */
    origelem = mother->elem;
    copyelem = copy_elem(origelem);
```

```
mothercopy->elem = copyelem;
mothercopylength = 1;
for(i=1;i<mothercross-1;i++){
  copyelem->next = copy_elem(origelem->next);
  copyelem = copyelem->next;
  origelem = origelem->next;
  mothercopylength++;
}
/* copy mothers crossoverpart onto fathercopy */
checksum = 0;
elemcopy = copy_elem(origelem->next);
fathercopy->elem = elemcopy;
origelem = origelem->next;
fathercopylength = 1;
if(origelem->func){checksum = checksum + origelem->func->arity-1;}
else{checksum--;}
for(;checksum != -1;){
  elemcopy->next = copy_elem(origelem->next);
  elemcopy = elemcopy->next;
  origelem = origelem->next;
  if(origelem->func){checksum = checksum + origelem->func->arity-1;}
  else{checksum--;}
  fathercopylength++;
}
/* copy father onto mothercopy */
copyelem->next = copy_elem(father->elem);
copyelem = copyelem->next;
mothercopylength++;
```

```
    /* copy the last part of mother onto mothercopy */
    for(;origelem->next != NULL;){
      copyelem->next = copy_elem(origelem->next);
      copyelem = copyelem->next;
      origelem = origelem->next;
      mothercopylength++;
    }
    fathercopy->length = fathercopylength;
    mothercopy->length = mothercopylength;
  }
}
else if(mother->length == 1){
  /* only mother has a length of 1 */
  fathercross = randomint(father->length);
  if(fathercross == 0){
    copy_parent(father,fathercopy);
    copy_parent(mother,mothercopy);
  }
  else{
    /* mother has the length of 1 and father has a crossoverpoint which is not 0*/
    /* copy the first part of the father onto fathercopy */
    elemorig = father->elem;
    elemcopy = copy_elem(elemorig);
    fathercopy->elem = elemcopy;
    fathercopylength = 1;
    for(i=1;i<fathercross-1;i++){
      elemcopy->next = copy_elem(elemorig->next);
      elemorig = elemorig->next;
```

```
  elemcopy = elemcopy->next;
  fathercopylength++;
}
/* copy fathers crossoverpart onto the mothercopy */
checksum = 0;
mothercopy->elem = copy_elem(elemorig->next);
elemorig = elemorig->next;
copyelem = mothercopy->elem;
mothercopylength = 1;
if(elemorig->func){checksum = checksum + elemorig->func->arity-1;}
else{checksum--;}
for(;checksum != -1;){
  copyelem->next = copy_elem(elemorig->next);
  copyelem = copyelem->next;
  elemorig = elemorig->next;
  if(elemorig->func){checksum = checksum + elemorig->func->arity-1;}
  else{checksum--;}
  mothercopylength++;
}
/* copy mother onto fathercopy */
elemcopy->next = copy_elem(mother->elem);
elemcopy = elemcopy->next;
fathercopylength++;
/* copy the last part of father onto fathercopy */
for(;elemorig->next != NULL;){
  elemcopy->next = copy_elem(elemorig->next);
  elemcopy = elemcopy->next;
  elemorig = elemorig->next;
```

```
      fathercopylength++;
    }
    fathercopy->length = fathercopylength;
    mothercopy->length = mothercopylength;
  }
}
else{
  /* neither of the parents has a length of 1*/
  mothercross = randomint(mother->length);
  fathercross = randomint(father->length);
  if((mothercross == 0) && (fathercross == 0)){
    copy_parent(father,fathercopy);
    copy_parent(mother,mothercopy);
  }
  else if(fathercross == 0){
    /* only the father has a crossoverpoint of 0 */
    /* copy the first part of the mother onto mothercopy */
    origelem = mother->elem;
    mothercopy->elem = copy_elem(origelem);
    copyelem = mothercopy->elem;
    mothercopylength = 1;
    for(i=1;i<mothercross-1;i++){
      copyelem->next = copy_elem(origelem->next);
      origelem = origelem->next;
      copyelem = copyelem->next;
      mothercopylength++;
    }
    /* copy the whole father onto the mothercopy */
```

```
elemorig = father->elem;
copyelem->next = copy_elem(elemorig);
copyelem = copyelem->next;
mothercopylength++;
for(i=1;i<father->length;i++){
  copyelem->next = copy_elem(elemorig->next);
  elemorig = elemorig->next;
  copyelem = copyelem->next;
  mothercopylength++;
}
/* copy the mothercrossoverpart onto the fathercopy */
checksum = 0;
fathercopy->elem = copy_elem(origelem->next);
origelem = origelem->next;
elemcopy = fathercopy->elem;
fathercopylength = 1;
if(origelem->func){checksum = checksum + origelem->func->arity-1;}
else{checksum--;}
for(;checksum != -1;){
  elemcopy->next = copy_elem(origelem->next);
  elemcopy = elemcopy->next;
  origelem = origelem->next;
  if(origelem->func){checksum = checksum + origelem->func->arity-1;}
  else{checksum--;}
  fathercopylength++;
}
/* copy the last part of the mother onto the mothercopy */
for(;origelem->next != NULL;){
```

```
    copyelem->next = copy_elem(origelem->next);
    copyelem = copyelem->next;
    origelem = origelem->next;
    mothercopylength++;
  }
  fathercopy->length = fathercopylength;
  mothercopy->length = mothercopylength;
}
else if(mothercross == 0){
  /* only the mother has a crossoverpoint of 0 */
  /* copy the first part of the father onto fathercopy */
  elemorig = father->elem;
  elemcopy = copy_elem(elemorig);
  fathercopy->elem = elemcopy;
  fathercopylength = 1;
  for(i=1;i<fathercross-1;i++){
    elemcopy->next = copy_elem(elemorig->next);
    elemorig = elemorig->next;
    elemcopy = elemcopy->next;
    fathercopylength++;
  }
  /* copy the whole mother onto the fathercopy */
  origelem = mother->elem;
  elemcopy->next = copy_elem(origelem);
  elemcopy = elemcopy->next;
  fathercopylength++;
  for(i=1;i<mother->length;i++){
    elemcopy->next = copy_elem(origelem->next);
```

```
  origelem = origelem->next;
  elemcopy = elemcopy->next;
  fathercopylength++;
}
/* copy the fathercrossoverpart onto the mothercopy */
checksum = 0;
mothercopy->elem = copy_elem(elemorig->next);
elemorig = elemorig->next;
copyelem = mothercopy->elem;
mothercopylength = 1;
if(elemorig->func){checksum = checksum + elemorig->func->arity-1;}
else{checksum--;}
for(;checksum != -1;){
  copyelem->next = copy_elem(elemorig->next);
  copyelem = copyelem->next;
  elemorig = elemorig->next;
  if(elemorig->func){checksum = checksum + elemorig->func->arity-1;}
  else{checksum--;}
  mothercopylength++;
}
/* copy the last part of the father onto the fathercopy */
for(;elemorig->next != NULL;){
  elemcopy->next = copy_elem(elemorig->next);
  elemcopy = elemcopy->next;
  elemorig = elemorig->next;
  fathercopylength++;
}
fathercopy->length = fathercopylength;
```

```
  mothercopy->length = mothercopylength;
}
else{
  /* none of the parents has a crossoverpoint of 0*/
  /* copy the first part of the father onto the fathercopy */
  elemorig = father->elem;
  fathercopy->elem = copy_elem(elemorig);
  elemcopy = fathercopy->elem;
  fathercopylength = 1;
  for(i=1;i<fathercross-1;i++){
    elemcopy->next = copy_elem(elemorig->next);
    elemorig = elemorig->next;
    elemcopy = elemcopy->next;
    fathercopylength++;
  }
  /* copy the first part of the mother onto the mothercopy */
  origelem = mother->elem;
  mothercopy->elem = copy_elem(origelem);
  copyelem = mothercopy->elem;
  mothercopylength = 1;
  for(i=1;i<mothercross-1;i++){
    copyelem->next = copy_elem(origelem->next);
    origelem = origelem->next;
    copyelem = copyelem->next;
    mothercopylength++;
  }
  /* copy the crossover part of the father onto the mothercopy */
  checksum = 0;
```

```
copyelem->next = copy_elem(elemorig->next);
elemorig = elemorig->next;
copyelem = copyelem->next;
mothercopylength++;
if(elemorig->func){checksum = checksum + elemorig->func->arity-1;}
else{checksum--;}
for(;checksum != -1;){
  copyelem->next = copy_elem(elemorig->next);
  copyelem = copyelem->next;
  elemorig = elemorig->next;
  if(elemorig->func){checksum = checksum + elemorig->func->arity-1;}
  else{checksum--;}
  mothercopylength++;
}
/* copy the crossover part of the mother onto the fathercopy */
checksum = 0;
elemcopy->next = copy_elem(origelem->next);
origelem = origelem->next;
elemcopy = elemcopy->next;;
fathercopylength++;
if(origelem->func){checksum = checksum + origelem->func->arity-1;}
else{checksum--;}
for(;checksum != -1;){
  elemcopy->next = copy_elem(origelem->next);
  elemcopy = elemcopy->next;
  origelem = origelem->next;
  if(origelem->func){checksum = checksum + origelem->func->arity-1;}
  else{checksum--;}
```

```
        fathercopylength++;
      }
      /* copy the last part of the father onto the fathercopy */
      for(;elemorig->next != NULL;){
        elemcopy->next = copy_elem(elemorig->next);
        elemcopy = elemcopy->next;
        elemorig = elemorig->next;
        fathercopylength++;
      }
      /* copy the last part of the mother onto the mothercopy */
      for(;origelem->next != NULL;){
        copyelem->next = copy_elem(origelem->next);
        copyelem = copyelem->next;
        origelem = origelem->next;
        mothercopylength++;
      }
      fathercopy->length = fathercopylength;
      mothercopy->length = mothercopylength;
    }
  }
  father->mate = 0;
  mother->mate = 0;
  /*  father->fitness = 16384;*/
  /*  mother->fitness = 16384;*/
};

/* We Need mutation!                                                    */
/* The mutation rate is set by MUTATIONRATE. Only one random function or */
```

```
/* terminal is mutated; if nessesary.                                        */
/* If a function is to be mutated, another function with the same arity! is  */
/* selected a replaceses the mutated gene. If it is a terminal, another radom */
/* terminal is selected to be used as replacement.                           */
void mutation(individual* indiv){

  int chance;
  int node,i,arity;
  element* tmpelem;
  function* tmpfunc;
  terminal* tmpterm;

  chance = randomint(1000);
  /* Lets mutate */
  if (chance <= MUTATIONRATE){
    node = randomint(indiv->length);
    tmpelem = indiv->elem;
    for(i=1;i<node;i++){
      tmpelem = tmpelem->next;
    }
    /* heres the element to be mutatet */
    /* function ? */
    if (tmpelem->func != NULL){

      tmpfunc = (function*) malloc(sizeof(function));
      arity = tmpelem->func->arity;
      switch (arity) {
      case 1:
```

```
      node = randomint(3);
      tmpfunc->name = functionset[node+6].name;
      tmpfunc->arity = functionset[node+6].arity;
      tmpfunc->expression = functionset[node+6].expression;
      break;
    case 2:
      node = randomint(5);
      tmpfunc->name = functionset[node].name;
      tmpfunc->arity = functionset[node].arity;
      tmpfunc->expression = functionset[node].expression;
    }
    free(tmpelem->func);
    tmpelem->func = tmpfunc;
  }
  /* If the node is a terminal just replace it.
  else{

                                                    */

    tmpterm = (terminal*) malloc(sizeof(terminal));
    node = randomint(((NUMBEROFCONSTANTS+NUMBEROFTERMINALS)-1));
    tmpterm->name = terminalset[node].name;
    tmpterm->value = &terminalset[node].value;
    free(tmpelem->term);
    tmpelem->term = tmpterm;
  }
 }
};
```

```
/* the main evolutionay loop, all selection is tournament, and the GP is       */
/* running as steady state.                                                    */
/* For each generation select two cadidates, the best will become the mother.  */
/* Do it again, and find the father. Do crossower adn mutaion. All until we run */
/* out of generations.                                                         */
/* Flip the leds each time a new individual is beeing testet.                  */
void evolution(stack* argumentstack){

  individual* mother;
  individual* father;
  individual* mothercopy;
  individual* fathercopy;
  individual* candidate1;
  individual* candidate2;

  short int debug,j,i,k,bestofgeneration,worstofgeneration;
  float averageofgeneration;

  for(i=0;i<NUMBEROFGENERATIONS;i++){

#ifdef DEBUG
    printf("Her starter generation %i\n",i);
#endif
    flip();

    /* in obstacle avoidance fitness should be maximised */

    candidate1 = select_individual(argumentstack);
```

```
flip();

candidate2 = select_individual(argumentstack);
if(candidate1->fitness > candidate2->fitness){ /* note which way does the fitness go? */
  mother = candidate1;
  mothercopy = candidate2;
}else{
  mother = candidate2;
  mothercopy = candidate1;
}
/* Find the father                                                    */
flip();

candidate1 = select_individual(argumentstack);
flip();

candidate2 = select_individual(argumentstack);
if(candidate1->fitness > candidate2->fitness){
  father = candidate1;
  fathercopy = candidate2;
}else{
  father = candidate2;
  fathercopy = candidate1;
}

/* Find the best, worst, and average of each generation */
bestofgeneration=0;
worstofgeneration=0;
```

```
averageofgeneration=0;

for(j=0;j<POPULATIONSIZE;j++){
  candidate1 = population[j];
  if(candidate1->fitness != ZEROFITNESSVALUE){
    bestofgeneration=candidate1->fitness;
    worstofgeneration=candidate1->fitness;
    averageofgeneration=((float) candidate1->fitness);
    break;
  }
}
k=j+1;
for(k;k<POPULATIONSIZE;k++){
  candidate1=population[k];
  if(candidate1->fitness != ZEROFITNESSVALUE){
    if(candidate1->fitness < worstofgeneration){
      worstofgeneration=candidate1->fitness;
    }else{
      if(candidate1->fitness > bestofgeneration){
        bestofgeneration=candidate1->fitness;
      }
    }
    averageofgeneration = (((float) j) * averageofgeneration + ((float) candidate1->fitness)) / ((float) (j+1));
  }
}
printf("%i : %i : %f : %i\n",i,bestofgeneration,averageofgeneration,worstofgeneration);
```

```
#ifdef DEBUG
    printf("\n I generation %i er fitness som følger:\n",i);
    printf("\tBEST: %i",bestofgeneration[i]);
    printf("\tAVERAGE: %f",averageofgeneration[i]);
    printf("\tWORST: %i\n",worstofgeneration[i]);
#endif

    /* Do the crossower and mutation.                                    */
    crossover(father,mother,fathercopy,mothercopy);

    mutation(fathercopy);
    mutation(mothercopy);
  }
  return;
};

void dump_data(){

  int i;

  mot_stop();
  var_on_led(0);
  var_on_led(1);
  tim_suspend_task(10000);
  /* print out the stuff */
  flip();
  tim_suspend_task(5000);
  printf("Generation\tBest\tAverage\tWorst\n");
```

```
  for(i=0;i<NUMBEROFGENERATIONS;i++){
    var_off_led(0);
    /*    printf("%i : %i : %f : %i\n",i,bestofgeneration[i],averageofgeneration[i],worstofgeneration[i]);*/
    var_on_led(0);
  }
  var_on_led(0);
  var_on_led(1);
  return;
}

/*****************************************************************************/
/* Main function                                                             */
/*****************************************************************************/

main(){

  stack* argumentstack;
  short int i;

  /* Do all the initialising of the robot.                                  */
  bios_reset();
  mot_reset();
  sens_reset();
  msg_reset();
  var_reset();
  ser_reset();
  str_reset();
  tim_reset();
```

```
  /* Hopefully setting the cpu to 16 Mhz                                */
  /* var_set_cpu_speed(0); */

  /*  srand(8913610860);*/

  /* Fire up the motors.                                                */
  tim_new_task(init_motors);

  /* Turn on both leds to see we are generation the population.         */
  var_on_led(0);
  var_on_led(1);
  /* Init pop.                                                          */
#ifdef DEBUG
  printf("begin initialise_population();\n");
#endif
  initialise_population();
#ifdef DEBUG
  printf("end initialise_population();\n");
#endif

  /* Turn off both leds to see we are through with the init pop         */
  var_off_led(0);
  var_off_led(1);

  /* fork off "debug" mode.                                             */
  /*  tim_new_task(alive); */

  /* init the stack.                                                    */
```

```
argumentstack= (stack *) malloc(sizeof(stack));
initialisestack(argumentstack);

                                                */

/* Run the main loop.
evolution(argumentstack);

  /* dump the stats */
  /*  dump_data();*/

};
```