In compliance with the
Canadian Privacy Legislation
some supporting forms
may have been removed from
this dissertation.

While these forms may be included
in the document page count,
their removal does not represent
any loss of content from the dissertation.

Hierarchical Learning Systems

Robotic Control Using Hierarchical Genetic Programming

By
Marcin L. Pilat

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfilment of
the requirements for the degree of
Master of Computer Science

Ottawa-Carleton Institute for Computer Science
School of Computer Science
Carleton University
Ottawa, Ontario

August 18, 2003

© Copyright
2003, Marcin L. Pilat

National Library
of Canada

Bibliothèque nationale
du Canada

Acquisitions and
Bibliographic Services

Acquisitions et
services bibliographiques

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

ISBN: 0-612-88973-4

Our file Notre référence

ISBN: 0-612-88973-4

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

Canada

Abstract

In this thesis, we study the use of hierarchical genetic programming techniques to evolve robotic controllers for a simulated Khepera miniature robot. We study GP chromosome representation methods of linear-genome and tree-based and HGP techniques of Automatically Defined Functions, Module Acquisition, and Adaptive Representation. We train robotic controllers in the tasks of obstacle avoidance, wall following, and light avoidance. Our results enable us to compare and contrast the five studied representation methods and to provide suggestions for their improvement.

Acknowledgements

First of all, I would like to thank my supervisor, Dr. Franz Oppacher, for his insightful comments and suggestions as well as his support for this thesis. Dr. Oppacher introduced me to Artificial Intelligence and especially to the area of Evolutionary Computation. It was an honour working with him.

I would like to thank the teaching staff at Carleton University for imparting on me their ideas and their enthusiasm for academic life. Every course I took was a learning experience that influenced and will influence my future actions. I would also like to thank the administrative staff at Carleton University for their support and a helping hand.

I would also like to thank Carleton University and the Natural Sciences and Engineering Research Council of Canada (NSERC) for their generous financial support. The research done for this thesis was mainly supported by an NSERC PGS-A Postgraduate Scholarship.

Finally, I would like to thank my family who have stuck with me and supported me throughout my studies. Their love and support drove me to complete my thesis and degree. They had to put up with my moods and busy schedule at times and thus I especially appreciate all they have done. I dedicate this thesis to my family.

Table of Contents

Acknowledgements	iv
1 Introduction	1
1.1 Motivation	2
1.2 Problem	2
1.3 Goals	3
1.4 Contributions	4
1.5 Outline	5
2 Background	6
2.1 Evolutionary Computation	6
2.2 Genetic Algorithms	7
2.2.1 Algorithm	8
2.2.2 Selection	9
2.2.3 Genetic Operators	10
2.2.4 Building Blocks Hypothesis	11
2.3 Genetic Programming	12
2.3.1 Chromosome Structure	13
2.3.2 Fitness	14
2.3.3 Genetic Operators	15
2.3.4 Building Blocks Hypothesis	16
2.4 Hierarchical Genetic Programming	17
2.4.1 Automatically Defined Functions	18
2.4.2 Module Acquisition	19

2.4.3	Adaptive Representation	19
2.4.4	Local Modules	20
2.5	Robot Control	21
2.5.1	Evolutionary Robotics	22
2.5.2	Genetic Programming	22
2.6	Khepera Robot	24
2.7	Robotic Simulators	25
2.7.1	Khepera Simulators	26
2.7.2	Khepera GP Simulator	27
2.8	Summary	27
3	Khepera GP Simulator	28
3.1	World Simulation	29
3.1.1	World Model	29
3.1.2	Robot Model	30
3.1.3	World Interaction	32
3.2	Evolutionary Controller	33
3.2.1	Tasks	33
3.2.2	Fitness Function Scripting	34
3.2.3	Run Statistics	34
3.3	User Interface	36
3.3.1	World Editor	37
3.3.2	Info Window	38
3.3.3	Simulator Interface	39
3.4	Code Structure	44
3.4.1	Application Classes	44
3.4.2	Data Classes	45
3.4.3	User Interface Classes	48
3.4.4	COM Objects and Interfaces	49
3.5	Summary	49
4	Robotic Controllers	50
4.1	Generic Architecture	50

4.2	Linear Genome GP	51
4.2.1	Chromosome Structure	53
4.2.2	Population Generation	54
4.2.3	Chromosome Evaluation	54
4.2.4	Genetic Operators	55
4.3	Tree-based GP	56
4.3.1	Chromosome Structure	57
4.3.2	Population Generation	58
4.3.3	Chromosome Evaluation	60
4.3.4	Genetic Operators	60
4.4	Automatically Defined Functions HGP	61
4.4.1	Chromosome Structure	61
4.4.2	Population Generation	62
4.4.3	Chromosome Evaluation	63
4.4.4	Genetic Operators	64
4.5	Module Acquisition HGP	65
4.5.1	Chromosome Structure	65
4.5.2	Population Generation	65
4.5.3	Chromosome Evaluation	66
4.5.4	Genetic Operators	66
4.6	Adaptive Representation HGP	68
4.6.1	Chromosome Structure	68
4.6.2	Subroutine Discovery	69
4.6.3	Evolutionary Algorithm	71
4.6.4	Population Generation	75
4.6.5	Chromosome Evaluation	75
4.6.6	Genetic Operators	75
4.7	Summary	76
5	Results	77
5.1	Obstacle Avoidance	77
5.1.1	Linear Genome GP	79

5.1.2	Tree-based GP	81
5.1.3	ADF HGP	85
5.1.4	MA HGP	89
5.1.5	ARL HGP	92
5.1.6	Random Test	95
5.1.7	Task Summary	97
5.2	Wall Following	98
5.2.1	Linear Genome GP	100
5.2.2	Tree-based GP	101
5.2.3	ADF HGP	105
5.2.4	MA HGP	106
5.2.5	ARL HGP	107
5.2.6	Random Test	112
5.2.7	Task Summary	112
5.3	Light Avoidance	113
5.3.1	Linear Genome GP	114
5.3.2	Tree-based GP	117
5.3.3	ADF HGP	119
5.3.4	MA HGP	120
5.3.5	ARL HGP	122
5.3.6	Random Test	127
5.3.7	Task Summary	127
5.4	Summary	128
6	Conclusions	129
6.1	Summary of Results	129
6.1.1	Linear Genome Approach	130
6.1.2	Hierarchical Genetic Programming Approaches	131
6.1.3	Multiple Symbolic Regression	133
6.2	Contributions	133
6.3	Improvements and Future Work	134
6.4	Summary	137

A Program Code Examples	138
Bibliography	146

List of Tables

4.1	Sample program code of an individual using the linear genome GP method.	54
4.2	Primitive values of each instruction part in the linear genome GP method as used in the experiments.	55
4.3	Values for the contents of the function set and terminal set available in the Simulator.	59
4.4	Test cases used in SFM calculation and their corresponding distance and light values.	73
4.5	Categorization of standardized fitness measure values.	74
5.1	Summary of parameters used during training of obstacle avoidance with the linear genome method.	80
5.2	Summary of parameters used during training of obstacle avoidance with the tree-based method.	83
5.3	Summary of parameters used during training of obstacle avoidance with the ADF HGP method.	87
5.4	Summary of parameters used during training of obstacle avoidance with the MA HGP method.	90
5.5	Summary of parameters used during training of obstacle avoidance with the ARL HGP method.	93
5.6	Summary of parameters used during training of wall following with the linear genome GP method.	102
5.7	Summary of parameters used during training of wall following with the tree-based GP method.	103

5.8	Summary of parameters used during training of wall following with the ADF HGP method.	105
5.9	Summary of parameters used during training of wall following with the MA HGP method.	107
5.10	Summary of parameters used during training of wall following with the ARL HGP method.	109
5.11	Summary of parameters used during training of light avoidance with the linear genome GP method.	116
5.12	Summary of parameters used during training of light avoidance with the tree-based GP method.	118
5.13	Summary of parameters used during training of obstacle avoidance with the ADF HGP method.	120
5.14	Summary of parameters used during training of light avoidance with the MA HGP method.	123
5.15	Summary of parameters used during training of light avoidance with the ARL HGP method.	124
6.1	Summary of results from our experiments for each of the studied methods. Behavioural performance was based on first occurrence of good behaviour. Performance of each method was relative to performance of other methods.	133

List of Figures

2.1	Pseudocode of a typical genetic algorithm.	8
2.2	Crossover operator in genetic algorithms. A random crossover point is chosen and parents exchange corresponding substrings to form children.	11
2.3	Structure of a typical GP chromosome. The tree evaluates to LISP S-expression $(+ (- 6 1) (* x (+ 1 7)))$	13
2.4	Crossover operator in genetic programming. Subtrees A and B are swapped between the two parents to form two children.	16
2.5	Khepera robot (left) and Khepera II robot (right). Images curtesy of K-Team S.A.	24
2.6	Schematic view of the Khepera robot. Sensors are labelled s0 to s7 and motors are labelled m1 and m2.	25
3.1	Screenshot of the Khepera GP Simulator for Windows®.	36
3.2	Simulator Toolbar with location of various tool and button groups.	37
3.3	Properties Dialogs for a Light Box and Robot object.	38
3.4	Task Library Editor enables the user to create and modify a library of learning tasks. Task Editor lets the user modify learning task properties.	40
3.5	Fitness Function Editor allows for editing of the fitness function definition written in JScript™.	41
3.6	Chromosome Structure Properties dialogs for linear and tree-based chromosomes.	42
3.7	Chromosome Browser allows the user to view the chromosomes in the current population.	43
3.8	Module Information dialog shows a list of modules or functions used by the current HGP-based population.	43

3.9	Current Population Statistics dialog displays statistical information gathered during the last generation of the GP controller simulation. .	44
4.1	Pseudocode of the genetic programming algorithm used in our research.	52
4.2	Schematic diagram of the generic GP robotic controller used in the Simulator.	53
4.3	Example of crossover between linear genome chromosomes.	56
4.4	Examples of different cases of gene mutation in a linear genome chromosome.	57
4.5	Structure of a minimal ADF chromosome with one function-defining branch and one result-producing branch. The dashed line represents boundary between invariant and non-invariant points.	62
4.6	Module creation in Module Acquisition. Part a) shows creation of a parameterless function and part b) creation of a function with 2 parameters.	67
4.7	Our implementation of the Subroutine-Discovery algorithm.	70
4.8	Subroutine creation in Adaptive Representation.	70
4.9	Pseudocode of our implementation of the ARL algorithm.	72
5.1	Training environment <i>Nordin2</i> and training and testing environment <i>squares</i> used in obstacle avoidance experiments.	79
5.2	Summary of behaviours learned during experimentation with the Khepera robot.	80
5.3	Graphs of entropy and average size vs. the number of generations in a sample run O9a.	81
5.4	Trace run of perfect obstacle avoidance behaviour from run O8c. . . .	82
5.5	Graphs of entropy and average size vs. the number of generations in a sample run O36.	83
5.6	Trace run of perfect obstacle avoidance behaviour from run O36a. . .	84
5.7	Graphs of entropy and average size vs. the number of generations in a sample run O152.	86
5.8	Graphs of entropy and average size vs. the number of generations in a sample run O160.	87

5.9	Graphs of entropy and average size, average SC, average EC vs. the number of generations in a sample run O52.	88
5.10	Graphs of entropy and average size, average SC, average EC vs. the number of generations in a sample run O66.	88
5.11	Trace run of perfect obstacle avoidance behaviour from run O52a. . .	89
5.12	Trace run of perfect obstacle avoidance behaviour from run O63a. . .	90
5.13	Graphs of entropy and average size, average SC, average EC vs. the number of generations in a sample run O76.	91
5.14	Graphs of entropy and average size, average SC, average EC vs. the number of generations in a sample run O88.	91
5.15	Trace run of perfect obstacle avoidance behaviour from run O94a. . .	92
5.16	Graphs of entropy and average size, average SC, average EC, and number of functions vs. the number of generations in a sample run O118. . .	94
5.17	Graphs of entropy and average size, average SC, average EC, and number of functions vs. the number of generations in a sample run O122. . .	95
5.18	Graphs of entropy and average size, average SC, average EC, and number of functions vs. the number of generations in a sample run O129. . .	96
5.19	Graphs of entropy and average size vs. the number of generations in a sample run R10.	97
5.20	Graphs of minimum, maximum, and average generations of first detection of good obstacle avoidance behaviour for each chromosome representation method.	98
5.21	Fitness Function for the wall following task.	99
5.22	Plot of sensor part values vs. the raw sensor values of three side sensors used in wall following fitness function.	100
5.23	Training environment <i>challways</i> and testing environment <i>maze1</i> used in wall following experiments.	101
5.24	Graphs of entropy and average size vs. the number of generations in a sample run W5.	102
5.25	Trace run of perfect maze-following behaviour from run W8a.	103
5.26	Graphs of entropy and average size vs. the number of generations in a sample run W25.	104

5.27	Graphs of entropy and average size, average SC, and average EC vs. the number of generations in a sample run W35.	106
5.28	Graphs of entropy and average size, average SC, and average EC vs. the number of generations in a sample run W57.	108
5.29	Trace run of near-perfect maze-following behaviour from run W58a. .	108
5.30	Graphs of entropy, average size, average SC, average EC, and number of functions vs. the number of generations in a sample run W73. . . .	110
5.31	Graphs of entropy, average size, average SC, average EC, and number of functions vs. the number of generations in a sample run W74. . . .	111
5.32	Graphs of minimum, maximum, and average generations of first detection of good wall following behaviour for each chromosome representation method.	113
5.33	Training environment <i>hider-seeker3</i> and testing environment <i>box-light</i> used in light avoidance experiments.	115
5.34	Graphs of entropy and average size vs. the number of generations in a sample run H4.	116
5.35	Graph of average fitness vs. the number of generations in a sample run H5.	117
5.36	Trace run of nice light avoidance behaviour from run H6a.	118
5.37	Graphs of entropy and average size vs. the number of generations in a sample run H23.	119
5.38	Graphs of entropy and average size vs. the number of generations in a sample run H43.	121
5.39	Trace run of nice light avoidance behaviour from run H43a.	122
5.40	Graphs of entropy and average complexity measures vs. the number of generations in a sample run H63.	123
5.41	Trace runs of nice light avoidance behaviour in different environments from run H64a.	124
5.42	Graphs of entropy, average size, average SC, average EC, and number of functions vs. the number of generations in a sample run H85. . . .	125
5.43	Trace run of interesting nice light avoidance behaviour from run H88a.	126

5.44	Trace runs of strange light avoidance behaviour in testing environment (box-lights) and nice behaviour in training environment (hider-seeker3) from run H92a.	126
5.45	Graphs of minimum, maximum, and average generations of first detection of good light avoidance behaviour (LAB) for each chromosome representation method.	128
A.1	Program code of an individual using the linear genome GP method to learn the obstacle avoidance task.	138
A.2	Program code of an individual using the tree-based GP method to learn the obstacle avoidance task.	139
A.3	Program code of an individual using the ADF HGP method to learn the obstacle avoidance task.	139
A.4	Program code of an individual using the ARL HGP method to learn the obstacle avoidance task.	140
A.5	Program code of an individual using the linear genome GP method to learn the wall following task.	140
A.6	Program code of an individual using the tree-based GP method to learn the wall following task.	141
A.7	Program code of an individual using the ADF HGP method to learn the wall following task.	141
A.8	Program code of an individual using the MA HGP method to learn the wall following task.	142
A.9	Program code of an individual using the ARL HGP method to learn the wall following task.	143
A.10	Program code of an individual using the linear genome GP method to learn the light avoidance task.	143
A.11	Program code of an individual using the tree-based GP method to learn the light avoidance task.	144
A.12	Program code of an individual using the ADF HGP method to learn the light avoidance task.	144
A.13	Program code of an individual using the MA HGP method to learn the light avoidance task.	145

A.14 Program code of an individual using the ARL HGP method to learn the light avoidance task.	145
---	-----

Chapter 1

Introduction

The area of evolutionary computation stems from research of biologists on natural evolution of life. Evolutionary theories provide an explanation of nature's ability to adapt to changing environments and of evolution of intelligent behavior in countless species. Those ideas gave birth to evolutionary computation which studies how theories of evolution can be used to solve computational problems.

Various evolutionary computation approaches currently exist with different methodologies and applications. We are interested in the area of genetic programming which uses evolutionary ideas to evolve computer programs. Genetic programming has been successfully used in a wide range of applications, yielding promising results.

Robotics is a relatively new area of research which focuses on building machines called robots. Robots are manufactured to improve the lives of humans; they are designed to perform repetitive or dangerous tasks with excellent precision and dependability. Robots are machines and thus require directions and programming to accomplish their goals.

In our research, we study the application of genetic programming techniques to the evolution of control programs for an autonomous miniature robot. We also present a software simulator for a Khepera miniature robot especially designed to study genetic programming based robotic controllers.

1.1 Motivation

Various robotic simulator software exists for simulation of miniature robots, including the Khepera robot. Free Khepera robot simulators exist but are written for a particular purpose and are difficult to modify to study other tasks. A good Khepera simulator exists for the UNIX[®] platform but it is written in procedural C language and the source code is difficult to modify.

Genetic programming is a powerful evolutionary search algorithm to evolve computer programs. However, it has not yet been demonstrated to perform well on complex real-world problems. The weakness of genetic programming might be due to the method it uses to compute solutions to problems. The standard genetic programming system tries to solve the entire problem at once. While this method is suitable for smaller problems, it is often not powerful enough to solve hard problems.

Hierarchical genetic programming (HGP) tries to overcome the weakness of the genetic programming method by introducing a divide-and-conquer approach to problems. Instead of solving a complex problem entirely, the method breaks the problem into smaller parts and solves each of the parts instead. The solutions of the smaller parts are then integrated into the solution to the overall problem. This modularization approach is based on what humans do to solve complex tasks.

Several hierarchical genetic programming methods have been proposed in the literature, each with its own advantages and disadvantages. In our research we were interested in studying the methods of Automatically Defined Functions [32], Module Acquisition [3], and Adaptive Representation [59].

1.2 Problem

Programming robots by humans can be a difficult endeavor and is not well suitable for complex real-world applications. The area of evolutionary robotics deals with automatic generation of control programs for robots using evolutionary techniques. Genetic programming has been used successfully to evolve subsumption-based robotic controllers.

We are interested in the reactive control of a Khepera robot using genetic programming techniques. In reactive control experiments, robots learn while travelling through the experimental environment. No separate fitness cases are used to calculate fitness and thus the robots do not need to be moved around in their environment for the purpose of fitness calculation. The reactive control problem is difficult since it requires dynamic fitness function evaluation where the individual fitness values depend on the local environment of the robot. However, the problem presents a more realistic dynamic learning environment.

The learning method used in an evolutionary algorithm can greatly influence the successfulness of the solution to the problem. Due to their beneficial properties, we feel that hierarchical genetic programming methods will offer advantages to the problem of reactive robotic control.

1.3 Goals

This thesis has two main goals. The first goal is the creation of a free Khepera Simulator for the Windows[®] operating system especially designed to be used for testing of genetic programming based robotic controllers. The Simulator source code must be well structured so that it can be easily modified to study other robotic controllers. The Khepera Simulator and its source code will be available free-of-charge for academic research purposes.

The second goal is the experimentation with hierarchical genetic programming methods to evolve reactive control programs for the Khepera robot. The purpose of the research is to evaluate the usefulness of the studied HGP controllers in the domain of reactive robotic control. Having analyzed the performance of HGP-based controllers, we hope to identify the advantages and disadvantages of the studied methods in order to improve their performance in the studied domain.

1.4 Contributions

We expect several contributions of our work in this thesis:

1. We will add substantial improvements to the Khepera Simulator for Windows[®]. The Simulator will contain genetic programming based robotic controllers using the methods of: linear genome GP, tree-based GP, Automatically Defined Functions HGP, Module Acquisition HGP, and Adaptive Representation HGP. The source code of the Simulator will be modular and easy to modify. The interface of the Simulator will be user friendly and provide the user with run-time modification of all parameters of the simulation engine and robotic controllers. The improved simulator will be made available free-of-charge for academic research purposes.
2. We will test and record the performance of the linear genome GP, tree-based GP, Automatically Defined Functions HGP, Module Acquisition HGP, and Adaptive Representation HGP methods on the problem of reactive robotic control.
3. Based on our experiments, we will compare and contrast the linear genome GP representation method studied by Nordin and Banzhaf [45] and the regular tree-based GP representation.
4. Based on our experiments, we will compare and contrast the tree-based GP method with the three studied hierarchical genetic programming methods. Our results will enable us to examine the advantages and disadvantages of the HGP methods for the problem of reactive robotic control.
5. For the tree-based GP method, we will compare and contrast the two studied representation methods for Multiple Symbolic Regression: using a dual tree, and using special settable variables. From our results, we will decide which method seems best suited for the problem.
6. Finally, based on our results, we will offer possible improvements to the HGP algorithms to improve their performance in the domain of robotic control.

1.5 Outline

Chapter 2 will provide a discussion of previous research related to our research area. We will introduce evolutionary algorithms through an introduction to genetic algorithms, genetic programming, and hierarchical genetic programming. We will also present the area of evolutionary robotics and discuss previous related research in the area. We will introduce the Khepera miniature robot and provide an overview of available Khepera Simulators.

In Chapter 3, we will introduce our Khepera GP Simulator. We will first discuss the simulation engine and robotic controllers used in the Simulator. Then, we will take a look at the user interface of the Simulator and its source code.

Chapter 4 will discuss the implementation of the robotic controllers used in our research. First, we will provide a generic evolutionary algorithm used by our controllers. Then we will discuss details of the studied GP and HGP methods and their implementations in our research.

Results of our research will be provided in Chapter 5. We will discuss each learning task separately and provide results of the experiments done with different robotic controllers.

In Chapter 6, we will offer conclusions reached by our research. We will also discuss further improvements to our Simulator and suggest improvements to the studied HGP methods.

Chapter 2

Background

In this chapter, we will introduce the ideas of evolutionary computation through a study of genetic algorithms and genetic programming. This will provide us with the knowledge required for detailed study of hierarchical genetic programming methods. We will also explore previous work related to the topic of robotic control using hierarchical genetic programming. Furthermore, the chapter will provide us with a description of the hardware used in our experimentation and available robotic simulation software packages.

We will begin with an introduction to genetic algorithms and shift to genetic programming. Then we will start our discussion of hierarchical genetic programming methods. We will follow with an introduction to robotic control and briefly present various approaches studied. Finally, we will talk about the hardware-based Khepera robot and robotic simulators available to the public.

2.1 Evolutionary Computation

As the name suggest, the area of evolutionary computation is based on the study of applying evolutionary techniques to problem solving. Using natural selection [15] as a blueprint, scientists have tried to create models of evolution that can be applied to solve common problems.

There are three main forms of evolutionary computation [29]: genetic algorithms [25, 22], evolution strategies [57, 5], and evolutionary programming [21, 20]. The

methods differ in the models of evolution that they employ and the types of structures they evolve. In this thesis, we will only concentrate on the genetic algorithm model of evolutionary computation and its derived model of genetic programming.

2.2 Genetic Algorithms

The idea of genetic algorithms was first introduced by Holland in [25]. Genetic algorithms are search algorithms modelled on natural evolution. The purpose of research on genetic algorithms was to create search algorithms that are robust in many different environments [22]. This robustness of genetic algorithms has been theoretically and empirically shown in the literature [25, 22].

Genetic algorithms search using a population of points in the search space. This parallel search technique is more efficient than the single-point search done by most standard optimization and search techniques (eg. calculus-based methods such as hill-climbing). It also improves the chance of finding an optimal (or best satisfied) solution in the search space and decreases the probability of getting stuck at local optima.

Problem domain parameters need to be encoded using strings of finite length over a finite alphabet to be used in a GA system. The usual encoding uses bit-strings where each character in the string comes from the alphabet $\{0,1\}$. The methods of parameter encoding are problem-specific and their discovery is often non-trivial.

The search process used by a pure genetic algorithm is sometimes called a blind search. No auxiliary knowledge about the problem domain or about the encoding is used by the algorithm. The search is guided by payoff information calculated from each string using an objective (fitness) function. By analyzing the values of its fitness function, the system can decide the relative performance of each string in the population. Based on this information, the algorithm can make decisions regarding the reproductive ability of each string.

The genetic algorithm uses probabilistic transition rules, compared to deterministic rules used by most other search techniques. However, the algorithm is not a random search since the random choices made by a genetic algorithm steer the search towards regions with possibly improved fitness.

GA Algorithm

1. Randomly create an initial population of strings
2. Iterate through the following items until termination criteria are met
 - (a) Assign a fitness value to each string in the population using the defined fitness function.
 - (b) Choose strings from the population with a probability based on string fitness.
 - (c) Apply the following genetic operators on the chosen strings to produce a new population of strings.
 - i. Reproduction: copy the chosen string into the new population.
 - ii. Crossover: create two new strings by genetic recombination (or crossover) on two chosen parent strings and add the new strings to the new population
 - iii. Mutation: randomly mutate character(s) of the chosen string based on a given probability and add mutated string to the new population.
3. Designate a result of the run (usually the best-so-far string) which represents a possible solution to the problem.

Figure 2.1: Pseudocode of a typical genetic algorithm.

2.2.1 Algorithm

The genetic algorithm is a simple algorithm working with simple string operations. The main string operations used are string copy and swapping of partial strings. Pseudocode of a typical genetic algorithm is provided in Figure 2.1.

There are four main steps required in order to apply a GA to solve a particular problem. The first step is the choice of a representation scheme used by the algorithm. The second is determining the fitness measure (objective function) that will evaluate the fitness of individuals in the population. The fitness measure is often inherent in the problem but for some tasks it can be non-trivial. The third step deals with the choice of parameters and variables to control the run of the algorithm. Those

parameters include the population size, maximum number of generations, and the frequencies of genetic operators. The fourth step is to determine how to designate the result of the run of the algorithm and the criteria for its termination.

2.2.2 Selection

Part of the driving force behind a genetic algorithm is the selection mechanism used to choose individuals of the GA population that will interact to form the individuals of the new population. The GA uses Fitness Proportionate Selection which chooses individuals with a probability based on their fitness. This method of selection assigns higher probability of selection to higher fitness individuals in the population based on the Darwinian notion of survival of the fittest.

The most common Fitness Proportionate Selection algorithm is the Roulette Wheel algorithm. In this algorithm, each individual of the population is assigned a portion of a roulette wheel proportional to its fitness. Thus, highly fit individuals will each have a larger percentage of the wheel compared to low fit individuals. The roulette wheel is then spun around to choose the individuals that will contribute to the new population. This method guarantees that individuals with high fitness will on average be chosen more frequently than the low fit individuals. However, since the selection is a probabilistic process, low fit individuals can also be chosen for the new population with a low probability.

Roulette Wheel Sampling has desirable average performance, but there can be large deviations between the number of selected individuals and the expected number of selected individuals based on the individual fitness. Stochastic Universal Sampling (SUS) [7] tries to minimize those fluctuations by placing n (where n is the population size) equally spaced pointers on the roulette wheel. This method ensures that approximately, the expected number of copies of each individual will be selected based on the individual's fitness.

The power of a fitness proportionate selection method depends on the fitness calculation. Raw fitness is not desired since it creates large fitness difference early in the run of the algorithm and small, unnoticed differences when the population is close to convergence. A few methods have been proposed to remove this drawback. Scaling methods map the raw fitness to a value that is more evenly distributed in the fitness

landscape. Ranking selection [6] assigns a rank to each individual in the population based on its fitness relative to other individuals. Ranking maintains diversity in the population by hiding the absolute fitness differences between the individuals. It is, however, slow due to the need for sorting of the population to find the individual ranks.

Tournament Selection was introduced to eliminate sorting and remove extra passes through the population for fitness evaluation. This selection method takes a random sample of individuals from the population and chooses few most fit individuals to advance to the new population. This can be thought of as a tournament of fitness between the selected individuals, thus the name of the method. Tournament Selection has been shown to perform as good as other selection methods[10] while lowering the number of fitness evaluations that need to be performed at each step of the GA.

2.2.3 Genetic Operators

Genetic operators modify the individuals of the population to create individuals of the new population in the next generation of a GA run. Similar to natural evolution, genetic operators are an important part of the mechanism of evolution in a GA system since they preserve the genetic makeup of the parents while introducing variability into the genetic makeup of the children. There are three main genetic operators used in classical genetic algorithms: reproduction, sexual recombination (crossover), and mutation.

Reproduction is the act of making identical copies of individuals available to the next generation. In a GA system, this preserves good genetic makeup of high fit individuals to the next generation but because of probabilistic selection, it can also introduce copies of low fit individuals to the new population.

Crossover is the main genetic operator in a genetic algorithm. It produces new individuals for the new population from parent individuals in the old population. In the canonical GA system, crossover is performed on two parents to produce two children who share parts of their genetic makeup with each parent. The operator randomly selects crossover point(s) in the two parent individuals and swaps the corresponding substrings to form two children. Single-point crossover, where only one crossover point is selected, is most widely used and is shown in Figure 2.2. Many

variations of the crossover operator exist and are usually dependent on the encoding structure of the chromosomes. The purpose of the crossover operator is to introduce new individuals into the population by recombining parts of well performing individuals.

Figure 2.2: Crossover operator in genetic algorithms. A random crossover point is chosen and parents exchange corresponding substrings to form children.

The mutation operator is of the lowest importance in genetic algorithms since GA systems have been shown to work without it. It usually consists of a modification to a character of the individual string based on a given probability. The probability of mutation is generally very small compared to the other genetic operators. The mutation operator can also vary depending on the chromosome encoding. This operator introduces variability into the population and is especially useful towards the end of a GA run when the population converges and its variability decreases.

Other genetic operators have been proposed that try to improve on the original three genetic operators. Those operators are usually domain specific and more complex in application. An introduction to genetic operator varieties and analysis of their performance can be found in [17].

2.2.4 Building Blocks Hypothesis

Holland [25] has tried to prove the usefulness of the canonical genetic algorithm by using the idea of a schema. A schema is a similarity template that categorizes strings based on their closeness (similarity). Schemas are made up of the characters in the chromosome string alphabet (usually 0,1) and a general symbol '*' that can be specialized into any symbol of the alphabet. The most general schema consists of all

'*' symbols and the least general schema does not contain any '*' symbols (i.e. the schema is a chromosome string).

Schemas are sometimes expressed as hyperplanes in dimension equal to the string length l . The most general schema is then the entire hypercube of dimension l and the least general is a point of the hypercube (with dimension 1).

For an alphabet size k and string length l , there are k^l valid strings and $(k + 1)^l$ valid schemas. Since any valid string contains l symbols from the alphabet and each symbol can be generalized to a '*' symbol in a schema, each valid alphabet string belongs to 2^l schemas. Thus, the performance of each string in the GA system indirectly affects the performance of each of its schemas.

Holland has shown [25] that the number of instances of an above-average schema in the population increases exponentially during the GA run until the schema instances occupy a substantial proportion of the population. By studying the effects of crossover and mutation on the schema, Holland proposed the Schema Theorem for genetic algorithms.

The Schema Theorem states that schema with above-average fitness, short defining length, and low order are exponentially rewarded by the GA system. Those special schemas with the given properties were called Building Blocks. Holland then hypothesized that the performance of the GA is based on the evolution of those building blocks.

2.3 Genetic Programming

The major drawback of genetic algorithms is the chromosome representation. The canonical GA works with fixed-length chromosome strings. A fixed string representation of a chromosome is restrictive in the decoded output it can generate. For many problems a richer chromosome representation generated using a more extensive alphabet is desired and necessary in order to produce useful results. Methods have been proposed to modify the GA chromosome structure and to introduce variable-length strings [13, 55] but those methods do not considerably change the fixed representation of the GA.

Genetic Programming was introduced by Koza [31] as an extension to genetic

algorithms in order to enrich the chromosome representation. Instead of fixed-length strings, GP evolves pieces of code written over a specified alphabet consisting of a certain set of functions and a set of variables (terminals). The chromosome encoding can be directly executed by the system or can be compiled or interpreted to produce machine executable code.

2.3.1 Chromosome Structure

The canonical GP implementation uses a tree-based chromosome representation [31, 32] as shown in Figure 2.3. The chromosome (originally coded as a LISP S-expression) represents a parse-tree that can be easily transformed into machine code. The internal nodes of the tree are chosen from a set of parameterized functions. The function parameters are subtrees of the specified node. Terminal nodes (nodes with no children) are chosen from the set of parameter-less functions and terminals. The terminal set is usually composed of variables and constants. Variables are place holders in the chromosome that are filled in with values during execution. Functions perform calculations or actions and can optionally have parameters.

Figure 2.3: Structure of a typical GP chromosome. The tree evaluates to LISP S-expression `(+ (- 6 1) (* x (+ 1 7)))`.

An important issue to consider when choosing the function and terminal set is its sufficiency. The contents of the function and terminal set must be balanced between being too small (not enough to solve the problem) and too large (excessive time

needed to solve the problem). Deciding on the sufficiency of terminals and functions for a problem can be a difficult problem in itself and is largely domain-specific.

Certain precautions need to be addressed in an implementation of a tree-based GP system. The maximum tree size/height needs to be specified in order to prevent the system from spending more time on tree execution than on population evolution. In order to support any combination of functions and terminals in tree nodes, care must be taken that each function and terminal return a value that can be used as parameters to other functions in the system. This special property is called closure.

Many different chromosome structures have been proposed. Nordin [44] provided a linear genome GP system which stores 32-bit instructions that can be executed directly on a processor. Nordin claimed the execution speed of the Compiling Genetic Programming System (CGPS) is several orders of magnitude faster than of an equivalent interpreted tree-based GP system [44]. The major disadvantage of the CGPS system is that it is only usable on a processor supporting the specific machine-code instruction set used. To be used on a processor with a different instruction set, the system needs to be either rewritten or interpreted. The CGPS was later called the Automatic Induction of Machine Code by Genetic Programming (AIMGP) system [50].

Poli [53] has developed a Parallel Distributed Genetic Programming (PDGP) system where the resulting chromosome structures are graphs. This graph-based GP approach is more general to the tree-based approach and can be used on a larger class of problems. Other approaches [26, 27] use hybrid chromosome representations and provide promising preliminary results.

2.3.2 Fitness

Fitness may be considered the driving force behind evolution; thus, fitness evaluation is an integral part of genetic programming. Fitness calculation can be expressed as either implicit or explicit. Implicit fitness measure (such as survival) is more commonly used in artificial life research[56]. Genetic programming mostly uses explicit fitness that can be measured from each individual in the population at any time.

The simplest form of explicit fitness measure is raw fitness. Raw fitness is a problem-specific fitness measure usually easily computed from the population. The

most common raw fitness evaluation is an error value between the observed results and correct results for each fitness case summed over all fitness cases.

Since raw fitness is problem-specific, the fitness values for multiple problems cannot be reasonably compared. Standardized fitness measure solves this problem by adjusting the raw fitness so that the lowest numerical value has the best fitness. Zero often denotes the best fitness value. Similar to genetic algorithms, scaling and ranking methods are also used to ensure the selection of an individual is proportional to its fitness and relative to the fitness of other individuals in the population.

2.3.3 Genetic Operators

Genetic programming uses genetic operators similar to genetic algorithms. The two main genetic operators used are reproduction and crossover.

The reproduction operator simply copies an individual into the next generation. The crossover (sexual recombination) operator produces new offspring for the future generation from parts of parents in the current generation (shown in Figure 2.4). The main difference between crossover in GP and in GAs is that the GP crossover is more destructive. A crossover point in GA chromosomes splits the parent chromosomes into corresponding sections. Because of the fixed representation of many genetic algorithms, the same parts of each parent will contribute to the same characteristics of the phenotype. This is not true in GP since the same crossover point usually has distinct meaning in each parent. Thus, crossover in GP can be thought of as a crossover with mutation.

It is because of the different role of crossover in GP, the mutation operator is seldom used. The crossover operator applied to two leaves of the parents acts like a point mutation in the offspring. Koza regards mutation in genetic programming as an unimportant secondary operator [31]. However, newer experimental evidence suggests that for some experimental settings, the performance difference between GP systems with crossover and those with mutation is small [2, 36].

Researchers have suggested other secondary genetic operations for genetic programming. Permutation, editing, encapsulation, and decimation are explained in [31].

Figure 2.4: Crossover operator in genetic programming. Subtrees A and B are swapped between the two parents to form two children.

2.3.4 Building Blocks Hypothesis

Schema-based analysis of genetic programming is more difficult than of genetic algorithms because of the variable chromosome structure employed in GP. O'Reilly and Oppacher [52] modified the GA schema theorem and proposed the GP Schema Theorem (GPST). They have also proposed a new GP Building Block Hypothesis (BBH) based on GPST and GP schemas. The research led to more questions whether the BBH is in fact viable in GP under the proposed schema theorem.

The Building Block Hypothesis for genetic programming [52] describes short and highly fit code fragments called building blocks and the role they play in a run of the genetic programming system. The hypothesis states that genetic programming combines those building blocks to create, over some generations, individuals with high fitness.

Recently, Poli [54] proposed a new general and exact schema theory for GP which provides an exact formulation for the expected number of instances of a schema at

the next generation. A special case of the theory is an exact schema theorem for standard crossover.

2.4 Hierarchical Genetic Programming

The main problem with genetic programming lies in its scalability. Genetic programming has been demonstrated to solve a variety of applications [32, 33] but it loses its effectiveness for more complex real-world problems [8].

GP solves problems differently than a human. When humans solve complex problems, we typically break the task into simpler sub-tasks and solve each sub-task. In contrast, regular GP tries to compute the entire solution to the problem at once. This decomposition technique of breaking down the task and solving its sub-tasks (called modularization) seems to be the right solution to overcome the complexity threshold of real-world problems.

Modularization techniques have been developed for GP but have generally employed a fixed decomposition structure provided by the experimenter. Hierarchical Genetic Programming introduces modularization techniques to the GP system so that the GP can evolve module solutions to problems without human-imposed structure. This automatic modularization technique should improve the performance of genetic programming on difficult problems.

Koza [32] identifies five techniques that can enable hierarchical problem solving to reduce the effort needed to solve a problem: hierarchical decomposition, recursive application, identical reuse, parameterized reuse, and abstraction. Hierarchical decomposition is the act of breaking a problem into smaller sub-problems, solving the sub-problems, and combining their solutions into a solution for the problem. Recursive application of hierarchical decomposition to a problem would be able to recursively break the problem down into small sub-problems that would be easy to solve by the system. Identical reuse is the process of using previously computed solutions to identical sub-problems, while parameterized reuse offers a way of applying the same problem solving mechanism to similar sub-problems via parameters. Abstraction deals with exclusion of irrelevant data from the problem environment.

Several Hierarchical Genetic Programming methods have been suggested, each

with its own advantages and disadvantages. The methods have been tested on various problems; however, current research does not adequately explain whether the studied HGP methods can, in general, outperform standard GP. The next subsections will provide an introduction into four HGP methods: Automatically Defined Functions (ADFs) [32], Module Acquisition (MA) [3], Adaptive Representation (AR) [59], and Local Modules (hGP) [8].

The goal of Hierarchical Genetic Programming is to help the GP system to find and capture building blocks (as defined in the BBH [52]) and to use them to compute better solutions to problems. Each HGP method presented here differs in the specification of the building blocks and the methods in which the building blocks are found.

2.4.1 Automatically Defined Functions

Koza's Automatically Defined Functions (ADFs) [32] method is the oldest and most widely used HGP method. The method automatically evolves function definitions while evolving the main GP program that is capable of calling the evolved functions.

An ADF chromosome consists of two distinct parts: the function defining branch, and the result producing branch. The function defining part is composed of one or more ADF definition branches which describe the structure of each ADF. The result producing branch contains the code of the resulting program. This code can call any function defined in the function defining branch of the same chromosome. More details on the ADF method and its implementation in the Simulator are provided in Chapter 4.

Automatically defined functions have been demonstrated to be advantageous in solving more complex versions of problems than possible by standard GP (e.g. 6-parity problem) [32]. They have also been shown to work well in many domains [32, 33]. The major disadvantage to ADFs is that the user must specify the structure of the ADF chromosomes (number of functions and arguments) and the function and terminal sets required by each function. In a true automatic hierarchical genetic programming system, this type of information should be evolved by the GP rather than provided by the user. Taking the downside of ADFs into consideration, current research is centered around operations that automatically modify the structure of the

ADF chromosome and the number of ADFs [33].

2.4.2 Module Acquisition

Introduced by Angeline and Pollack, Module Acquisition [3] uses standard tree-based chromosome structure and two new genetic operators to create and manage a collection of subroutines. The compression operator creates subroutines from subtrees of the chromosome program tree. Since the compression operator lowers the diversity of the population by removing subtrees, an expansion operator is also provided to counteract the negative effects. The expansion operator reverses the process of the compression operator by substituting original subtrees for subroutine calls in the chromosome tree.

Subroutines are called from within the bodies of the chromosome trees. The subroutines contained in the subroutine collection are frozen in time and cannot be modified during evolution of the program trees. Details about the use and implementation of the Module Acquisition method are provided in Chapter 4.

The Module Acquisition method automatically generates a hierarchical module structure [1]; however, no clear advantages of the method have yet been provided. Kinnear has compared MA to ADFs on the even-4-parity problem [30] and concluded that the method does not offer improvement in space or time over the ADF method.

2.4.3 Adaptive Representation

Rosca and Ballard proposed the Adaptive Representation method to dynamically extend the function set with identified building blocks [59]. The method uses standard tree-based representation and searches for blocks of code (defined as subtrees of a given maximum height). Blocks are parameterized into functions by substituting each occurrence of a terminal by a variable.

The method works by incrementally checking the population for fit building blocks. Block fitness is dependant on the performance of the individual where the block resides (and, thus, the block) or the performance of a part of the individual (e.g. using a block fitness function). Evolution is done in epochs which are defined as sequences of consecutive generations where no fit building blocks are discovered. At the end of

each epoch (i.e. after a discovery of a candidate building block) a user specified proportion of the population (constituting the lowest performing individuals) is replaced by individuals that are randomly generated from the new extended function set.

Rosca and Ballard provide theoretical discussion on the usefulness of their approach in improving the speed of evolution over standard GP [59]. It is unclear, however, how to discover candidate building blocks from the blocks encountered during evolution without additional domain knowledge. The method also suffers from the inability to delete low performing subroutines and thus overgrowing of the function set.

To alleviate the problems associated with AR, Rosca and Ballard provide an improved algorithm called Adaptive Representation through Learning (ARL) [60]. The modified algorithm uses the notions of differential fitness and block activation to decide on the usefulness of building blocks. Differential fitness measures the improvement of a chromosome over its least fit parent. Block activation is a counter of the number of times a specific block was executed by the system. Blocks with large differential fitness, high activation, and small height are selected as candidate blocks and added to the subroutine collection.

2.4.4 Local Modules

Banzhaf et al. [8] introduce the idea of local and context-sensitive modules in their hGP algorithm. The method uses hierarchical levels of populations of individuals evolving at different speeds. The top-level population is the main individual population producing results. Subsequent levels consist of modules that are called by individuals on higher levels.

Each level in the Local Modules approach evolves using a typical GP system. Crossover is only allowed between individuals at the same hierarchical level. The speed of evolution at each level is inversely proportional to the depth of the level (i.e. the top-level is subjected to the quickest evolution). Modules are identified at the end of a generation using the best individuals of the population. The modules are then inserted into the population of the lower level. Module selection criteria is based on differential fitness. Fitness of a module is equal to the fitness of the individual in a higher level that calls the module.

The Local Modules method leaves some unanswered questions. The impact of hierarchical levels of evolution on the performance of the method is unclear and provided as an open issue [8]. The method was compared to standard GP using function regression and even-N-parity problems and was found to outperform the GP. It is unclear how the method would compare to other hierarchical genetic programming methods under the same circumstances.

2.5 Robot Control

The field of Robotics has been a cradle for testing of novel ideas and frameworks of learning. Robots, being mostly autonomous and acting in the real environment, offer an authentic test-bed for learning technologies. Compared to the theoretical approach to learning, robotics offers a practical approach where the limits of current technology can be tested. Research with robots is thus becoming increasingly popular and sought after.

Robotic control deals with the programming of robots to perform actions in the environment. These control programs decide on actions and behaviours that should be taken by the robot based on the robot's view of the environment. Many different approaches to robot control have been presented in the literature.

The area of robot control is often subdivided into three sub-areas: reactive, behavior-based, and hybrid [4]. Reactive control uses a simple set of condition-action pairs that define how the robot reacts to a stimulus. Brooks [11] proposed a multi-layer subsumption architecture where higher-level layers can subsume and block lower-level layers from action. Behavior-based architecture [38] uses a collection of interacting behaviours that can take input from the robot's environment sensors or other behaviours and produce output to the robot's effectors or other behaviours. More information on behavior-based robotics can be found in [4]. Hybrid control strategies exist that offer a compromise between purely-reactive and behavior-based strategies.

2.5.1 Evolutionary Robotics

Most applications of the robotic control strategies discussed above deal with control programs that are written by humans prior to experimentation. This method is adequate for some experiments but is not realistic for complex real-world applications [23]. A need exists for the control programs to be generated automatically and/or to be modified automatically while performing a task. Simulated evolution offers a way to automatically generate control programs and is the basis of a new field of research called Evolutionary Robotics.

Before the advent of genetic programming, the most commonly used technique in Evolutionary Robotics was to evolve neural networks using genetic algorithms. Research in the field continues and a sample of neuro-controller research can be found in [14, 18, 19, 39, 24, 41, 43, 65].

In this thesis, we will concentrate on using genetic programming to create control programs for robots. Nolfi and Floreano [42] offer an extensive survey of the field of Evolutionary Robotics.

2.5.2 Genetic Programming

Brooks [12] introduced the idea of using Artificial Life techniques to evolve control programs for mobile robots. Although no experimental results were presented, Brooks identified genetic programming as a hopeful technique for control program evolution. Koza [34] presented results of using GP to evolve emergent wall following behavior for an autonomous mobile robot. The control program was based on the subsumption architecture and demonstrated that GP can be used to evolve control programs for mobile robots. In [35], Koza and Rice demonstrated that genetic programming can be used to automatically create a control program to perform a box moving task. The paper also offered a good comparison between GP techniques and reinforcement learning techniques in accomplishing the task.

Reynolds [58] has used genetic programming to evolve a controller program for tiny critters in a simulated environment. The critter tasks were to manoeuvre in a static obstacle environment (obstacle-avoidance) and avoid a predator. In this ALife-inspired predator-pray paradigm, the fitness criteria was based on the sum of the

critter lifetimes. Results showed interesting partial solutions to the task but failed to show herding behavior such as observed in animals.

Nordin and Banzhaf [45, 48, 46, 49, 47] have experimented with a simulated and real Khepera miniature robot to evolve control programs using genetic programming. They used the Compiling Genetic Programming System (CGPS) [44] which worked with a variable length linear genome composed of machine code instructions. The system evolved machine code that was directly used on the robot without the need of an interpreter.

The initial experiments of Nordin and Banzhaf [45, 46] have used a memory-less genetic programming system with reactive control of the robot. The system used a type of function regression to evolve a control program that would provide the robot with 2 motor (actuator) values from an input of 8 (or more) sensor values. GP successfully evolved control programs for simple control tasks such as: obstacle avoidance, wall following, and light-seeking. The work was extended [49, 50] to include memory of previous actions and a two-fold system architecture composed of a planning process and a learning process. Speed improvements over the memory-less system were observed in the memory-based system and the robots exhibited more complex behaviours [49]. Summary of the techniques used and tasks studied can be found in [9].

Olmer et al. [51] have researched a control task hierarchy where more complex tasks were solved by a combination of easier sub-tasks. The research used a separate population for each action primitive and a control population that provided the robot with a choice of an action primitive for each step of the experiment. Results indicated that this kind of hierarchical control architecture can be used to evolve robotic control programs but more research in the field was required.

Ebner [16] experimented with applying genetic programming to evolve a control architecture for a large physical mobile robot. The main experiment on the physical robot was only done once because of the lack of time. Results looked promising but more experimentation was required. No proof was given as to the alleged hierarchical nature of the evolved control architecture.

Martin [37] applied genetic programming to create a reactive obstacle avoidance system for an autonomous mobile robot using a camera. The GP algorithm was

only used to decide on the position of ground in the camera images. A hand-written control program was used on the output of the GP in order to operate the robot.

2.6 Khepera Robot

The Khepera robot is a miniature mobile robot created and sold by K-Team S.A.¹ - a Swiss company specializing in development and manufacture of mobile mini-robots. Recently, K-Team has created a new Khepera II robot with an improved micro-processor and memory and a wider range of capabilities. Our research is based on the original Khepera robot.

Khepera is circular, with a diameter of 55mm and height of 30mm. The robot can sense its environment with 8 built-in infra-red proximity and ambient light sensors. Two motors with controllable acceleration are used to move the robot in the environment. Figure 2.5 shows a picture of the Khepera robots and Figure 2.6 provides a schematic diagram of the robot's sensors and motors.

Figure 2.5: Khepera robot (left) and Khepera II robot (right). Images courtesy of K-Team S.A.

The brain of the Khepera robot is a 16Mhz Motorola 68331 micro-processor with 256 KB of RAM and 128-256 KB of reprogrammable ROM memory. The ROM

¹K-Team S.A. can be contacted at the company website: <http://www.k-team.com>.

Figure 2.6: Schematic view of the Khepera robot. Sensors are labelled s0 to s7 and motors are labelled m1 and m2.

contains a simple operating system and communication interface to a host computer. The robot can execute its own programming that can be either provided through a serial connection or downloaded into the robot's memory.

The Khepera robot can be equipped with a variety of extension turrets that provide it with abilities to perform more complex tasks. Some extension turrets are: gripper turret used for object recognition and manipulation, video turret for on-board camera ability, and I/O turrets for improved communication with the host computer and other turrets.

For financial reasons, we were unable to acquire a Khepera robot for our experimentation. However, we have used a software simulation of the original Khepera robot and its environment. The simulation included a realistic and tested environment simulation engine and a genetic programming engine that, with few modifications, can be ported to a real Khepera robot.

2.7 Robotic Simulators

Robotic simulators play an important role in robotic experimentation. Robotic equipment can be costly and require proper experimentation facilities. Software simulators

offer the experimenter a test-bed for robotic technologies when a physical robot cannot be acquired. Some simulators provide a very accurate model of the environment and the robot's interactions in the environment. Such simulators can be used as valid substitutions for real robots for testing various robotic tasks.

Some robotic research on physical robots requires painstaking experimentation where the robots need to be supervised and may need to be rearranged in the environment during the experiment. That is one area where a robotic simulator has an advantage to the physical robot. The simulator can be left unsupervised and can be programmed to automatically perform human actions such as relocation of the robot in its environment. This can considerably speed up experimentation time and requires less human time.

The main disadvantage of software simulators is the inexact model of the environment. A real physical environment contains noisy data that can greatly influence the results of an experiment. A typical software simulator lacks this environmental noise or provides noisy routines that do not exactly mirror the physical environment. One of the goals of using a robotic software simulator is to be able to reproduce similar results on the physical robot. Thus, the software environment must contain noise comparable to the real physical environment so that the software-based experimentation can be comparable to experimentation on the real robot.

2.7.1 Khepera Simulators

The original Khepera Simulator² was developed by Olivier Michel at the Microprocessor Systems Lab (LAMI) of the Swiss Federal Institute of Technology (EPFL). The latest version of the simulator (version 2.0) is available free-of-charge for research use and it is written exclusively for the UNIX[®] platform.

Many other software simulators for the Khepera robots are currently available³. Olivier Michel has founded a company called Cyberbotics⁴ that specializes in development of 3D simulation software for mobile robots. The software - Webots [40] - supports a variety of robots rendered in a 3-dimensional environment.

²The Khepera Simulator is available at <http://diwww.epfl.ch/lami/team/michel/khep-sim>.

³A list of Khepera simulators can be found at the K-Team S.A. website.

⁴Cyberbotics can be contacted at the company website: <http://www.cyberbotics.com>.

2.7.2 Khepera GP Simulator

Khepera Genetic Programming Simulator was created by Marcin L. Pilat in 2001 as a port of the original Khepera Simulator to Windows[®]. In 2003, the Simulator was improved and adapted for simulating GP-based tasks on Khepera robots. Khepera GP Simulator and its source code is free for academic use. The current version of the Khepera GP Simulator is version 3.0 and can be downloaded from the author's website⁵. More information about the GP Khepera Simulator can be found in Chapter 3 of this thesis.

2.8 Summary

This chapter provided an introduction to evolutionary computation. We discussed genetic algorithms to set a stage for an introduction to genetic programming. We identified problems with the original genetic programming approach and listed four recent approaches to hierarchical genetic programming. Through a condensed survey of genetic programming techniques used in evolutionary robotics, we have introduced previous approaches similar to our research. Finally, we talked about the Khepera robot and introduced the idea of robotic simulators.

Chapter 3 will detail the GP Khepera Simulator that we have used in our research. In Chapter 4 we will describe how we have used the mentioned hierarchical genetic programming approaches to evolve control programs for the Khepera robot. Results of our research will follow in Chapter 5.

⁵The home page for GP Khepera Simulator for Windows[®] is available at <http://www.pilat.org/khepgpsim>.

Chapter 3

Khepera GP Simulator

The Khepera GP Simulator for Windows[®] is a software package to simulate Khepera robots in their environment. The software was designed to use the genetic programming paradigm to automatically generate control programs for the robots. Thus, the simulator can be used for testing of GP techniques in the domain of robotic control.

The simulation engine of the Khepera GP Simulator is based on the original Khepera Simulator for UNIX[®] by Olivier Michel¹. Over the development time of the GP Simulator, the engine has been modified to improve the structure of the simulator and to add extra functionality.

Version 2.0 of the Khepera GP Simulator was created in 2002. We have made many improvements to the previous version of the simulator in order to study a wider variety of robotic controllers and learning tasks. Version 3.0 of the simulator contains full implementations of five GP-based robotic controllers compared to only one controller in the previous version. We have added a statistics engine to log vital run statistics and a robot path tracer to capture the path of the robot through the environment. We have made numerous user interface modifications and additions in order to improve the ease-of-use of the software. We have also made code modifications to improve the modularity of the code and ease the addition of new functionality.

Khepera GP Simulator for Windows[®], version 3.0, is available free for educational purposes and can be downloaded from the author's website². The source code is also

¹The Khepera Simulator is available at <http://diwww.epfl.ch/lami/team/michel/khep-sim>.

²The home page for Khepera GP Simulator is available at <http://www.pilat.org/khepgpsim>.

available and can be modified by researchers for specific experiments. The code was written using Microsoft[®] Visual C++[®], Microsoft[®] Foundation Class (MFC) Library, and Component Object Model (COM). The Simulator is only available for the Windows[®] platform.

In this chapter, we will take a detailed look at the Khepera GP Simulator for Windows[®]. First, we will discuss the simulator engine that simulates a robot in its environment. Second, we will talk about the robotic controllers used to control the robots in order to learn a task. Third, we will take a look at the user interface of the Simulator and finally, we will study the internal code structure of the Simulator.

3.1 World Simulation

The main purpose of the Khepera GP Simulator is to simulate a physical Khepera robot in its environment. The simulation includes sensing of the environment using the robotic sensors and interacting with the environment using the robotic actuators (motors powering the wheels).

The simulation engine uses a simple physics model to broaden the interactivity of the robot with its environment. Multiple Khepera robots can be simulated together thus allowing the study of more complex robotic behaviours requiring interaction between the robots (e.g. box-pushing, following, collective garbage collection).

3.1.1 World Model

The environment of the robot is modelled in a rectangular working area. The dimensions of this working area can be specified by the user. The outside of the working area is considered to be solid and should not be used by the robot. The entire environment can be scaled to the desire of the user.

All items in the environment are treated as objects. There are three types of objects - building objects, light objects, and robot objects. Robot objects are simulated Khepera robots placed in the environment and are discussed in more detail in the following section.

Brick and cork objects are considered building objects. Rectangular bricks can be

used to form the walls of the environment and can be rotated around their centers. Circular cork objects can be used as small obstacles for the robots that are difficult to detect by the robot sensors.

The light objects are composed of lamps and light boxes. Lamps are not physical objects in the environment and can be thought of as overhead lamps that cannot be touched by the robot. Light boxes are large square bricks of low weight with adjustable rotation. The light objects can be lit so that they are detected by the light sensors in the robots.

The most important feature of the world model in the Simulator is its dynamics. Any object can be manipulated in the environment in real-time during a simulation run. The simulation acts directly on the world objects in the environment eliminating the need for an internal digitized map of the world (as in the original Khepera Simulator for UNIX[®]).

3.1.2 Robot Model

The physical Khepera robot is a miniature mobile robot as described in Section 2.6. The robot uses 8 infra-red proximity sensors to detect distances to objects in its environment. It also uses 8 ambient light sensors to detect the amount of light present in the environment. Figure 2.6 shows the location of the sensors on the Khepera robot.

Sensors of the simulated Khepera robot were based on the sensor model used by the original Khepera simulator for UNIX[®]. Documentation for the original Khepera simulator stated that its simulation is a reasonable approximation of the real world verified by experiments against a physical Khepera robot. Care was taken to ensure that the sensor simulation in the Khepera GP Simulator was comparable to the sensor simulation used by the original simulator. The simulation was not tested against a physical Khepera robot due to inability to acquire the robot.

Each of the proximity sensors was simulated as a sampling of 15 points within a conical area with the sensor as origin and in the direction of the sensor angle. Each such sampling point was assigned a distance value corresponding to its location within the conical area. The distance value of a sampling point was used if the sampling point was contained in an object in the environment; otherwise, a distance value of

0 was assigned. The sum of all 15 sampling point distance values was given as the proximity (distance) value for that sensor in the range $[0, 1024)$. The proximity value was proportional to the closeness to an object in the environment with 1023 denoting maximum closeness.

Calculation of each ambient light sensor was based on the amount of light surrounding the sensor. Each ambient light sensor had a 60 degree field of vision centered at the sensor and in the direction of the sensor angle. If a lit object was found within the field of vision of the sensor, a light value was calculated as a ratio of the distance to the light source, angle to the light source, and a constant maximum value. The light value of each lit object in the field of vision of the sensor was subtracted from 500 resulting in the ambient light sensor value in the range $[50, 500]$. The light sensor value was inversely proportional to the amount of light in the environment with 500 denoting maximum darkness.

Noise is an important factor in simulations. Sensors on the physical Khepera robot are noisy and fluctuate due to unforeseen environmental factors. This was imitated in the Simulator by adding a small random value (up to $\pm 10\%$ for proximity sensors and up to $\pm 5\%$ for ambient light sensors) to the final sensor values. The used noise values corresponded to the values used by the original Khepera Simulator by Olivier Michel.

The Khepera robot uses two motors to drive two wheels situated on the robot. Simulated motor speeds were in the range $[-10, 10]$ with the sign denoting the direction of the spin (with negative as backwards). Noise was added to the motor values while calculating the effect of the motors on the environment (up to $\pm 10\%$ for speed amplitude and up to $\pm 5\%$ for motion direction).

The position of the robot in its environment and its direction angle can be modified by the user. Each robot can have a unique name so that it can be easily distinguished among all other robots in the environment. With multiple robots in the environment, many distinct tasks can be studied simultaneously. Each robot can be set to work with a specific task. Multiple robots can interact with each other in the environment and each robot is treated as a world object by other robots.

3.1.3 World Interaction

All the objects in the world model can interact with each other. Sensing abilities of the simulated Khepera robot were discussed in the previous section. Through the robot sensors, all objects in the environment indirectly interact with the robot. Most important interactions take place between the robot and the objects in the world and are initiated by the modification of the robotic actuators.

The robotic controller provides a set of motor values to be used by a robot during each step of the simulation. Given motor values are processed to yield a force vector. The force vector specifies the direction of the motion and the amount of force the robot applies in the world. The force vector is then used to calculate the next position and rotation of the robot. If there are no objects blocking the movement, the robot is moved to its new position and its rotation may be changed.

If the movement of the robot places the robot in collision with other objects, special handling of the collision is required. In the static case, where the robot interacts with an object that cannot be moved in the environment (such as the brick object), the robot is not allowed to complete its movement and the collision flag is set. In the dynamic case, where the object is a moveable object (such as the light box object), a simple physics model is applied to calculate the resulting force vectors of the objects involved. Those force vectors are then used recursively on affected objects until all affected objects are modified.

The simple physics model uses simple vector-based collision equations to calculate resulting force vectors from incoming force vectors and object weights. Properties of the physics model can be modified by modifying the weights of objects in the world.

As an example of the physics model, we consider the result of a collision between a robot and a light box object. Depending on the angle of collision, the rotation of the light box, the magnitude of the force, and the weight of the box, the box will either: stay solid (resisting the robot), rotate around its axis, translate its position (2D movement), or perform a mixture of translation and rotation. All objects connected to the light box will be considered in the calculations and acted upon if necessary.

The physics model in the Khepera GP Simulator handles only simple interactions between the involved objects. A more detailed physics model would be required in order to create the complex simulations that happen in the real world. The physics

model should also be tested against the real world to verify that it accurately portrays the object interactions of the world.

3.2 Evolutionary Controller

A robotic simulator has little purpose without a robotic controller. The Khepera GP Simulator was specifically designed to study GP-based robotic controllers but can be easily used with any type of a robotic controller. The controller dictates the actions of the robot in the environment. The GP controllers included with the Simulator modify a population of robotic control programs in order to evolve certain tasks (or behaviours).

The five GP controllers included with the Simulator use different GP methodologies to encode the control programs. The *GP Linear* controller uses a linear genome to encode the control programs in the population while the *GP Tree* controller uses tree-based chromosome representation. The *HGP ADF* controller is a hierarchical genetic programming controller using the method of Automatically Defined Functions (ADFs). The *HGP Module Acquisition* controller is based on the method of Module Acquisition (MA) and the *HGP Adaptive Representation* controller uses the method of Adaptive Representation through Learning (ARL). Details about each of the GP controllers types are provided in Chapter 4.

3.2.1 Tasks

Each learning task (such as obstacle avoidance, wall following) is represented in the Simulator as a Task entity. A task is of a fixed GP controller type chosen from the five controller types included with the Simulator. The controller of a particular task cannot be modified once the task is created. Many tasks can be created at one time and can be stored in a Task Library. The Task Library and each separate task can be saved to file and loaded at a later time.

The GP controller type of a task specifies the chromosome structure and how the chromosomes in the population interact during evolution. Multiple tasks can use the same GP controller type with different specifications of the chromosome structure.

Chromosome properties depend on the GP controller type and contain information on the building blocks of each chromosome (terminal and function sets), chromosome height restrictions, and creation methods.

Parameters of the GP controller can be modified through the task entity. Crossover and mutation rates, tournament size for tournament selection, evolution toggle, and population size are examples of GP controller parameters. A task contains a population of chromosomes; thus, it can be used to store snapshots of the population during evolution.

Each task also contains a fitness function which provides guidelines for the evolution of the population of control programs. The fitness function can be thought as a formal definition of the learning task.

3.2.2 Fitness Function Scripting

Fitness functions in the Simulator are dynamic and can be easily modified at run-time. The fitness function definitions must be written using a scripting language - Microsoft[®] JScript[™]. This scripting language is based on Java[™] and is available free-of-charge from Microsoft[®] Corporation.

JScript[™] provides the user with a rich scripting language to define the fitness function. The language supports a variety of pre-defined functions and the ability to create variables. A *robot* object is made available to the scripting engine to access the sensor and motor values of the robot being studied. Detailed information on fitness function scripting is provided with the Simulator.

3.2.3 Run Statistics

The GP controllers in the Simulator gather statistical information during the run of the evolutionary algorithm. Statistical information for each generation is saved and can be stored into a comma-delimited text data file. This file can then be viewed by any text editor or spreadsheet software.

Statistical information is used in order to analyze the performance of an evolutionary run. For each generation, average and best population fitness values are stored as statistics. The number of bumps (collisions of the robot with other objects) is also

recorded.

Complexity of the chromosomes in the population are stored using three complexity measures. For each measure, the average value and the value of the best-of-generation individual is stored. The size measure specifies the raw size of the chromosomes. The raw size $Size(F)$ is defined as the number of instructions in a linear genome chromosome or the number of tree nodes in the tree-based chromosome representation (where F is the list of instructions or program tree representing the individual).

For hierarchical GP methods, the complexity is also measured using *structural complexity* $SC(F)$ and *evaluational complexity* $EC(F)$ [63]. The structural complexity measure includes the sizes of all unique function (module) trees called from an individual. Evaluational complexity of an individual is measured recursively and includes sizes of all function (module) trees embedded in the individual. This measure approximates the number of computational units required for execution of the individual program. For a program tree F_0 containing calls to program trees $F_1, F_2, \dots, F_m$, the measures can be formally expressed as:

$$SC(F_0) = \sum_{0 \leq j \leq m} Size(F_j) \quad (3.1)$$

$$EC(F_0) = Size(F_0) + \sum_{0 < j \leq m} |Calls(F_0, F_j)| \cdot EC(F_j) \quad (3.2)$$

where $|Calls(F_0, F_j)|$ is the number of calls to F_j from F_0 .

Structural complexity and evaluational complexity of non-hierarchical individuals (linear genome or simple tree-based) is equal to the raw size measure. With the use of hierarchical measures and functions (modules), the two measures could be distinct and provide a more realistic complexity measure than raw individual program tree size.

The last statistic measured and stored during each generation is the value of *population entropy* [61]. Population entropy measures the state of a dynamic system represented by the population. Detailed description of population entropy is provided in Section 4.6.3.

3.3 User Interface

The Graphical User Interface (GUI) of the Khepera GP Simulator, as shown in Figure 3.1, is composed of the *World Editor*, the *Info Window*, and the *Simulator Interface*. The *World Editor* is used to modify the simulation environment to study a specific learning task. The *Simulator Interface* enables the user to create learning tasks, choose learning methods, and run simulations of robotic controllers while the *Info Window* provides information about a simulation run.

Figure 3.1: Screenshot of the Khepera GP Simulator for Windows[®].

3.3.1 World Editor

The *World Editor* has a dual role of an editor for the world environment and a viewer of the simulation run. The editor provides the user with the ability to modify the world environment by manipulating objects contained in the world. During a run of the simulation, the editor dynamically shows the location of the robot in the world environment and the effect of the robot on other objects in the world. Menus and toolbars are used to access the functionality of the *World Editor* and the Simulator. The toolbar is shown in Figure 3.2.

Figure 3.2: Simulator Toolbar with location of various tool and button groups.

The Khepera GP Simulator was built using a multiple document framework that allows the user to work with multiple testing environments (world documents). Each world document can contain any variety of objects including robot objects. World documents can be serialized (saved and loaded) through world files. An import option is also provided to load objects from world file formats used in previous version of the Khepera GP Simulator and in the original Khepera Simulator for UNIX[®]. All options dealing with the world documents and files are found in the *File* menu and the *Document* group of the toolbar (see Figure 3.2). Properties of the world environment, such as dimensions, offset, and scale, can be modified through the World Properties dialog under the view menu.

Object manipulation in the *World Editor* starts with object creation. The *Tools* menu and *Drawing Tools* group of the toolbar (see Figure 3.2) contain five drawing tools - one for each of the world objects: brick, cork, lamp, light box, and robot. Once a drawing tool is selected, an object of the selected type can be drawn on the world working area at a click of the left mouse button.

The *Select Tool* allows the user to manipulate objects in the Editor. The tool can be found under the *Tools* menu or on the toolbar. Multiple selection of objects is allowed by multiple clicking on objects or by drawing a selection rectangle. Selected objects can be moved around the editor environment or deleted from the environment

by using the *Remove Object Tool*. Selected objects can be cut, copied, or pasted from the clipboard by using options under the *Edit* menu or the *Editing* section of the toolbar.

Each object type has properties that can be modified through the Editor. Properties dialogs of each object, as seen in Figure 3.3, can be shown by selecting *Properties* from the context menu that appears by right-clicking on an object. Object properties include the position in the world environment, rotation, and object-specific properties such as light toggle for light objects and global rotation setting for brick objects.

Figure 3.3: Properties Dialogs for a Light Box and Robot object.

The Robot Properties dialog includes a *Reset* button that resets the position and direction of the robot to a fixed default value. The dialog also allows the user to choose the specific task to be learned by the robot. The task specifies the type of GP controller that is used to move the robot in the environment.

3.3.2 Info Window

The *Info Window* is situated on the left side of the *World Editor* and can be hidden using a toggle under the *View* menu. This window displays information about a selected robot and the simulation run. The *Message Window* can be used to display simulation messages to the user. Currently, the window displays fitness values of evaluated individuals in the population and collision warnings. The window update

can be turned off using a toggle. A sample of simulation statistics is provided under the *Message Window* and includes last generation count, average fitness value of last generation, and the entropy value of the population.

The *Robot View* section of the *Info Window* includes a graphical and text-based interface showing the robotic sensor and motor values of a selected robot. The information is provided on a schematic drawing of the Khepera robot showing relative positions of the sensors on the robotic body.

For each sensor, the upper value represents proximity sensor value and the lower represents ambient light value. Proximity sensors values are also shown in a blue gradient ranging from white (sensor value 0) to dark blue (sensor value 1023). Ambient light sensor values are represented using a red gradient ranging from white (sensor value 500 or over) to dark red (sensor value 50 or lower). Motor values are presented near the center of the robot with corresponding direction indicators. A square patch of color near the center of the robot diagram represents a collision sensor and changes its color from green to red during a collision.

3.3.3 Simulator Interface

The most important part of the Simulator is the simulation of the Khepera robot through its environment using a robotic controller. The simulation GUI allows the user to modify properties of the simulation through Tasks and control the running state of the simulation.

Tasks can be edited through the Task Library Editor shown in Figure 3.4. This dialog displays a list of tasks that are currently being studied with the Simulator. The entire library and each separate Task can be stored to file and loaded at a later time through the functionality in the Task Library Editor. The editor also allows creation of new tasks and deletion of unwanted tasks.

New Tasks can be created through the *New* button in the Task Library Editor. A dialog is then displayed asking for a chromosome representation type. This choice dictates the type of GP controller used by the new Task and cannot be modified after the Task is created.

The Task Editor allows the user to modify Task properties and is also shown in Figure 3.4. Task properties are divided into four categories: properties of the GP

Figure 3.4: Task Library Editor enables the user to create and modify a library of learning tasks. Task Editor lets the user modify learning task properties.

controller engine, chromosome structure properties, population properties, and run statistics. The GP controller properties specify parameters of the GP system such as crossover rate, mutation rate, and tournament size for tournament selection. The Disable Evolution option allows the simulation to run in read-only mode so that the population is not evolved.

The fitness function can be edited through the Fitness Function Editor (shown in Figure 3.5) accessible through the Task Editor. Fitness function definitions in JScriptTM can be typed into the editor window. Saving and loading functionality allow for storage and retrieval of fitness function definitions in text format. The Test button allows the user to verify the syntax of definition code.

The Chromosome Structure Properties dialogs can be used to modify the structure and building blocks of chromosomes in the population. Due to different structures of chromosomes for different GP methods, three different Chromosome Structure Properties dialogs are used: linear (for GP Linear type), tree (for GP Tree, HGP Module Acquisition, HGP Adaptive Representation types), and tree ADF (for HGP ADF type). Samples of the dialogs are provided in Figure 3.6.

The linear dialog allows modification of the structure of each linear genome instruction by choosing the types of parameters that can be used for each part of the

Figure 3.5: Fitness Function Editor allows for editing of the fitness function definition written in JScript™.

instruction. The maximum length of instructions in each chromosome and the maximum constant value can also be specified.

The tree dialog allows modification of the contents of the function and terminal sets. It also specifies the chromosome creation method (full, grown, or ramped half-and-half) [31]. *Maximum Creation Height* defines the maximum height of any chromosome at creation while *Maximum Overall Height* specifies the maximum height of any chromosome during evolution. The *Number of Outputs* value specifies the number of values returned from the chromosome program tree. Two values are normally used and the one value setting must be accompanied by corresponding motor *SetMx* functions to set the values of the robotic motors internally. In addition to the tree dialog, the ADF dialog contains definitions of the contents of ADF terminal and function sets, number of ADF function in each chromosome, and the number of arguments for each ADF function.

Population options in the *Task Editor* define the size of the population and let the user generate the initial population. The program trees of chromosomes in the

Figure 3.6: Chromosome Structure Properties dialogs for linear and tree-based chromosomes.

current population can be viewed using the Chromosome Browser as seen in Figure 3.7. For Hierarchical GP methods, modules or functions can be viewed through the Module Information dialog shown in 3.8.

Statistics of a GP controller run can be accessed using the Statistics section of the *Task Editor*. Current generation statistics can be seen through the Current Population Statistics dialog shown in Figure 3.9. Statistics can be saved into a comma-delimited text file that can be viewed by a text editor or spreadsheet application. Statistical information can be reset using the appropriate button.

Once the world environment is built and a robot placed into the environment, a learning task can be chosen for the robot through the GP Controller section of the Robot Properties dialog. Tasks for all active robots have to be chosen before the simulation can begin. The user can start the simulation engine from the *Simulation* menu or the *Simulation* section of the toolbar. The Run command starts the simulation engine until it is stopped by the Stop command. The Step command allows for step-by-step execution of the simulation engine. Only one world document can be simulated at any time in the current version of the Simulator. When the simulation

Figure 3.7: Chromosome Browser allows the user to view the chromosomes in the current population.

Figure 3.8: Module Information dialog shows a list of modules or functions used by the current HGP-based population.

Figure 3.9: Current Population Statistics dialog displays statistical information gathered during the last generation of the GP controller simulation.

engine is started, the user can watch the simulation taking place through the *World Editor*.

3.4 Code Structure

The Khepera GP Simulator for Windows[®] was fully created in Microsoft[®] Visual C++[®] 6.0 (MSVC) using the Microsoft[®] Foundation Class Library (MFC). The code was written using object-oriented methodology and software patterns. The main emphasis of the development was for reusable and modular code that can be easily modified and expanded. Classes within the Simulator can be divided into sections based on their utility.

3.4.1 Application Classes

Application classes define the Simulator application and its behavior. The class `CSimulatorApp` defines the main application process and handles application-based behavior. The `CSimulatorDoc` class represents a document that stores all data corresponding to a loaded world environment.

The `CWT_Simulator` class represents a simulation thread used to run the simulation engine. The simulation is run on a separate thread from the main application thread so that the user is able to easily watch the progress of the simulation through the application GUI. Currently, only one simulation thread is used; thus, only one document can be simulated at one time.

3.4.2 Data Classes

Data classes are used for storage of data and provide functionality for displaying the data to the user through UI and Dialog classes. Many data classes in the Simulator are derived from the MFC class `CObject` that provides easy serialization mechanisms. The data classes are divided into class hierarchies with common behavior and purpose.

The `CDS_World` class stores the world environment objects for each world document and provides interaction between the world objects. The class also stores specifications of each world environment (such as dimensions and scale). All objects in the Simulator are stored in world coordinates which could differ from screen coordinates (pixels). World coordinates are more accurate than pixels and allow for very detailed simulations executing in small screen windows. The `CDS_World` class provides conversion functions between world and screen coordinates using the scale parameter defining a ratio of world to screen coordinates.

All world object classes in the Simulator derive from a pure abstract class `CDS_Object` defining the interface to all world object classes. A class exists for each type of world object used by the Simulator: `CDS_Brick`, `CDS_Lamp`, `CDS_Cork`, `CDS_LightBox`, and `CDS_Robot`. Each class stores data required by the corresponding world object and provides functions required for object drawing and interaction with the environment and other objects.

The `CDS_Robot` class represents the special Khepera robot object. The class contains functions to calculate the sensor and motor values of the robot using the world environment. Helper classes `CDS_IRSensor` and `CDS_Motor` define the robotic sensors and motors.

The abstract `CDS_Task` class provides an interface for Task classes. Each derived Task class represents a learning Task and a robotic controller. There are five Task classes derived from `CDS_Task` that represent five different GP learning methods:

CDS_TaskLinear, CDS_TaskTree, CDS_TaskADF, CDS_TaskAR, and CDS_TaskMA. Each class can be instantiated to study a learning task using a particular GP representation method. The Task Library is represented by the CDS_TaskLib class.

Each Task class stores a fitness function definition represented by the CDS_FitnessFn class and stored in text format. Task classes also store populations of chromosomes. To improve code structure and reduce code coupling, five population classes are used, one for each GP representation method: CDS_PopLinear, CDS_PopTree, CDS_PopADF, CDS_PopAR, and CDS_PopMA. An abstract CDS_Population class provides interface to the concrete population classes.

The *encoding set* specifies the structure and properties of newly created chromosomes. Due to differences in chromosome structure between the studied GP methods, a class hierarchy was created for the encoding set. The abstract class CDS_EncodingSet sits at the top of the hierarchy. Encoding set for creation of linear genome based chromosomes is enclosed in the CDS_EncSetLinear class. The CDS_EncSetTree class uses the regular tree-based chromosome representation stored in a helper class CDS_EncSetBasic that does not derive from CDS_EncodingSet. The CDS_EncSetBasic class specifies the contents of the terminal set and function set for creation of tree-based chromosomes. The tree-based encoding set is also used by the Module Acquisition method since creation of new chromosomes in the HGP method is similar to that in regular tree-based GP. The Adaptive Representation method uses the CDS_EncSetAR class that stores a list of AR functions used in creation of new individuals. The CDS_EncSetADF class is used for creating chromosomes in the ADF representation since a separate definition of the result producing branch and the ADF branches is required.

Evaluation of chromosome program code is done with the help of the *evaluation set* defined by the CDS_EvalSet class. The evaluation set contains variable-to-value bindings required for evaluation of code. A derived class is used for each of the HGP method based chromosomes; the classes are: CDS_EvalSetADF, CDS_EvalSetAR, and CDS_EvalSetMA. Additional to the variable bindings, those classes contain bindings of function/module names to function/module code that must be evaluated in order to evaluate the chromosome code.

Chromosomes are stored in the chromosome class hierarchy rooted at the abstract class `CDS_Chromosome` that defines the interface to chromosome objects. The hierarchy is built according to the structures of chromosomes in each GP representation method. Linear genome based chromosomes are represented by the `CDS_ChromLinear` class while the tree-based chromosomes by the `CDS_ChromTree` class. Classes `CDS_ChromADF`, `CDS_ChromAR`, and `CDS_ChromMA` are derived from the `CDS_ChromTree` class and represent the chromosomes for the corresponding GP representation methods.

Chromosome program code for the linear genome chromosomes is stored in the chromosome object. The code for tree-based chromosome representations is stored in a tree data structure contained within the chromosome classes. The tree data structure class hierarchy is rooted at the abstract class `CDS_TreeNode`. Tree node classes define a recursive tree storage facility where each tree node is represented by a separate object. This implementation of the program tree nodes and the Genome Interpreter is analogous to the Virtual Function Tree approach as described in [28]. Keith and Martin state that this pointer-based approach provides effective and reusable code with a small time and space overhead.

The `CDS_TreeNode` class hierarchy includes: a binary tree node class `CDS_TreeNodeBinary`, a multi-nary tree node class `CDS_TreeNodeMultinary`, and other classes with special requirements. The `CDS_TreeNodeVar` and `CDS_TreeNodeConst` classes define variable and constant terminals, respectively. The `CDS_TreeNodeIFLTE` class represents an *if-less-then-else* conditional node with four children. The ADF chromosome representation uses the `CDS_TreeNodeDefun` class for a *defun* node of an ADF tree.

Multi-nary tree nodes can contain any number of children nodes. In the Simulator, we use the abstract `CDS_TreeNodeMultinary` class to denote tree nodes that have a varying number of children. Not all node classes are made to derive from the `CDS_TreeNodeMultinary` class due to code speed and memory space considerations. The multi-nary tree node classes `CDS_TreeNodeProgn` and `CDS_TreeNodeValues` are used by the ADF representation chromosome trees. The classes `CDS_TreeNodeModCall`, `CDS_TreeNodeFn`, and `CDS_TreeNodeARFn` are used to

represent function/module calls in program trees used by HGP representation methods: Module Acquisition, ADFs, Adaptive Representation, respectively.

Binary tree nodes have exactly two children and are represented by the abstract class `CDS_TreeNodeBinary`. The binary tree node classes include mathematical functions: `CDS_TreeNodeAdd`, `CDS_TreeNodeSub`, `CDS_TreeNodeMul`, `CDS_TreeNodeDiv`, boolean functions: `CDS_TreeNodeAND`, `CDS_TreeNodeOR`, `CDS_TreeNodeXOR`, left and right shift operators: `CDS_TreeNodeLS`, `CDS_TreeNodeRS`, and a list function with 2 arguments: `CDS_TreeNodeList2` [31].

Special `CDS_TreeNodeSetVar` class is used to set the value of a global variable inside the program code. The idea represented by the class is that of settable variables (see [31], sections 18.2, 19.7 and [32]). The `CDS_TreeNodeSetVar` class was used in our research for setting the values of robotic motor variables $m1$ and $m2$.

The studied HGP methods require the use of functions and modules. The `CDS_BlockAR` class stores the program code for a function (also called a building block) used by the Adaptive Representation method. The `CDS_GLibMod` class is analogous to a module used by the Module Acquisition method. The ADF method does not require extra data storage classes since the ADF program code tree is stored together with each corresponding chromosome.

Other miscellaneous classes and structures exist in the Simulator to extend its modularity and functionality. Statistical information for a GP run is stored in the `CDS_Stats` class. The `CDPoint` class represents a point on the world environment where the coordinates are in double precision floating point values. The `CDVect` class denotes a pair of points making a 2D vector (or a rectangle) in space while the derived `CDVectDir` class includes a direction angle that is used by the physics engine of the Simulator.

3.4.3 User Interface Classes

The user interface of the Simulator is composed of over twenty dialog classes. Each dialog class name is prefixed with `CUID` denoting the dialog class type. Dialogs are used to view and modify the data contained in the data classes. The MFC-derived classes `CMainFrame`, `CChildFrame`, and `CSimulatorView` provide the window functionality of the Simulator application. The `CUI_InfoBar` class represents the Info

Window of the Simulator.

The user interface tools used to manipulate the world environment are all based on the abstract `CUIT_Generic` class that provides common interface to all the tools. Concrete tool classes are derived from this abstract class and provide their corresponding functionality. The drawing tool classes are: `CUIT_DrawBrick`, `CUIT_DrawCork`, `CUIT_DrawLight`, `CUIT_DrawLightBox`, and `CUIT_DrawRobot`. The select tool is represented by the `CUIT_Select` class. This hierarchical tool class structure provides clean implementation of the tool functionality and simplifies the addition of future tools.

3.4.4 COM Objects and Interfaces

The Simulator contains one COM object represented by the `CDS_RobotSet` class and stored by the population classes. This class contains up-to-date values of the simulated Khepera robot's sensors and motors. The values are used by the scripting engine responsible for evaluating of the fitness function. The COM object is also represented in the scripting language by the object *robot*.

The robot COM object can be manipulated by using the `IDS_EncodingSet` interface. The interface provides get and set functions for the robotic sensors and motors. This interface is used by the scripting function to gain access to the properties of the *robot* object.

3.5 Summary

In this chapter, we have taken a closer look at the Khepera GP Simulator. We have described the simulation of the physical robot and its environment and introduced the evolutionary robotic controller. We have shown the user interface of the Simulator and finally, described the internal code structure of the Simulator application. The next chapter will take a detailed look at the evolutionary robotic controllers used in the Simulator.

Chapter 4

Robotic Controllers

In this thesis, we address the problem of robotic control using various genetic programming and hierarchical genetic programming approaches. To study this problem, we require robotic controllers that use various GP methods to evolve robotic control programs for a simulated Khepera robot.

This chapter will discuss robotic controllers. First, we will take a look at the generic robotic controller architecture used in our research. Next, we will begin with Genetic Programming controllers: linear genome based GP controller and tree-based GP controller. Finally, we will study Hierarchical Genetic Programming controllers using the methods of Automatically Defined Functions, Module Acquisition, and Adaptive Representation.

4.1 Generic Architecture

In this section, we present the generic robotic controller architecture used by the various GP approaches in the following sections. Modifications to this architecture required by each GP method are discussed in the sections corresponding to the method.

Our research into robotic controllers built on research done by Nordin and Banzhaf [49] on using GP to evolve robotic controllers for the Khepera robot. Nordin and Banzhaf were able to evolve controllers for various learning tasks (such as obstacle avoidance and wall following). In our research, we compared the linear genome GP

method they have used in their experiments to tree-based GP and three most popular Hierarchical Genetic Programming methods: Automatically Defined Functions, Module Acquisition, and Adaptive Representation.

The GP system in the robotic controller evolves control programs that best approximate a desired solution to a pre-defined problem. This procedure of inducing a symbolic function to fit a specified set of data is called Symbolic Regression [47]. The goal of the system is to approximate the function

$$f(s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7) = \{m_1, m_2\} \quad (4.1)$$

where the function input is the robotic sensor data and the output is the speed of the motors controlling the motion of the robot. The control program code of each individual constitutes the body of the function. The results are compared using a behaviour-based fitness function that measures the accuracy of the approximation by the deviation from desired behavior of the robot.

In our research, we dealt with a population of control programs for the Khepera robot. Evolutionary algorithms were applied to the population in order to evolve control programs that accomplish specified learning tasks. The evolutionary algorithm used in the GP robotic controller was a steady-state tournament selection algorithm. The execution cycle of the generic GP algorithm used by the Simulator for one robot is shown in Figure 4.1. A schematic view of the algorithm is shown in Figure 4.2.

When performing a simulation of multiple Khepera robots in the same environment, the robots take turns in executing the GP algorithms. This provides a simulated real-time multi-tasking environment where interactions between the robots can be studied.

4.2 Linear Genome GP

The linear genome GP system using binary machine code was introduced in [44] as Compiling Genetic Programming System (CGPS). The method was later used in [47] to evolve a robotic controller for Khepera robots. The structure of our linear genome GP controller closely resembles the controller used by Nordin and Banzhaf.

GP Algorithm

1. Initially generate a random population of individuals using the encoding set specific to the GP method used.
2. Repeat the following until specified termination criteria occur:
 - (a) Randomly select k individuals from the population with no replacement (k indicates tournament size).
 - (b) For each of the k selected individuals:
 - i. Retrieve robot sensor data from the simulated environment using the resulting position of the robot from previous evaluation.
 - ii. Using the evaluation set, evaluate the program code of the individual based on the given sensor input and retrieve resulting motor values.
 - iii. Set the motor values into the simulated robot and step the robot one time unit. The position of the robot and position of other objects in the environment might change depending on the motion of the robot.
 - iv. Retrieve new robotic sensor data from the resulting simulated environment.
 - v. Compute the fitness of the individual using the new sensor data and fitness function definition. Up to 4 fitness values from previous evaluations are stored in the individual and the average is used to designate final individual fitness.
 - (c) Create two new individuals by copying the 2 fittest individuals from the selected k individuals.
 - (d) Apply genetic operators of mutation and crossover on the two new individuals, based on specified probabilities.
 - (e) Replace the 2 worst fit individuals from the selected k individuals with the resulting two new individuals.
 - (f) Update generation statistics.

Figure 4.1: Pseudocode of the genetic programming algorithm used in our research.

Figure 4.2: Schematic diagram of the generic GP robotic controller used in the Simulator.

4.2.1 Chromosome Structure

The original linear genome GP controller by Nordin and Banzhaf [47] used variable length strings of 32 bit instructions for a register machine performing arithmetic operations on a small set of registers. The initial versions of the Simulator used a 32-bit integer representation of each instruction. In the current version, each instruction is represented as a text string and is processed by a Genome Interpreter prior to evaluation. This change was made to improve the readability of the Simulator code. A loss in performance was noticed since processing of the string based instructions is more time intensive than for numerical representations. However, it was decided that the performance of the new representation was sufficient for the purpose of the research.

In the linear genome GP system, each individual was composed of a series of instructions (genes). We used two different instruction types: *P1* and *P2*. The instructions were of the following format:

$$P1 : resvar = var1 \ op \ var2 \quad (4.2)$$

$$P2 : resvar = var1 \ op \ const \quad (4.3)$$

where *resvar* was the result variable and *op* was a binary operator working on either two variables *var1* and *var2* or a variable *var1* and a constant *const*. Sample of the program code used in the linear genome method is provided in Table 4.1.

P2:	f = s4 >> 4857
P1:	b = s3 << s2
P2:	d = s4 + 481
P1:	m1 = s0 + s2
P1:	m2 = s2 ^ s5

Table 4.1: Sample program code of an individual using the linear genome GP method.

4.2.2 Population Generation

Individuals of the population were created randomly using settings provided through the *Linear Chromosome Structure Properties* dialog. Each individual was randomly assigned a height (number of instructions) from 1 to the maximum height specified in the settings. Instructions were randomly chosen to be of either type *P1* or *P2*, with equal probability. For each part of an instruction, a value was selected randomly from a set of primitive values. The sets of primitive values for each instruction part can be modified through the settings dialog.

The primitive values of the *result* and *variable* types of instruction parts were chosen from the set of 8 proximity (distance) sensors: *s0* – *s7*, set of 8 ambient light sensors: *l0* – *l7*, set of 2 motors, and a set of 6 intermediate variables: *a*, *b*, *c*, *d*, *e*, *f* (with default value of 0). The *operator* values were chosen from a fixed set of 8 arithmetic, logical, and shifting operators. The integer values of constants in *P2* instructions were chosen randomly from the range $[0, max)$ where *max* was the maximum constant value specified in the settings dialog. Table 4.2 provides primitive value sets of the instruction parts used in our experiments.

4.2.3 Chromosome Evaluation

Individual program code evaluation was done by a chromosome object storing the set of instructions defining the program code. The evaluation set provided the chromosome object with values of all variables that were used in the instructions. Each

Instruction Part	Instruction Value Set
res	motor values ($m1, m2$); intermediate variables ($a - f$)
var1, var2	distance sensor values ($s0 - s7$) [proximity experiments]; light sensor values ($l0 - l7$) [light experiments]; intermediate values ($a - f$)
op	add (+), subtract (-), multiply (*), left shift (SHL) right shift (SHR), XOR (^), OR (), AND (&)
const	numerical value: 0-8191

Table 4.2: Primitive values of each instruction part in the linear genome GP method as used in the experiments.

instruction was evaluated separately and results were stored back into the evaluation set.

The Genome Interpreter split each text-based instruction into its corresponding parts. The *operator* part was used to choose the function applied to the rest of the parts. The values of each *variable* instruction part were extracted from the evaluation set and the values of constant parts were converted from string to a numerical value. The corresponding function was then performed on the values and the resulting value was stored into the evaluation set under the *result* instruction part.

4.2.4 Genetic Operators

The linear genome GP method used three genetic operators: reproduction, crossover and mutation. Crossover was applied to the two fittest individuals of a tournament, according to a probability specified by the user. If crossover was not selected, reproduction occurred and copies of fittest individuals were inserted into the new population.

The crossover operator used a simple variable length 2-point crossover applied to the list of instructions (genes) of an individual. Genes were treated as atomic units by the crossover operator and were not modified internally. Two crossover points were randomly chosen for each individual denoting the start and end of a selection of genes. The two sets of genes from both parents were swapped between the individuals to create two children as shown in Figure 4.3. If the length of a child exceeded the allowed maximum chromosome length, a copy of the primary parent that created the

child was placed into the next generation.

Figure 4.3: Example of crossover between linear genome chromosomes.

Mutation modified the contents of a gene. The probability of mutation was specified per gene and multiple mutations were allowed in one individual. Once a gene was selected for mutation, the instruction was decoded to its four parts: *res*, *var1*, *op*, *var2* (or *const*). One of the parts was then randomly chosen, with equal probability. If the chosen part was *res*, *var1*, or *op*, a random value of the same part was selected from the encoding set and the value was substituted into the gene. However, if the chosen part was *var2* (or *const*), the type of instruction was modified with the probability of 0.1. A new value of either part *var2* or *const* was selected based on the modified instruction type *P1* or *P2*, respectively. Examples of the mutation operation are provided in Figure 4.4.

4.3 Tree-based GP

The tree-based GP system used the canonical tree representation of chromosomes. This GP method was used in the experimentation for two reasons: to compare the performance of linear genome GP to tree-based GP and to set a performance base

Figure 4.4: Examples of different cases of gene mutation in a linear genome chromosome.

for the HGP methods using the tree-based chromosome representation.

4.3.1 Chromosome Structure

As in the canonical GP implementation described in Section 2.3.1, the control program code in the tree-based GP method was composed of rooted trees. In the Simulator, trees were defined as recursive node structures where:

1. the root of the tree was a node
2. every node in the tree had 0 or more children nodes.

The recursion stopped at terminal nodes that did not have any children nodes. The internal nodes of the program trees contained parameterized functions while the leaf nodes contained parameter-less functions and terminals.

In our research, we required two return values from the program code evaluation. The two values were used to drive the motors of the Khepera robot. We studied two methods of returning the vector of two values. The first method represented the

program code as a single rooted tree with no output value. The robot motor values were set explicitly inside the tree using settable variables (see [31], sections 18.2, 19.7 and [32], section 18.3) and corresponding functions *SetM1* and *SetM2*. The second method did not use special functions but stored the program code as two rooted trees, one for each motor value (see [31], section 19.1). Results from the two trees were used as values to the motors.

The nodes of the program tree in tree-based chromosome representation were selected from a set of functions and a set of terminals. The function set defined a collection of parameterized functions that were used to build the internal nodes of the tree. Each parameter of a function node was stored in a child subtree. Parameter-less functions were only available for leaf nodes. The terminal set defined a collection of terminals that were used in the leaf nodes of the tree. Terminals were either variables or constant numerical values.

4.3.2 Population Generation

Program trees of each individual were created in a recursive manner. Three methods were provided for the creation of the initial random population: *full*, *grow*, or *ramped half-and-half* [31].

The *full* method created trees where all leaf nodes were at equal depth. The height of the trees was equal to the maximum creation height specified in the *Chromosome Structure Properties* dialog. The trees were created by selecting the tree nodes at less than the maximum depth from the function set of parameterized functions. The nodes at maximum depth were chosen from the terminal set. All resulting trees had the greatest allowable height at creation time.

The *grow* method grew trees of variable shape. The tree nodes at less than the maximum depth were chosen from a union of the function set and the terminal set. The nodes at maximum depth were chosen from the terminal set to ensure the maximum tree height at creation was respected.

The *ramped half-and-half* method was a mixture of the *full* and *grow* methods. At each tree depth (between 2 and maximum specified depth) an equal number of trees were created. For each specific depth value, half of the trees were created using the *full* method and the other half were created using the *grow* method. This method

created trees of various heights and shapes and it was documented to perform well over a broad range of problems [31].

In our research, we have experimented with various function and terminal combinations. The list of available functions and terminals used in our experiments is provided in Table 4.3.

Function Code	Description	Parameters
Add	binary integer addition	2
Sub	binary integer subtraction	2
Mul	binary integer multiplication	2
Div	protected binary integer division	2
AND	binary AND logical operator	2
OR	binary OR logical operator	2
XOR	binary XOR logical operator	2
<<	binary left shift operator	2
>>	binary right shift operator	2
IFLTE	if-less-than-or-equal conditional	4
SetM1	set motor m1 (only if single output)	1
SetM2	set motor m2 (only if single output)	1
Terminal Code	Description	Parameters
Var [s0-s7]	8 proximity sensors of the robot	0
Var [l0-l7]	8 ambient light sensors of the robot	0
Const [0-8192]	constant integer value in given range	0
Const [0-100]	constant integer value in given range	0
Const [0-10]	constant integer value in given range	0

Table 4.3: Values for the contents of the function set and terminal set available in the Simulator.

If the number of outputs specified in the chromosome properties dialog was 1, a single rooted tree was created by randomly selecting node labels from the function and terminal sets (depending on the creation method and depth). Each selection from the terminal or the function set was made with equal probability. If the number of outputs was specified as 2, two rooted trees were created, each of height one less than the maximum height allowed at creation. The trees were joined by a *values* node acting as a root of the program tree.

4.3.3 Chromosome Evaluation

Chromosome evaluation was done with the help of the evaluation set which stored the values of all variables available in the program tree. The evaluation was done recursively on the tree, starting with the root node. With the help of the virtual function model in C++, evaluation of each node was passed onto the *Eval* method of the corresponding node class. Each class provided handling of the *Eval* method that acted as a Genome Interpreter. The *variable* nodes extracted values of stored variables from the evaluation set and the *constant* nodes returned internally stored integer values.

The tree node evaluation structure was created to respect the closure property. The return value of all tree nodes was of the same type; thus, any return value from a node was able to be used as an argument to other nodes in the tree. Some nodes ignored the return values of their children but most nodes processed the return values and returned the result up the tree. With the representation using two rooted trees, the return value from each tree was used to set the corresponding *m1* or *m2* variable. The one tree representation ignored the root return value and relied on the *SetM1* and *SetM2* nodes to set the corresponding motor variables in the evaluation set.

4.3.4 Genetic Operators

Only two genetic operators were used in the tree-based chromosome representation: reproduction and crossover. The mutation operator for tree-based chromosomes was not used in our research because of the role played by the crossover operator (see Section 2.3.3). Crossover was applied to the two fittest individuals in a tournament, with a given probability. Reproduction occurred in place of crossover if the random coin toss did not select for crossover to take place.

The crossover operator was a single subtree switching crossover between the two parents as shown in Figure 2.4. A random node was chosen independently as the crossover point in both parents. The subtrees rooted at the chosen nodes were then swapped between the parent trees to produce two children. If the height of the resulting children exceeded the maximum overall height, specified in the chromosome properties dialog, the crossover operation defaulted to reproduction for the primary

parent of the child. In the case of multiple output values and dual trees, the *values* root node was restricted from being selected for a crossover point.

4.4 Automatically Defined Functions HGP

The Automatically Defined Function (ADF) HGP method used in our research was based on the ADF method proposed by Koza [32] (see Section 2.4.1). The ADF method is the most widely used HGP method with many instances of documented performance gain over the canonical tree-based GP method (for details see [32, 33]). We have used the method in our experiments to examine its performance in the area of reactive robotic control.

4.4.1 Chromosome Structure

The ADF method was an extension of the tree-based GP method and shared its basic structure. A typical structure of an ADF chromosome is shown in Figure 4.5. The structure was composed of two types of nodes: invariant and non-invariant. The invariant points (above dotted line in Figure 4.5) were fixed and present in every ADF chromosome. Non-invariant points (below dotted line in Figure 4.5) defined the bodies of the ADF definitions and result producing branch and were modified during evolution.

In contrast to generic GP, ADF chromosomes used two terminal and two function sets. The result producing branch was built using a standard terminal set and standard function set augmented with the ADFs contained in the same chromosome. Separate terminal and function sets were used by the function defining branches to define the ADFs. These secondary sets did not have access to the original terminals provided by the program and worked on dummy variables denoting the arguments to the function call. The size complexity measure of the program tree included the ADF definition portions of the tree; thus, the SC measure was equivalent to the size measure in our implementation.

All ADFs defined in an individual were only available locally to the program tree of the same individual. ADF chromosomes could contain multiple ADF definitions

Figure 4.5: Structure of a minimal ADF chromosome with one function-defining branch and one result-producing branch. The dashed line represents boundary between invariant and non-invariant points.

(multiple *defun* branches) which specified multiple functions available to the result producing branch. The number of ADFs present in each chromosome and the number of arguments for each ADF were specified as parameters set by the user before population initialization. Multiple result producing branches, expressed as children of the *values* node, were also allowed. The extra result outputs were used for problems where it was beneficial to express the final result as a vector of values.

Zero or multiple ADFs could be called from within the result producing branch. Some recursive ADF implementations allow calling of ADFs from within other ADFs. This leads to problems with circular evocation of ADFs and requires extra protection. Due to the increase of implementation complexity, we did not allow ADF calls inside ADF definitions in our implementation of the ADF method.

4.4.2 Population Generation

Generation of individuals using the ADF method was similar to the generation of individuals in the tree-based GP method. The invariant nodes of ADF program trees (see Figure 4.5) were identical for each chromosome in the population while the non-invariant nodes differed between chromosomes.

The number of ADF functions used in each program tree and the number of

arguments to all ADF functions were modified using the *ADF Chromosome Structure Properties* dialog of the Simulator. The ADF functions were stored using *defun* nodes that contained the name, arguments, and a subtree specifying the body of the ADF definition. The *progn* node acted as the root of the entire program tree. The result producing branch was the first child of the *progn* node and was followed by one or more sibling ADF definition branches. For the case where multiple outputs from the program tree were required, multiple result producing branches were used.

The function and terminal sets decided the contents of the program tree. We have used two different function and terminal sets for the ADF definition branches and the result producing branches. The function and terminal sets that we have used for the result producing branch were composed of the same primitives that we have used for the tree-based chromosomes (as shown in Table 4.3). The original function set was extended with ADF function calls of the ADF functions defined in the same chromosome. The function calls used a special *Fn* tree node that was used in the evaluation of the program tree.

The ADF definition branches used a restricted function set that did not contain the *SetM1* and *SetM2* functions. The terminal set used by the ADF definition branches was composed of the argument variables to the ADFs and constant numerical values. The modifications to the terminal and function sets of the ADF definition branches were made to hide the original variables used in the result producing branches.

As with the tree-based chromosomes, three chromosome creation methods were used: full, grow, and ramped half-and-half. The non-invariant nodes of each ADF body and result producing branch body were randomly and recursively created by the same method as trees in the tree-based method. The maximum creation height of ADF chromosome program trees was used to set the maximum creation height of the non-invariant node trees. Thus, the overall maximum creation height of ADF chromosomes was two levels higher due to the invariant nodes.

4.4.3 Chromosome Evaluation

ADF chromosome evaluation was similar to the tree-based chromosome evaluation. Only the body of the result producing branch was evaluated and its result values were used to set the motor values of the evaluation set. The difference in the ADF

chromosome evaluation arose from the ADF call nodes in the result producing branch.

Before the recursive evaluation of the result producing branch was initiated, the evaluation set was extended with ADFs of the individual. An F_n node stored the name of the corresponding ADF and arguments to the function call as subtrees of the node. Evaluation of the F_n node required the evaluation of all subtrees resulting in a set of values. The values were used as corresponding arguments (named ARG_n , where n was the corresponding function number) of the function through the use of a temporary evaluation set. The ADF tree was then evaluated using the temporary argument bindings. The result of the function evaluation was used as the result value of the F_n node and was propagated up the program tree by the recursive evaluation procedure.

4.4.4 Genetic Operators

As in the tree-based chromosomes, only reproduction and crossover operators were used in the ADF chromosomes. Crossover was applied to the two fittest chromosomes in a tournament, based on a specified probability. Reproduction was the default action if crossover was unable to take place.

The sub-tree swapping crossover operator used in ADF chromosomes was more complex than in tree-based chromosomes. To preserve the structure of the chromosome, the operator was only allowed to swap non-invariant nodes and nodes of similar type.

Two major methods exist of assigning type information to non-invariant nodes of program trees [32]. *Branch typing* assigns a different type to nodes contained in different branches of the ADF chromosome. This ensures that crossover takes place between the same result producing branches or the same function defining branches in the two parents. Branch typing is the default process to assign type information to nodes. *Point typing* is sometimes used in situations where the architecture of the program is evolved during the run of the GP. This method assigns a distinct type to a node based on the function set, terminal set, and other structural information of the node.

In our research, we used branch typing to assign type information to ADF program tree nodes. The result producing branch was assigned type 0 and each ADF branch

was assigned a unique type greater than 0. A crossover point was first chosen in the first parent and the branch type of the crossover point node was calculated. The crossover point of the same branch type was then chosen from the second parent. Once the crossover points were known, the crossover was applied as in the tree-based GP method.

4.5 Module Acquisition HGP

The Module Acquisition (MA) method of Angeline and Pollack [3] uses two new operators of compression and expansion to modularize the program code into subroutines. The resulting hierarchical module structure makes this method interesting to study. We have used the MA method in our research to compare its performance against the performance of the basic tree-based GP method and the other HGP methods studied.

4.5.1 Chromosome Structure

The chromosome program tree structure used in the Module Acquisition method was identical to the program tree structure of the original tree-based chromosomes. Unlike in the tree-based GP method, MA chromosomes could contain module call nodes that call stored subroutines. Modules in the MA method were local to a chromosome and were only propagated through the population by reproduction and crossover.

Each MA chromosome stored a list of modules used in the program tree of the chromosome. Modules were stored in separate objects from the chromosome to increase code modularity. Module objects contained their own program trees that defined the code of each module. Module call nesting was allowed inside program trees of other modules; however, by the nature of their creation, modules were never recursive. The non-recursive nature of modules eliminated problems with circular evaluation and infinite loops.

4.5.2 Population Generation

The initial random chromosome population using the MA method was created in the same way as in the tree-based GP method. Properties of the chromosomes were

edited using the *Chromosome Structure Properties* dialog. Contents of the function set and terminal set could also be modified and the available primitives are provided in Table 4.3.

4.5.3 Chromosome Evaluation

MA chromosome evaluation was similar to the chromosome evaluation used by the tree-based GP method. The MA chromosome program trees could contain *ModCall* nodes. These nodes represented module calls and required access to the modules stored in the chromosome. Before the evaluation of the program tree occurred, the modules used by the chromosome were placed into the evaluation set.

During the evaluation of the program tree, *ModCall* nodes performed module evaluation. All subtrees of the *ModCall* node were first evaluated and their return values recorded. The return values were then used as arguments to the evaluation of the corresponding module retrieved from the evaluation set. The return value of the module evaluation was used as the return value of the *ModCall* node.

4.5.4 Genetic Operators

The Module Acquisition method used four genetic operators: reproduction, crossover, and two mutation operators of compression and expansion. The reproduction and crossover operators performed as outlined in Section 4.3.4 for tree-based chromosomes. Additional update of the chromosome module lists was performed for each of the chromosomes undergoing crossover. This update was made to ensure the module lists contain exactly the modules used in the program trees of the chromosomes or recursively in the program trees of other modules used by the chromosome.

The compression operator created a new subroutine from a randomly selected subtree of an individual in the population. We have used the method of depth compression [1] in our research. In our implementation of the compression operator, we first selected a random maximum depth value from a specified range of values. Unless stated otherwise, we have used a range of [2,5] to choose the random maximum depth value in our experiments. We also selected a random node of the tree to act as a root of the subtree undergoing compression.

If the chosen subtree was not beyond the chosen maximum depth, we created a module (subroutine) with no parameters and the chosen subtree as its program tree. A subroutine call replaced the chosen subtree in the chromosome program tree (as shown in Figure 4.6a). The module was then added to the list of modules for the chromosome. If the chosen subtree was too large, branches beyond the maximum depth were used as parameters to the new subroutine (as shown in Figure 4.6b).

Figure 4.6: Module creation in Module Acquisition. Part a) shows creation of a parameterless function and part b) creation of a function with 2 parameters.

The expansion operator was used to undo the work of the compression operator. First, a search was done through the chromosome program tree to find a module call. If a module call was found, it was replaced with the original subtree stored in the corresponding module. The module was removed from the module list of the chromosome if it was no longer used by the chromosome. In our implementation, we have also included another form of the expansion operator that recursively expanded all modules in the chromosome.

The mutation operators of compression and expansion were used after the reproduction or crossover operators produced offspring of the two fittest chromosomes in a tournament. Compression and expansion were applied to each of the two offspring

independently, with a specified probability. The probability of compression decided the amount of compression used in the system and was modifiable by the user. The probability of expansion (which was usually much lower than that of compression) was also modifiable.

4.6 Adaptive Representation HGP

The Adaptive Representation (AR) HGP method by Rosca and Ballard [59] extended the canonical tree-based GP system by introducing parameterized blocks (i.e. functions). Unlike in the ADF HGP approach, the functions were discovered automatically and without human-imposed structure. The method differed from the MA HGP approach by the algorithms used in function discovery and management of a global function library. Our implementation of the AR method was based on the improved Adaptive Representation through Learning (ARL) algorithm [60]. We have used the ARL method in our research to examine its performance in the domain of robotic control.

4.6.1 Chromosome Structure

The structure of the ARL chromosome program trees was identical to the structure of trees in the tree-based GP method. An additional function node was used in the trees to enable ARL function calls. Unlike the MA method, the ARL functions were stored in a global library and were accessible to all the chromosomes in the population. The function library was dynamically grown by the execution of the evolutionary algorithm. The library was stored in the population object; thus, different populations contained a different set of ARL functions. A local function library for each chromosome contained links to functions used directly or indirectly by the chromosome. This library was used to easily access the functions without accessing the global function library.

The ARL functions were stored outside of the chromosomes in special block objects. The block objects stored their own program trees and shared a common evaluation interface with the chromosomes. Nesting of function blocks was allowed; however,

recursive function calls were not possible due to the function creation methods.

4.6.2 Subroutine Discovery

The main idea behind the ARL algorithm was the automatic discovery of useful subroutines. Rosca [62] defined the concepts of *differential fitness* and *block activation* used in subroutines discovery. The algorithm for subroutine discovery used in our research is provided in Figure 4.7.

Differential fitness was defined as the difference in fitness between an individual and its least fit parent and can be stated as:

$$DiffFitness(i) = Fitness(i) - \min_{p \in Parents(i)} \{Fitness(p)\} \quad (4.4)$$

where i is an individual in the population with raw fitness $Fitness(i)$. We were interested in individuals with maximum differential fitness:

$$\max_i \{DiffFitness(i)\} > 0 \quad (4.5)$$

Rosca stated that large differences in fitness can be the result of useful combinations of blocks of code in the individual [62].

Block activation was defined as the number of times a block of code was executed during evaluations of the individual. Rosca stated that only blocks with high block activation values should be considered candidate blocks. We have not used the concept of block activation because of the large performance overhead on the system. In most of our experiments, we have not used any conditional functions, thus the block activation values for every block in an individual were equal.

In our implementation of the *Subroutine-Discovery* algorithm, we selected one promising individual from the set of promising individuals discovered during the last generation. Selection was done by applying Equation 4.5 to find the most promising individual. This choice of only one individual was made to limit the number of subroutines created during an epoch. The most promising individual was processed to produce a set of candidate building blocks. Candidate building blocks were defined as blocks (subtrees) of code with a specified block height.

Subroutine-Discovery(B, S^{new}, P^{dup})

1. Select the most promising individual i (with greatest differential fitness) of the generation from the set of promising individuals B .
2. Create a set of candidate building blocks BB by selecting all blocks of small height (set by experimental parameter).
3. For each block $b \in BB$:
 - (a) Generalize the code of block b to create a subroutine s using a terminal set of local subroutine variables.
 - (b) Create a new individual p^{dup} as a copy of individual i with block b replaced with a call to the new subroutine s .
 - (c) Update $S^{new} = S^{new} \cup \{s\}$ and $P^{dup} = P^{dup} \cup \{p^{dup}\}$.

Figure 4.7: Our implementation of the Subroutine-Discovery algorithm.

Generalization of a candidate block was done by creating a new subroutine and creating a duplicate individual using the new subroutine. The process is shown in Figure 4.8. The new subroutine contained the tree of the candidate block with terminals replaced by the subroutine parameters. Duplicate terminals in the candidate block were generalized to one parameter of the subroutine. The new subroutine was added to the function set of the population. The most promising individual was cloned to produce a duplicate individual. The candidate block of the duplicate individual was substituted by a function call to the new subroutine. The duplicate individual was added to a set of duplicate individuals that was used in the intermediate population.

Figure 4.8: Subroutine creation in Adaptive Representation.

Rosca [62] used the idea of a subroutine utility that was analogous to schema fitness for subroutines. The utility was used for accumulation of rewards for a subroutine over a fixed time window and was calculated by a special utility function. Using subroutine utility, low performing subroutines were removed from the function set and individuals using the removed subroutines were modified by uncompressing the subroutine calls. This maintenance of the set of subroutines was done in order to prevent the set from growing too large.

In our research we have used a simpler measure of subroutine utility. We have assigned to each subroutine an integer utility value with a default value specified by parameter W . The utility value denoted the number of generations until an unused subroutine was removed from the function set. Unused subroutines had their utility value decremented each generation until the value reached 0 and the subroutine was removed from the population. Subroutines used by the individuals in the population were not removed.

4.6.3 Evolutionary Algorithm

Implementation of the ARL method required changes to the generic evolutionary algorithm specified in Section 4.1. Our ARL evolutionary algorithm used a generational (non steady-state) algorithm with tournament selection to select individuals for the mating pool. Population size was kept constant between generations but a larger intermediate population was used during each generation. Our implementation of the ARL algorithm is outlined in Figure 4.9.

The most important part of the ARL algorithm was the Adapt Representation step. This step automatically extended the function set by analyzing blocks of code in individuals with promising performance. The run of the AR algorithm was divided into epochs which were defined as sequences of consecutive generations in which no new candidate building blocks were discovered [59]. The ARL algorithm provided a concrete definition of epoch creation using *population entropy* [61].

Population entropy provided a measure of the state of a dynamic system represented by the population. Rosca [61] compared the population-based dynamic system to a physical or informational system with similar behavior. Shannon's information entropy formula [64] was used to capture the entropy value.

ARL Algorithm

1. Initialize parameters: $t = 0$, $S_0 = \emptyset$, $S_0^{new} = \emptyset$, $P_0^{new} = \emptyset$, $P_0^m = \emptyset$, $P_0^{dup} = \emptyset$, $P'_0 = \emptyset$, $B_0 = \emptyset$.
2. Initially generate a random population P_0 of ARL individuals using the terminal set T and function set F .
3. Repeat the following until specified termination criteria occur:
 - (a) Generate a new population P_{t+1} by tournament selection, reproduction, and crossover of individuals in P_t .
 - i. Create intermediate population $P'_t = P_t \cup P_t^{new} \cup P_t^{dup}$.
 - ii. Create the mating pool P_t^m by repeating $|P_t|$ times:
 - A. Randomly select k individuals from the intermediate population P'_t with replacement, where k indicates tournament size.
 - B. Perform individual evaluation for the k selected individuals as in the generic algorithm in Figure 4.1.
 - C. Add a copy of the fittest individual of the tournament to the mating pool P_t^m .
 - D. Find the most promising individual of the tournament using differential fitness and add to the set of most promising individuals B_t .
 - iii. Apply genetic operations of reproduction and crossover to individuals in the mating pool P_t^m to produce offspring P_{t+1} .
 - (b) Adapt Representation:
 - i. Update subroutine utilities S_t and delete subroutines of low utility.
 - ii. Initialize the set of new subroutines $S_t^{new} = \emptyset$, set of duplicate individuals $P_t^{dup} = \emptyset$, and set of new individuals $P_t^{new} = \emptyset$.
 - iii. Calculate population entropy E_t of population P_t .
 - iv. If entropy value E_t passes the entropy threshold test:
 - A. Discover new subroutines S_t^{new} and create duplicate individuals P_t^{dup} by calling Subroutine-Discovery($B_t, S_t^{new}, P_t^{dup}$).
 - B. Randomly generate new individuals P_t^{new} using the terminal set T and function set $F \cup S_{t+1}$.
 - v. Create the set of discovered subroutines for the next generation: $S_{t+1} = S_t + S_t^{new}$.
 - (c) Proceed to the next generation: $t = t + 1$.

Figure 4.9: Pseudocode of our implementation of the ARL algorithm.

Entropy was measured by grouping individuals of the population into a set of classes based on their behavior (phenotype). Shannon's formula was then used to calculate the entropy:

$$E(P) = - \sum_k p_k \cdot \log p_k \quad (4.6)$$

where p_k is the proportion of the population P grouped into partition k . Individual raw fitness is usually used in entropy calculation. We could not use raw fitness in our calculation because of the dynamic nature of our fitness calculation. We have used a standardized fitness measure SFM to calculate a value used for individual classification. The SFM was calculated for each individual using three test cases. Each test case provided a fixed set of values to robotic sensors of the individual and evaluated the individual to acquire the two resulting motor values. Test case sensor values are provided in Table 4.4. The motor values from each test were averaged to yield a pair of SFMs.

Test Case	Distance Values	Light Values
1	$s0 - s7 = 5$	$l0 - l7 = 5$
2	$s0 - s7 = 500$	$l0 - l7 = 250$
3	$s0 - s7 = 1000$	$l0 - l7 = 500$

Table 4.4: Test cases used in SFM calculation and their corresponding distance and light values.

We have used 20 categories to partition the individuals into classes based on the value of their SFM. Table 4.5 provides the classification of SFM values used in our experiments. Exponential scale was used for category specification because of the large range of SMF values. Population entropy value was calculated (see Equation 4.6) independently for each of the two fitness measures using the corresponding categories. The final entropy value was calculated as the average of the two entropy values from the two fitness measures.

The measure of population entropy is important since it correlates to the state of diversity in the population during a GP run. Drop in population entropy signifies a drop in population diversity. The ARL method tries to counteract the drops in

Category	SFM value from	SFM value to
0	-9,999,999,999	-1,000,000,000
1	-999,999,999	-100,000,000
2	-99,999,999	-10,000,000
3	-9,999,999	-1,000,000
4	-999,999	-100,000
5	-99,999	-10,000
6	-9,999	-1,000
7	-999	-100
8	-99	-10
9	-9	-1
10	0	9
11	10	99
12	100	999
13	1,000	9,999
14	10,000	99,999
15	100,000	999,999
16	1,000,000	9,999,999
17	10,000,000	99,999,999
18	100,000,000	999,999,999
19	1,000,000,000	9,999,999,999

Table 4.5: Categorization of standardized fitness measure values.

population entropy by creation of new individuals. The entropy measure was used to decide when the Subroutine Discovery step (see Figure 4.9) should be performed and a new epoch started.

We have used two methods to decide whether a new epoch should begin based on the entropy values. The first method was a simple *static* entropy threshold with a specified threshold value. The Adaptive Representation step of the algorithm was performed when the last calculated entropy value fell below the threshold. Our second method of deciding on a new epoch was by using a *dynamic* entropy threshold. The dynamic threshold did not use any fixed threshold values but a decrease in entropy over the last four consecutive generations initiated the AR step of the algorithm.

After the discovery of new subroutines, the function set contained new functions. The ARL method generated random individuals using the new function set. The

new individuals were placed in the intermediate population and fought for survival with the individuals in the population. The number of new individuals to create was specified by the replacement fraction parameter.

4.6.4 Population Generation

The initial population of chromosomes was created as for the tree-based method. Properties of the chromosomes were edited through the *Chromosome Structure Properties* dialog in the Simulator. The terminal set and the initial function set used for the ARL chromosomes was chosen from the primitives specified in Table 4.3.

Generation of chromosomes during the run of the evolutionary algorithm was similar to that of the initial population. The function set available for the new chromosomes contained already defined ARL functions stored in the function library. Thus, the newly created chromosomes could take advantage of the already evolved functions.

4.6.5 Chromosome Evaluation

Chromosome evaluation was done using the evaluation set as for the tree-based GP method. The defined ARL functions were inserted into the evaluation set in order to be called by special *ARFn* nodes in the program tree. The *ARFn* nodes invoked the evaluation of program trees of corresponding block objects. Subtrees of the *ARFn* nodes were evaluated and results used as parameters to the block object function calls. The result of the function call was returned as the result of the *ARFn* node evaluation.

4.6.6 Genetic Operators

Only the reproduction and crossover operators were used for the ARL chromosomes. Since the ARL chromosome program trees shared their structure with the tree-based GP chromosomes, the same crossover operator was used. Crossover was performed on the two fittest individuals of the tournament based on a specified probability. The reproduction operator was used if the crossover operator was not used or when the

trees generated by the crossover operator had height exceeding the maximum overall program tree height.

The function library local to each chromosome needed to be updated after invocation of the crossover operator. This was done to ensure each chromosome had knowledge of all the functions that it used either directly or indirectly. No update of the global function library was done.

4.7 Summary

In this chapter, we have presented the evolutionary methods we have used in our robotic controllers. First, we presented the evolutionary algorithm used in most of the methods. Then, we talked about the genetic programming methods using linear genome and tree-based chromosome representations. Following, we introduced three hierarchical genetic programming methods: Automatically Defined Functions, Module Acquisition, and Adaptive Representation through Learning. For each method, we have described chromosome structures, chromosome creation and evaluation, genetic operators, and other relevant information.

In Chapter 5 we will present our results of using the robotic controller implementations discussed in the current chapter. Performance with various learning tasks will be discussed and GP and HGP methods will be compared. Experimental settings of all parameters will be provided with the results of each experiment.

Chapter 5

Results

In this chapter, we present the results of our experiments. We have experimented with three learning tasks: obstacle avoidance, wall following, and light avoidance. Results for each learning task are presented in a corresponding section.

5.1 Obstacle Avoidance

The task of obstacle avoidance is very important for many real-world robotic applications. Any robotic exploratory behavior requires some degree of obstacle avoidance to detect and manoeuver around obstacles in the environment. In this research, obstacle avoidance was defined as robotic behavior steering the robot away from obstacles in the testing environment. For the Khepera robot, this task was equivalent to minimizing the values of the proximity sensors while moving about in the environment.

We have tested a variety of fitness functions and parameters in our experiments to find settings that provided best experimental performance. We have used a fitness function based on the work of Banzhaf et al. [9]. The function was composed of two opposite parts: pain and pleasure. The pleasure part of the fitness function was computed from motor values and encouraged the robot to move about in the environment using straight motion. The pain part was composed of sensor values and punished the robot for object proximity. The fitness function can be expressed as an equation:

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

where m_1 and m_2 are motor values and s_0 to s_7 are proximity sensor values. Unless otherwise stated, we have used a default value of 10 for α and default value of 1 for β in our experiments.

After the fitness calculation, the fitness value was saved in a fitness list of the individuals. Each individual kept up to 4 latest fitness values. Current fitness of an individual was calculated as the average of the stored fitness values. This calculation was performed in order to neutralize some of the effects of the dynamic fitness landscape.

The quality of the learned behavior and its training time is dependant on the structure of the training environment. In our research, we have used training environments that produced best results in test trials. Two training environments were used for the obstacle avoidance task and are shown in Figure 5.1. The *Nordin2* training environment was based on the training environment used by Nordin and Banzhaf in [49]. The *squares* environment was used to train some runs of the simulation but because of its variably sized squares, it was mainly used for testing learned obstacle avoidance behaviours.

Various behaviours of the robot were noticed while learning the obstacle avoidance task. We have subdivided the learned behaviours into groups based on the complexity and success rate of each behavior. The simplest behaviours were solely based on the movement of the robot anywhere in the environment. The *straight* behaviour expressed the robot moving straight forward whereas the backup behaviour expressed straight movement backwards. Curved behaviour defined non-straight movement either forward or backwards.

The simple behaviours did not take into consideration the sensors of the robot. The second level of behaviour was composed of circling, bouncing, and forward-backup behaviours. Circling represented spinning the robot in circles of various diameter and bouncing represented turning behaviour at the obstacle boundaries (usually touching the obstacle while turning). Forward-backup behaviour was a combination of forward with backup behaviours where the robot moved back and forth.

The highest level of behaviour was called sniffing and it represented the robot

Figure 5.1: Training environment *Nordin2* and training and testing environment *squares* used in obstacle avoidance experiments.

using its sensor data to detect and avoid an obstacle before touching it. Avoidance of obstacle was done either by circling away from it or by some backup behaviour. Various level of sniffing behaviour were observed. The perfect sniffing behaviour involved obstacle sniffing and straight motion behaviours that combined into smooth obstacle avoidance motion around the entire testing environment. Summary of the observed behaviours is provided in Figure 5.2.

5.1.1 Linear Genome GP

The first GP representation method studied was linear genome. Training was done using the *Nordin* (variation of *Nordin2*) environment and the default blank square environment. Experimental parameters used for the linear genome GP method in the task of obstacle avoidance are provided in Table 5.1.

Only a small variation in average size was noticed in the average size graphs from training runs. The entropy values oscillated around a fairly stable level during each

Figure 5.2: Summary of behaviours learned during experimentation with the Khepera robot.

Terminal Set:	Result Primitives: motor; temp
Function Set:	Variable Primitives: distance; temp
Maximum Code Height:	Constants: Integers in the range [0,8192]
Fitness Function:	ADD, SUB, MUL, SHL, SHR, XOR, OR, AND
Training Env.:	256
Testing Env.:	Equation 5.1 ($\alpha = 10, 100$)
Population Size:	Nordin, blank
Crossover Prob.:	squares
Mutation Prob.:	50
Tournament Size:	90%
	5%
	7

Table 5.1: Summary of parameters used during training of obstacle avoidance with the linear genome method.

Figure 5.3: Graphs of entropy and average size vs. the number of generations in a sample run O9a.

run. The typical observed changes in entropy and average size are presented in graphs of Figure 5.3.

Circling behaviours were observed starting from generation 20. Most runs developed some wall-bouncing behaviour, either 1 or 2 sided. Some backup behaviour was also observed. Wall-sniffing behaviour was noticed from generation 100. Few perfect behaviours were observed starting from generation 39. Sample trace run of the evolved perfect behaviour in the testing environment is shown in Figure 5.4.

Using a maximum constant integer value of 10 did not seem to produce different results than using the default maximum constant value of 8192. We suspect this was due to the additional shifting operators that could easily modify an integer value by few orders of magnitude. We have found that using a population of size 50 produced the best results because of the small size and stable entropy values during a run. Sample program code of an individual from a population that learned some obstacle avoidance behaviour is available in Figure A.1.

5.1.2 Tree-based GP

The second GP representation method studied was the canonical tree-based GP. Training was done using the *Nordin2* environment and the upper-left square of the *squares* environment. Experimental parameters used for the tree-based GP method in the task of obstacle avoidance are provided in Table 5.2.

With the half-and-half initial population creation method, a large drop in average

Figure 5.4: Trace run of perfect obstacle avoidance behaviour from run O8c.

size and entropy was quickly observed. With the full method, usually a gradual drop in average size was observed (from 60 to about 20 over more than 200 generations). Population entropy values gradually decreased but were above a reasonable level (0.6-1.0) until over generation 200. Thus, we have used the full method in our experiments since tests with the larger individuals created by the full method produced lower drops in entropy and average size. Plots of entropy and average size values over a typical run of the tree-based algorithm are shown in Figure 5.5.

Using the conditional *IFLTE* operator did not improve the results of the experiments. Population size of 200 seemed to provide the overall best results. Lower population sizes produced premature convergence. Population sizes of 500 and 1000 produced good results within a few generations (less than 50) but infrequently. Forward-backup behaviour seemed quite common with the large populations. This behaviour was probably due to the initial diversity of the large populations.

Simple circling behaviour was first noticed in generation 30. The most frequent

Terminal Set:	Var [s0-s7], Const [0-10]
Function Set:	Arithmetic: Add, Sub, Mul, Div; Logical: AND, OR, XOR, SHL, SHR; Conditional: IFLTE; 1 output: SetM1, SetM2
Maximum Creation Height:	6
Maximum Overall Height:	10
Number of Outputs:	2 (or 1)
Creation Method:	Full
Fitness Function:	Equation 5.1 ($\alpha = 1$)
Training Env.:	Nordin2; squares (upper-left)
Testing Env.:	squares
Population Size:	50, 100, 200, 500, 1000
Crossover Prob.:	90%
Mutation Prob.:	0%
Tournament Size:	7

Table 5.2: Summary of parameters used during training of obstacle avoidance with the tree-based method.

Figure 5.5: Graphs of entropy and average size vs. the number of generations in a sample run O36.

behaviour was one sided wall-bouncing. Wall-sniffing was observed from generation 40 (using ObsAvoid-nodiff fitness function) and generation 12 (using ObsAvoid-diff fitness function). Few perfect behaviours were found during the experiments. One evolved perfect behaviour, shown in Figure 5.6, used a backup strategy to turn around near corners without touching the wall of the test environment. Sample program code of an individual from a population that learned some obstacle avoidance behaviour is available in Figure A.2.

Figure 5.6: Trace run of perfect obstacle avoidance behaviour from run O36a.

Initial population creation was first done using an equal probability to select between a fixed constant terminal and variable terminals found in the terminal set. Since there were over 10 variable terminals, this probability produced trees with very few constant values. We believe that constant values are important in the program trees and thus we modified the initial population creation to create a constant terminal 30% of the time and a variable terminal 70% of the time. The value of 30% was chosen after some initial testing experiments.

For most of the experiments, we have used the terminal set composed of arithmetic operators. We have noticed that using constants from the range $[0,10]$ produces better results with the arithmetic operators than using constants from range $[0,8192]$. This was probably due to the fact that the only valid output values from program tree evaluation were in the range $[-10,10]$.

We have used terminal sets composed of logical operators in some of the runs to compare their performance with the terminal sets composed of arithmetic operators. From our tests, the logical operators did not perform as well as arithmetic operators yielding only wall-bouncing and simple wall-sniffing behaviours.

The average chromosome size values dropped sharply (from around 60 to around 20 within first 50-70 generations) when using the logical terminal set. After, the average size values remained fairly stable or dropped lower to values below 10. With a population size of 200, the entropy dropped below value 1.0 within 80 generations. However, with population size of 100, a similar drop was observed within 30 generations. Beyond the large drops, the entropy values dropped more slowly. Sample graphs of entropy and average size using the logical terminal set are shown in Figure 5.7.

Most of the tree-based GP (and HGP) experiments we performed used 2-output program trees. We have tested the other method of Symbolic Multiple Regression using one tree output and *SetMx* functions. We found that the entropy values dropped below 1.0 within 50 generations then usually remained stable at around value 0.6. Average size values remained quite stable or increased to a reasonably stable level. The initial average size of the population was half of the size used by the 2-output method due to the structure of the program trees. Sample plots of the runs with the one output method are shown in Figure 5.8.

The performance of the one tree output method did not improve on the performance of the 2-output method. Only simple wall-bouncing behaviours were noticed. Not much difference was noticed with the conditional *IFLTE* operator.

5.1.3 ADF HGP

The first HGP method tested was Automatically Defined Functions (ADF). Training was done using the *Nordin2* environment and the upper-left squares of the *squares*

Figure 5.7: Graphs of entropy and average size vs. the number of generations in a sample run O152.

environment. Experimental parameters used for the ADF method in the task of obstacle avoidance are provided in Table 5.3.

The entropy values dropped below 1.0 within the first 20 generations using one ADF with two arguments. Sometimes the entropy was seen to recover and stabilize around a low value. The average program tree size and SC decreased slowly to about value 50 (from initial value of 100) in about 200 generations. This decrease was smaller than the one in the tree-based GP method experiments. The average EC values started at about twice the values of the average size and almost converged with the average size values within 200 generations. Figure 5.9 shows typical plots of entropy, average size and SC, and average EC in experiments using one ADF per chromosome.

With two or three ADFs, the entropy values dropped below 1.0 within the first 10 generations or less. The average size values remained fairly constant within 200 generations. The average EC values behaved similar to the average size values but were initially 2 to 4 times higher. Sample graphs of the entropy, average size and SC,

Figure 5.8: Graphs of entropy and average size vs. the number of generations in a sample run O160.

RPB Terminal Set:	Var [s0-s7], Const [0-10]
RPB Function Set:	Add, Sub, Mul, Div
ADF Terminal Set:	ARGx, Const [0-10]
ADF Function Set:	Add, Sub, Mul, Div
Maximum Creation Height:	6
Maximum Overall Height:	10
Number of Outputs:	2
Creation Method:	Full
No. of ADFs:	1; 2; or 3
No. of Arguments:	2
Fitness Function:	Equation 5.1
Training Env.:	Nordin2; squares (upper-left)
Testing Env.:	squares
Population Size:	50, 100, 200
Crossover Prob.:	90%
Mutation Prob.:	0%
Tournament Size:	7

Table 5.3: Summary of parameters used during training of obstacle avoidance with the ADF HGP method.

and average EC behaviours with three ADFs are shown in Figure 5.10.

Figure 5.9: Graphs of entropy and average size, average SC, average EC vs. the number of generations in a sample run O52.

Figure 5.10: Graphs of entropy and average size, average SC, average EC vs. the number of generations in a sample run O66.

Simple circling and wall-bouncing behaviour was noticed from generation 15. Not much difference was observed between behaviours acquired using population of size 50, 100, or 200. Wall-sniffing was first noticed from generation 20 (one ADF), 10 (two ADFs), and 35 (three ADFs). Few perfect behaviours were observed starting from generation 16. Example trace of one perfect strategy observed is shown in Figure 5.11. Interesting sniffing and forward-backing behaviour was seen in generation 108 of one run and a trace of the behaviour is given in Figure 5.12. Sample program code of an individual with two ADFs from a population that learned some obstacle

Figure 5.11: Trace run of perfect obstacle avoidance behaviour from run O52a.

avoidance behaviour is available in Figure A.3.

5.1.4 MA HGP

The second HGP method tested was Module Acquisition (MA). Training was done using the *Nordin2* environment and the upper-left square of the *squares* environment. Experimental parameters used for the MA method in the task of obstacle avoidance are provided in Table 5.4.

Usually, the entropy values dropped below 1.0 by about generation 50. Most observations showed large drops in average size values (less than half the original size in less than 50 generations). Using 10% compression resulted in larger difference between average size and average EC values than with using 5% compression. Distinctions between average SC and average EC values were more noticeable with 10% compression.

Initial average EC values were equal to the average size values because of the lack of modules in the initial population. The average EC values grew to twice the size of average size values within 100 generations and eventually converged to the average size values in some instances. Figures 5.13 and 5.14 show plots of entropy, average

Figure 5.12: Trace run of perfect obstacle avoidance behaviour from run O63a.

Terminal Set:	Var [s0-s7], Const [0-10]
Function Set:	Add, Sub, Mul, Div, (and IFLTE)
Maximum Creation Height:	6
Maximum Overall Height:	10
Number of Outputs:	2
Creation Method:	Full
Fitness Function:	Equation 5.1
Training Env.:	Nordin2; squares (upper-left)
Testing Env.:	squares
Population Size:	50, 100, 200
Crossover Prob.:	90%
Mutation Prob.:	0%
Compression Prob.:	5%; or 10%
Expansion Prob.:	0.5%; or 1%
Tournament Size:	7

Table 5.4: Summary of parameters used during training of obstacle avoidance with the MA HGP method.

size, average SC, and average EC for typical runs using compression of 5% and 10%, respectively.

Figure 5.13: Graphs of entropy and average size, average SC, average EC vs. the number of generations in a sample run O76.

Figure 5.14: Graphs of entropy and average size, average SC, average EC vs. the number of generations in a sample run O88.

Using the conditional *IFLTE* operator did not seem to improve the results. More complex behaviours were achieved by training in the *squares* environment than in the *Nordin2* environment. This indicated that the learned behaviour was quite dependent on the training environment. Best results were achieved by using population size 200 (instead of 50 or 100). This might have been due to the large drops in entropy and program sizes evident especially in smaller populations.

Simple circling and wall-bouncing behaviours first appeared in generation 15 to

Figure 5.15: Trace run of perfect obstacle avoidance behaviour from run O94a.

30. Many wall-sniffing behaviours were found early - starting with generation 8 until generation 46. More sniffing behaviour was noticed with 10% compression. Perfect behaviour was noticed at generation 65 and consisted of a snake-like trail and a combination of wall-sniffing and backup near corners of the test environment as shown in Figure 5.15.

5.1.5 ARL HGP

The last HGP method studied was the Adaptive Representation with Learning (ARL) method. Training was done using the *Nordin2* environment and the upper-left square of the *squares* environment. Experimental parameters used for the ARL method in the task of obstacle avoidance are provided in Table 5.5.

First, we consider the static entropy threshold case with threshold value of 1.5. Little difference was found in plots of entropy and average size measurements using populations 50, 100, or 200. Average entropy values remained stable around a fixed value (about 1.1 for population size 50, about 1.4 for 100, and about 1.6 for 200) until at least generation 100.

Average size values remained fairly constant for at least 100 generations. Average

Terminal Set:	Var [s0-s7], Const [0-10]
Function Set:	Add, Sub, Mul, Div
Maximum Creation Height:	6
Maximum Overall Height:	10
Number of Outputs:	2
Creation Method:	Full
Fitness Function:	Equation 5.1
Training Env.:	Nordin2; squares (upper-left)
Testing Env.:	squares
Population Size:	50, 100, 200
Crossover Prob.:	90%
Mutation Prob.:	0%
Entropy Threshold:	static (1.5); static (1.0); dynamic
Default Utility W:	3
Replacement Fraction:	20%
Block Height:	fixed (3)
Tournament Size:	7

Table 5.5: Summary of parameters used during training of obstacle avoidance with the ARL HGP method.

EC values behaved similar to the average size value but were 2 to 10 times higher. The average SC values remained between the EC and size values, sharing a similar behaviour. Function creation initiated at around generation 10. The plot of number of functions correlated with the three average sizes plots. The number of functions did not grow monotonically over the entire run but showed intervals of increase and decrease. Sample graphs of entropy, average size, average SC, average EC, and number of functions are provided in Figure 5.16.

Simple circling and wall-bouncing behaviours started to appear from generation 6 to 90. Wall-sniffing was noticed quite often starting with gen 5 to 70. Perfect behaviour was not noticed in our tests. It seemed that good results were easier to achieve with the ARL method but they were also easier to lose.

Next, we look at the static entropy threshold case with threshold value of 1.0. The entropy values behaved quite stable but below 1.0 during most of our experiments. Function creation usually started at around generation 15.

The number of functions created was similar to the case with entropy threshold of 1.5. Average size values were usually stable and increasing. The plots of average

Figure 5.16: Graphs of entropy and average size, average SC, average EC, and number of functions vs. the number of generations in a sample run O118.

SC and average EC separated from the plot of average size when new functions were first discovered. Typical plots of entropy, average size, average SC, average EC, and number of function changes are provided in Figure 5.17.

Simple circling or wall-bouncing behaviour was noticed from generation 6 to 50. Some wall-sniffing behaviour was observed at generation 8 to 55. No perfect behaviour was noticed.

Finally, we take a look at experiments with the dynamic entropy threshold. In all the runs, a large drop in entropy was noticed. The peak number of functions found was overall a few times smaller than that of the static methods. The average SC and average EC values were very close to the values of average size. Sample plots of entropy, average size, average SC, average EC, and number of functions are found in Figure 5.18. The only observed behaviour was wall-bouncing starting at generation 12 to 29.

Figure 5.17: Graphs of entropy and average size, average SC, average EC, and number of functions vs. the number of generations in a sample run O122.

Overall, we observed that the dynamic entropy threshold did not perform well. A static threshold with threshold value of 1.5 seemed to provide the best results and to keep the entropy around a constant level. Population size of 50 provided the best overall behavioural results. Sample program code of an individual from a population that learned some obstacle avoidance behaviour is available in Figure A.4. The code contains two functions containing simple arithmetic operations; such functions were common in populations exhibiting good behaviour in our experiments.

5.1.6 Random Test

To ensure the validity of the fitness function we have chosen for our experiments, we have run some tests with a random fitness function that returned a random number

Figure 5.18: Graphs of entropy and average size, average SC, average EC, and number of functions vs. the number of generations in a sample run O129.

from range $[0,1000]$. The experimental setup was exactly the same as the real experiments with our obstacle avoidance fitness function. We have only used the GP methods of linear genome and tree-based representation in these experiments.

The entropy values using the random fitness function oscillated around a constant value until at least generation 200. This shows us that the population was not converging quickly to any solution. The graphs of average size varied greatly in each experiment and cannot be concluded on. The only useful behaviour found was temporary wall-bouncing probably caused by the turning motion of the motors in some individuals of the population.

We have expected quite randomized behaviour with some learned behaviour. The results proved our expectations. Sample plots of entropy and average size are provided in Figure 5.19.

Figure 5.19: Graphs of entropy and average size vs. the number of generations in a sample run R10.

5.1.7 Task Summary

To summarize the results of experiments with the obstacle avoidance task, we compared the five representation methods in terms of run statistics and evolved behaviours.

The representation method with the most stable entropy values was the ARL method with a static threshold value of 1.5. The linear genome and ADF methods also provided long, stable entropy values but with larger variations. The MA and tree-based representations provided the worst stability with large drops of entropy values. Most stability in the average chromosome size values was seen with the linear genome method. Among the HGP methods, the most stable method was the ARL method with a static threshold of 1.5. The worst average size value stability was seen with the MA method.

We have used the highest level of behaviour (sniffing behaviour) to calculate average generation values of first occurrence of the stated behaviour. We have not used the simple behaviour categories because they contained behaviours not directly applicable to the studied task. Summary of the results of our behaviour calculation can be found in Figure 5.20. The figure shows minimum, maximum, and average generation values for each representation method. The method with best (smallest) minimum, maximum, and average values was the ARL HGP method. The method with worst (largest) minimum, maximum, and average values was the linear genome GP method.

Overall, the HGP methods performed quite well and comparable with the tree-based GP method. Note that the generation values were taken from experiments with various experimental settings.

Figure 5.20: Graphs of minimum, maximum, and average generations of first detection of good obstacle avoidance behaviour for each chromosome representation method.

5.2 Wall Following

The task of wall following allows the robot to perform more difficult and interesting behaviours such as maze navigation. The purpose of the wall following task was to teach the robot to walk around the boundaries of obstacles with a certain desirable distance from the boundaries. The learned task should have included some obstacle avoidance behaviour; however, that was not the main requirement of the experiments.

The wall following fitness function was composed of two parts: sensor part and motor part. The sensor part computed a sensor value from a subset of the robotic sensor values. The motor part was calculated by computing an absolute motor sum minus the absolute value of the difference. The fitness function is provided in Figure 5.21. In our experiments, unless stated otherwise, we have used the following values for the free parameters of the fitness function: $\forall_i a_i = 1$, $\alpha = 100$, $\beta = 1$. Last 4 fitness values of an individual were used to calculate the individual fitness average value as in the obstacle avoidance task.

Only six sensors ($s_0 - s_5$) were used in calculating the sensor part of the fitness

Wall Following Fitness Function

```

Left =  $a_0 \cdot s_0 + a_1 \cdot s_1 + a_2 \cdot s_2$ ;
Right =  $a_5 \cdot s_5 + a_4 \cdot s_4 + a_3 \cdot s_3$ ;

if (Right > 1023)
    RightSensorPart = 1000 - Right;
else if (Right < 20)
    RightSensorPart = (1000/20) * Right;
else
    RightSensorPart = 1000;

if (Left > 1023)
    LeftSensorPart = 1000 - Left;
else if (Left < 20)
    LeftSensorPart = (1000/20) * left;
else
    LeftSensorPart = 1000;

MotorPart =  $|m_1| + |m_2| - |m_1 - m_2|$ ;
Fitness =  $\alpha \cdot \text{MotorPart} + \beta \cdot (\text{RightSensorPart} + \text{LeftSensorPart})$ ;

```

Figure 5.21: Fitness Function for the wall following task.

calculation. The sensors represented the side and front sensors of the robot. The calculated sensor part value acted as either pleasure or pain depending on the values of the three sensors on the right and left sides of the robot. Thus, the robot was punished when it was either too far away from a wall or too close to it. The sensor part value of three right sensors was calculated separately from the sensor part value of the three left sensors. A graph of the sensor part values over the raw sensor range for three side sensors (either left or right) is shown in Figure 5.22.

We have used the *challways* world for the training environment for the wall following task. The environment consisted of a long, straight stretch of corridor and curved environment boundaries. We did not place too many straight wall edges in the environment since manoeuvring through jagged edges was not in our task definition and the robot seemed to learn better with soft edges. The *challways* world

Figure 5.22: Plot of sensor part values vs. the raw sensor values of three side sensors used in wall following fitness function.

was also used for a testing environment of some behaviours but the main testing environment was the *maze1* world. The *maze1* world contained a soft edged corridor maze that allowed testing of complex maze navigation behaviour. The training and testing environments are shown in Figure 5.23.

We have observed a variety of behaviours during our experimentation. Summary of the behaviours is provided in Figure 5.2. We have placed the behaviours into categories based on their relative performance and success. The first category was of poor wall following behaviour and it consisted of simple wall-bouncing and circling behaviour. The category of good wall following behaviour consisted of wall-sniffing and some maze following behaviour. The best behaviour category consisted of perfect maze following behaviour without wall touching and usually using both sensor sides of the robot.

5.2.1 Linear Genome GP

The first GP method studied was linear genome GP. Experimental parameters used for the linear genome GP method in the task of wall following are provided in Table 5.6.

In our experiments, we have noticed large oscillations in entropy and but also some stability around average value of 1.0 by at least generation 200. The average size oscillated but remained at a fairly constant level (averaging about 120). Sample

Figure 5.23: Training environment *challways* and testing environment *maze1* used in wall following experiments.

plots of the entropy and average height behaviour are shown in Figure 5.24.

The most frequent behaviour was wall bouncing. Some maze following behaviour was seen at generation 11 to 77. Perfect maze following behaviour was observed at generation 8 to 90. Overall, the results were very good. Best results were achieved by using population of size 50. A sample trace of evolved perfect maze following behaviour is shown in Figure 5.25. Sample program code of an individual from a population with wall following behaviour is available in Figure A.5.

5.2.2 Tree-based GP

The second GP method considered was the canonical tree-based GP method. Experimental parameters used for the tree-based GP method in the task of wall following are provided in Table 5.7.

With population of size 50, entropy dropped below value 1.0 by about generation 20. Using population of size 100, the same entropy drop was observed at about

Terminal Set:	Result Primitives: motor; temp
Function Set:	Variable Primitives: distance; temp
Maximum Code Height:	Constants: Integers in the range [0,8192]
Fitness Function:	ADD, SUB, MUL, SHL, SHR, XOR, OR, AND
Training Env.:	256
Testing Env.:	Figure 5.21
Population Size:	challways
Crossover Prob.:	maze1
Mutation Prob.:	50, 100
Tournament Size:	90%
	5%
	7

Table 5.6: Summary of parameters used during training of wall following with the linear genome GP method.

Figure 5.24: Graphs of entropy and average size vs. the number of generations in a sample run W5.

Figure 5.25: Trace run of perfect maze-following behaviour from run W8a.

Terminal Set:	Var [s0-s7], Const [0-10]
Function Set:	Arithmetic: Add, Sub, Mul, Div; Logical: AND, OR, XOR, SHL, SHR; Conditional: IFLTE;
Maximum Creation Height:	6
Maximum Overall Height:	10
Number of Outputs:	2
Creation Method:	Full
Fitness Function:	Figure 5.21 ($\alpha = 10$)
Training Env.:	challways
Testing Env.:	maze1
Population Size:	50, 100, 200
Crossover Prob.:	90%
Mutation Prob.:	0%
Tournament Size:	7

Table 5.7: Summary of parameters used during training of wall following with the tree-based GP method.

generation 60. Thus, to keep entropy values high, we used population sizes of 100 or 200. The average size seemed to decrease faster than the entropy but the results varied with each run. To minimize the large drops of entropy and average size, we used a value of 10 for the β variable in the fitness function (see Figure 5.21). This provided larger emphasis on the sensor part of the fitness than on the motor parts. Sample plots of the entropy and average size are provided in Figure 5.26.

Figure 5.26: Graphs of entropy and average size vs. the number of generations in a sample run W25.

Using the conditional *IFLTE* operator seemed to keep entropy quite stable but did not show behavioural improvements. Using logical operators in the function set did not seem to improve the results of using arithmetic operators. Best results were observed using population of size 100.

Simple circling and wall-bouncing behaviour was noticed at generation 15 to 60. Some maze following or wall-sniffing behaviour was observed at generation 6 to 53. Perfect maze following was only seen in one run at generation 205. Sample program code of an individual from a population that learned some wall following behaviour

RPB Terminal Set:	Var [s0-s7], Const [0-10]
RPB Function Set:	Add, Sub, Mul, Div
ADF Terminal Set:	ARGx, Const [0-10]
ADF Function Set:	Add, Sub, Mul, Div
Maximum Creation Height:	6
Maximum Overall Height:	10
Number of Outputs:	2
Creation Method:	Full
No. of ADFs:	1; 2
No. of Arguments:	2
Fitness Function:	Figure 5.21 ($\alpha = 10$)
Training Env.:	challways
Testing Env.:	maze1
Population Size:	50, 100
Crossover Prob.:	90%
Mutation Prob.:	0%
Tournament Size:	7

Table 5.8: Summary of parameters used during training of wall following with the ADF HGP method.

is available in Figure A.6.

5.2.3 ADF HGP

The first HGP method studied was the Automatically Defined Functions (ADF) method. Experimental parameters used for the ADF method in the task of wall following are provided in Table 5.8.

With population of size 50, the entropy value dropped below 1.0 within generation 30. The entropy value was kept above 1.0 within 100 generations using population of size 100. Using additional ADFs did not seem to affect the entropy drop. Average size seemed to decrease slowly initially but decreased noticeably after about generation 100. The initial average EC values were about 3 times higher than the initial average size values when using 1 ADF. The average EC values converged back to the average size as early as generation 60. With 2 ADFs, average EC values started at about 4 to 5 times the values of average size and behaved similar as for the 1 ADF case. Sample plots of entropy, average size and SC, and average EC are shown in Figure 5.27.

Figure 5.27: Graphs of entropy and average size, average SC, and average EC vs. the number of generations in a sample run W35.

Best results were observed with populations of size 50 and 1 ADF with 2 arguments. Additional ADFs did not seem to produce better results. Simple wall-bouncing or circling behaviour was first noticed at generation 10 to 50. Some maze following behaviour was observed at generation 15 to 97. Perfect maze-following was only seen once at generation 135. Sample program code of an individual with one ADF from a population that learned some wall following behaviour is available in Figure A.7.

5.2.4 MA HGP

The second HGP method considered was the Module Acquisition (MA) method. Experimental parameters used for the MA method in the task of wall following are provided in Table 5.9.

Experiments with population of size 100 showed nice entropy variation that usually dipped under value 1.0 after generation 100. The average size values decreased quickly,

Terminal Set:	Var [s0-s7], Const [0-10]
Function Set:	Add, Sub, Mul, Div
Maximum Creation Height:	6
Maximum Overall Height:	10
Number of Outputs:	2
Creation Method:	Full
Fitness Function:	Figure 5.21 ($\alpha = 10$)
Training Env.:	challways
Testing Env.:	maze1
Population Size:	50, 100, 200
Crossover Prob.:	90%
Mutation Prob.:	0%
Compression Prob.:	10%
Expansion Prob.:	1%
Tournament Size:	7

Table 5.9: Summary of parameters used during training of wall following with the MA HGP method.

halving the initial values by generation 100. The values were below 10 by generation 140. Average SC and average EC values started the same as average size, diverged slightly during the run of the algorithm, and converged back to the average size values by generation 140. Sample plots of entropy, average size, average SC, and average EC are available in Figure 5.28.

The best results were achieved with population of size 100. We have used compression value of 10% for all the experiments. Some simple wall-following or simple maze-following behaviour was observed at generation 11 to 242. No perfect maze-following behaviour was found but interesting near perfect behaviour with backup and wall-sniffing was found in generation 59. A trace of this interesting behaviour is shown in Figure 5.29. Sample program code of an individual from a population that learned some wall following behaviour is available in Figure A.8. Most functions from good performing individuals were simple arithmetic functions.

5.2.5 ARL HGP

The last HGP method studied was the Adaptive Representation with Learning (ARL) method. Experimental parameters used for the ARL method in the task of wall

Figure 5.28: Graphs of entropy and average size, average SC, and average EC vs. the number of generations in a sample run W57.

Figure 5.29: Trace run of near-perfect maze-following behaviour from run W58a.

Terminal Set:	Var [s0-s7], Const [0-10]
Function Set:	Add, Sub, Mul, Div
Maximum Creation Height:	6
Maximum Overall Height:	10
Number of Outputs:	2
Creation Method:	Full
Fitness Function:	Figure 5.21 ($\alpha = 10$)
Training Env.:	challways
Testing Env.:	maze1
Population Size:	50
Crossover Prob.:	90%
Mutation Prob.:	0%
Entropy Threshold:	static (1.0); static (1.5); dynamic
Default Utility W:	3
Replacement Fraction:	20%; 50%
Block Height:	fixed (3)
Tournament Size:	7

Table 5.10: Summary of parameters used during training of wall following with the ARL HGP method.

following are provided in Table 5.10.

First, we consider the case with static entropy threshold with a threshold value of 1.0. The entropy remained at around value 1.2 until at least generation 50 then it slowly dropped down to 0 at about generation 70. The average size usually dropped down steeply after generation 15 but has been observed to recover slightly in certain runs.

Average EC values remained the same as average size until functions were first found by about generation 50. The average EC values did not seem to increase more than twice of the corresponding average size values. Average SC values stayed between the average SC and average size values. The number of functions was observed to be quite low (less than 10) before entropy dropped to 0. For plots of entropy, average size, average SC, average EC, and function count, see Figure 5.30.

Next, we consider the static entropy threshold with threshold value of 1.5. Entropy remained at around value 1.4 but was also observed in some runs to quickly fall down to 0. Average size was seen to initially increase for about 30 generations and then slowly decrease. Average SC and average EC values followed the trend of the average

Figure 5.30: Graphs of entropy, average size, average SC, average EC, and number of functions vs. the number of generations in a sample run W73.

size values with values 2 to 3 times larger. Eventual convergence back to the average size values was seen in some runs.

Function creation began early, before generation 10. Functions seemed to number up to 120 but function count increased and decreased with time. The three size graphs mirrored behaviour seen in the number of functions graph. Sample of entropy, three average sizes, and number for functions plots are shown in Figure 5.31. With the replacement factor of 0.5 (instead of the default 0.2), the entropy values and size values were kept around a constant value. Function counts increased and decreased

with a high of 250. No good behaviours were noticed with the high replacement factor.

Figure 5.31: Graphs of entropy, average size, average SC, average EC, and number of functions vs. the number of generations in a sample run W74.

Finally, we cover the dynamic entropy threshold. The entropy was noticed to drop quite evenly and quickly and reached 0 by generation 100. Average size values remained close to the starting value until about generation 50, then slowly dropped. Average SC and average EC values mirrored the behaviour of average size with values up to twice of the average size. The number of functions noticed was quite small at

less than 10.

With static entropy threshold of 1.0, some maze following behaviour was first seen from generation 22 to 43. Perfect maze following was found once in generation 28. With threshold of 1.5, some maze following was found in generation 25 to 32 and no perfect behaviour was found. With the dynamic threshold function, no maze or wall following behaviour was observed. From our experiments, the best behaviour was found in experiments using the static threshold with threshold value of 1.0. We have only used population sizes of 50 due to a shorter simulation time and our previous results with the ARL method. Sample program code of an individual from a population that learned some wall following behaviour is available in Figure A.9 and shows a characteristic simple arithmetic function.

5.2.6 Random Test

As in the obstacle avoidance case, we have tried to verify the usefulness of our fitness function by using a randomized fitness function. We have only performed experiments with the linear genome and tree-based chromosome representations. All other parameters were kept as in original experiments.

We have found that the entropy either oscillated around a constant value for at least 200 generations or dropped slowly. Large drops in average size values were noticed in all experiments. No useful wall following behaviour was observed in the tests.

5.2.7 Task Summary

To summarize the results of experiments with the wall following task, we compared the five representation methods in terms of run statistics and evolved behaviours.

The most stable entropy behaviour was noticed in experiments using the ARL HGP method. The worst stable entropy behaviour was noticed using the ADF method and included a large initial drop of entropy to a stable level. Most stability of average size values was also seen in the linear genome GP method. Relatively good stability was also found in the ARL HGP method with static threshold value of 1.5. The largest drops in average chromosome size were noticed with the MA method and the

ARL method with a static threshold value of 1.0.

We have used the good and best behaviour categories to calculate average generation values of first occurrence of the stated behaviour. We have not used the first, poor behaviour category because the category contains behaviours not directly applicable to the studied task. Summary of our behaviour calculation can be found in Figure 5.32. The Figure shows minimum, maximum, and average generation values for each representation method. The ARL method produced the best average and maximum results with the smallest difference between the minimum and maximum values. The best minimum generation results came from the tree-based method. The worst performance was seen using the MA method with the largest maximum and average values. The largest minimum value belonged to the ARL method.

Figure 5.32: Graphs of minimum, maximum, and average generations of first detection of good wall following behaviour for each chromosome representation method.

5.3 Light Avoidance

The light avoidance task is similar to the obstacle avoidance task but uses the ambient light sensors of the robot instead of the proximity sensors. Lights used in the training and testing environments are overhead lamps that cannot be touched by the robot. The robot must learn to stay inside an unlit section of the world environment while moving as much as possible.

The fitness function used for light avoidance was similar to the fitness function

used for obstacle avoidance. The function contained a pleasure part computed from the motor values of the robot and a pain part computed from the light sensors. Proximity sensors were not part of the fitness function. A formal definition of the function is given as:

$$Fitness = \alpha(|m_1| + |m_2| - |m_1 - m_2|) - \beta(4000 - \sum_{i=0}^7 l_i) \quad (5.2)$$

where m_1 and m_2 are motor values and l_0 to l_7 are ambient light sensor values. Unless otherwise stated, we have used a default value of 10 for α and default value of 1 for β in our experiments. Because of the definition of light sensor values (with 0 as maximum light and 500 as minimum), we used the value of 4000 (8 sensor times 500 value) to make the fitness function behave similar to the fitness function for obstacle avoidance. As with the other tasks, up to 4 latest fitness values of an individual were used to calculate the final fitness value of the individual.

The training environment *hider-seeker3* was composed of a rectangle of darkness surrounded by lights and a circular light island in the middle of the darkness area. The testing environment *box-light* was composed of a similar dark area rectangle without the middle island. The training and test environments can be seen in Figure 5.33.

As with other tasks, we have subdivided the learned behaviours of the robots into two categories. The *Some LAB* category of behaviour consisted of some light avoidance behaviour (LAB) usually composed of circular, oval or uneven robot maneuvers with some degree of light detection and avoidance. The *Nice LAB* behaviour category consisted of nice light avoidance behaviour with definite light detection and avoidance. Nice LAB motion usually consisted of travelling around the boundary of the dark area in the testing environment. Summary of possible behaviours can be found in Figure 5.2.

5.3.1 Linear Genome GP

The first GP method studied was linear genome GP. Experimental parameters used for the linear genome GP method in the task of light avoidance are provided in Table 5.11.

Figure 5.33: Training environment *hider-seeker3* and testing environment *box-light* used in light avoidance experiments.

We have observed the same entropy behaviour using populations of size 50, 100, and 200. Entropy was stable around a value above 1.0 for over 100 generations. The average size values were relatively stable, oscillating around a fixed value. Sample entropy and average size plots are shown in Figure 5.34.

Sample average fitness plot is provided in Figure 5.35. From the graph in Figure 5.35, it is evident that the average fitness is not a good indication of a convergence of the population to a solution for a problem using real-time dynamic fitness calculations. The fitness provides the evolution engine with information to decide between different individuals in a similar state of the environment. Individuals with different robot positions in the environment cannot be compared using such a fitness measure.

Some LAB category behaviour was first noticed in generation 82 to 143. Nice LAB category behaviour was noticed from generation 96 to 227. Using a maximum constant of 100 (instead of the default 10) provided overall worse behavioural performance. Good behaviour was observed with population sizes of either 50 or 100. Sample trace of Nice LAB behaviour is provided in Figure 5.36. Sample program code of an

Terminal Set:	Result Primitives: motor; temp Variable Primitives: light; temp Constants: Integers in the range [0,10]
Function Set:	ADD, SUB, MUL, SHL, SHR, XOR, OR, AND
Maximum Code Height:	256
Fitness Function:	Equation 5.2
Training Env.:	hider-seeker3
Testing Env.:	box-light
Population Size:	50, 100, 200
Crossover Prob.:	90%
Mutation Prob.:	5%
Tournament Size:	7

Table 5.11: Summary of parameters used during training of light avoidance with the linear genome GP method.

Figure 5.34: Graphs of entropy and average size vs. the number of generations in a sample run H4.

Figure 5.35: Graph of average fitness vs. the number of generations in a sample run H5.

individual from a population that learned Some LAB category behaviour is available in Figure A.10.

5.3.2 Tree-based GP

The second GP method studied was tree-based GP. Experimental parameters used for the tree-based GP method in the task of light avoidance are provided in Table 5.12.

With population of 50 individuals, the entropy value dropped quickly. With 100 or 200 individuals, entropy seemed rather stable until at least generation 100 or slightly decreased over time. Average size values were nicely correlated with the entropy values and thus were fairly stable when the entropy value was stable. Experiments with the conditional IFLTE function produced sharp entropy drops below value 1.0 within 50 generations. Using logical operators in the function set produced plots similar to the arithmetic operators. Sample graphs of entropy and average size are provided in Figure 5.37.

Best results using the tree-based representation method were seen with populations of size 100. The arithmetic function set with conditional IFLTE and the logical function set did not seem to improve the results over the basic arithmetic function set. Some LAB category behaviour was first noticed from generation 29 to 88. Nice LAB category behaviour was observed at generation 46 to 129. Sample program code

Figure 5.36: Trace run of nice light avoidance behaviour from run H6a.

Terminal Set:	Var [10-17], Const [0-10]
Function Set:	Arithmetic: Add, Sub, Mul, Div; Logical: AND, OR, XOR, SHL, SHR; Conditional: IFLTE;
Maximum Creation Height:	6
Maximum Overall Height:	10
Number of Outputs:	2
Creation Method:	Full
Fitness Function:	Equation 5.2
Training Env.:	hider-seeker3
Testing Env.:	box-light
Population Size:	50, 100, 200
Crossover Prob.:	90%
Mutation Prob.:	0%
Tournament Size:	7

Table 5.12: Summary of parameters used during training of light avoidance with the tree-based GP method.

Figure 5.37: Graphs of entropy and average size vs. the number of generations in a sample run H23.

of an individual from a population that learned Some LAB category behaviour is available in Figure A.11.

5.3.3 ADF HGP

The first HGP method studied was the Automatically Defined Functions (ADF) method. Experimental parameters used for the ADF method in the task of light avoidance are provided in Table 5.13.

The entropy value dropped down considerably within 100 generations. Average size remained fairly stable but was seen to drop over time. Average EC values 2 to 3 times the values of average size (and average SC). The tendency of the average EC plot to converge to the plot of average size was noticed in higher generations. Experiments with additional ADFs provided similar entropy and average size behaviours to ones observed with 1 ADF. Sample plots of entropy, average size and SC, and average EC are provided in Figure 5.38.

Best results were observed with population of size 100 and 1 ADF with 2 arguments. Some LAB category behaviour was found in generation 47 to 111. Nice LAB

RPB Terminal Set:	Var [10-17], Const [0-10]
RPB Function Set:	Add, Sub, Mul, Div
ADF Terminal Set:	ARGx, Const [0-10]
ADF Function Set:	Add, Sub, Mul, Div
Maximum Creation Height:	6
Maximum Overall Height:	10
Number of Outputs:	2
Creation Method:	Full
No. of ADFs:	1; 2; or 3
No. of Arguments:	2
Fitness Function:	Equation 5.2
Training Env.:	hider-seeker3
Testing Env.:	box-light
Population Size:	50, 100, 200
Crossover Prob.:	90%
Mutation Prob.:	0%
Tournament Size:	7

Table 5.13: Summary of parameters used during training of obstacle avoidance with the ADF HGP method.

category behaviour was observed starting from generation 47 to 71. Some results clearly showed adaptation to the training environment with navigation behaviour around the middle island and produced strange behaviour in the testing environment without the island. A trace of interesting Nice LAB behaviour learned using the ADF representation is shown in Figure 5.39. Sample program code of an individual from a population that learned Some LAB category behaviour is available in Figure A.12.

5.3.4 MA HGP

The second HGP method considered was the Module Acquisition (MA) method. Experimental parameters used for the MA method in the task of light avoidance are provided in Table 5.14.

Entropy remained fairly stable but was also observed to drop abruptly in some runs. Average size dropped within 50 generations to about a value of 20 (from over 60). Average EC behaved similar to average size with equal initial values. The EC values diverged to about twice the average size values, and eventually converged

Figure 5.38: Graphs of entropy and average size vs. the number of generations in a sample run H43.

again with drops in entropy. Average SC values were either the same as average EC values or slightly below. Entropy values dropped quickly using 20% probability of compression (and 5% probability of expansion). With 5% probability of compression (0.5% probability of expansion), entropy values behaved the same as with the default 10% probability. Sample plots of entropy, average size, average SC, and average EC are shown in Figure 5.40.

Best results were observed using population of size 50, 10% probability of compression, and 1% probability of expansion. Some LAB category behaviour was first noticed in generation 54 to 99. Nice LAB category behaviour was observed in generation 23 to 69. Adaptation to the middle island of the training environment is clearly seen in the traces of Figure 5.41. Sample program code of an individual from a population that learned Some LAB category behaviour is available in Figure A.13.

Figure 5.39: Trace run of nice light avoidance behaviour from run H43a.

5.3.5 ARL HGP

The last HGP method studied was the Adaptive Representation through Learning (ARL) method. Experimental parameters used for the ARL method in the task of light avoidance are provided in Table 5.15.

First, we consider the static threshold case with a threshold value of 1.0. The entropy remained fairly stable but often dropped below a value of 1.0. Similar entropy behaviour was observed with populations of size 50 and 100. Average size values remained stable until at least generation 100 and sometimes increased slightly over time. Average EC followed the trends of average size with 3 to 8 times the value at peak. Average SC shared the behaviour of average size with a value between average size and average EC.

Function production started at generation 10. Function count was large at peak (up to 300) but the value increased and decreased over each run. Sample plots of entropy, average size, average SC, average EC, and number of functions are shown in Figure 5.42.

Best results were found with population size of 100. Some LAB category behaviour was noticed at generation 7 to 57. Nice LAB category behaviour was first seen

Terminal Set:	Var [10-17], Const [0-10]
Function Set:	Add, Sub, Mul, Div
Maximum Creation Height:	6
Maximum Overall Height:	10
Number of Outputs:	2
Creation Method:	Full
Fitness Function:	Equation 5.2
Training Env.:	hider-seeker3
Testing Env.:	box-light
Population Size:	50, 100
Crossover Prob.:	90%
Mutation Prob.:	0%
Compression Prob.:	10%; 20%; or 5%
Expansion Prob.:	1%; 5%; or 0.5%
Tournament Size:	7

Table 5.14: Summary of parameters used during training of light avoidance with the MA HGP method.

Figure 5.40: Graphs of entropy and average complexity measures vs. the number of generations in a sample run H63.

Figure 5.41: Trace runs of nice light avoidance behaviour in different environments from run H64a.

Terminal Set:	Var [10-17], Const [0-10]
Function Set:	Add, Sub, Mul, Div
Maximum Creation Height:	6
Maximum Overall Height:	10
Number of Outputs:	2
Creation Method:	Full
Fitness Function:	Equation 5.2
Training Env.:	hider-seeker3
Testing Env.:	box-light
Population Size:	50, 100
Crossover Prob.:	90%
Mutation Prob.:	0%
Entropy Threshold:	static (1.0); static (1.5); dynamic
Default Utility W:	3
Replacement Fraction:	20%
Block Height:	fixed (3)
Tournament Size:	7

Table 5.15: Summary of parameters used during training of light avoidance with the ARL HGP method.

Figure 5.42: Graphs of entropy, average size, average SC, average EC, and number of functions vs. the number of generations in a sample run H85.

in some runs at generation 9 to 39. Figure 5.43 shows a trace of interesting Nice LAB behaviour. Figure 5.44 shows strange behaviour in the testing environment but proper behaviour in the training environment due to over-adaptation to the training environment.

For the case of static entropy threshold with a value of 1.5, the behaviour of entropy, sizes, and function counts was similar to that of experiments with threshold value of 1.0. Entropy values were usually higher in the runs with threshold value of 1.5. Best results were observed using population of size 50. Some LAB category behaviour was seen from generation 10 to 78. Nice LAB category behaviour was found once at generation 28.

Using the dynamic entropy threshold calculation provided no useful results. Entropy was seen to drop abruptly with low function production.

For the light avoidance task, static threshold with threshold value of 1.0 and population of size 100 provided the overall best results. Some results with low generation

Figure 5.43: Trace run of interesting nice light avoidance behaviour from run H88a.

Figure 5.44: Trace runs of strange light avoidance behaviour in testing environment (box-lights) and nice behaviour in training environment (hider-seeker3) from run H92a.

of first detection did not use any functions since no functions were yet created. Sample program code of an individual from a population that learned Some LAB category behaviour is available in Figure A.14. The sample program code shows six characteristic functions using some combination of the division and addition operators.

5.3.6 Random Test

Our tests using a random fitness function and other test parameters equal to those in regular experiments yielded quite random results. Tests were only run using linear genome and tree-based chromosome representations. The entropy value oscillated at a fixed level to at least generation 300. Big decreases in average size measures were often seen and no useful light avoidance behaviour was noticed.

5.3.7 Task Summary

To summarize the results of experiments with the light avoidance task, we compare the five representation methods in terms of run statistics and evolved behaviours.

The most stable entropy behaviour was noticed with the linear genome method. The worst stable entropy behaviour was observed with the ADF method. Most stable average size values were noticed using the ARL method. The linear genome and tree-based representations also provided quite stable average size behaviour. The worst average size behaviour was seen with the MA representation method.

We have used the Some LAB and Nice LAB categories to calculate average generation values of first occurrence of the stated behaviour. Summary of our behaviour calculation results can be found in Figure 5.45. The Figure shows minimum, maximum, and average generation values for each representation method. The best (lowest) minimum, maximum, and average values came from experiments using the ARL method. The worst (highest) minimum, maximum, and average values were from linear genome experiments. The HGP methods performed very well and comparable to the tree-based method.

Figure 5.45: Graphs of minimum, maximum, and average generations of first detection of good light avoidance behaviour (LAB) for each chromosome representation method.

5.4 Summary

In this chapter, we have provided our experimental results. Three learning tasks were discussed: obstacle avoidance, wall following, and light avoidance. For each learning task, we have outlined the results of each of the five chromosome representation methods studied. We have supplemented our result descriptions with graphs of run statistics and application screen-shots with traces of learned behaviours.

The next chapter will conclude our research by summarizing our overall results and offering explanations of observed behaviours. We will also discuss future research in the area and improvements to the studied algorithms.

Chapter 6

Conclusions

In this chapter, we will draw conclusions from our experimental results provided in the previous chapter. We will also verify the contributions made by our research. Finally, we will outline improvements to the studied methods and future work in the research area.

6.1 Summary of Results

In chapter 5, we have described the results of our experiments with three learning tasks. In this section, we will try to generalize the results for each of the GP and HGP methods as applied to evolution of robotic controllers.

Our research was concerned with the evolution of robotic controllers for the Khepera robot to perform tasks. We were more interested in the population of individuals making up the robotic controller than in the program codes of individuals in the population. However, sample program codes from evolved individuals are provided in Appendix A.

The reactive robotic control problem provides a challenge to the genetic programming paradigm. With the lack of test cases for fitness function evaluation, the fitness of an individual can differ greatly depending on the immediate neighbourhood of the robot. We have noticed that the definition of the fitness function can greatly influence the population contents and thus the resulting behaviours. Because of the constantly changing local environment, even good performing behaviour was observed

to eventually be replaced by worse behaviours.

The number of collisions with an obstacle was not a good performance measure during learning by the robotic controller. However, the collisions can be used in the testing environment to provide a measure of success in learning proper obstacle avoidance behaviour.

The training environment is an important part of robotic training. We have noticed that the robotic controllers often adapted too much to the training environment. This problem of overfitting is a common problem in genetic programming. A choice of proper training environment for a particular task is thus very important. From our obstacle avoidance and wall following task learning experiments, we have noticed that sharp corners of the environment formed an area of difficulty for the robotic controller. This was probably due to a part of the corner fitting between the fields of view of the proximity sensors.

The entropy value was an important indicator of population diversity in our experiments. We have noticed that good behaviour was found in populations with relatively high entropy value (above about 0.6). Low entropy value signified convergence in the population which usually accompanied a convergence to a low average chromosome size. Populations of individuals with low chromosome size did not contain enough information to successfully search for a good solution.

6.1.1 Linear Genome Approach

Throughout most of our experiments, the linear genome method enforced a stable level of entropy and average chromosome size. A stable level was defined as oscillations around a certain value and slight decrease in the average values. This behaviour was probably due to the different crossover operator in the linear genome method than in the tree-based methods and by the mutation operator. Because of the stable entropy levels, populations of 50 individuals were enough to provide stable behaviour for many generations.

With the tree-based methods, entropy value stability depended on the definition and parameter values of the fitness function. Tendency toward smaller program sizes was seen with the half-and-half chromosome creation method or small sized populations. To keep program sizes and entropy at high values for reasonable time, we

have used population sizes of 100 or more individuals (with the exception of the ARL method discussed below).

Occurrence of behaviour in linear genome experiments was similar to other methods. The average generation values of initial good behaviour occurrence were usually higher using the linear genome methods. However, the methods used different population sizes and individual sizes so it was difficult to draw conclusions from the raw results. To gain a better perspective on the raw data, we have calculated the values of average chromosome evaluations until a good solution was found. The average chromosome evaluation values for the linear genome method were higher than all other methods for the light avoidance task but were low or average in the obstacle avoidance and wall following tasks. Summary of method performance is available in Table 6.1.

With our implementation of the linear genome method (using a genome interpreter) we have found the evolution time similar to the time using the tree-based representation (with equivalent population size and tree size settings). The main difference between the methods was the contents of the function sets in linear genome and tree-based method experiments. Tests with logical operators in the tree-based method did not provide clear improvements over the arithmetic methods, thus we have used the standard arithmetic operators in our experiments.

6.1.2 Hierarchical Genetic Programming Approaches

We have tested three HGP learning methods: Automatically Defined Functions (ADF), Module Acquisition (MA), and Adaptive Representation through Learning (ARL). Robotic controllers using each method were able to evolve some degree of proper behaviour for each learning task.

The best entropy and best average size stability was seen with experiments using the ARL method and a static threshold value of 1.5. The worst entropy behaviour was seen mainly with the ADF method but sometimes with the MA and tree-based methods. The worst average size behaviour was seen with the MA method for all the studied tasks. Summary of method performance is available in Table 6.1.

The ADF method used a predefined, constant function set containing one or more ADFs. The only method of function call acquisition was through crossover with

another individual of the population. The ADF program trees did not mate with the program trees of the result producing branches of the individuals. The ADFs inside individuals showing nice behaviour were usually quite large and complex with no noticeable patterns. It is possible that in our experiments, the ADFs had not much purpose other than to provide few extra tree levels of instructions. In our experiments, the ADF method runs provided performance that was usually below that of the tree-based method and sometimes the worst of all HGP methods.

The slowest method of function creation was the MA method. Even with 20% probability of function creation, the function set did not grow very fast. Most of the individuals in the populations with good behaviour did not use any of the functions in the module set. The creation of functions produced program size loss which in turn often lowered the entropy of the population. The behavioural performance of the MA method was usually worse than that of the tree-based method. Since similar experimental settings were used for the two methods, we can deduce that the function creation of our MA method disrupted the task learning instead of helping it.

The performance of the ARL method was surprisingly good. The method displayed the most stable entropy and average chromosome size behaviour in most experiments. This behaviour was observed only when function creation was taking place, thus we think that the function creation and new individual creation processes were responsible for the stability. The method also achieved the best time and smallest deviation to reach good behaviour in most experiments.

The number of functions created by the ARL algorithm depended on each run but did not grow monotonically as first expected. The function set grew and shrank throughout the runs of the algorithm. The functions usually contained simple arithmetic operators working on function parameters. Many of the functions from populations with good behaviour contained division and addition operators and seemed to calculate some form of ratio of the function parameters (see Figure A.14). Since such ratios would be helpful in all of our studied tasks, we think that some of the evolved functions were of benefit to the individuals.

Method	Entropy Stability	Size Stability	Behavioural Performance
Linear GP	excellent	excellent	poor
Tree GP	average	average	average
ADF HGP	poor	average	average
MA HGP	average	poor	average
ARL HGP	excellent	excellent	excellent

Table 6.1: Summary of results from our experiments for each of the studied methods. Behavioural performance was based on first occurrence of good behaviour. Performance of each method was relative to performance of other methods.

6.1.3 Multiple Symbolic Regression

We have tested two methods of structuring program trees in order to perform multiple symbolic regression calculations. Our control programs required output of two motor values as given in Equation 4.1. The first method (2-output method) used a special program tree structure with a separate tree for each output variable. The second method (1-output method) used only one rooted program tree and special *SetMx* functions to set the two motor output variables of the robot.

We have performed most of our experiments with the 2-output method but we have experimented with the 1-output method using the tree-based representation. From our experiments, the 2-output method outperformed the 1-output method in both statistical and behavioural measures.

6.2 Contributions

In this section, we take a second look at the proposed contributions of our research in order to summarize them and to ensure they were all met.

1. We have added substantial improvements to the Khepera GP Simulator for Windows[®] and created version 3.0 of the simulator. The Simulator contained GP-based robotic controllers using the methods of: linear genome GP, tree-based GP, Automatically Defined Functions HGP, Module Acquisition HGP, and Adaptive Representation HGP. The source code of the Simulator was made modular and easy to modify. The interface of the Simulator was made user

friendly and provided the user with run-time modification of all parameters of the simulation engine and robotic controllers. The new version of the simulator was made available publicly free-of-charge for academic research purposes.

2. We have tested and recorded the performance of the linear genome GP, tree-based GP, Automatically Defined Functions HGP, Module Acquisition HGP, and Adaptive Representation HGP methods on the problem of reactive robotic control using obstacle avoidance, wall following, and light avoidance tasks.
3. Based on our experiments, we have compared and contrasted the linear genome GP representation method and the regular tree-based GP representation.
4. Based on our experiments, we have compared and contrasted the tree-based GP method with the three studied hierarchical genetic programming methods. Our results enabled us to describe the advantages and disadvantages of the HGP methods for the problem of reactive robotic control.
5. For the tree-based GP method, we have compared and contrasted the two studied representation methods for Multiple Symbolic Regression: using a dual tree, and using special settable variables. From our results, we have concluded that the method of dual tree seems best suited for the problem.
6. Finally, based on our results, we have offered possible improvements to the HGP algorithms to improve their performance in the domain of robotic control.

6.3 Improvements and Future Work

Algorithms and strategies of solving problems can usually be improved to yield better solutions. Our research enabled us to indicate areas of possible improvement to the studied genetic programming algorithms in order to improve their performance in the domain of robotic controller generation.

We feel that entropy stability, chromosome size stability, and proper fitness evaluation are the most important attributes of a well functioning genetic programming robotic controller training system. Entropy and chromosome size values should be

relatively stable (i.e. should not decrease abruptly) so that they remain at reasonable levels for a reasonable number of generations. Stability of those values depends on the definition of the fitness function and on the genetic algorithm used in the experiments.

We have seen that modification of fitness function parameters led to strong statistical and behaviour changes in the evolving population. Definition of the fitness function is thus a very important aspect of evolution of correct solutions. We have tested variations of the fitness functions used in our experiments to choose those with best performance. However, more testing of fitness functions and their parameters should be done to identify the optimal fitness function definition for each learning task.

The Automatically Defined Functions (ADF) method by Koza [32] did not perform well in our experiments. The ADF method we have used contained ADFs of size equivalent to the main program body. We feel that smaller building blocks or functions are more useful for robotic controllers. We noticed that the sizes of the ADFs did not decrease well enough before the population converged prematurely. We think that it would be best to specify a smaller initial and maximum size of the ADFs so that the functions require less time to find optimal configurations.

The method of Module Acquisition (MA) by Angeline and Pollack [3] provided the worst performance in most of our experiments. We feel that this poor performance was due to the creation of modules thus lowering the average program tree size. Since no mechanism existed to counteract this loss of program size and accompanying loss of entropy, the population often converged prematurely to suboptimal solutions. Probability-based compression and expansion operator invocation might be replaced by need-based operator invocation similar to those found in the ARL method. This new operator invocation should lead to better behaviour through adjustments of operator frequencies based on population needs.

The Adaptive Representation through Learning (ARL) method by Rosca and Ballard [63] contained a mechanism to neutralize the bad effects of function creation. Thus, the method exhibited very stable entropy and average size behaviours while producing nice and quick learning behaviour. The creation of random individuals using the enriched function set at the start of a new epoch provided the genetic algorithm with fresh search material. The functions found in the adaptive representation

step of the algorithm were small and seemed better building blocks than the functions in the ADF method. Even though our tests of our dynamic entropy threshold were not very successful, we still feel that best performance should be achieved by some kind of a dynamically changing entropy threshold calculation.

Influx of random individuals to the population during evolution can lead to problems. Too many random individuals would destabilize good solutions present in individuals of the previous population. We think that a low replacement fraction used with elitism of best individuals should produce the optimal evolutionary balance. Elitist individuals would always be copied into the population and would ensure the fittest individuals are not lost between generations.

Variation in the population can also be achieved by using a mutation operator for the tree-based representation methods. The mutation operator can quickly add subtle variety to the population. The crossover operator can perform similar mutations but with a lower probability of success based on the size and structure of the program tree.

Future work with the Khepera GP Simulator involves formulation of a proper physics model to study object interaction tasks. Modification of the simulation engine for multi-threaded robot simulations would enable proper real-time multi-robot simulation. With the use of a real Khepera robot, we hope to add serial Khepera interface to the Simulator and validate the correctness of our Khepera simulation engine.

In this thesis, we have evolved reactive, memoryless robotic controllers. Our results indicate that the controllers can be trained to exhibit some level of proper behaviour for the studied tasks. The extension to this research would be to study memory-based robotic controllers that can store previous actions and use them to decide future behaviour. Such controllers using the linear genome method have been shown in [49] to successfully and quickly evolve more complex behaviours than a memoryless controller.

We would also like to use a real Khepera robot to verify our results. Physical robots train in a noisy and sometimes unpredictable environment and would provide a real world test case for our research. Because of the reactive learning system, the Simulator and robotic controllers can be easily modified to perform experiments with

a real Khepera robot.

6.4 Summary

This chapter concluded our thesis by summarizing our findings, comparing and contrasting the linear-genome GP method with the tree-based GP representation, and comparing and contrasting the tree-based GP representation with the three HGP methods. We have also revisited the contributions made by our thesis, provided improvements to the HGP methods, and discussed future work in the research area.

Appendix A

Program Code Examples

Individual 2493	P1: f = s0 * c	P1: f = s0 ^ a	P2: f = b ^ 7443	P1: c = c s4
P2: a = f * 7526	P1: d = d * s0	P2: f = b ^ 7443	P1: c = s1 << s0	P1: d = s5 s4
P2: b = s3 ^ 4939	P1: d = s4 e	P1: b = s2 >> s4	P1: a = f e	P1: m2 = a >> d
P1: m1 = s2 - s5	P1: m2 = f d	P1: m2 = s7 & s0	P1: d = c + s2	P1: c = d >> s4
P2: f = f >> 286	P1: m2 = s3 s4	P1: m1 = c >> s0	P1: m2 = s0 * d	P1: m2 = f * d
P1: f = s0 * c	P1: f = s7 << s6	P1: d = s3 * s4	P1: m2 = s6 >> d	P1: m2 = s1 a
P1: d = d * s0	P2: b = s4 * 7709	P1: a = f ^ d	P1: d = s5 s4	P2: b = s7 << 6453
P1: d = s4 e	P2: f = d ^ 4751	P1: m2 = s2 s4	P1: a = a >> d	P1: d = s4 * s6
P1: m2 = f d	P1: c = c - s4	P1: f = s7 << a	P2: c = s1 >> 7528	P1: d = f * d
P1: m2 = s3 s4	P1: c = s2 ^ s3	P1: c = s2 ^ s3	P1: b = a * s6	P1: d = s5 s6
P1: f = s7 << s6	P2: c = s4 - 5	P2: c = s4 & 5	P2: d = f >> 4632	P1: m2 = s3 - d
P2: b = s4 * 6240	P2: f = s7 + 39	P2: f = s7 + 39	P1: c = d ^ s2	P1: m2 = s2 s4
P2: f = d ^ 4751	P1: m2 = c - c	P1: m2 = c - c	P1: m1 = s1 << s0	P2: c = b ^ 947
P1: c = c - s4	P2: m2 = s7 ^ 4201	P2: m2 = e + 7709	P1: d = d - s3	
P1: d = s5 s4	P2: c = e >> 3155	P2: a = s2 >> 582	P1: c = s0 * s3	
P1: m2 = a >> d	P1: c = s0 & s3	P1: f = f - s5	P1: a = f & s2	
P2: e = d >> 7528	P1: f = s3 >> c	P2: d = s2 ^ 5219	P1: f = s2 >> s4	
P1: m1 = s2 - s5	P2: d = e * 1268	P2: c = s3 - 655	P2: b = s4 * 7709	
P2: f = f >> 286	P2: c = s3 ^ 655	P1: c = s0 ^ a	P2: f = d ^ 4751	

Figure A.1: Program code of an individual using the linear genome GP method to learn the obstacle avoidance task.

```

values      /      +      -
-          *      *      *
+          +      +      *
+          6      +      -
/          s6     s6     s5
s2         *      s2     8
s5         s2     *      /
*          s2     /      /
s4         -      -      -
s1         *      -      3
-          s2     /      s2
+          s2     /      s7
*          -      4      s5
8          s5     -
2          s1     s2
5          s6     *
+          s1     s1
+          s2     s7
s5         +      8
*          2
s4
s1
s1

```

Figure A.2: Program code of an individual using the tree-based GP method to learn the obstacle avoidance task.

```

values      defun ADF0 (ARG0, ARG1)      defun ADF1 (ARG0, ARG1)
+          values                          values
8          -                              /
/          +                              -
/          /                              ARG0
s3         8                              /
s1         /                              -
/          +                              6
*          s4                              3
/          -                              +
s4         s4                              6
s3         s3                              ARG0
s7         *                              /
s2         s4                              +
*          1                              /
/          2                              0
s4         /                              0
s3         /                              ARG0
/          s4                              6
s4         +
+          -
-          2
2          4
4          *
*          1
1          2
2

```

Figure A.3: Program code of an individual using the ADF HGP method to learn the obstacle avoidance task.

```

values
+
s0
9
-
+
/
+
2
s4
+
4
s4
*
-
s5
s3
9
Fn12541144&0&686
s6
s6
s7
s1

/
-
/
s7
s1
+
1
0
+
Fn12386896&4&862
8
s3
6
s4
+
6
6

Fn12541144&0&686
-
+
d0
d1
-
d2
d3
Fn12386896&4&862
+
-
d0
d1
+
d2
d3

```

Figure A.4: Program code of an individual using the ARL HGP method to learn the obstacle avoidance task.

Individual 573	P1: e = d << d	P2: f = s3 >> 2926
P1: b = s3 s0	P1: f = s6 + e	P1: e = d - b
P2: m1 = e >> 7169	P1: d = b + d	P1: d = s3 ^ s2
P1: a = s5 ^ f	P1: e = d - b	P2: m2 = s0 & 2621
P2: m1 = c + 5179	P2: m2 = s0 * 2621	P2: a = d & 1611
P2: e = e ^ 3699	P2: d = f & 1611	P1: f = s6 + e
P1: m1 = s3 + s7	P1: f = s6 + e	P2: c = s1 & 3872
P2: f = s3 >> 2926	P2: c = s1 & 3872	P2: d = s3 - 214
P1: e = d - b	P2: d = s3 - 2390	P2: f = s2 2653
P1: d = s3 ^ s2	P1: m2 = s4 ^ s7	P2: c = s1 & 3872
P2: m2 = s0 & 2621	P1: c = a >> f	P2: d = s3 - 214
P2: d = f & 1611	P2: f = c + 57	P2: f = e 7033
P1: b = e s0	P2: m1 = d & 4058	P2: f = s2 2653
P1: m2 = d - d	P2: f = s6 & 3523	P2: c = s6 - 3972
P1: c = s3 f	P2: e = s7 - 3535	

Figure A.5: Program code of an individual using the linear genome GP method to learn the wall following task.

```

values
s1
/
-
+
s0
-
s0
s0
s1
-
s1
s1
s2

```

Figure A.6: Program code of an individual using the tree-based GP method to learn the wall following task.

```

values
ADF0
s0
ADF0
*
s4
s4
1
s4

defun ADF0 (ARG0, ARG1)
values
+
7
+
/
ARG1
ARG1
-
/
7
ARG1
-
/
7
7
/
0
*
7
2

```

Figure A.7: Program code of an individual using the ADF HGP method to learn the wall following task.

```

values
/
/
+
Fn1056679512&6909
Fn1056679510&11493
/
4
Fn1056679514&15777
/
/
/
4
7
+
s2
4
+
+
s7
s3
-
s5
0
/
+
s3
6
-
s2
7

```

```

Fn1056679510&11493
+
s4
s3
Fn1056679514&15777
-
s4
+
s1
s7
Fn1056679512&6909
*
2
s4

```

Figure A.8: Program code of an individual using the MA HGP method to learn the wall following task.

Figure A.9: Program code of an individual using the ARL HGP method to learn the wall following task.

Individual 3124	P2: a = f >> 9	P1: b = c ^ f
P1: a = l1 + l0	P2: c = l2 ^ 7	P1: c = l7 + l6
P2: m1 = c << 5	P2: m2 = l1 & 6	P2: a = f * 6
P1: a = d + c	P2: m2 = b >> 9	P1: f = l1 & b
P2: m1 = c << 5	P1: m2 = l7 ^ l1	P2: a = d & 4
P2: a = f >> 9	P1: c = l7 ^ b	P1: e = c - l4
P2: c = l2 ^ 7	P2: m1 = l6 & 3	P1: m2 = d - l7
P2: a = f * 6	P1: f = d << c	P2: a = d >> 7
P1: f = l1 & b	P1: d = c << l2	P1: d = l1 + l3

Figure A.10: Program code of an individual using the linear genome GP method to learn the light avoidance task.

Figure A.11: Program code of an individual using the tree-based GP method to learn the light avoidance task.

```

values
-
  ADF0
  +
  +
  16
  14
  ADF0
  12
  14
-
  ADF0
  12
  9
  0
/
  14
  14
ADF0
+
  ADF0
  0
  11
  13
/
  10
-
  13
  2

defun ADF0 (ARG0, ARG1)
values
-
  /
  *
  /
  9
  1
-
  ARG1
  9
-
  /
  6
  ARG1
-
  ARG1
  ARG0
-
  ARG1
  9

```

Figure A.12: Program code of an individual using the ADF HGP method to learn the light avoidance task.

```

values
9
+
-
/
9
+
11
-
3
13
/
17
11
Fn1056243547&5720

-
15
-
/
17
0
+
13
13
+
+
17
17
+
12
13

Fn1056243547&5720
/
+
7
16
*
0
0

```

Figure A.13: Program code of an individual using the MA HGP method to learn the light avoidance task.

```

values
10
+
Fn34114040&4&94
Fn32499840&7&431
Fn32499840&0&442
14
3
3
5
/
16
10
+
13
13
Fn32499840&3&378
+
0
/
14
13
+
17
9
Fn32499840&6&406
4
13
10
16
9
/
Fn32499840&6&406
16
7
11
17
Fn32499840&4&968
14
14
13
16

Fn32499840&3&378
+
d0
d1
/
d2
d3
Fn32499840&7&431
+
+
d0
d1
/
d2
d0
Fn32499840&4&968
/
+
d0
d1
d2

Fn32499840&6&406
/
d0
d1
d2
d3
Fn34114040&4&94
+
+
d0
d1
+
d2
d3
Fn32499840&0&442
+
/
d0
d1
+
d2
d3

```

Figure A.14: Program code of an individual using the ARL HGP method to learn the light avoidance task.

Bibliography

- [1] P. J. Angeline. Genetic programming and emergent intelligence. In Kenneth E. Kinneer, Jr., editor, *Advances in Genetic Programming*, chapter 4, pages 75–98. MIT Press, 1994.
- [2] P. J. Angeline. Subtree crossover: Building block engine or macromutation? In J. Koza, K. Deb, M. Dorigo, D. Fogel, M. Garzon, H. Iba, and R. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 9–17, San Francisco, CA, 22-25 1997. Morgan Kaufmann.
- [3] P. J. Angeline and J. B. Pollack. The evolutionary induction of subroutines. In *Proceedings of the Fourteenth Annual Conference of the Cognitive Science Society*, Bloomington, Indiana, USA, 1992. Lawrence Erlbaum.
- [4] R. C. Arkin. *Behavior-Based Robotics*. MIT Press, Cambridge, MA, 1998.
- [5] T. Bäck, F. Hoffmeister, and H.-P. Schwefel. A survey of evolution strategies. In Rick Belew and Lashon Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 2–9, San Mateo, CA, 1991. Morgan Kaufman.
- [6] J. E. Baker. Adaptive selection methods for genetic algorithms. In John J. Grefenstette, editor, *Proceedings of the First International Conference on Genetic Algorithms and their Applications*, pages 101–111, Hillsdale, 1987. Lawrence Erlbaum Associates.

- [7] J. E. Baker. Reducing bias and inefficiency in the selection algorithm. In *Proceedings of the Second International Conference on Genetic Algorithms*, Hillsdale, 1987. Lawrence Erlbaum Associates.
- [8] W. Banzhaf, D. Banscherus, and P. Dittrich. Hierarchical genetic programming using local modules. Series Computational Intelligence, Internal Report of SFB 531, No. 56/99 ISSN 1433-3325, Univ. of Dortmund, D-44221 Dortmund, Germany, 1999.
- [9] W. Banzhaf, P. Nordin, and M. Olmer. Generating Adaptive Behavior for a Real Robot using Function Regression within Genetic Programming. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 35–43, Stanford University, CA, USA, 13-16 1997. Morgan Kaufmann.
- [10] T. Blickle and L. Thiele. A mathematical analysis of tournament selection. In Larry Eshelman, editor, *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 9–16, San Francisco, CA, 1995. Morgan Kaufmann.
- [11] R. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1), 1986.
- [12] R. Brooks. Artificial life and real robots. In F. J. Varela and P. Bourguine, editors, *Toward a Practice of Autonomous Systems: Proceedings of the first European Conference on Artificial Life*, pages 3–10, Cambridge, MA, 1992. MIT Press-Bradford Books.
- [13] D. S. Burke, K. A. De Jong, J. J. Grefenstette, C. Loggia Ramsey, and A. S. Wu. Putting more genetics into genetic algorithms. *Evolutionary Computation*, 6(4):387–410, Winter 1998.
- [14] D. T. Cliff, P. Husbands, and I. Harvey. Evolving visually guided robots. In J. Meyer, H. L. Roitblat, and S. W. Wilson, editors, *From Animals to Animats II: Proceedings of the Second International Conference on Simulation of Adaptive Behavior*, Cambridge, MA, 1993. MIT Press-Bradford Books.

-
- [15] C. Darwin. *On the Origin of Species by Means of Natural Selection*. John Murray, London, 1859.
 - [16] Marc Ebner. Evolution of a control architecture for a mobile robot. In Moshe Sipper, Daniel Mange, and Andrs Prez-Urbe, editors, *Proceedings of the Second International Conference on Evolvable Systems: From Biology to Hardware (ICES 98)*, volume 1478, pages 303–310, Lausanne, Switzerland, 23-25 1998. Springer Verlag.
 - [17] D. Ferguson. Evolution to the xtreme: Evolving evolutionary strategies using a meta-level approach. Master's thesis, Carleton University, Ottawa, Canada, 2002.
 - [18] D. Floreano and F. Mondada. Automatic creation of an autonomous agent: Genetic evolution of a neural-network driven robot. In *From Animals to Animats 3, Proc. of 3rd Intl. Conf. on Simulation of Adaptive Behavior, SAB'94*, Cambridge, MA, 1994. MIT Press-Bradford Books.
 - [19] D. Floreano and F. Mondada. Evolution of homing navigation in a real mobile robot. *IEEE Transactions on Systems, Man, and Cybernetics-Part B*, 26:396–407, 1996.
 - [20] D. B. Fogel. On the philosophical differences between evolutionary algorithms and genetic algorithms. In D.B. Fogel and W. Atmar, editors, *Proceedings of the Second Annual Conference on Evolutionary Programming*, pages 23–29, La Jolla, California, 1993.
 - [21] L. J. Fogel, A. J. Owens, and M. J. Walsh. *Artificial Intelligence through Simulated Evolution*. John Wiley & Sons, New York, 1966.
 - [22] D. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, Reading, Mass., 1989.
 - [23] I. Harvey, P. Husbands, and D. Cliff. Issues in evolutionary robotics. In J. Meyer, H. Roitblat, and S. Wilson, editors, *From Animals to Animats II: Proceedings of the Second International Conference on Simulation of Adaptive Behavior*, pages 364–373, Cambridge, MA, 1992. MIT Press-Bradford Books.

- [24] Inman Harvey. Artificial evolution: A continuing SAGA. *Lecture Notes in Computer Science*, 2217:94–??, 2001.
- [25] J. H. Holland. *Adaptation in Natural and Artificial Systems*. MIT Press, 1992.
- [26] W. Kantschik and W. Banzhaf. Linear-tree GP and its comparison with other GP structures. In Julian F. Miller, Marco Tomassini, Pier Luca Lanzi, Conor Ryan, Andrea G. B. Tettamanzi, and William B. Langdon, editors, *Genetic Programming, Proceedings of EuroGP'2001*, volume 2038 of *LNCS*, pages 302–312, Lake Como, Italy, 18-20 April 2001. Springer-Verlag.
- [27] W. Kantschik and W. Banzhaf. Linear-graph GP—A new GP structure. In James A. Foster, Evelyne Lutton, Julian Miller, Conor Ryan, and Andrea G. B. Tettamanzi, editors, *Genetic Programming, Proceedings of the 5th European Conference, EuroGP 2002*, volume 2278 of *LNCS*, pages 83–92, Kinsale, Ireland, 3-5 April 2002. Springer-Verlag.
- [28] Mike J. Keith and Martin C. Martin. Genetic programming in c++: Implementation issues. *Advances in Genetic Programming*, 1994.
- [29] K. E. Kinneær, Jr., editor. *Advances in Genetic Programming*. MIT Press, Cambridge, MA, 1994.
- [30] K. E. Kinneær, Jr. Alternatives in automatic function definition: A comparison of performance. In Kenneth E. Kinneær, Jr., editor, *Advances in Genetic Programming*, chapter 6, pages 119–141. MIT Press, 1994.
- [31] J. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, 1992.
- [32] J. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge, MA, 1994.
- [33] J. Koza, F. H. Bennett III, D. Andre, and M. A. Keane. *Genetic Programming III: Darwinian Invention and Problem Solving*. Morgan Kaufmann, San Francisco, 1999.

- [34] J. R. Koza. Evolution of subsumption using genetic programming. In F. J. Varela and P. Bourguine, editors, *Proceedings of the First European Conference on Artificial Life. Towards a Practice of Autonomous Systems*, pages 110–119, Cambridge, MA, 11-13 1992. MIT Press-Bradford Books.
- [35] J. R. Koza and J. P. Rice. Automatic programming of robots using genetic programming. In *Proceedings of Tenth National Conference on Artificial Intelligence*, pages 194–201. AAAI Press/MIT Press, 1992.
- [36] S. Luke and L. Spector. A revised comparison of crossover and mutation in genetic programming. In John R. Koza, Wolfgang Banzhaf, Kumar Chellapilla, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max H. Garzon, David E. Goldberg, Hitoshi Iba, and Rick Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 208–213, University of Wisconsin, Madison, Wisconsin, USA, 22-25 1998. Morgan Kaufmann.
- [37] Martin C. Martin. Visual obstacle avoidance using genetic programming: First results. In Lee Spector, Erik D. Goodman, Annie Wu, W. B. Langdon, Hans-Michael Voigt, Mitsuo Gen, Sandip Sen, Marco Dorigo, Shahram Pezeshek, Max H. Garzon, and Edmund Burke, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 1107–1113, San Francisco, California, USA, 7-11 2001. Morgan Kaufmann.
- [38] M. J. Mataric. Behavior-based control: Examples from navigation, learning, and group behavior. *Journal of Experimental and Theoretical Artificial Intelligence*, 9 (2–3), 1997.
- [39] L. A. Meeden. An incremental approach to developing intelligent neural network controllers for robots. *IEEE Transactions on Systems, Man, and Cybernetics-Part B*, 26(3):474–485, 1996.
- [40] O. Michel. Webots: a powerful realistic mobile robots simulator. In *Proceeding of the Second International Workshop on RoboCup. LNAI*. Springer-Verlag, 1998.

- [41] F. Mondada and D. Floreano. Evolution of neural control structures: some experiments on mobile robots. *Robotics and Autonomous Systems*, 16:183–195, 1995.
- [42] S. Nolfi and D. Floreano. *Evolutionary Robotics: The Biology, Intelligence, and Technology of Self-Organizing Machines*. MIT Press/Bradford Books, Cambridge, MA, 2000.
- [43] S. Nolfi, D. Floreano, O. Miglino, and F. Mondada. How to evolve autonomous robots: Different approaches in evolutionary robotics. In *Proceedings of the Fourth Workshop on Artificial Life*, pages 190–197, Cambridge, MA, 1994. MIT Press.
- [44] J. P. Nordin. A compiling genetic programming system that directly manipulates the machine-code. In K. E. Kinneer, Jr., editor, *Advances in Genetic Programming*, chapter 14, pages 311–331. MIT Press, Cambridge, MA, 1994.
- [45] P. Nordin and W. Banzhaf. Genetic programming controlling a miniature robot. In E. V. Siegel and J. R. Koza, editors, *Working Notes for the AAAI Symposium on Genetic Programming*, pages 61–67, MIT, Cambridge, MA, USA, 10–12 1995. AAAI.
- [46] P. Nordin and W. Banzhaf. A genetic programming system learning obstacle avoiding behavior and controlling a miniature robot in real time, sysreport 4/95. Technical report, University of Dortmund, Fachbereich Informatik, 1995.
- [47] P. Nordin and W. Banzhaf. Real time evolution of behavior and a world model for a miniature robot using genetic programming, sysreport 5/95. Technical report, Dept. of CS, University of Dortmund, 1995.
- [48] P. Nordin and W. Banzhaf. An on-line method to evolve behavior and to control a miniature robot in real time with genetic programming. *Adaptive Behavior*, 5(2):107–140, 1997.
- [49] P. Nordin and W. Banzhaf. Real time control of a khepera robot using genetic programming. *Cybernetics and Control*, 26(3):533–561, 1997.

- [50] P. Nordin, W. Banzhaf, and M. Brameier. Evolution of a world model for a miniature robot using genetic programming. *Robotics and Autonomous Systems*, 25(1-2):105–116, 1998.
- [51] M. Olmer, P. Nordin, and W. Banzhaf. Evolving real-time behavioral modules for a robot with gp. In M. Jamshidi, F. Pin, and P. Dauchez, editors, *Proc. 6th International Symposium on Robotics And Manufacturing (ISRAM-96): Robotics and Manufacturing*, pages 675–680, New York, 1996. Asme Press.
- [52] U. M. O'Reilly and F. Oppacher. The troubling aspects of a building block hypothesis for genetic programming. Technical Report 94-02-001, Santa Fe: Santa Fe Institute, 1399 Hyde Park Road Santa Fe, New Mexico 87501-8943 USA, 1992.
- [53] R. Poli. Evolution of graph-like programs with parallel distributed genetic programming. In Thomas Back, editor, *Genetic Algorithms: Proceedings of the Seventh International Conference*, pages 346–353, Michigan State University, East Lansing, MI, USA, 19-23 July 1997. Morgan Kaufmann.
- [54] R. Poli. General schema theory for genetic programming with subtree-swapping crossover. In Julian F. Miller, Marco Tomassini, Pier Luca Lanzi, Conor Ryan, Andrea G. B. Tettamanzi, and William B. Langdon, editors, *Genetic Programming, Proceedings of EuroGP'2001*, volume 2038, pages 143–159, Lake Como, Italy, 18-20 2001. Springer-Verlag.
- [55] C. Loggia Ramsey, K. A. De Jong, J. J. Grefenstette, A. S. Wu, and D. S. Burke. Genome length as an evolutionary self-adaptation. *Lecture Notes in Computer Science*, 1498:345–356, 1998.
- [56] T. S. Ray. Is it alive, or is it GA? In R. K. Belew and L. B. Booker, editors, *Proceedings of the 1991 International Conference on Genetic Algorithms*, pages 527–534, San Mateo, CA, 1991. Morgan Kaufmann.
- [57] I. Rechenberg. *Evolutionsstrategie*. Friedrich Frommann Verlag (Günther Holzboog KG), Stuttgart, 1973.

- [58] C. Reynolds. An evolved, vision-based behavioral model of coordinated group motion. In J. Meyer, H. L. Roitblat, and S. W. Wilson, editors, *From Animals to Animats II: Proceedings of the Second International Conference on Simulation of Adaptive Behavior*, Cambridge, MA, 1993. MIT Press-Bradford Books.
- [59] J. P. Rosca and D. H. Ballard. Hierarchical self-organization in genetic programming. In *Proceedings of the Eleventh International Conference on Machine Learning*. Morgan Kaufmann, 1994.
- [60] J. P. Rosca and D. H. Ballard. Discovery of subroutines in genetic programming. In P. J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 9, pages 177–202. MIT Press, Cambridge, MA, USA, 1996.
- [61] Justinian P. Rosca. Entropy-driven adaptive representation. In Justinian P. Rosca, editor, *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, pages 23–32, Tahoe City, California, USA, 9 1995.
- [62] Justinian P. Rosca. *Hierarchical Learning with Procedural Abstraction Mechanisms*. PhD thesis, University of Rochester, 1997.
- [63] Justinian P. Rosca and Dana H. Ballard. Genetic programming with adaptive representations. Technical Report TR 489, University of Rochester, Computer Science Department, Rochester, NY, USA, feb 1994.
- [64] C.E. Shannon and W. Weaver. *The Mathematical Theory of Communication*. University of Illinois Press, Urbana, Illinois, 1949.
- [65] I. G. Sprinkhuizen-Kuyper. Artificial evolution of box-pushing behaviour, CS 01-02. Technical report, Universiteit Maastricht, Faculty of General Sciences, IKAT/Department of Computer Science, Maastricht, The Netherlands, 2001.