3

Evolution of Khepera Robotic Controllers with Hierarchical Genetic Programming Techniques

Marcin L. Pilat[1] and Franz Oppacher[2]

[1] Department of Computer Science, University of Calgary,
2500 University Drive N.W.,
Calgary, AB, T2N 1N4, Canada,
pilat@cpsc.ucalgary.ca, http://www.pilat.org
[2] School of Computer Science, Carleton University,
1125 Colonel By Drive,
Ottawa, ON, K1S 5B6, Canada
oppacher@scs.carleton.ca, http://www.scs.carleton.ca/~oppacher

In this chapter, we evolve robotic controllers for a miniature mobile Khepera robot. We are concerned with control tasks for obstacle avoidance, wall following, and light avoidance. Robotic controllers are evolved through canonical GP implementation, linear genome GP system, and hierarchical GP methods (Automatically Defined Functions, Module Acquisition, Adaptive Representation through Learning). We compare the different evolutionary strategies based on their performance in evolution of robotic controllers. Experiments are performed on the Khepera GP Simulator for Windows. We develop the simulator as a user and developer friendly software to study GP and other robot controllers.

3.1 Introduction

Evolutionary computation 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.

robotics focuses on building machines to improve the lives of humans. Robots are designed to perform repetitive or dangerous tasks with excellent precision and dependability. However, robots require directions and programming to accomplish their goals.

In this chapter, 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 the Khepera miniature robot designed to study genetic programming based robotic controllers.

3.2 Genetic Programming

Genetic Programming (GP) was introduced by Koza [9] 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 set of functions and a set of terminals. The chromosome encoding can be directly executed by the system or can be compiled or interpreted to produce machine executable code.

The main problem with genetic programming lies in its scalability. Genetic programming has been demonstrated to solve a variety of applications [11, 13] but it appears to lose its effectiveness for more complex real-world problems [5]. When we 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. While this method is suitable for smaller problems, it is often not powerful enough to solve difficult problems. The problem 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 (HGP) 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 [11] 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 is 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. In our research, we study the HGP methods: Automatically Defined Functions (ADFs) [11], Module Acquisition (MA) [2], Adaptive Representation (AR) [25].

3.3 Robotic Control

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.

The area of robotic control is often subdivided into three sub-areas: reactive, behavior-based, and hybrid [3]. Reactive control uses a simple set of condition-action pairs that define how the robot reacts to a stimulus. Brooks [6] proposed a multi-layer subsumption architecture where higher-level layers can subsume and block lower-level layers from action. Behavior-based architecture [14] 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. Hybrid control strategies exist that offer a compromise between purely-reactive and behavior-based strategies.

Brooks [7] 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 [10] 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 evolve control programs for mobile robots. In [12], Koza and Rice demonstrated that genetic programming can 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 [22] has used genetic programming to evolve a controller program for tiny critters in a simulated environment. The critter tasks were to manoeuver 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 [16, 19, 17, 20, 18] 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) [15] which worked with a variable length linear genome composed of machine code instructions. The system evolved machine code that was directly run on the robot without the need of an interpreter.

The initial experiments of Nordin and Banzhaf [16, 17] were based on a memory-less genetic programming system with reactive control of the robot. The system performed a type of symbolic regression to evolve a control program that would provide the robot with 2 motor 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 [20, 21] 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 [20]. Summary of the techniques used and tasks studied can be found in [4].

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 robot positions do not need to be reset 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.

3.4 Khepera Simulators

Robotic simulators play an important role in robotic experimentation. Robotic equipment can be costly and requires proper 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 of 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 constant supervision and periodical rearrangement of the robots within the environment. For such research, robotic simulators have an advantage to physical robots. Simulators can be left unsupervised and can be programmed to automatically perform human actions such as relocation of robots in the simulated 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. 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.

3.4.1 Khepera Robot

The Khepera robot is a miniature mobile robot created and sold by K-Team S.A. (http://www.k-team.com) - 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, more 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. Fig. 3.1 provides a schematic diagram of the robot's sensors and motors.

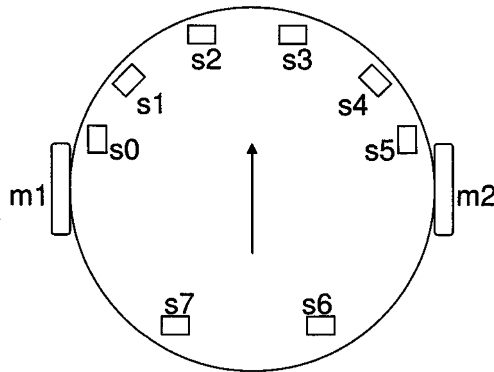

Fig. 3.1. Schematic view of the Khepera robot. Sensors are labelled s0 to s7 and motors are labelled m1 and m2.

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 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 onboard 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.

3.4.2 Khepera Simulator

The original Khepera Simulator (`http://diwww.epfl.ch/lami/team/michel` /khep-sim) 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 available. Cyberbotics (http://www.cyberbotics.com) specializes in development of 3D simulation software for mobile robots. The software - Webots - supports a variety of robots rendered in a 3-dimensional environment.

3.4.3 Khepera GP Simulator

The Khepera GP Simulator for Windows® is a software package to simulate Khepera robots in their environment. The software is 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 simulator was created by Marcin L. Pilat in 2001 as a port of the original Khepera Simulator to the Windows® platform. In 2003, the simulator was improved and adapted for simulating GP-based tasks on Khepera robots. Version 3.0 is available free for educational purposes and can be downloaded from the author's website (http:// www.pilat.org/ khepgpsim). The source code is also 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.

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). Noise is added to the simulation in order to approximate a noisy 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).

The environment of the robot is modeled as a scalable rectangular working area. All items in the environment are treated as objects. There are three types of objects - building objects (bricks, corks, boxes), light objects (lamps, light boxes), and robot objects. Robot objects are simulated Khepera robots placed in the environment. Any object can be manipulated in the environment in real-time during a simulation run.

The Khepera GP Simulator is specifically designed to study GP-based robotic controllers but can be easily adapted to non-GP controllers. 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 robotic controller provides a set of motor values to be used by a robot during each step of the simulation. The motor values are processed to yield a force vector that 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. Collisions are handled by a simple vector-based collision engine with modifiable parameters.

Each learning task (such as obstacle avoidance, wall following) can be evolved with any type of GP controller. The controller type specifies the chromosome structure and chromosome interactions during evolution. Multiple tasks can be evolved by the same GP controller type with different specifications of the chromosome structure. A task contains a population of chromosomes; thus, it can be used to store snapshots of the population during evolution.

Each task 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. Fitness functions in the simulator are dynamic and can be easily modified at runtime. The fitness function definitions are 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.

The GP controllers in the simulator gather statistical information during the run of the evolutionary algorithm. This information is stored in order to analyze the performance of an evolutionary run. For each generation, the statistics engine stores average and best population fitness, robotic collisions, chromosome complexity, and population entropy values. Population entropy [23] measures the state of a dynamic system represented by the population and can be correlated with the state of population diversity.

Complexity of the chromosomes in the population is stored using three complexity measures: size, structural complexity, and evolutional complexity [27]. The size measure specifies the raw size of the chromosomes defined as the number of instructions in a linear genome chromosome or the number of tree nodes in the tree-based chromosome representation. The structural complexity measure includes the sizes of all unique function trees called from within an individual. Evaluational complexity of an individual is measured recursively and includes sizes of all function trees embedded in the individual. This measure approximates the number of computational units required for execution of the individual program.

3.5 Robotic Controllers

Our research into robotic controllers builds on research done by Nordin and Banzhaf [20] to evolve GP 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 compare the linear genome GP method they have used in their experiments to canonical tree-based GP representation 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 [18]. The goal of the system is to approximate the function:

$$f(s0, s1, s2, s3, s4, s5, s6, s7) = \{m1, m2\} \tag{3.1}$$

where the function input is the robotic sensor data (*s0*-*s7*) and the output is the speed of the motors controlling the motion of the robot (*m1*-*m2*). 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 deal with a population of control programs for the Khepera robot. A steady-state tournament selection GP algorithm is applied to the population in order to evolve control programs that accomplish the specified learning tasks.

3.5.1 Tree-based GP

The canonical GP implementation uses a tree-based chromosome representation [9, 11]. 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 program tree are chosen from a set of parameterized functions with parameters as subtrees. Leaf nodes 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. To generate tree-based chromosomes, we use the function and terminal sets as shown in Table 3.1.

Program trees of each individual are created in a recursive manner. Three methods have been suggested for the creation of the initial random population: *full*, *grow*, or *ramped half-and-half* [9]. The *full* method creates trees with all leaf nodes at equal depth and is the method used in our implementation. The

Table 3.1. Contents of the function set and terminal set used by tree-based chromosome representations.

Function Set:	`Add, Sub, Mul, Div, AND, OR, XOR, <<, >>, IFLTE`
Terminal Set:	`[s0-s7]` (8 proximity sensors), `[l0-l7]` (8 ambient light sensors), `[0-8192]` (constants in given range)

grow method grows trees of variable size and the *ramped half-and-half* method creates a mixture of trees with different heights through either the *full* or the *grow* method.

Two genetic operators are used in the tree-based chromosome representation: reproduction and crossover. Reproduction copies a chromosome into the next generation. Single subtree switching crossover is applied to the two fittest individuals in a tournament, with a given probability. We use a crossover probability of 0.9 in all tree-based chromosome representation experiments.

3.5.2 Linear Genome GP

Nordin [15] 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 [15]. 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 [21].

The linear genome method was applied by Nordin and Banzhaf [18] 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. We represent each instruction as a text string and process it through a genome interpreter prior to evaluation. This encoding improves the readability of the program code compared to the binary approach of Nordin and Banzhaf but suffers a loss in performance due to processing of the string based instructions. However, the performance of the string-based representation is sufficient for the purpose of our research.

In the linear genome GP system, each individual is composed of a series of instructions (genes). The instructions are of the following format:

$$resvar = var1 \;\; op \;\; \{var2|const\} \tag{3.2}$$

where *resvar* is the result variable and *op* is a binary operator working on either two variables (*var1* and *var2*) or a variable and a constant (*var1* and *const*).

Each individual is randomly assigned a height (number of instructions) from 1 to the maximum specified height. For each part of an instruction, a value is selected randomly from a set of primitive values. Table 3.2 provides primitive value sets of the instruction parts used in our linear genome experiments.

Table 3.2. Primitive values of instruction parts in the linear genome GP method.

Part	Primitive Value Set
res	motor values $(m1, m2)$ intermediate variables $(a - f)$
var1, var2	proximity sensor values $(s0 - s7)$ light sensor values $(l0 - l7)$ intermediate values $(a - f)$
op	add (+), subtract (-), multiply (*), left shift (SHL) right shift (SHR), XOR (^), OR (|), AND (&)
const	integer value in range: 0-8191

The linear genome GP method employs three genetic operators: reproduction, crossover and mutation. The crossover operator uses a simple variable length 2-point crossover applied to the list of instructions (genes) of two fittest individuals of a tournament. Genes are treated as atomic units by the crossover operator and are not modified internally. Simulated bit-wise mutation modifies the contents of a gene. Crossover probability of 0.9 and mutation probability of 0.05 are used in the linear genome experiments.

3.5.3 Automatically Defined Functions HGP

Koza's Automatically Defined Functions (ADFs) [11] 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 functions. The ADF HGP method implemented in our research is based on the ADF method proposed by Koza.

The ADF method has been demonstrated to be advantageous in solving more complex versions of problems than possible by standard GP (e.g. 6-parity problem) [11]. The major disadvantage to the method 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 HGP 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 [13].

The method is an extension of the tree-based GP method and shares its basic structure. An ADF chromosome consists of two distinct parts: the func-

tion 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. Invariant nodes are fixed structural nodes and are present in every ADF chromosome. Non-invariant nodes define the bodies of the ADF definitions and the result producing branch and are modified during evolution.

All ADFs defined in an individual are available locally to the program tree of the same individual. The number of ADFs present in each chromosome and the number of arguments for each ADF are specified as parameters. We use chromosomes with one, two, and three ADF definitions and two function arguments. Zero or multiple ADFs can 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 do not allow ADF calls inside ADF definitions in our implementation.

The result producing branch is built using a standard terminal set and standard function set (shown in Table 3.1) augmented with the ADFs contained in the same chromosome. Separate terminal and function sets are used by the function defining branches to define the ADFs. The ADF branch function set is identical to that of the tree-based chromosome representation whereas the ADF terminal set is composed of ADF argument variables and constants.

Tree-based reproduction and crossover genetic operators are used in the ADF chromosomes. The crossover operator can only swap non-invariant nodes of the same type using branch typing [11].

3.5.4 Module Acquisition HGP

The Module Acquisition (MA) method of Angeline and Pollack [2] employs two new operators of compression and expansion to modularize the program code into subroutines. The subroutines contained in the subroutine collection are frozen in time and cannot be modified during evolution of the program trees. 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 [8] and concluded that the method does not offer improvement in space or time over the ADF method.

The chromosome structure is identical to that of the original tree-based chromosomes with standard tree-based function and terminal sets. Modules (subroutines) are created locally for each chromosome from subtrees of the program tree and propagate through the population solely by reproduction and crossover. Module nesting is allowed inside program trees of other modules; however, by the nature of their creation, modules are not recursive.

The Module Acquisition method employs four genetic operators: reproduction, crossover, and two mutation operators of compression and expansion. The reproduction and crossover operators perform as for tree-based chromosomes. The compression operator creates a new subroutine from a randomly selected subtree of an individual in the population using depth compression [1]. We use a maximum depth value from range 2-5. Branches beyond the maximum depth are used as parameters to the new subroutine.

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 the original subtree for a subroutine call in the chromosome tree. The subroutine is removed from the module list of the chromosome if it is no longer used.

The special mutation operators are applied after the standard tree-based reproduction and crossover operators. We set the probability of compression to 0.1 and probability of expansion to 0.01.

3.5.5 Adaptive Representation HGP

Rosca and Ballard proposed the Adaptive Representation method to dynamically extend the function set with identified building blocks [25]. 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. Unlike in the ADF HGP approach, the functions are discovered automatically and without human-imposed structure. The method differs from the MA HGP approach by the algorithms used in function discovery and management of the function library. Our implementation of the AR method is based on the improved Adaptive Representation through Learning (ARL) algorithm [26].

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 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 [25]. It is unclear, however, how to discover candidate building blocks without additional domain knowledge.

The structure of the ARL chromosome program trees is identical to that of the tree-based GP method. The function set is dynamically extended by the evolutionary algorithm through creation of new functions. Nesting of functions

is allowed; however, recursive function calls are not possible due to the function creation method.

The main advantage of the ARL algorithm is the automatic discovery of useful subroutines through the concepts of differential fitness and block activation [24]. Differential fitness is defined as the difference in fitness between an individual and its least fit parent. Rosca states that large differential fitness can be the result of useful combinations of blocks of code in the individual [24]. Block activation is defined as the number of times a block of code is executed during evaluations of the individual. Rosca states that only blocks with high block activation values should be considered candidate blocks. We do not implement the concept of block activation because of the large performance overhead on the system.

In our implementation of the ARL algorithm, we select the most promising individual (based on differential fitness) from the set of promising individuals discovered during the last generation. Candidate blocks of small height (tree height of 3) are chosen from the most promising individual. The blocks are generalized into subroutines which extend the function set.

Rosca [24] computes subroutine utility which is analogous to schema fitness for subroutines. The utility is defined as the accumulation of rewards for a subroutine over a fixed time window and is calculated by a special utility function. Using subroutine utility, low performing subroutines are removed from the function set. We implement a simpler measure of subroutine utility by assigning to each subroutine an integer utility value denoting the number of generations until an unused subroutine is removed from the function set. Utility value of each unused subroutine is decremented each generation until it reaches 0 and the subroutine is removed from the population.

The run of the ARL algorithm is divided into epochs which were defined as sequences of consecutive generations in which no new candidate building blocks are discovered [25]. The ARL algorithm provides a concrete definition of epoch creation using population entropy [23] which provides a measure of the state of a dynamic system represented by the population. Rosca [23] compares the population-based dynamic system to a physical or informational system with similar behavior.

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

$$E(P) = -\sum_{k} p_k \cdot \log p_k \tag{3.3}$$

where p_k is the proportion of the population P grouped into partition k. Entropy is usually computed based on raw individual fitness; however, we could not use raw fitness because of the dynamic nature of our fitness calculation.

We compute a standardized fitness measure through an average of three fixed test cases. Each test case provides resulting robotic motor values from individual evaluation on a fixed set of input sensors. We then partition the individuals into 20 categories based on their standardized fitness measures. The population entropy value is calculated by applying Shannon's formula on the partition categories.

The measure of population entropy is important since it correlates to the state of diversity in the population during a GP run. Drops in population entropy signify drops in population diversity. The ARL method tries to counteract the drops in population entropy by creation of new individuals. The start of a new epoch is decided using a static entropy threshold of 1.5. New epoch begins and subroutines are discovered when the entropy value of the population falls below the threshold.

After the discovery of new subroutines, the function set is extended by the new functions. The ARL method generates random individuals using the new function set. The new individuals replace a fixed proportion of the worst performing individuals in the population. We use a replacement fraction of 0.2 in our experiments. Genetic operators of reproduction and crossover are similar as for the tree-based method.

3.6 Results

3.6.1 Obstacle Avoidance

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

We select a fitness function based on the work of Banzhaf et al. [4]. The function is composed of two opposite parts: pain and pleasure. The pleasure part is computed from motor values and encourages the robot to move in the environment using straight motion. The pain part is composed of sensor values and punishes the robot for object proximity. The fitness function can be expressed as an equation:

$$Fitness = \alpha(|m_1| + |m_2| - |m_1 - m_2|) - \beta \sum_{i=0}^{7} s_i \tag{3.4}$$

where m_1 and m_2 are motor values and s_0 to s_7 are proximity sensor values. The value of α is set to 10 and value of β to 1. Parameter values were chosen based on tuning experiments.

Various robotic behaviours are observed while learning the obstacle avoidance task. We subdivide the learned behaviours into groups based on the complexity and success rate of each behavior. The simplest (Type 1) behaviours are solely based on the blind movement of the robot (straight, backup, curved). The second level (Type 2) of behaviour (circling, bouncing, forward-backup) includes behaviours with noticeable use of sensor data. The highest level (Type 3) of behaviour is called sniffing and demonstrates obstacle detection and avoidance. The perfect sniffing behaviour involves obstacle sniffing and straight motion behaviours that combine into smooth obstacle avoidance motion around the entire testing environment. Summary of the observed behaviours is provided in Fig. 3.2.

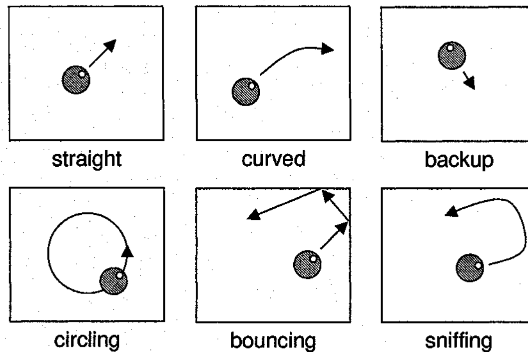

Fig. 3.2. Summary of behaviours learned during experimentation with the Khepera robot.

In our analysis of method performance, we examine population entropy stability, average chromosome complexity stability, and average generation of initial behaviour occurrence. Entropy and complexity stability is defined as a gradual change of the measured values over time without large abrupt value changes.

For the obstacle avoidance task, the representation method with the most stable entropy values is the ARL method. The linear genome and ADF methods also provide long, stable entropy values but with larger variations. The MA and tree-based representations provide the worst stability with large drops of entropy values. Most stability in the average chromosome size values is seen with the linear genome method. Among the HGP methods, the most stable complexity measures are seen with the ARL method and least stable with the MA method.

Type 2 and 3 behaviours are analyzed to calculate average generation values of first occurrence of the behaviours. We do not take into consideration Type 1 behaviours since they are not directly applicable to the studied task. Summary of the results of our behaviour calculation can be found in Fig.

3.3. The method with best (smallest) values is the ARL HGP method and with worst (largest) values is the linear genome GP method. Overall, the HGP methods perform comparable to the tree-based GP method. Trace run of perfect evolved obstacle avoidance behaviour is shown in Fig. 3.4.

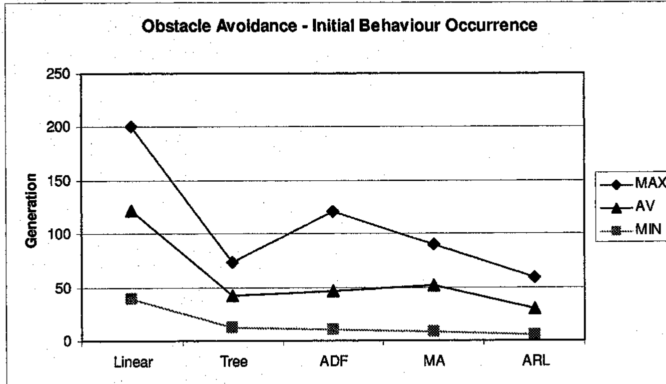

Fig. 3.3. Graphs of minimum, maximum, and average generations of first detection of Type 2 and 3 obstacle avoidance behaviour for each chromosome representation method.

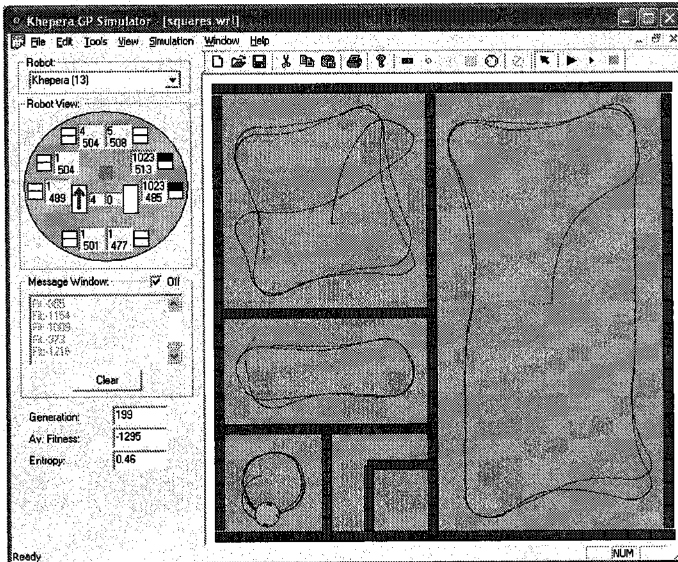

Fig. 3.4. Trace runs of perfect evolved obstacle avoidance behaviour in various testing environments.

3.6.2 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 is to teach the robot to walk around the boundaries of obstacles with a certain desirable distance away from the obstacles. The learned task should include some obstacle avoidance behaviour; however, that is not the main requirement of the experiments.

The wall following fitness function is composed of a sensor part and a motor part. The sensor part computes a sensor value from a subset of the robotic sensor values. The motor part is calculated by computing an absolute motor sum minus the absolute value of the difference. The fitness function is provided in Fig. 3.1. In our experiments, we set the values for the free parameters of the fitness function as follows: $\forall_i a_i = 1, \alpha = 100, \beta = 1$. Parameter values and fitness function definition were chosen based on tuning experiments.

Algorithm 3.1 Wall Following Fitness Function

input: Left $= a_0 \cdot s_0 + a_1 \cdot s_1 + a_2 \cdot s_2$,
Right $= a_5 \cdot s_5 + a_4 \cdot s_4 + a_3 \cdot s_3$,
MotorPart $= |m_1| + |m_2| - |m_1 - m_2|$;
output: Fitness;
1. if (Right ¿ 1023)
2. RightSensorPart = 1000 - Right;
3. else if (Right ¡ 20)
4. RightSensorPart = (1000/20) * Right;
5. else
6. RightSensorPart = 1000;
7. if (Left ¿ 1023)
8. LeftSensorPart = 1000 - Left;
9. else if (Left ¡ 20)
10. LeftSensorPart = (1000/20) * left;
11. else
12. LeftSensorPart = 1000;
13. Fitness = $\alpha\cdot$ MotorPart + $\beta\cdot$ (RightSensorPart + LeftSensorPart);

Only six sensors ($s_0 - s_5$) are used in calculating the sensor part of the fitness calculation. The sensors represent the side and front sensors of the robot. The calculated sensor part value acts as either pleasure or pain depending on the sensor values. The robot is punished when it is either too far away from a wall or too close to it. The training environment consists of a long, straight stretch of corridor and curved environment boundaries.

Summary of observed behaviours is provided in Fig. 3.2. We partition the behaviours into categories based on their relative performance and success. The Type 1 category is of poor wall following behaviour and consists of simple wall-bouncing and circling behaviours. The Type 2 category of good

wall following behaviour contains wall-sniffing and some maze following. The best behaviour category, Type 3, consists of perfect maze following behaviour without wall touching.

The most stable entropy is noticed in experiments using the ARL HGP method. The least stable entropy is observed using the ADF method and includes a large initial drop of entropy values to a low, stable level. Good stability of average size values is seen in the linear genome GP and ARL HGP methods. The largest drops in average chromosome size are noticed with the MA method.

We calculate average generation values of first occurrence of Type 2 and 3 behaviours. Type 1 category bahaviour is not directly applicable to the studied task. Summary of our behaviour calculation can be found in Fig. 3.5. The ARL method produces the best average results with smallest deviation whereas the worst performance is seen using the MA method. Trace run of perfect evolved maze-following behaviour is shown in Fig. 3.6.

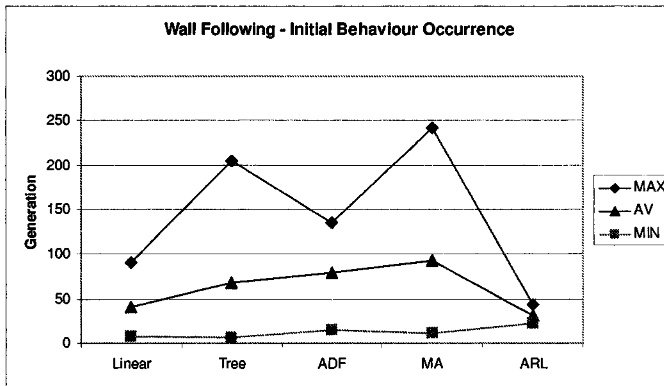

Fig. 3.5. Graphs of minimum, maximum, and average generations of first detection of Type 2 and 3 wall following behaviour for each chromosome representation method.

3.6.3 Light Avoidance

The light avoidance task is similar to the obstacle avoidance task but relies on the ambient light sensors of the robot instead of the proximity sensors. The source of light in the training and testing environments is composed of 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 for light avoidance is derived from the fitness function for the obstacle avoidance task. The function contains a pleasure part

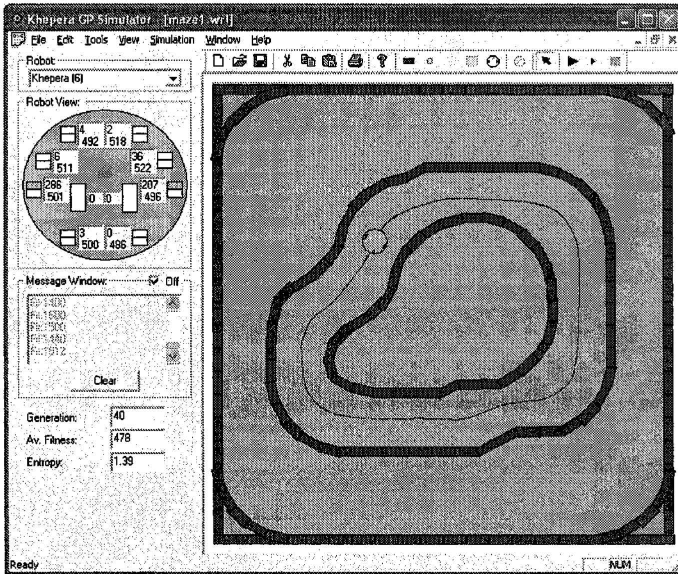

Fig. 3.6. Trace run of perfect evolved maze-following behaviour.

computed from the motor values of the robot and a pain part computed from the light sensors. Proximity sensors are not part of the fitness evaluation. A formal definition of the function is given as:

$$Fitness = \alpha(|m_1| + |m_2| - |m_1 - m_2|) - \beta(4000 - \sum_{i=0}^{7} l_i) \tag{3.5}$$

where m_1 and m_2 are motor values and l_0 to l_7 are ambient light sensor values. We set 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 subtract the sensor sum from 4000 (8 sensors of 500 value each) to make the fitness function behave similar to the fitness function for obstacle avoidance. Parameter values were chosen based on tuning experiments.

The training environment is composed of a rectangle of darkness surrounded by lights and a circular light island in the middle of the darkness area. The testing environment contains a similar dark rectangular area without the middle island.

We subdivide the learned behaviours of the robots into two categories. The Type 2 category of behaviour consists of circular, oval or uneven robot maneuvers with low degree of light detection and avoidance. Type 3 behaviour classifies definite light detection and avoidance behaviours. Perfect behaviour usually consists of travelling around the boundary of the dark area in the

testing environment. Summary of possible behaviours can be found in Fig. 3.2 with the substitution of light boundaries for obstacle boundaries.

The most stable entropy is noticed with the linear genome method and the least stable with the ADF method. Most stable average size values are noticed using the ARL method. The linear genome and tree-based representations also provide quite stable average size behaviour. The worst average size stability is seen with the MA representation method.

Results with Type 2 and 3 light avoidance behaviour are processed to calculate average generation values of first occurrence of the behaviour. Summary of our behaviour calculation results can be found in Fig. 3.7. The best (lowest) values are from experiments using the ARL method while the worst (highest) values are from linear genome experiments. The HGP methods perform comparable to or better than the tree-based method. Trace runs of perfect evolved light avoidance behaviour are shown in Fig. 3.8.

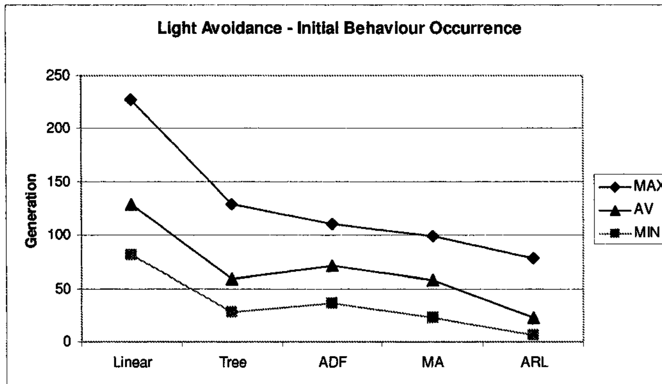

Fig. 3.7. Graphs of minimum, maximum, and average generations of first detection of Type 2 and 3 light avoidance behaviour for each chromosome representation method.

3.7 Summary

Our research deals with evolution of robotic controllers for the Khepera robot. We are interested in the population of individuals making up the robotic controller. 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. The definition of the fitness function can influence the population contents and thus the resulting behaviours.

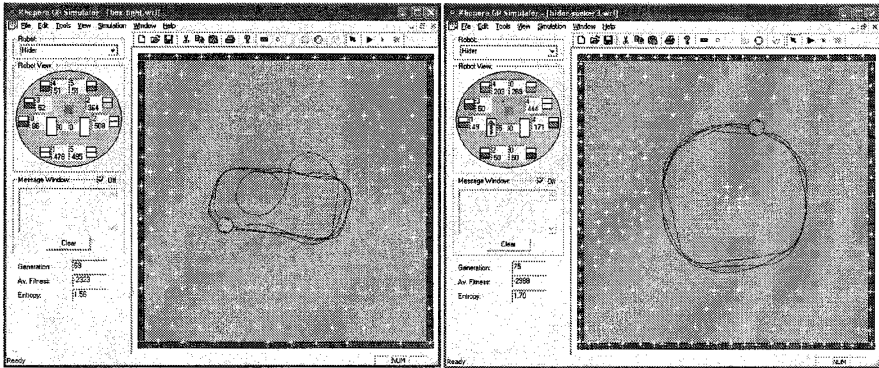

Fig. 3.8. Trace runs of perfect evolved light avoidance behaviour in different testing environments.

Robotic controllers often over-adapt 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 notice that sharp corners of the environment form an area of difficulty for the robotic controller. This is probably caused by a corner fitting between the fields of view of the proximity sensors.

The population entropy value is an important indicator of population diversity in our experiments. Good trained behaviour is found in populations with relatively high entropy value (above 0.6). Low entropy value signifies convergence in the population which usually accompanies a convergence to a low average chromosome size. Populations of individuals with low chromosome size do not contain enough information to successfully search for a good solution. No special measures are taken to prevent bloating in our experiments; however, a maximum tree height (or maximum number of instructions for the linear genome method) is specified for each chromosome.

We examine three HGP learning methods: Automatically Defined Functions (ADF), Module Acquisition (MA), and Adaptive Representation through Learning (ARL) and two GP methods: tree-based and linear genome. Robotic controllers using each method are able to evolve some degree of proper behaviour for each learning task. Summary of method performance is available in Table 3.3. We treat the tree-based method as a basis for evaluating the performance of the linear genome and HGP methods. We define the behaviour of the tree-based method as average. Sample plots of population entropy and chromosome complexity observed in tree-based experiments are provided in Fig. 3.9.

The best entropy and best average size stability is seen with experiments using the ARL method (see Fig. 3.10). The worst entropy behaviour is seen mainly with the ADF method (as shown in Fig. 3.11) but also with the MA

Table 3.3. Summary of results from our experiments for each of the studied methods. Behavioural performance is based on first occurrence of good evolved behaviour. Methods are compared based on relative performance.

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

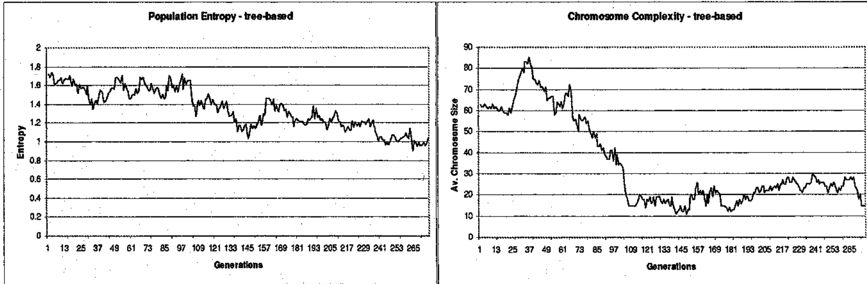

Fig. 3.9. Graphs of entropy and average size vs. the number of generations in a sample run using the tree-based method.

and tree-based methods. The worst average size behaviour is noticed with the MA method for all the studied tasks (see Fig. 3.12).

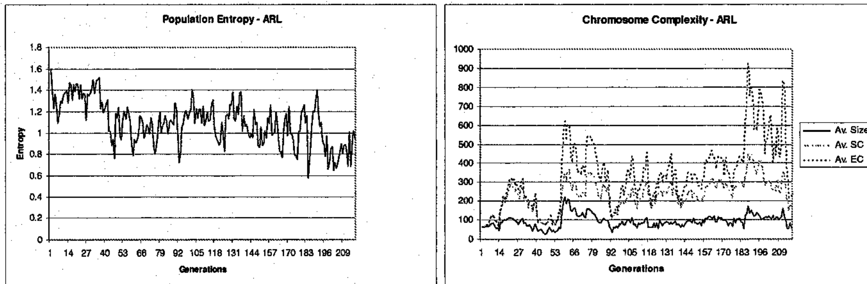

Fig. 3.10. Graphs of entropy, average size, average SC and average EC vs. the number of generations using the ARL method.

Throughout most of our experiments, the linear genome method enforces a stable level of entropy and average chromosome size (as seen in sample plot of Fig. 3.13). This behaviour is probably due to the different crossover operator in the linear genome method than in the tree-based methods and by the additional mutation operator. Because of the stable entropy levels,

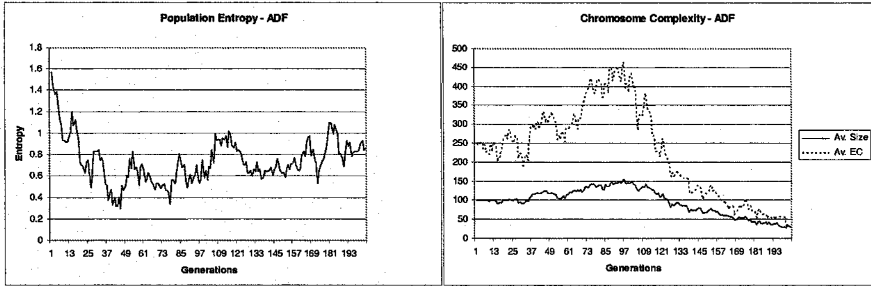

Fig. 3.11. Graphs of entropy, average size and average EC vs. the number of generations in a sample run using the ADF method.

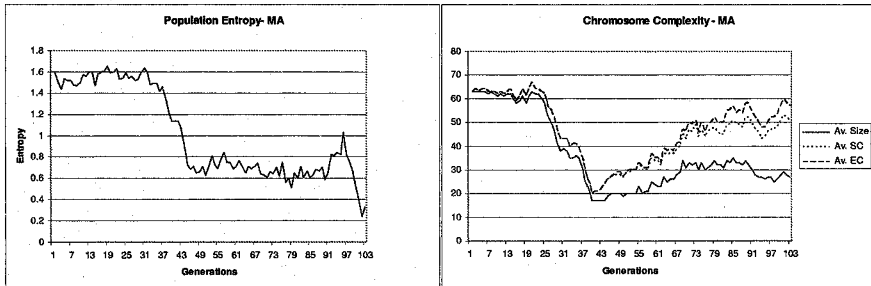

Fig. 3.12. Graphs of entropy, average size, average SC and average EC vs. the number of generations in a sample run using the MA method.

populations of 50 individuals are enough to provide stable behaviour for many generations.

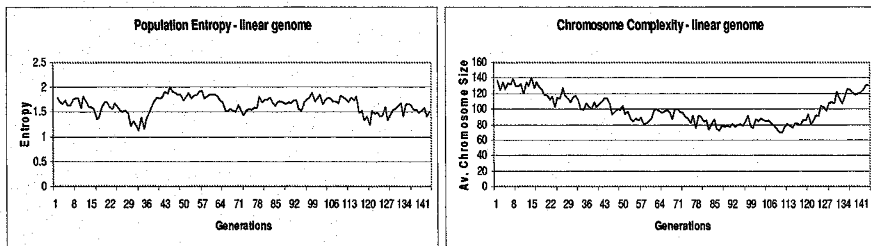

Fig. 3.13. Graphs of entropy, average size, average SC and average EC vs. the number of generations using the linear genome method.

With the tree-based chromosome representations, entropy value stability depends on the definition and parameter values of the fitness function. Tendency toward smaller program sizes is seen with the half-and-half chromosome creation method or the full method with small sized populations. To keep pro-

gram sizes and entropy at high values for reasonable time, we need to evolve populations of 100 or more individuals (with the exception of the ARL method discussed below).

The average generation values of initial good behaviour occurrence are usually highest with the linear genome method. However, the methods are based on different population sizes and individual sizes so it is difficult to draw conclusions from the raw results. With our implementation of the linear genome method (through a genome interpreter) the evolution time is similar to the time using the tree-based representation (with equivalent population size and tree size settings). The main difference between the methods is the contents of the function sets.

The ADF method uses a predefined, constant function set containing one or more ADFs. Function call acquisition occurs only through crossover with individuals of the population. The ADFs inside individuals showing proper evolved behaviour are usually quite large and complex with no noticeable patterns. It is possible that in our experiments the ADFs only provide few extra tree levels of instructions. The ADF method runs provided performance that was usually below that of the tree-based method and sometimes the worst of all HGP methods. Fig. 3.14 shows the code of a sample ADF program taken from a population with learned light avoidance behaviour.

The slowest method of function creation is the MA method. Most of the individuals in the population with proper evolved behaviour do not contain any of the functions in the module set. The creation of functions produces program size loss which in turn often lowers the entropy of the population. The behavioural performance of the MA method is usually worse than that of the tree-based method. Since similar experimental settings are used for the two methods, we can deduce that the function creation of our MA method disrupted the task learning instead of helping it. Fig. 3.15 shows the code of a sample MA program taken from a population with learned wall following behaviour.

The ARL method displays the most stable entropy and average chromosome size behaviour in most experiments. This stable behaviour is observed only with function creation, thus we think that the function creation and new individual creation processes are responsible for the stability. The method also achieves the best time and smallest deviation to reach good evolved behaviour in most experiments.

The number of functions created by the ARL algorithm depend on each run but do not grow monotonically as first expected. The function set grows and shrinks throughout the runs of the algorithm. The functions usually contain simple arithmetic operators working on function parameters. Many of the functions from populations with proper evolved behaviour contain division and addition operators that seem to calculate some form of ratio of the function parameters. Since such ratios can be helpful in all of our studied tasks, we think that some of the evolved functions are of benefit to the indi-

```
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
```

Fig. 3.14. Code of a sample ADF program from population showing light avoidance behaviour. ADF function definition shows characteristic large size and complex format.

viduals. Code of a sample ARL program taken from a population with learned obstacle avoidance behaviour can be found in Fig. 3.16.

Algorithms and strategies of solving problems can usually be improved to yield better solutions. Our research enables us to indicate areas of possible improvement to the studied genetic programming algorithms for the domain of robotic control. We feel that population diversity (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 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 controller settings.

Modification of fitness function parameters leads to strong statistical and behaviour changes in the evolving population. Definition of the fitness function is thus a very important aspect for evolution of correct solutions. We choose the fitness function definitions and parameter values that produce the best performance in trial runs. However, more testing of fitness functions and their

```
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
```

Fig. 3.15. Code of a sample MA program from population showing wall following behaviour.

```
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
```

Fig. 3.16. Code of a sample ARL program from population showing obstacle avoidance behaviour. Characteristic ARL functions are visible.

parameters should be done to identify the optimal settings for each learning task.

The ADF method builds ADFs of initial size equivalent to the main program body. We feel that smaller building blocks (functions) are more useful for robotic controllers. The sizes of the ADFs do not decrease well enough before the population prematurely converges. 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.

We feel that the poor performance of the MA method is due to the creation of modules which lowers the average program tree size. Since no mechanism exists to counteract this loss of program size and accompanying loss of entropy, the population often converges prematurely to suboptimal solutions. We propose that the probability-based compression and expansion operator invocation of the MA method be replaced by a need-based operator invocation (similar to that found in the ARL method). This new operator invocation should lead to better performance through adjustments of operator frequencies based on population needs.

The ARL method contains a mechanism to neutralize the bad effects of function creation. Thus, the method exhibits very stable entropy and average size behaviours while quickly evolving high performing robotic controllers. The creation of random individuals using the enriched function set at the start of a new epoch provides the genetic algorithm with fresh search material. The functions found in the adaptive representation step of the algorithm are small and seem better building blocks than the functions in the ADF method. We feel that best performance can be achieved by some kind of a dynamically evolving entropy threshold calculation.

Influx of random individuals to the population during evolution can lead to problems. Too many new random individuals can 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 a new population and would ensure that 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 work, we evolve 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 [20] 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.

References

1. P.J. Angeline, Genetic Programming and Emergent Intelligence, In K.E. Kinnear, Jr. (ed.), Advances in Genetic Programming, chapter 4, pp. 75–98, MIT Press, 1994
2. P. J. Angeline and J. B. Pollack, The evolutionary induction of subroutines, In Proceedings of the Fourteenth Annual Conference of the Cognitive Science Society, Lawrence Erlbaum, Bloomington, Indiana, USA, 1992
3. R.C. Arkin, Behavior-Based robotics, MIT Press, Cambridge, MA, 1998
4. W. Banzhaf, P. Nordin and M. Olmer, Generating Adaptive Behavior using Function Regression within Genetic Programming and a Real Robot, In J.R. Koza, K. Deb, M. Dorigo, D.B. Fogel, M. Garzon, H. Iba, and R.L. Riolo (eds.), Genetic Programming 1997: Proceedings of the Second Annual Conference, pp. 35–43, Morgan Kaufmann, San Francisco, CA, USA, 1997
5. W. Banzhaf, D. Banscherus, and P. Dittrich, Hierarchical Genetic Programming using Local Modules, Series Computational Intelligence, Internal Report of SFB 531, No. 56/99, Univ. of Dortmund, D-44221 Dortmund, Germany, 1999
6. R. Brooks, A robust layered control system for a mobile robot, IEEE Journal of Robotics and Automation, **2**(1), 1986
7. R. Brooks, Artificial Life and Real Robots, In F. J. Varela and P. Bourgine (eds.), Toward a Practice of Autonomous Systems: Proceedings of the first European Conference on Artificial Life, pp. 3–10, MIT Press-Bradford Books, Cambridge, MA, 1992
8. K.E. Kinnear, Jr., Alternatives in Automatic Function Definition: A Comparison Of Performance, In K.E. Kinnear, Jr. (ed.), Advances in Genetic Programming, chapter 6, pp. 119–141, MIT Press, 1994
9. J.R. Koza, Genetic Programming: On the Programming of Computers by Means of Natural Selection, MIT Press, Cambridge, MA, 1992
10. J.R. Koza, Evolution of subsumption using genetic programming, In F. J. Varela and P. Bourgine (eds.), Proceedings of the First European Conference on Artificial Life, Towards a Practice of Autonomous Systems, pp. 110–119, MIT Press-Bradford Books, Cambridge, MA, 1992
11. J.R. Koza, Genetic Programming II: Automatic Discovery of Reusable Programs, MIT Press, Cambridge, MA, 1994

12. J.R. Koza and J.P. Rice, Automatic programming of robots using genetic programming, In Proceedings of Tenth National Conference on Artificial Intelligence, pp. 194–201, AAAI Press/MIT Press, 1992
13. J.R. Koza, F. H. Bennett III, D. Andre, and M. A. Keane, Genetic Programming III: Darwinian Invention and Problem Solving, Morgan Kaufmann, San Francisco, 1999
14. M. J. Mataric, Behavior-Based Control: Examples from Navigation, Learning, and Group Behavior, Journal of Experimental and Theoretical Artificial Intelligence, **9**(2–3),1997
15. P. Nordin, A Compiling Genetic Programming System that Directly Manipulates the Machine-Code, In K.E. Kinnear, Jr. (ed.), Advances in Genetic Programming, chapter 14, pp. 311–331, MIT Press, Cambridge, MA, 1994
16. P. Nordin and W. Banzhaf, Genetic Programming Controlling a Miniature Robot, In E.V. Siegel and J.R. Koza (eds.), Working Notes for the AAAI Symposium on Genetic Programming, pp. 61–67, AAAI, MIT, Cambridge, MA, USA, 1995
17. P. Nordin and W. Banzhaf, A genetic programming system learning obstacle avoiding behavior and controlling a miniature robot in real time, SysReport 4/95, University of Dortmund, Fachbereich Informatik, 1995
18. P. Nordin and W. Banzhaf, Real Time Evolution of Behavior and a World Model for a Miniature Robot using Genetic Programming, SysReport 5/95, Dept. of CS, University of Dortmund, 1995
19. 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), pp. 107–140, 1997
20. P. Nordin and W. Banzhaf, Real Time Control of a Khepera Robot using Genetic Programming, Cybernetics and Control, **26**(3), pp. 533–561, 1997
21. 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), pp. 105–116, 1998
22. C. Reynolds, An Evolved, Vision-Based Behavioral Model of Coordinated Group Motion, In J. Meyer, H.L. Roitblat and S.W. Wilson (eds.), From Animals to Animats II: Proceedings of the Second International Conference on Simulation of Adaptive Behavior, MIT Press-Bradford Books, Cambridge, MA, 1993
23. J.P. Rosca, Entropy-Driven Adaptive Representation, In J.P. Rosca (ed.), Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications, pp. 23–32, Tahoe City, California, USA, 1995
24. J.P. Rosca, Hierarchical Learning with Procedural Abstraction Mechanisms, PhD thesis, University of Rochester, 1997
25. 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
26. J.P. Rosca and D.H. Ballard, Discovery of Subroutines in Genetic Programming, In P.J. Angeline and K.E. Kinnear, Jr. (eds.), Advances in Genetic Programming 2, chapter 9, pp. 177–202, MIT Press, Cambridge, MA, USA, 1996
27. J.P. Rosca and D.H. Ballard, Genetic Programming with Adaptive Representations, Technical Report TR 489, University of Rochester, Computer Science Department, Rochester, NY, USA, 1994