An On-Line Method to Evolve Behavior and to Control a Miniature Robot in Real Time with Genetic Programming

Peter Nordin Wolfgang Banzhaf
Fachbereich Informatik
Universität Dortmund
44221 Dortmund, Germany
email: nordin,banzhaf@ls11.informatik.uni-dortmund.de

Abstract

We present a novel evolutionary approach to robotic control of a real robot based on genetic programming (GP). Our approach uses genetic programming techniques that manipulate machine code to evolve control programs for robots. This variant of GP has several advantages over a conventional GP system, such as higher speed, lower memory requirements and better real time properties. Previous attempts to apply GP in robotics use simulations to evaluate control programs and have difficulties with learning tasks involving a real robot. We present an on-line control method that is evaluated in two different physical environments and applied to two tasks using the Khepera robot platform: obstacle avoidance and object following. The results show fast learning and good generalization.

1 Introduction

Autonomous robots or agents have a large potential for the future. There are many situations where they could relieve humans from dangerous, difficult or monotone tasks. We are convinced that many of these future agents will need to show adaptive behavior in order to successfully master their tasks. The real world is simply too complex and unpredictable to be managed without learning. In nature we can see how higher animals with adaptive capabilities have been very successful in almost all environments. The most flexible animal with the highest capability of learning – man – has been almost too successful in its attempt to affect and control its environment.

There are several problems that an autonomous robot has to cope with in a real world situation:

1. Its model of the world is often inaccurate and out of date,
2. the sensors of the robot will be noisy and the actuators imprecise,
3. the environment is likely to be dynamic and constantly changing.

The process of designing an agent can be significantly simplified if the designer does not have to explicitly add every detail of the environment, the robot or the suitable behavior and, instead, can rely on the learning ability of the agent.

A variety of adaptive and learning control architectures have been proposed over the years. Many of these approaches are built on reinforcement learning techniques, see (Kröse, 1995) and (Kaelbling, Littmann & Moore, 1996). Reinforcement learning is a term used in connection with agents learning behavior through "trial-and-error" interactions in their (dynamic) environment. In other words, the agents have to search through their behavioral space in order to find an appropriate behavior. Many different techniques exist to search a large space like the behavioral space of an agent. Thus, the number of reinforcement learning variants is manifold, given the enormous number of different search techniques which could be used.

A large class of those techniques systematically use knowledge generated within the system and supported by outside methods, e.g., statistics (for a review, see Kaelbling et al., 1996) or PAC learning (Greiner & Isukapalli, 1996).

Algorithms inspired by natural processes, on the other hand, are applicable as well. Many connectionist approaches have been proposed to achieve learning for autonomous agents / robots. Millán in (Millán, 1996) has worked on a neural network architecture for the acquisition of efficient navigation capabilities of a robot in very short time.

Evolutionary algorithms provide an efficient framework for this kind of search as well, since the fitness function used by these algorithms could be interpreted as the reinforcement signal of reinforcement learning. Genetic algorithms and

genetic programming are hence potentially very useful in the area of reinforcement learning. A considerable number of researchers have been examining this potential over recent years.

Meeden (Meeden, 1996) uses a combination of connectionist and evolutionary approaches to develop a robot controller that is eventually able to build up memory-dependent behavior, characteristic of planning systems. Floreano and Mondada (Floreano, & Mondada, 1996) show how the development of an internal neural topographic map allows a robot to choose an appropriate trajectory, given a certain starting location and energy level. Neural Network controllers have been evolved even earlier with genetic algorithms (Cliff, 1991; Harvey, Husbands & Cliff, 1993; Floreano & Mondada, 1994) with considerable success. Evolutionary algorithms for creating an artificial neural network which controlled an autonomous land vehicle have been tested in applications such as road following tasks (Baluja, 1996).

A more general class of models is known under the heading selectionist neural networks. Selectionist approaches to neural networks have a decade old tradition starting from the seminal work of Edelman on *Neural Darwinism* (Edelman, 1987).

Donnart and Meyer (Donnart & Meyer, 1996) consider the control and adaptation of behavior in the context of classifier systems. Both reactive and planning rules are implemented using what they call a motivationally autonomous animat which behaves according to its state of perception of the environment. Colombetti, Dorigo and Borghi, (Colombetti, Dorigo & Borghi, 1996) even propose "Behavior Engineering" as a new technological area whose aim is to provide methodologies and tools for developing autonomous robots.

An interesting variant of learning is the SAMUEL system (Grefenstette, Ramsey & Schultz, 1990; Potter, De Jong & Grefenstette, 1995) which uses an evolutionary system to evolve sequential decision rules for a simulated robot.

Each of these approaches is a reasonable starting point for developing learning techniques for real-time control of robots. However, we feel that there are overwhelming advantages to having each controller represented as a computer program in a symbolic programming language. Consequently, our research uses Genetic Programming (GP) as the basis for an adaptive control method.

1.1 The Advantages of Genetic Programming

Genetic programming uses a genetic algorithm to evolve *computer programs*. The complete controller of the robot is evolved as a computer program in a general purpose programming language. GP has the following advantages as a learning control algorithm:

- GP produces purely symbolic output (programs). This can be beneficial if we want to analyze specific solutions – if we want to know what it has

learned and why the robot is behaving in a certain way. Understanding the behavior of ANNs or classifier systems may be more difficult.

- Using a general program as the data structure defining behavior is flexible. Any behavior can be represented in a general purpose programming language. If, for instance, special function primitives are desirable then they can be directly added to the *function set* of the system. This can also reduce the need for preprocessing of sensor data and postprocessing of actuator commands.

- In our case we use a special variant of GP which evolves binary machine code and allows for very efficient implementation. This approach is up to 2000 times faster than a LISP-based GP system and it has, in certain instances, been shown to learn significantly faster than a neural network (Nordin, 1994).

- The GP variant we use is also very memory efficient. It uses only 32 KBytes (KB) for the kernel and another 50 KB to store the population. The complete system can run on a micro-controller. In order to fully exploit their application areas autonomous adaptive robots must be able to efficiently use computer resources. In the future, micro and nano technology robotics may increase the pressure towards efficient use of computer resources. Furthermore, the memory consumption of our approach is constant and does not need garbage collection. This makes it well suited to real-time applications.

- The objective of the robot's learning task is defined in a single fitness function making it straight forward to modify the objective. There is no procedural or representational knowledge necessary to formulate the fitness function.

- Under the right circumstances a GP system shows excellent generalization capabilities, (Nordin, 1994; Nordin & Banzhaf, 1995a; Nordin, Francone & Banzhaf, 1996; Francone, Nordin & Banzhaf, 1996). Good generalization will give the agent high probability of coping with a changing dynamic environment.

- The genetic programming technique usually gives little risk of *over-learning*, i.e. over-specialization to a certain situation.

- Genetic programming can be applied with a minimum of a priori knowledge.

- Cliff et al. advocate "that the primitives manipulated in the evolutionary process should be at the lowest level possible" (Cliff, Harvey & Husbands, 1993). This will give minimal constraints for solutions and will reflect

as few of the designer's prejudices as possible. A machine code GP system manipulates the lowest level language possible in a computer and the design issues of the language used for evolution are thus not affected by robot researcher's prejudices. Instead, the language is defined by the manufacturer of the micro processor. It could be argued that this "objective" representation ensures that results obtained are less the product of the system design and more the result of adaptive behavior.

1.2 GP in the Real World

Most previous experiments with GP and robot control have employed a simulated robot and environment. Judging from our own experience and from the general opinion in the research field (Brooks, 1992) we strongly believe in experiments with *real* robots for several reasons.

Simulations often display several weaknesses when compared to experiments in the real world. The simulated world is often too ideal, resulting in the agent learning to react to small details in the simulation that will not be reliably detected in a real environment. For example, the agent might react to a specific exact sensor value that occurs all the time in a simulation but occurs only occasionally in the real world with real noisy sensors.

Problems of this kind are prevalent in most research using GP to evolve robot control programs. As a result, the control algorithms evolved by GP are often brittle. Reynolds acknowledges this issue in his discussion of control programs for obstacle avoidance behavior evolved by GP:

> However, those evolved control programs were found to be quite "brittle". They do not solve the general obstacle avoidance problem. They can only guide the vehicle through one specific obstacle course. They will not even work in the *same* obstacle course if any of the vehicle's initial conditions (position or orientation) is slightly perturbed. These brittle controllers are a bit like a "house of cards" which stands as long as absolutely nothing changes (Reynolds, 1994).

Other experiments display a similar brittleness in behavior. One way to improve the simulation and to evolve more robust controllers is to add noise to the environment. This reduces the brittleness but does not eliminate it (Reynolds, 1994). Further examples of GP and simulated control include (Koza, 1992; Atkin & Cohen, 1994; Fraser & Rush, 1994; Handley, 1994; Sims 1994; Spencer, 1994; Teller, 1994).

More serious than the brittleness of controllers is the problems of moving the experiments from a simulated robot to a real one. The normal way of doing GP and control is to create a population of random programs. Each program then controls the robot for a predefined number of time steps. The robot is reset exactly to the same location before a program gets evaluated.

The individual program's performance is judged according to a goodness criterion. When all programs are evaluated the best performing are reproduced into the next generation. Subsequently, the genetic operators crossover (recombination) and mutation are applied. These steps are repeated until a good solution is found. Normally about 100 generations are needed for convergence to a good solution. If we have a population size of 500 solution candidates (programs) and each individual is evolved during 100 time steps then we need $100 \times 500 \times 100 = 5,000,000$ time steps of evaluation before a good controller is found. This is impractical for a real robot. The dynamic response from the environment generally requires at least 300 *ms* for meaningful feed-back. This means 1,500,000 seconds, or two weeks of evolution. The robot also needs to be reset to its initial position before each program evaluation, in this case 50,000 times. A *simulation* could, on the other hand, be run much faster than real time and resetting the position would not be a problem either.

In this paper we present the first on-line version of a GP control architecture which allows us to efficiently use GP to control a real robot. The principle is based on a probabilistic sampling of the environment where different individuals are evaluated in different situations. Though this approach gives an *unfair* judgment in individual situations, and selection will be based on a comparison of seemingly incommensurable situations, the overall better performing program individuals will survive in the long run.

Motivation Summary

In summary, our reasons for the experiments described in this paper were:

- We believe that adaptive and learning behavior is necessary for the successful application of autonomous agents in real world situations.
- We are also convinced that experiments with real robots are needed in order to evaluate different approaches to robot control.
- Genetic programming might have advantages for control in certain situations. Advantages could be speed, generalization, memory consumption, analysis of solutions and flexible representation.
- No efficient implementation of GP with real robots exists up to date.

We have therefore suggested a method for on-line GP with a real robot and also introduced the machine code manipulating GP technique in robot control.

Reading Guidance

The rest of this paper is structured as follows: We start by defining our methods with a description of genetic programming (section 2.1) and an introduction to the compiling genetic programming (section 2.1.1) approach. The on-line control

method is presented in section 2.2, while the platform for our experiment, the Khepera robot, is presented in section 2.3. The objectives, the tasks of obstacle avoidance and object following, are defined in section 2.4. The results of our experiments are found in section 3. Finally we summarize our conclusions and outline future work in section 4.

2 Methods

2.1 Genetic Programming

Genetic programming (Koza, 1992) uses an evolutionary technique to *breed* programs. First, a goal is defined by specifying a quality function. This so-called fitness function could, for instance, be the error in a function regression[1]. The population – a set of solution candidates – is initialized with random content, i.e., with random programs. In each "generation" the fittest individual programs are selected for reproduction. These relatively fit individuals produce offspring through recombination, often called crossover, and through mutation. Various methods exist for selection and reproduction. The idea is always that better individuals and their offspring gradually replace the worse performing individuals[2].

The individual solution candidate can be represented as a tree which also constitutes its genome. This tree can be seen as the parse tree of a program in a programming language. Recombination is performed by two parents exchanging subtrees, as shown in Figure 1.

A typical application of GP is symbolic regression. Symbolic regression is the task of inducing a symbolic equation, function or program which fits given numerical data. Genetic programming is ideal for symbolic regression and most GP applications could be reformulated as variants of symbolic regression. A GP system performing symbolic regression takes a number of numerical input/output relations, called fitness cases, and produces a function or program that is consistent with these fitness cases. Consider, for example, the following fitness cases:

```
f(2) = 6
f(4) = 20
f(5) = 30
f(7) = 56
```

One of the infinite number of perfect solutions would be $f(x) = x^2 + x$. The

[1] In this case the aim would be to minimize this error function.

[2] Genetic programming has some similarities with "beam search" (Lowerre & Reddy, 1980, Rosenbloom, 1987), if the population is regarded as the memory buffer and the fitness as a stochastically assigned priority.

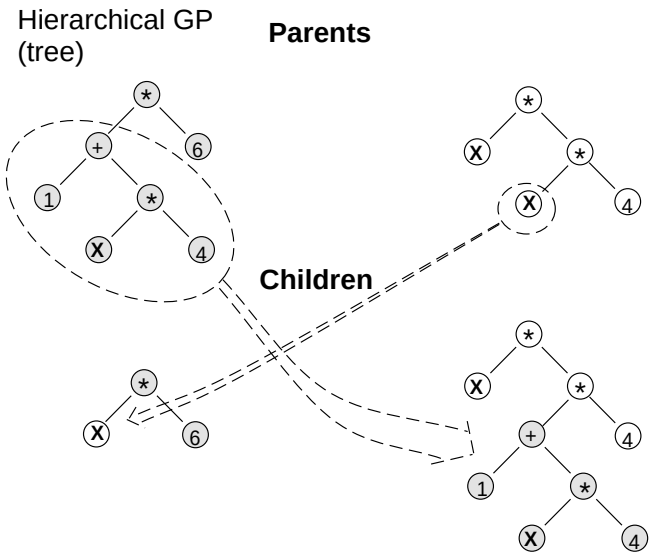

Figure 1: Hierarchical crossover in genetic programming: Subtrees are exchanged between parents.

fitness function might be the sum of the differences between an individual's (function's) actual output and the target output over all the fitness cases.

The *function set* or the function primitives could, in this case, be arithmetic operators $+, -, \times, /$, as shown in Figure 1. Terminals could be chosen to be the input value x and constants.

The two most important decisions to be made before the training of a genetic programming system can start are to choose a good fitness function and to choose the right function and terminal set. The fitness function should allow for a gradual improvement during evolution, since it should give meaningful feedback to the GP induction system. The function set should contain primitives relevant to the problem domain. Each function in the function set should also be syntactically closed – it should be able to gracefully accept all possible inputs in the problem domain.

2.1.1 Compiling Genetic Programming

The Evolutionary Algorithm we use in this paper is an advanced version of the Compiling Genetic Programming system (CGPS) described in (Nordin, 1994). The structures that undergo evolution are variable length strings of 32 bit instructions for a CPU of a standard computer with von Neumann architecture. This architecture is also known as a register machine architecture. A register

machine performs operations on a small set of registers. The 32 bits in the instruction represent simple arithmetic or logic operations such as "$a = b + c$" or "$c = b\&5$". The genetic operators are able to manipulate binary code directly. The actual format is the machine code format of a SUN-4 (Sparc, 1991).

Most other genetic programming approaches use a technique where a problem–specific language is executed by an interpreter. The individuals in the population are decoded at run time by a virtual machine. The data structures in those programs often have the form of a tree. This solution gives high flexibility and the ability to customize the language depending on the constraints of the problem at hand.

The disadvantage of this paradigm is that interpreting the program involves a large overhead. There is also a need to define more complicated genetic operators than with CGPS, which further decreases speed of the evolutionary process. Often the complete system and the genetic operators themselves are written in an interpreted language like LISP (Koza, 1992). This reduces performance in most hardware environments. Recently some systems have been presented written in compiler languages like C or C++, parsing structures equivalent to the programs used in a LISP implementation (Keith & Martin, 1993). This gives increased performance while it preserves the ability to be flexible with the representation and selection of a problem specific functions in the programs (function set). Most of these systems still interpret the programs in the population and that involves considerable overheads both in execution time and memory consumption.

We have implemented the idea of using the lowest level binary machine code as the "programs" in the population. Every individual is a piece of machine code that is called and manipulated by the genetic operators. There is no intermediate language or interpreting part of the program. This approach has enhanced performance by up to 2000 times compared to a conventional system using an interpreted language (Nordin & Banzhaf, 1995b). We call our approach "compiling" as the system generates binary code from the training set and there are no interpreting steps. The idea is to use the real machine instead of a virtual machine and it can be expected that the loss in flexibility will be well compensated for by increased efficiency.

Data and programs reside in the same memory in a computer with von Neumann architecture. Both, data and programs, are represented internally by fixed length numbers. These number will be interpreted either as data or as program depending on the state of the processor (see Figure 2). Each instruction defines an operation between the processors internal registers or between memory and internal registers. The idea behind compiling genetic programming is to treat the lowest level binary code just as an array of numbers when genetic operators are applied.

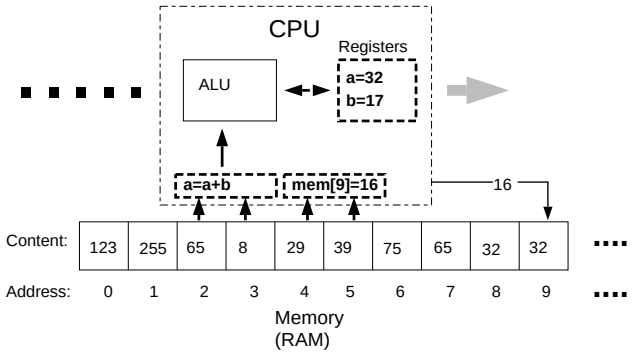

Figure 2: The operation of the CPU. Memory content is used as data or interpreted as instructions to manipulate data.

2.1.2 The structure of a machine code function callable as a C-function

The individuals in the population consist of machine code sequences resembling standard C-functions. The two most significant differences between a usual GP system and CGPS are that CGPS has a linear genome and that the crossover/mutation operators work on linear structures. These differences are forced by the use of machine code.

Figure 3 illustrates the structure of a GP individual in CGPS. There are four major parts:

1. The *Header* deals with the administration, needed when entering a function called by a C (main) routine. This normally means manipulation of the stack, for instance getting the arguments for the function from the stack. There could also be some processing to ensure consistency of processor registers. This part is often constant and can be added at the beginning of the initialization of the machine code individual in the population. Mutation and crossover operators must be prevented from changing this part of the code during evolution.

2. The *Footer* is similar to the header but operates in opposite order and "cleans up" after the function call. The footer must also be protected from change through genetic operators. The return instruction forces the system to leave the function and to return program control to the calling procedure. If variable length programs are desired, then the return operator could be allowed to move within a range defining minimum and maximum program size.

3. The *function body* consists of the actual program that evaluates the func-

tion.

4. A *buffer* is reserved at the end of each individual to allow for length variations of the program.

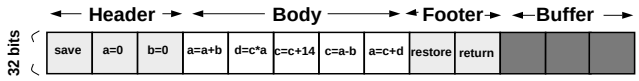

Figure 3: Structure of a program individual. It consists of 4 parts: Header, body, footer and buffer.

2.1.3 The genetic operators

The evolutionary algorithm has the following three operators:

- *Selection*: CGPS uses tournament selection described in section 2.2.1 below.
- The *mutation* operator changes the content of an instruction by varying constants or register references. It randomly changes one bit of the instruction provided certain criteria are fulfilled. If hitting the operation code (op-code) mutation ensures that the the resulting instruction is a member of the set of approved instructions. This avoids jumps, illegal instructions, bus errors or loops etc. Furthermore it also assures some arithmetic consistency, when, for instance, division by zero is prevented.
- The *crossover* operator works on variable length individuals. Crossover is only allowed *between* instructions, i.e. at 32–bit intervals in the binary string. The genome is snipped between 32–bit machine instructions and only blocks of code are exchanged that are separated at instruction boundaries. So crossover looks more like Genetic Algorithm crossover than it does resemble GP subtree crossover (Nordin, Francone & Banzhaf, 1996). Figure 4 illustrates the crossover method of CGPS (Nordin & Banzhaf, 1995b).

All operators ensure syntactic closure during evolution, i.e. they do not cause the computer to exit execution.

It is interesting to note that the DNA genome in nature has a linear structure as well. It is a double helix of four different organic molecules resembling a string of letters in a four letter alphabet. The linear sequence is made up of larger units called genes. Each gene codes for a specific protein or function. A gene could thus be seen as a segment of the genome which can be interpreted – and has a meaning – in itself. In our case there are lines of code each representing an

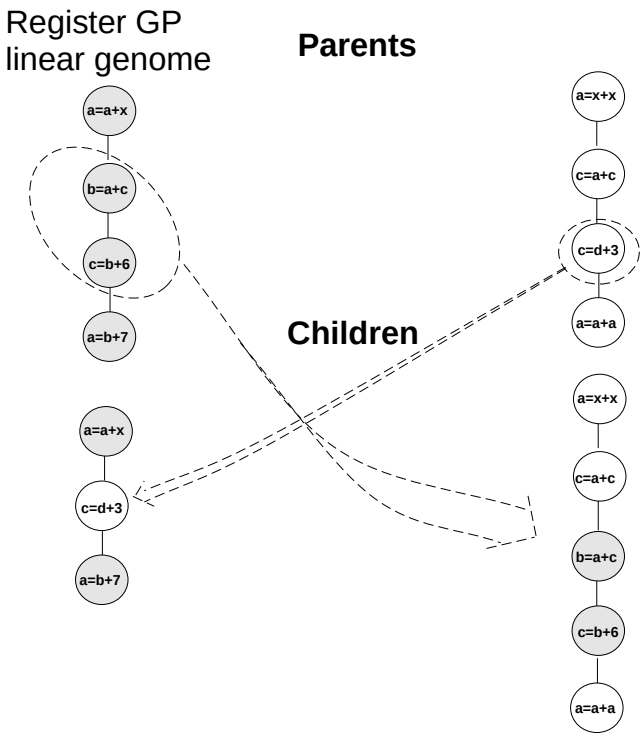

Figure 4: Crossover of a machine code program with a linear genome. Crossover takes place between instructions symbolized by circles.

instruction or command. Such a command is syntactically closed in the sense that it is possible to execute it independently of the others. This, therefore, has some similarity to a gene in nature – it consists of a syntactically closed independent structure which has a defined starting and ending point. It is in both cases treated as a structure separate from the whole genome structure. The method of applying crossover only between instructions/genes and mutation only within instructions/genes can also be used with computer languages other than machine code, see for instance (Nordin & Banzhaf, 1996a).

2.1.4 An Example Program

As we have seen above, a CGPS program is no more than a sequence of 32 bit binary machine instructions in memory. For example, an evolved CGPS program might be three 32 bit machine instructions. When executed, those

three instruction will cause the CPU to perform three simple operations on the CPU's hardware registers:

```
register2 = register1 + register2    (1)

register3 = register1 * 128           (2)

register3 = register2 Div register3  (3)
```

In the remainder of this section, we will follow the CGPS program individual using the above three instructions (1) - (3) through a single fitness evaluation.

Let's say that we have three independent variables with values of, respectively, 3, 2650, and 10 for the single fitness case. Let's also assume that we have chosen register 3 as the single output register. Our three instruction CGPS program would be evaluated for fitness in the following steps:

1. The hardware registers would be initialized with the values of the independent variables.
2. Each of the three instructions (1) - (3) would be executed in order.
3. The value of register 3 would then be used as the output of the system for the purpose of fitness evaluation.

Here is how the values of the hardware registers would change during the above fitness evaluation.

Register 1	*Register 2*	*Register 3*	*Description of Step*
3	2650	10	Initialize Registers With Independent Variables
3	2653	10	Execute Instruction (1)
3	2653	384	Execute Instruction (2)
3	2653	6	Execute Instruction (3)

Table 1: Execution of program (1)−(3) under specific starting conditions for evaluation purposes.

So the output of the above program for the fitness case of 3, 2650, 10 is 6 - that is, the value in the register that was chosen as the output register, Register 3. This output can now be compared to a target output and its fitness can be computed.

2.1.5 Summary

It should be clear by now that there is no specific data structure in CGPS such as trees or S-expressions that are evolved and which are then compiled or

interpreted into machine instructions. The only representation we use for the programs is the array of machine code instructions itself.

Of the many advantages of CGPS, we feel that the principal advantage is speed. Executing machine code directly is much faster than interpreting languages or dynamically creating programs. In fact, CGPS can perform about 240,000 fitness evaluations per second on a single processor SUN workstation.

The kernel of CGPS requires only 32 KBytes. Its small system size is partly due to the fact that knowledge of the language used is supplied by the CPU designer in hardware – there is no need to define the language and its interpretation. Another reason for the compactness is that the system manipulates the individual as a linear array of integers which is much more efficient than the more complex symbolic tree structures used in other GP system. The final reason for low memory consumption is the large amount of work done by CPU manufacturers to ensure that their machine instruction codes are compact and efficient.

The properties of CGPS make it ideally suited for real-time control in low-end processor architectures such as one-chip embedded control. Furthermore, functions of great complexity can be evolved with just simple arithmetic or logic operations in a register machine (Nordin & Banzhaf, 1995b).

2.2 The On-line GP approach

Our method for using GP with a real-time application is based on a probabilistic sampling of the environment. Different solution candidates (programs) are evaluated in different situations. This is unfair because a good individual dealing with a hard situation can be rejected in favor of a bad individual dealing with a very easy situation. Our experience is, however, that a good overall individual tends to survive and reproduce in the long term. The somewhat paradoxical fact is that sparse training data sets or probabilistic sampling in evolutionary algorithms often both increase speed toward the goal and keep the diversity high enough to escape local optima during search.

The remarkable claim that evolutionary algorithms might do better with noisy fitness functions is effectively illustrated in (Fitzpatrick, Grefenstette & van Gucht, 1984). A genetic algorithm is used there to match two digital pictures each consisting of 10,000 pixels. A trade-off exists here between the number of pixels to be compared in one matching operation (one would prefer less in order to gain speed) and the noise which is inevitably induced by selecting a subset of the images for the matching operation (one would prefer more pixels in order to keep the match accurate). The most efficient sample size, i.e. number of pixels in the fitness evaluation turned out to be 10 (out of 10,000) pixels. In other words, the fastest run to reach a good solution only looked at 1/1000 of the available data in each individual evaluation.

Our own experiments with GP also point to the fact that very sparse data sets are useful for learning in evolutionary algorithms (Banzhaf, Francone &

Nordin, 1996; Francone, Nordin & Banzhaf, 1996).

2.2.1 The evolutionary system

The population of programs is initialized with random content at the beginning of training. Tournament selection is used to determine who will survive and have offspring. We use a steady-state GP population (Syswerda, 1991; Reynolds, 1994) rather than discrete generations. Figure 5 gives a schematic view of the control architecture and the GP-system. Input to the GP system are sensor values communicated from the robot, output of the GP system are motor values controlling its behavior.

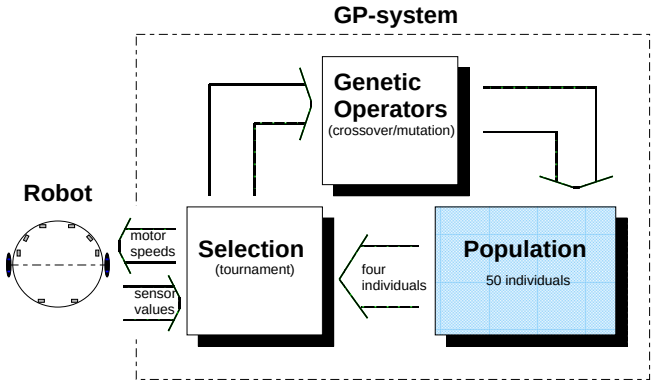

Figure 5: Schematic view of the control system. Input to the GP system are sensor values communicated from the robot, output of the GP system are motor values controlling its behavior.

The GP execution cycle is:

1. Select four members from the population at random.
2. For each member of the tournament in succession do:
   (a) Read from input and provide the data to the individual program.
   (b) Execute the individual and store the results.
   (c) Send output to the system being controlled.
   (d) Sleep for some time (here: 400ms) in order to await the results of the action.
   (e) Read from input again and compute fitness, see Section 2.4.

3. Replace the two worst performing individuals with copies of the two best ones.

4. Do mutation and crossover on the offspring (copies).

5. Goto step 1.

Each individual is thus tested against a different real-time situation leading to a unique fitness case. This results in "unfair" comparison where individuals have to navigate in situations with very different possible outcomes. However, our experiments show that over time averaging tendencies of this learning method will even out the random effects of probabilistic sampling and a set of good solutions will survive.

2.3 The Khepera Robot

Our experiments were performed with a standard autonomous miniature robot, the Swiss mobile robot platform Khepera (Mondada, Franzi & Ienne, 1993). It is equipped with eight infrared proximity sensors (Figure 6). The mobile robot has a circular shape, a diameter of 6 cm and a height of 5 cm. It possesses two motors and an on-board power supply. The motors can be independently controlled by a PID controller. The eight infrared sensors are distributed around the robot in a circular pattern (Figure 7). They emit infrared light, receive the reflections and in this way they are able to measure distances in a short range of 2-5 cm. The robot is also equipped with a Motorola 68331 micro-controller which can be connected to a workstation via serial cable.

It is possible to control the robot in two ways. The controlling algorithm could be run on the workstation with data and commands communicated through a serial cable. Alternatively, the controlling algorithm could be down-loaded to the robot which then runs the complete system in a stand-alone fashion. We use both versions of the system.

The micro-controller has 256 KB of RAM and a ROM containing a small operating system. The operating system has simple multi-tasking capabilities and manages the serial communication with the host computer.

2.4 Objectives

Our two sample tasks for illustration of the control approach are obstacle avoidance and object following. In both of our tasks the goal of the learning robot is to find a control function (program) which reacts to the sensor data and produces a vector of two motor speeds. The only change in the architecture between the two tasks is the fitness function. With obstacle avoidance the fitness function gives the agent maximal "pleasure" by staying away from obstacles. In the object following task the best fitness is achieved when the agent is at a specified distance from an object.

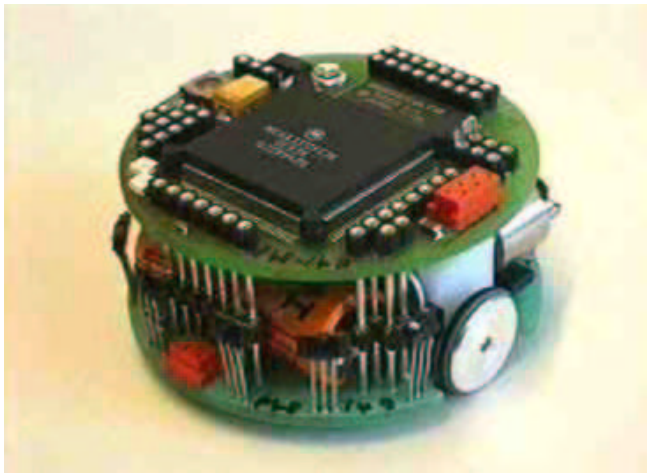

Figure 6: The Khepera Robot.

2.4.1 Training Environment

We wanted to make sure that our results and the evolved controllers were not brittle. This had previously been a problem with robot control simulation and genetic programming, as mentioned in the introduction. We therefore used two different environments in each experiment. The robot was alternatively trained in one environment and then placed in the other environment to study if it had been able to generalize. The design of the environment was deliberately done so that the two environments expressed different properties regarding friction against floor and walls, reflection of objects, size and shape. The second environment was in addition equipped with movable walls to increase the number of possible variations.

Figure 8 shows the first environment and its complex features. Its dimensions are about 70 cm $\times$ 90 cm. It has an irregular boarder with different angles and four dead-ends in each corner that were supposed to deceive the robot during navigation. In the larger open area in the middle loose objects can be placed. The friction between wheels and surface is considerably lower than in the second environment, enabling the robot to slip with its wheels during a collision with an obstacle. There is an increase in friction with the walls, making it hard for the circular robot to turn while in contact with a wall.

The second environment is larger, about 100 cm $\times$ 100 cm, and is shown in Figure 9. It has a higher friction between surface and wheels but a lower friction between walls and the circular robot. The walls of this environment are highly reflective to the infrared light emitted by the sensors and they are movable to

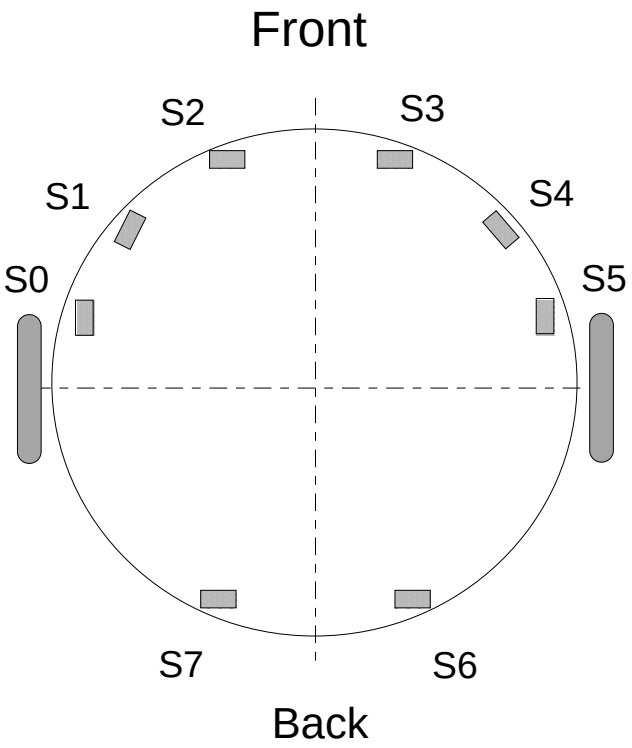

Figure 7: Position of the 8 infrared Siemens SFH 980 proximity sensors $s_0, ..., s_7$ on the Khepera robot.

allow for different paths and configurations.

2.4.2 Obstacle Avoidance

The goal of the controlling CGPS system in the first experiment is to evolve obstacle avoiding behavior in a reactive context. The system operates in real-time and tries to learn obstacle avoiding behavior using real noisy sensor data from the 8 proximity sensors. See (Braitenberg, 1984; Reynolds, 1988; Mataric, 1993; Zapata, Lepinay, Novales & Deplanques, 1993) for a discussion of this problem domain.

In the obstacle avoiding application we search for an appropriate control function that takes the sensor values as input and returns a vector of two motor speeds:

$$f(s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7) = \{m_1, m_2\} \tag{3}$$

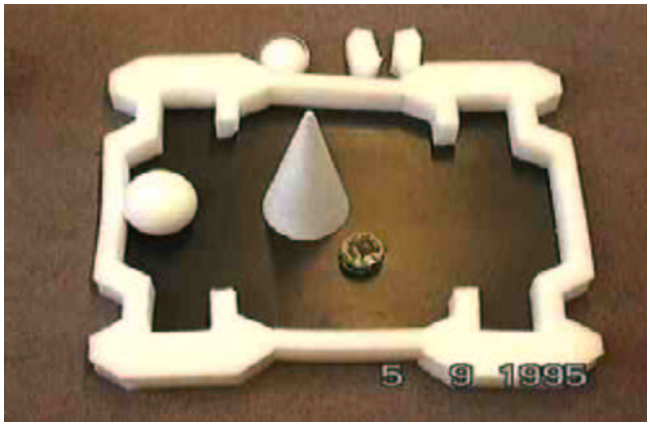

Figure 8: The first environment (training).

In our experiments, the evolved programs are true functions because no side-effects like storage in variables are allowed. It might be possible to evolve programs that can store information in memory for later use, but for the sake of simplicity we have chosen to limit the experiments to side-effect-free programs here. A program that could use memory will potentially be much more powerful. It could use memory to build a world model and to adapt faster to changing environments. This approach will be the subject of further studies in the future.

The controlling CGPS uses a small population size, typically less than 50 individuals. The individuals use eight values from the sensors as input and produce two output values which are transmitted to the robot as motor speeds. Each individual program controls the robot independently of the others.

Fitness calculation

The fitness in the obstacle avoiding task has a pain and a pleasure part. The negative contribution to fitness, called pain, is simply the sum of all proximity sensor values. The closer the robot's sensors are to an object, the more it will experience pain. In order to keep the robot from standing still or rotating, there is a positive contribution to fitness called pleasure. The robot receives pleasure from going straight and fast.

Let s_i be the values of the proximity sensors ranging from $0 - 1023$ where a higher value means closer to an object. Let m_1 and m_2 be effector values, namely the left and right motor speeds resulting from an execution of an individual. The values of m_1 and m_2 are in the range between $0 - 15$. Fitness can then be

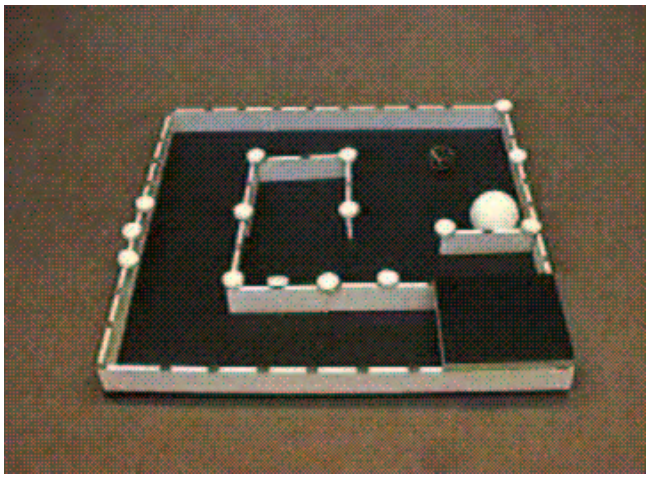

Figure 9: The second environment (testing).

expressed more formally as:

$$fitness = \alpha(m_1 + m_2 - \mid m_1 - m_2 \mid) - \beta \sum_{i=0}^{7} s_i \tag{4}$$

Thus, motor speed values minus the absolute value of their difference and sensor values enter the fitness function. The weights $\alpha = 16, \beta = 1$ have been used in these experiments.

2.4.3 Object Tracking

In this second experiment the robot's task is to follow moving objects without colliding with them. This more complex task includes a component of obstacle avoidance when the robot comes too close to an object. The task does not contain any reward for moving so the robot moves only if the object is moving and stops when near enough to the object. The training of this behavior was performed in the same two environments as the obstacle avoidance task but now including moving objects. The objects were moved by hand once the robot had come far enough in its learning process to be able to detect an object and to approach it.

Fitness calculation

Calculating fitness for the object tracking task is based solely on the four sensors facing forward. In order to define the desired behavior we need to give a function

which attracts the robot to objects far away but repels the robot if it is too close. The fitness function should thus have an U-shape where the lowest pain is at a predefined distance from an object. The fitness function we used for this task is:

$$fitness = (s_1 + s_2 + s_3 + s_4 - 1000)^2 \quad (5)$$

The value of 1000 represents the ideal distance from an object. When the sum of the front facing sensors is 1000 then the agent perceives its "ideal feeling" and the fitness is zero.

3 Results

3.1 Obstacle Avoidance

The robot shows exploratory behavior from the first moment. This is a result of the diversity in behavior residing in the first generation of programs which has been generated randomly. Naturally, the behavior is erratic at the outset of a run.

During the first minutes, the robot keeps colliding with different objects, but as time goes on the collisions become less and less frequent. The first intelligent behavior usually emerging is some kind of backing up after a collision. Then the robot gradually learns to steer away in an increasingly more sophisticated manner.

On average the robot learns in 90% of the experiments to reduce collisions to less than 2 collisions per minute. The reason for the collisions not always dropping to zero is that the infrared sensors are not very well doing when encountering sharp edges or corners.

The learning time is about one hour. It takes 40-60 minutes, or 200-300 generation equivalents, to evolve a good obstacle avoiding behavior. This time compares very well to other evolutionary approaches on the same platform. The evolution of obstacle avoiding behavior using a genetic algorithm to train a neural network is reported to take 100 times as long (Floreano & Mondada, 1994). These two experiments might not be directly comparable but one has at least an indication of the strength of the combination between probabilistic sampling and a machine code manipulating GP system in robotic control.

Figure 10 shows how the number of collisions per minute diminishes as the robot learns and the population becomes dominated by good control strategies.

The moving robot gives the impression of performing a very complex behavior. Its behavior resembles a bug or an ant exploring its environment with irregular small moves around the objects. Data such as Figure 15 (discussed in section 3.5) also suggest that the evolved behavior is complex. Though it is hard to classify the different kinds of behavior the robot demonstrates during evolution we found a few distinguishable classes:

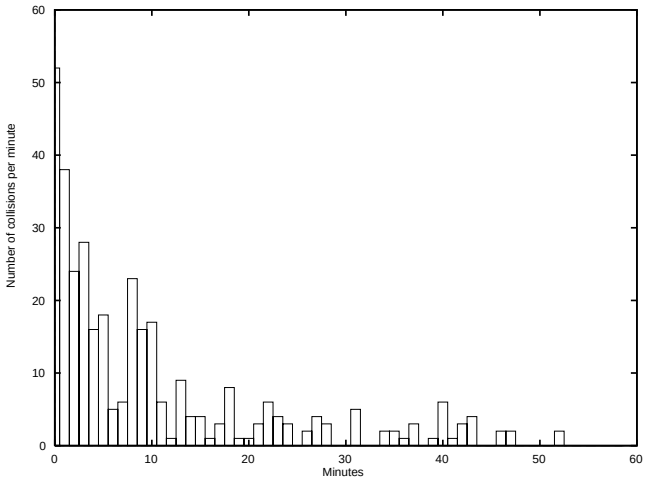

Figure 10: The number of collisions per minute in a typical training run with the environment of Figure8.

1. The initial wall-bumping behavior. This is the childhood of the robot where it must make mistakes in order to learn for the rest of its life. It tries several impossible strategies such as trying to force itself through the walls.

2. The backing–up behavior. This is the first effective technique that allows the robot to avoid obstacles. However, this technique has several drawbacks. It risks placing the robot in the same situation if it moves forward again.

3. The reflection behavior. Here the robot gradually turns away from an obstacle as it approaches it. It looks as if the robot bounces like a ball at something invisible, close to the obstacle. This behavior gives a minimal speed change in the robot's path.

4. The sniffing behavior. The robot stops immediately as it just senses an object. It then starts to sniff around by very small movements as if it was determining which way will be the most favorable to go.

All these behaviors can be distinguished but they also appear mixed and as part of very complex movements.

The task of obstacle avoidance could at first glance be regarded as a simple task to learn. After all, it is relatively easy to write programs by hand that solve the problem efficiently. One should, however, keep in mind that the GP

system learns the task with no a priori knowledge. The major difficulty is to sort out how to interpret its eight input sensors. The system must in a similar way learn the dynamics of its actuators and the physical dynamics of its body and environment.

3.2 Object Tracking

The behavior of the robot is now profoundly different due only to a slight change in the fitness function. Surprisingly, it learns obstacle avoidance faster than the obstacle avoiding system. The reason is presumably that the pleasure part of the fitness function is missing. There is no reward for going straight and fast in this case, which might simplify the learning task, because there are no conflicting signals in difficult situations.

The object following behavior is learned in about 30 minutes. The robot appears to "sniff" around an object as soon as it finds one. It moves slightly in all directions to try to find the optimal distance to the object. Noise from the sensors and from stochastic sampling result in constant small movements. When an object is moved the robot acts as if it were connected to the object with an elastic spring. It follows the object with a certain delay and when the object stops it oscillates slightly around the ideal distance. A similar behavior is achieved when the object is pushed against the robot - it backs off as if there were a spring connected to the object.

It is interesting to note how differently the robot behaves by only changing a few terms in its fitness function and how the robot adapts to its rewards relatively quickly without almost any procedural knowledge. The robot just has a "blind" feed-back from its fitness function, a "reinforcement signal" telling it if something is good or bad, not what and why.

3.3 Robustness

Given the brittleness of other GP control experiments we found it important to evaluate each training run in an unseen environment. No degradation in behavior is visible, when the robot is lifted from the training environment and placed in the testing environment. In a similar way the robot showed the same performance if it was lifted and placed in a different starting point in the environment.

3.4 A Sample Evolved Program

As seen earlier in Section 2.1.1, each node in the genome represents an operation between a small set of registers. The structure of a node corresponds to a machine code instruction in the processor and is stored in such a way that it can be executed directly. Below we see how a program individual for our robot control task may look if printed as a 'C' program for better readability. Each

individual is composed of simple instructions (program lines) between input and output parameters and temporary variables. The actual robot control function takes eight values as input (the sensors $s_0 - s_7$) and produces two values as output (the motor speeds m_1, m_2), see also Figure 7 for a location of the sensors.

The program below is a program actually evolved by the system where $a - e$ are registers for temporary storage of input values and $motor1, motor2$ are registers for the resulting speed values that will be sent to the motors. The operators in the program below are the arithmetic operators plus "<<", ">>", which denote arithmetic shift operations, and "&" and "|" which are logical "and" and "or"[3]. Table 2 gives a summary of the parameters used in the experiments.

```
a=s3 + 4;
d=s2 >> s1;
b=s1 - d;
d=s2 + 2;
c=d >> 1;
b=d - d;
a=c - 3;
c=s4 << a;
d=a * 4;
a=a ^ 5;
e=d + s4;
c=b & c;
d=d + d;
c=d | 8;
d=d * 10;
b=e >> 6;
d=b & 0;
e=c >> b;
motor2=a | e;
c=d | 7;
motor1=c * 9;
c=e & e;
```

3.5 Visualizations of results for obstacle avoidance

In order to illustrate the control strategies of the different evolved programs and to analyze how the robot achieves obstacle avoidance we have used a graphic technique to plot the relationship between sensor input and motor output. The

[3]The code has been edited for clarity. Sensor and motor values are in reality stored in the same register used during calculation. Register "a" contains the value of s_0 when evaluation starts and registers "b" contains the value of s_1 . Correspondingly the motor speed values sent to the motors are what is left in register "a" and "b" at the end of evaluation.

Objective :	Obstacle avoidance and object following
Terminal set :	Integers in the range 0-8191
Function set :	ADD, SUB, MUL, SHL, SHR, XOR, OR, AND
Fitness :	Pleasure subtracted from pain value
Maximum population size :	50
Crossover Prob :	90%
Mutation Prob :	5%
Selection :	Tournament Selection
Termination criteria :	None
Maximum number of instructions:	256

Table 2: Summary of parameters used in the experiments. Note that each instruction corresponds to 4 nodes in a tree-based GP system.

control function takes eight values as input (the sensors) and produces two values as output (the motor speeds). The control function is therefore too high dimensional for a direct visualization in a diagram. Instead we simulate a uniform stimulus on the left side and the right side of the robot, as shown in Figure 11. The output of an individual program in the population is examined then and the steering direction of the robot is plotted in terms of the difference between left and right motor speeds.

A simple control strategy which would avoid obstacles to some extent would be a plane sloping from the upper left corner of the diagram down to the lower right corner, as in Figure 12. If the robot would be controlled by such an algorithm it would steer more to the right the more the sensors on the left side are stimulated. Many plots from actually evolved control programs show similarities to this plane, e.g. Figure 13 and Figure 14. The diagrams are more complex but the slope from upper left to lower right is evident. A good control strategy should, however, accomplish more than simply steering away. It should for instance have the ability to back away from obstacles and perform even more complex maneuvers. In many cases we do not have an explanation for the complex look of these diagrams. We do not know whether they are the product of mere chance or whether the complex features help the individual in the competition to control the robot.

Below we see the individual control program which generated Figure 15 when examined systematically, i.e. when presented with all combinations of evenly distributed left and right side stimuli.

```
a=s3 + 4;
d=s2 >> s1;
b=s1 - d;
d=s2 + 2;
```

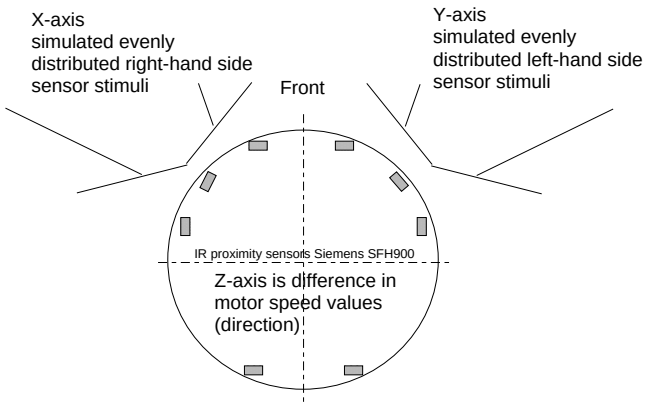

Figure 11: Method for visualizations of results. Input on both halves of the sensors is equally strong. These values are plotted on the $X-$ and $Y-$ axis of Figures12 to 15

```
c=d >> 1;
b=d - d;
a=c - 3;
c=s4 << a;
d=a * 4;
a=a ^ 5;
e=d + s4;
c=b & c;
d=d + d;
c=d | 8;
d=d * 10;
b=e >> 6;
d=b & 0;
e=c >> b;
motor2=a | e;
c=d | 7;
motor1=c * 9;
c=e & e;
```

This complex control strategy does not use all of the sensor values, for instance s_0 is missing. The sensors from the back of the robot are not used either in this example. This could be the result of too little exposure to backing into obstacles during training. It is typical for solutions of genetic programming[4] to be parsimonious in its output and only use the concepts which it "thinks" are

[4] After neutral code has been removed.

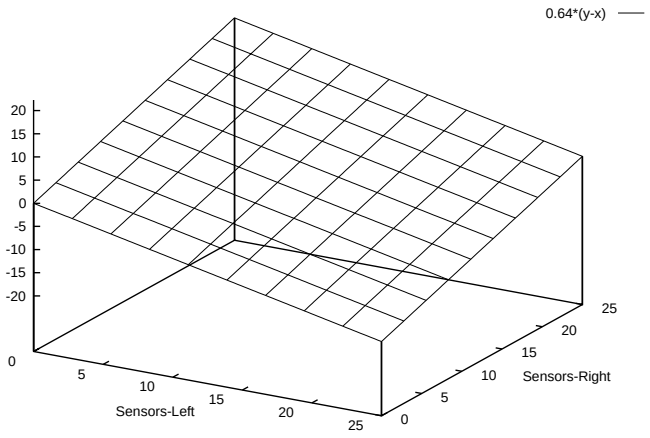

Figure 12: An example of a simple controlling function which would display simple obstacle avoiding behavior.

really needed to achieve the defined goal. Most evolved programs are about the size of the program above, 10-20 instructions equaling 40-80 bytes of memory.

3.6 GP on board of the Khepera in autonomous mode

There is no consensus what an autonomous robot really is. Some would argue that autonomy is a property of the controlling algorithm while others would argue that physical autonomy is needed. Since we share the latter opinion, we have ported a special version of the system to the micro-controller on board of the Khepera.

It is possible to download this system via the serial cable to the robot. With the batteries switched on, the robot can then be disconnected from the workstation and can run completely autonomous. The Motorola 68331 micro-controller then runs the full GP system.

As mentioned earlier, this micro controller has 256 KB of RAM memory. The kernel of the GP system occupies 32 KB and each individual requires 1 KB in this setup. The complete system with 50 individual then occupies 82 KB which is well within the limits of the on-board system.

The results with the autonomous system are almost identical to those with the system connected to the workstation. The only small difference is that the cable no longer disturbs the system. When the cable was connected to the robot, it sometimes physically restricted the robot's movements. The cable sometimes got in the way of the sensors and caused a "tail chasing" behavior. The problems with the cable are an example of the many small disturbances which are almost

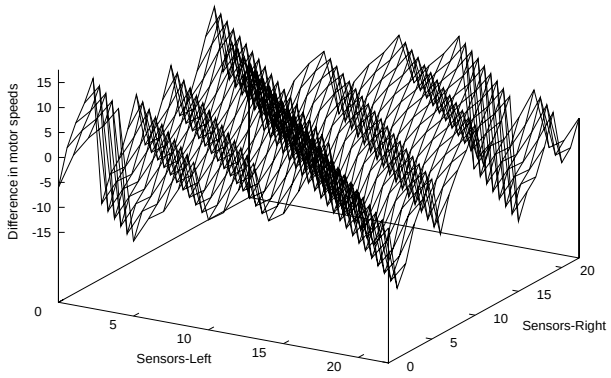

Figure 13: Example of controlling function with similarities with the plane in Figure 12.

impossible to simulate but must be experienced with a real robot.

The batteries are the main limitations of the autonomous system. They can power the miniature robot and the system for about 40-60 minutes which is close to the minimum time required for training. The learning GP system in itself is identical to the system on the workstation with identical cycle time and performance.

The micro-controller version demonstrates that the compactness of the compiling GP system enables relatively powerful solutions in weak architectures such as those used in embedded control.

4 Conclusions and Future Work

The combination of probabilistic sampling with a GP technique has two important advantages. It enables the efficient use of a GP technique on-line and at the same time accelerates the learning.

The system works well when implemented on a real robot receiving noisy input with real-time constraints. The evolved algorithm shows robust performance even if the robot is lifted and placed in a significantly different environment or if objects are moved around.

We believe that the robust behavior of the robot partly could be attributed to the built-in generalization capabilities of the genetic programming system (Nordin & Banzhaf, 1995a). Partly it could be attributed to the probabilistic sampling: No individual can simply memorize a path to move on because it does not know how and in what situation the next action will have to be evaluated.

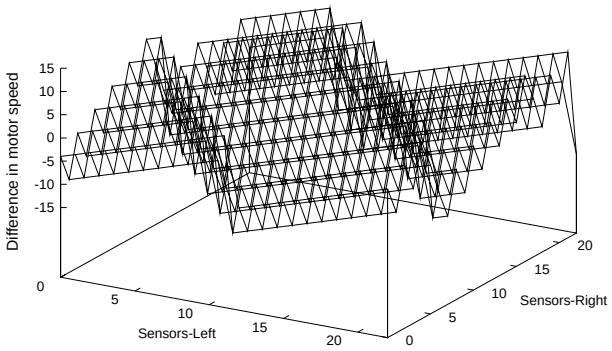

Figure 14: Example of controlling function.

Evolutionary search methods have been proven to do well and robustly in multi-modal and difficult search domains. This could be one reason for the robust performance in the complex search domain of symbolic control programs. The concept of keeping a population of solution candidates with a varying fitness could help to avoid local minima in search. Simpler approaches, for instance a hill climbing method, might get caught easier in a local minimum which would prevent good final results.

We have also cross-compiled the GP-system and run it in the same set-up on the micro-controller on-board the robot. This demonstrates the applicability of Compiling Genetic Programming to control tasks on low-end architectures. The technique could potentially be applied to many one-chip control applications in, for instance, consumer electronics etc. Thanks to the use of machine language the system does not have to incorporate the knowledge of the language in software – it is provided in hardware.

The fixed machine language of the system could be said to be an advantage because it omits most of the designer's prejudices of why an application should work. On the other hand it could be argued that the machine language approach is less flexible then other GP approaches because the language is fixed[5]. However, GP systems have been shown not to bite the bait we give it in the form of complex specialized and predefined functions. Instead, the system often chooses to evolve its own approximations with lower level primitives (Koza, 1992; Nordin & Banzhaf, 1996b).

An important extension to the system would be to eliminate the 400 ms

[5]Even though it is possible to extend CGPS with any function or procedure as part of the function set

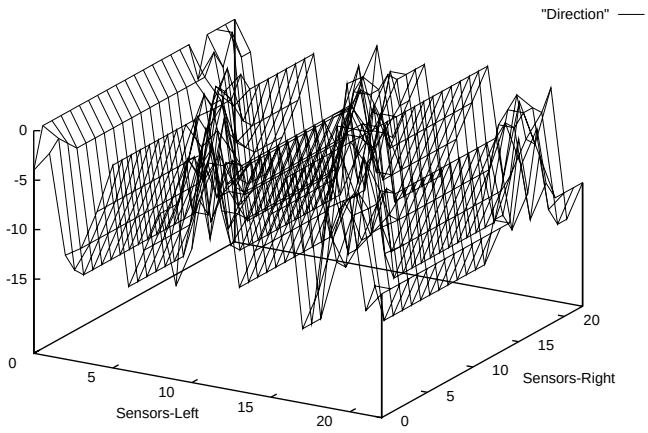

Figure 15: Sample behavior of an evolved control program.

reaction delay time, during which the system is waiting for the result of its action. The need to wait for the response of the environment is the major drawback of our system. If we would like to learn a more complex task we would most probably have to raise the population size and wait much longer for convergence of the algorithm. The speed of the algorithm is defined by the response needed from the environment and one would have to wait longer for convergence almost regardless of machine resources.

This problem could be addressed by allowing the system to memorize previous stimulus-response pairs and by enabling it to self-inspect memory later on in order to learn directly from past experiences without a need to wait for results of its actions. Early results show that this history based method can speed up the GP algorithm by a factor of at least 1000. The learning of obstacle avoiding behavior is accelerated by a factor of 40 enabling learning within a few minutes (Nordin & Banzhaf, 1995c). The difference comes about by the fixed requirement of waiting for feedback from the environment.

We are also investigating a different approach for more complex problems. A useful strategy to create robust behavior is to divide complex actions into *action primitives* which only perform specialized tasks like avoiding obstacles or moving ahead, and then combine them again for the creation of higher *levels of competence* (Brooks, 1992). In our approach the GP system first learns the sub-tasks and then evolves a higher–level action selection strategy for deciding which of the evolved lower–level algorithms should be in control. The lower level sub-tasks are:

- Obstacle avoidance.

- Wall following.
- Homing behavior. Find a dark "nest" (visible in the lower right in Figure 9).
- Traveling open spaces. The agent learns to efficiently travel an open space without obstacles.

Preliminary results show that the robot is indeed able to evolve both the control algorithms for the different lower-level task and the strategy for the selection of tasks (Banzhaf, Nordin & Olmer, 1996; Olmer, Nordin & Banzhaf, 1996). This could potentially be a feasible approach when a very complex behavior needs to be adaptive.

Acknowledgment

We are grateful to Dr. Bill Langdon for correcting an earlier version of this manuscript and to the unknown referees for helping improve the paper substantially. We would like to thank Dr. Lashon Booker for his efforts in refining the final version of the manuscript. Partial support from the Deutsche Forschungsgemeinschaft (DFG) under grant Ba 1042/5-1 is gratefully acknowledged.

References

Atkin, M., & Cohen, P. R. (1994). Learning Monitoring Strategies: A Difficult Genetic Programming Application. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence.* IEEE Press, Piscataway, NY.

Baluja, S. (1996). Evolution of an Artificial Neural Network based Autonomous Land Vehicle Controller. *IEEE Transactions Systems, Man and Cybernetics - Part B, Special Issue on Learning Autonomous Robots, 26,* 450 — 463.

Banzhaf, W., Francone, F., & Nordin, P. (1996). The Effect of Extensive Use of the Mutation Operator on Generalization in Genetic Programming using Sparse Data Sets. In W. Ebeling, I. Rechenberg, H.-P. Schwefel and M. Voigt (Eds.), *Proceedings of the PPSN-IV.* Springer, Berlin.

Banzhaf, W., Nordin, P., & Olmer, M. (1996). Generating Adaptive Behavior using Function Regression within Genetic Programming and a Real Robot. Unpublished manuscript.

Braitenberg, V. (1984). *Vehicles.* MIT Press, Cambridge, MA.

Brooks, R. (1992). Artificial Life and Real Robots. In F.J. Varela and P. Bourgine (Eds.), *Proceedings of the First European Conference on Artificial Life.* MIT Press, Cambridge, MA.

Cliff, D. (1991). Computational Neuroethology: A Provisional Manifesto. In J. A. Meyer and S. Wilson (Eds.), *From Animals To Animats: Proceedings of the First International Conference on simulation of Adaptive Behavior.* MIT Press, Cambridge, MA.

Cliff, D., Harvey, I., & Husbands, P. (1993). Explorations in Evolutionary Robotics. *Adaptive Behavior, 2,* 73 — 110.

Colombetti, M., Dorigo, M., & Borghi, G. (1996). Behavior Analysis and Training: A methodology for Behavior Engineering. *IEEE Transactions Systems, Man and Cybernetics - Part B, Special Issue on Learning Autonomous Robots, 26,* 365 — 380.

Donnart, J., & Meyer, J. (1996). Evolution of Homing Navigation in a Real Mobile Robot. *IEEE Transactions Systems, Man and Cybernetics - Part B, Special Issue on Learning Autonomous Robots, 26,* 381 — 395.

Edelman, G. (1987). *Neural Darwinism.* Basic Books, New York.

Fitzpatrick, J. M., Grefenstette, J., & Van Gucht, D. (1984). Image Registration by Genetic Search. In *Proceedings of IEEE Southeast Conference.* IEEE Press, Piscataway, NY.

Floreano, D., & Mondada, F. (1994). Automatic Creation of an Autonomous Agent: Genetic Evolution of a Neural-Network Driven Robot. In D. Cliff, P. Husbands, J. A. Meyer & S. Wilson (Eds.), *From Animals to Animats III: Proceedings of the Third International Conference on Simulation of Adaptive Behavior.* MIT Press, Cambridge, MA.

Floreano, D., & Mondada, F. (1996). Evolution of Homing Navigation in a Real Mobile Robot. *IEEE Transactions Systems, Man and Cybernetics - Part B, Special Issue on Learning Autonomous Robots, 26,* 396 — 407.

Francone, F., Nordin, P., & Banzhaf, W. (1996). The Effect of Extensive Use of the Mutation Operator on Generalization in Genetic Programming using Sparse Data Sets. In J. Koza, D. Goldberg, D. Fogel & R. Riolo (Eds.), *Proceedings of the first annual Conf. on Genetic Programming.* MIT Press, Cambridge, MA.

Fraser, A.P., & Rush, J.R. (1994). Putting INK into a BIRo: A Discussion of Problem Domain Knowledge for Evolutionary Robotics. In *Proceedings of the Workshop on Artificial Intelligence and Simulation of Behavior Workshop on Evolutionary Computing.*

Grefenstette, J., Ramsey, C., & Schultz, A. (1990). Learning Sequential Decision Rules using Simulation Models and Competition. *Machine Learning, 5,* 355 — 381.

Greiner, R., & Isukapalli, R. (1996). Learning to Select Useful Landmarks. In *IEEE Transactions Systems, Man and Cybernetics - Part B, Special Issue on Learning Autonomous Robots, 26,* 437 — 449.

Handley, S. (1994). The Automatic Generation of Plans for a Mobile Robot via Genetic Programming with Automatically Defined Functions. In K. Kinnear (Ed.), *Advances in Genetic Programming.* MIT Press, Cambridge, MA, 391 — 407.

Harvey, I., Husbands, P., & Cliff, D. (1993). Issues in Evolutionary Robotics. In J. A. Meyer & S. Wilson (Eds.), *From Animals To Animats 2: Proceedings of the Second International Conference on Simulation of Adaptive Behavior.* MIT Press, Cambridge, MA.

Kaelbling, L. P., Littman, M. L., & Moore, A. W. (1996). Reinforcement Learning: A Survey. *J. of Artificial Intelligence Research, 4,* 237 — 285.

Keith, M.J., & Martin, C.M. (1993). Genetic Programming in C++: Implementation and Design Issues. In K. Kinnear (Ed.), *Advances in Genetic Programming.* MIT Press, Cambridge, MA, 285 — 310.

Koza, J. (1992). *Genetic Programming.* MIT Press, Cambridge, MA.

Kröse, B. (1995). Learning from Delayed Response. *Robotics and Autonomous Systems, 15,* 233 — 235.

Lowerre, B. T., & Reddy, R. D. (1980). The Harpy Speech Understanding System. In W. A. Lea (Ed.), *Trends in Speech Recognition.* Prentice-Hall, NY.

Mataric, M.J. (1993). Designing Emergent Behaviors: From Local Interactions to Collective Intelligence. In J. A. Meyer & S. Wilson (Eds.), *From Animals To Animats 2: Proceedings of the Second International Conference on Simulation of Adaptive Behavior.* MIT Press, Cambridge, MA.

Meeden, L. (1996). An Incremental Approach to Developing Intelligent Neural Network Controllers for Robots. *IEEE Transactions Systems, Man and Cybernetics - Part B, Special Issue on Learning Autonomous Robots, 26,* 474 — 484.

Millán, R.J. del (1996). Rapid, Safe, and Incremental Learning of Navigation Strategies. *IEEE Transactions Systems, Man and Cybernetics - Part B, Special Issue on Learning Autonomous Robots, 26,* 408 — 420.

Mondada, F., Franzi, E., & Ienne, P. (1993). Mobile Robot Miniaturization: A Tool for Investigation in Control Algorithms. In *Proceedings of the 3rd International Symposium on Experimental Robotics (ISER '93), Kyoto, Japan, October 1993.*

Nordin, P. (1994). A Compiling Genetic Programming System that Directly Manipulates the Machine-Code. In K. Kinnear (Ed.), *Advances in Genetic Programming.* MIT Press, Cambridge, MA, 311 — 331.

Nordin, P., & Banzhaf, W. (1995a). Complexity Compression and Evolution. In L. Eshelman (Ed.), *Proceedings of Sixth International Conference of Genetic Algorithms.* Morgan Kaufmann, San Mateo, CA.

Nordin, P., & Banzhaf, W. (1995b). Evolving Turing Complete Programs for a Register Machine with Self-Modifying Code. In L. Eshelman (Ed.), *Proceedings of Sixth International Conference of Genetic Algorithms.* Morgan Kaufmann, San Mateo, CA.

Nordin, P., & Banzhaf, W. (1995c). Real Time Evolution of Behavior and a World Model for a Miniature Robot using Genetic Programming. *Technical Report SysReport SYS-5/95.* Dept. of Computer Science, University of Dortmund, December 1995.

Nordin, P., & Banzhaf, W. (1996a). Image Recognition and Image Encoding Using Paint Primitives and Genetic Programming. Unpublished manuscript.

Nordin, P., & Banzhaf, W. (1996b). Programmatic Compression of Images and Sound. In J. Koza, D. Goldberg, D. Fogel & R. Riolo (Eds.), *Proceedings of the first annual Conf. on Genetic Programming.* MIT Press, Cambridge, MA.

Nordin, P., Francone, F., & Banzhaf, W. (1996). Explicitly Defined Introns in Genetic Programming. In P. Angeline & K. Kinnear (Eds.), *Advances in Genetic Programming II.* MIT Press, Cambridge, MA, 111 — 134.

Olmer, M., Nordin, P., & Banzhaf, W. (1996). Evolving Real-Time Behavior Modules for a Real Robot with Genetic Programming. In J. Grefenstette & A. Shultz (Eds.), *Proceedings of the International Symposium on Robotics and Manufacturing, Montpellier, France, 1996.* In press.

Potter, M. A., De Jong, K. A., & Grefenstette, J. (1995). In L. Eshelman (Ed.), *Proceedings of Sixth International Conference of Genetic Algorithms.* Morgan Kaufmann, San Mateo, CA.

Reynolds, C. W. (1988). Not Bumping into Things. In: *Notes for the SIGGRAPH'88 Course Developments in Physically-Based Modeling.* ACM-SIGGRAPH.

Reynolds, C.W. (1994). Evolution of Obstacle Avoidance Behavior. In K. Kinnear (Ed.), *Advances in Genetic Programming.* MIT Press, Cambridge, MA, 221 — 241.

Rosenbloom, P. (1987). Best First Search. In S. Shapiro (Ed.), *Encyclopedia of Artificial Intelligence, Vol. 2.* Wiley, New York.

Sims, K. (1994). Evolving 3D Morphology and Behavior by Competition. In: A. Brooks & P. Maes (Eds.), *Proc. Artificial Life IV.* MIT Press, Cambridge, MA.

Sparc (1991). *The SPARC Architecture Manual.* SPARC International Inc, Menlo Park, CA.

Spencer, G. (1994). Automatic Generation of Programs for Crawling and Walking. In K. Kinnear (Ed.), *Advances in Genetic Programming.* MIT Press, Cambridge, MA, 335 — 353.

Syswerda, G. (1991). A Study of Reproduction in Generational Steady-State Genetic Algorithms. In G. J. E. Rawlings (Ed.), *Foundations of Genetic Algorithms.* Morgan Kaufmann, San Mateo, CA.

Teller, A. (1994). The Evolution of Mental Models. In K. Kinnear (Ed.), *Advances in Genetic Programming.* MIT Press, Cambridge, MA, 199 — 219.

Zapata, R., Lepinay, P., Novales, C., & Deplanques, P. (1993). Reactive Behaviors of Fast Mobile Robots in Unstructured Environments: Sensor-based Control and Neural Networks. In J. A. Meyer & S. Wilson (Eds.), *From Animals To Animats 2: Proceedings of the Second International Conference on Simulation of Adaptive Behavior.* MIT Press, Cambridge, MA.