Real Time Evolution of Behavior and a World Model for a Miniature Robot using Genetic Programming

Peter Nordin Wolfgang Banzhaf

Fachbereich Informatik

Universität Dortmund

44221 Dortmund, GERMANY

email: nordin,banzhaf@ls11.informatik.uni-dortmund.de

Abstract

A very general form of representing and specifying an autonomous agent's behavior is by using a computer language. The task of planning feasible actions could then simply be reduced to an instance of automatic programming. We have evaluated the use of an evolutionary technique for automatic programming called Genetic Programming (GP) to directly control a miniature robot. To our knowledge, this is the first attempt to control a real robot with a GP based learning method. Two schemes are presented. The objective of the GP-system in our first approach is to evolve real-time obstacle avoiding behavior from sensorial data. This technique enables real time learning with a real robot using genetic programming, it has, however, the drawback of the learning time being limited by the response dynamics of the environment. To overcome this problems we have devised a second method, learning from past experiences stored in memory. This new system allows speeds up of the algorithm by a factor of more than 2000. The emergence of the obstacle avoiding behavior is also speeded up by a factor of 40 enabling learning of this task in 1.5 minutes. This learning time is several orders of magnitudes faster then comparable experiments with other control architectures. The used algorithm is furthermore very compact and can be fitted into the micro-controller of the autonomous mobile miniature robot.

1 Introduction

A computer language is one of the most general forms of representing and specifying behavior. We can use a suitable computer programming language to specify any behavior of an autonomous agent. Genetic Programming (GP) is a method using an evolutionary algorithm to evolve computer programs. It is thus a probabilistic method of automated programming. Given a goal in the form of a fitness function and a set of instructions to be used, the genetic programming system will try to evolve a program that solves the tasks specified by the fitness function. In this paper we present a general method for on-line learning from past experiences with a mobile robot in real time. It is built on experiences from earlier experiments using a genetic programming system to control a real robot. The older method, briefly described below, does not provide an explicit possibility to learn from past experiences and is thus limited by the mechanical dynamics of the robot to get feed-back from its actions. The learning speed is consequently limited by this dynamics. If an agent's brain processes events faster then the time taken to perform actions in the real world it is feasible to start remembering past experiences and try to form a world model based on memory entries. This world model could then serve as the basis for decisions regarding future actions.

To use a genetic process as the architecture for mental activities could, at first, be considered awkward. As far as we know today, genetic information processing is not directly involved in information processing in brains, though the idea of genetics as a model of mental processes is not new. William James, the father of American psychology, argued just 15 years after Darwin published *The origin of Species*, in 1874, that mental processes could operate in a Darwinian manner [14]. He suggested that ideas "compete" with each other in the brain leaving only the best or fittest. Just as Darwinian evolution shaped a better brain in a couple of million years, a similar Darwinian process operating within the brain might shape intelligent solutions to problems on the time scale of thought and action. This allows "our thoughts to die instead of ourselves". More recently, selectionist approaches to learning have been studied in detail by Gerald Edelman and his collaborators (see [7] and references therein).

The use of an evolutionary method to evolve controller architectures has been reported previously in a number of variants. Robotic controllers have, for instance, been evolved using dynamic recurrent neural nets [4], [12]. Several experiments have also been performed where a controller program has been evolved directly through genetic programming [15], [23], [11].

Previous experiments with genetic programming and robotic control, however, have been performed with a simulated robot and a simulated environment. In such a set-up, the environment and the robot can be reset easily to an initial state in order to ensure that each individual in the population is judged starting from the same state. Apart from being practically infeasible for a real robot, this method could result in over-specialization and failure to evolve a behavior that can generalize to unseen environments and tasks. To overcome the latter problem noise is sometimes artificially added to the simulated environment.

We have earlier reported on the first experiments using GP to control a real robot trained in real–time with actual sensor values [21]. In such an environment, the system has to evolve robust controllers because noise is present everywhere and the number of real-life training situations is infinite. In addition, it is highly impractical to reset the robot to a predefined state before evaluating a fitness case. Consequently, we were forced to devise a method which ensures learning of behavior while the environment is probabilistically sampled with new real-time fitness cases for each individual evaluation.

Over time, fluctuations of the environment cancel out and the system approaches a robust state. The advantage of this method, which we call probabilistic sampling, is a considerable acceleration of evolutionary process. We propose to use it in other evolutionary applications, too.

Using this approach the evolution of a successful control program is driven by continuous interaction with, and feed–back from, the environment. There is, however, no memory of past experience other then the information implicitly stored in the genetic material. Hence, the main disadvantage of this method is that learning speed becomes dependent on the mechanical dynamics of the environment not on the speed of the algorithm or the processor. In our example the learning algorithm consumes less than 0.1% of the available CPU time – the rest of the time is spent waiting for feed–back from the environment.

What we describe below is an extension of this method using a memory buffer to store and use past experiences, enabling learning at a speed only determined by the processor. The execution of the GP system is consequently accelerated by a factor larger then 2000 and the agent learns its task as soon as there are enough experiences collected in memory. Convergence towards feasible behavior requires only a few minutes in the obstacle avoidance experiments described below.

The rest of this paper is organized as follows: In Sections 2 and 3 we start by briefly introducing the genetic programming paradigm and the real–time variant we use here. We present the miniature robot used in these experiments and the environment it is trained in, in Sections 4 and 5. The objective of the training in these experiments, obstacle avoiding behavior, as well as the fitness function defining it, are described in Section 6. Our earlier approach towards a control architecture is briefly summarized in Section 7. This architecture represents the first application of a real time GP system to a real autonomous robot. The new method is presented in Section 8, with a GP control architecture learning form past experiences and enabling fast convergence. We present the results from learning obstacle avoiding behavior with this architecture in Section 9 and relate it to the performance of our previous approach and other evolutionary learning techniques. Finally we summarize our results, discuss conclusions and present ideas for future work.

2 Genetic Programming

Evolutionary Algorithms mimic aspects of natural evolution, to optimize a solution towards a defined goal. Darwin's principle of natural selection and survival of the fittest is thought to be responsible for the evolution of all life forms on earth. This principle has been employed successfully on computers over the past 30 years. Different research subfields have emerged, such as Evolution Strategies [24], Genetic Algorithms [13] and Evolutionary Programming [9], indicating that they all mimic various aspects of natural evolution. In recent years, these methods have been applied successfully to a spectrum of real-world and academic problem domains, such robot control as seen above.

A comparatively young and growing research topic in this field is Genetic Programming (GP). Genetic Programming uses the mechanisms behind natural selection for *evolution of computer programs*. Instead of a human programmer programming the computer, the computer can self-modify, through genetic operators, a population of programs in order to finally generate a program that solves the defined problem. This technique, like other adaptive techniques, has applications in problem domains where theories are incomplete and insufficient for the human programmer, or when there is not enough time or resources available to allow for human programming.

Methods with similarities to GP were suggested as far back as in the 1950s [10]. For various reasons these experiments never were a complete success even if partial results were achieved [6]. It was a breakthrough when J. Koza formulated his approach based on program individuals as tree structures represented by LISP S-expressions [15]. His hierarchical subtree exchanging genetic crossover operator guaranteed syntactic closure during evolution. Koza has been instrumental in developing GP and was the first to demonstrate the feasibility of the technique with dozens of applications.

In this notion, genetic programming evolves programs stored as trees in a population by exchanging subtrees during crossover. Selection is performed according to a fitness function. Figure 1 illustrates the crossover method used with tree-based genetic programming. A simple genetic programming system thus

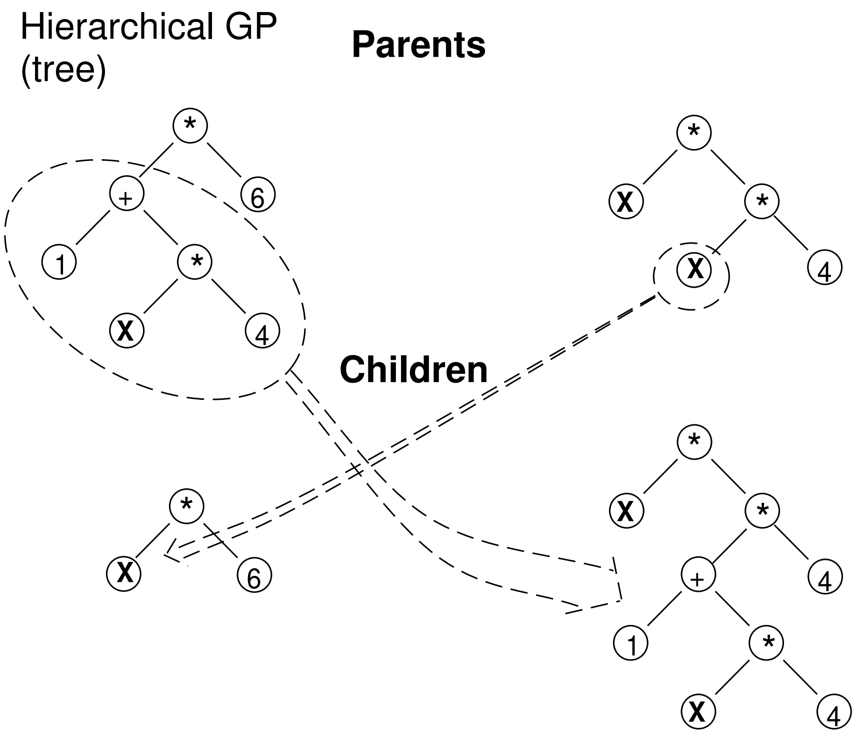

Figure 1: Hierarchical GP and effect of crossover. Subtrees are exchanged thus guaranteeing syntactic closure.

consists of:

1. A population of solution candidates (programs) where population size could vary between 30 and 50,000 individuals. The population is normally initiated to a random content.

2. A fitness measure defining the desired task of the evolved programs.

3. A recombination or crossover operator allowing for exchange of solution segments.

4. A mutation operator changing code more locally than the crossover operator.

5. A set of basic constants,instructions, procedures or function used as the atomic parts of the evolved programs.

In the experiments described below we use another genetic programming system which has properties well suited for real–time applications.

2.1 Genetic Programming and Machine Code

The GP system we use has a linear genome and stores the individuals of the population as binary machine code in memory. This results in a speed-up of several orders of magnitude. The method is also memory efficient requiring only 32KB for the GP kernel. The individuals can be stored in an economic way and memory consumption is stable during evolution without any need for garbage collection etc. All these properties make the system ideally suited for real-time control in low-end processor architectures such as one-chip embedded control. The Evolutionary Algorithm we use in this paper is an advanced

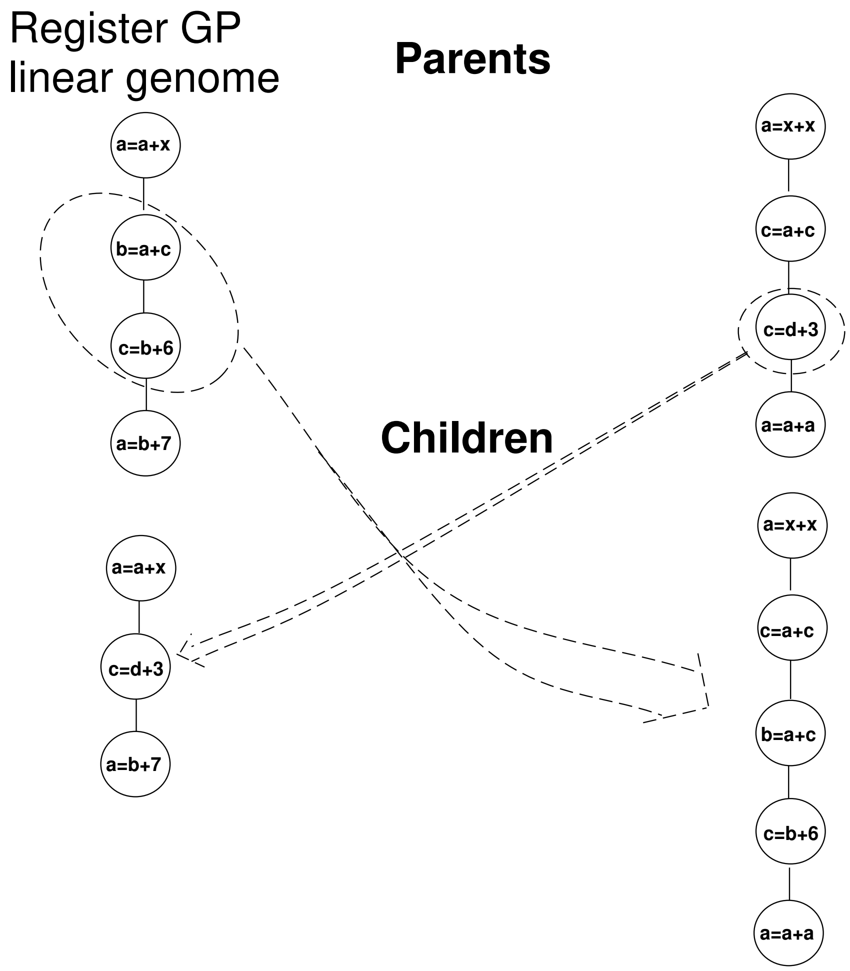

Figure 2: Linear GP and crossover. Instructions or sequences of instructions are exchanged.

version of the CGPS described in [18], composed of variable length strings of 32 bit instructions for a register machine. Each node in the genome is an instruction for a register machine. The register machine performs arithmetic operations on a small set of registers. Each instruction might also include a small integer constant of maximal 13 bits. The 32 bits in the instruction thus represent simple arithmetic operations such as "a=b+c" or "c=b*5". The actual format of the 32 bits corresponds to the machine code format of a SUN-4 [25], which enables the genetic operators to manipulate binary code directly. For a more thorough description of the system and its implementation, see [19].

The set-up is motivated by fast execution, low memory requirements and a linear genome which makes reasoning about information content less complex. This compact system is a prerequisite for a micro-controller version.

The machine code manipulating GP system uses two-point string crossover. A node is the atomic crossover unit in the GP structure. Crossover can occur on either or both sides of a node but not within a node. Because our particular implementation of GP works with 32 bit machine code instructions, a node is a 32 bit instruction. Figure 2 illustrates the crossover method used in our experiments. Mutation flips bits inside the 32-bit node. The mutation operator ensures that only instructions in the function set with valid ranges of registers and constants are the result of a mutation. All genetic operators ensure syntactic closure during evolution.

The instructions used in our experiments, are all low-level machine code instructions. The function set consists of the arithmetic operations ADD, SUB and MUL, the shift operations SLL and SLR and finally the logic operations AND, OR and XOR. All these instructions operate on 32-bit registers.

Below we see how an individual program may look if printed as a 'C' program.

```
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;
```

```
(``>>'', ``>>''=Shift   ``|''=Or   ``&''=And  ``^''=Exor
```

Each individual is composed of simple instructions (program lines) between variables and input and output parameters. The input in the form of sensor values are represented as register variables (S_i). The resulting action as motor speeds are also given as registers (motor1 and motor2).

3 The evolutionary algorithm

The same basic evolutionary algorithm is at the heart of both the simple system learning directly from actual sensor input and the system learning form past experiences.

The population of programs is initialized with random content at the outset. A simple tournament is used for the competitive selection of individuals to produce the right distribution of offspring. The GP-system with its simple steady state tournament selection algorithm [23], [26] has the following execution cycle:

1. Select four arbitrary programs from the population.
2. For each of the programs calculate fitness.
3. Make two copies (offspring) of the two individuals with highest fitness and let the copies be subject to crossover and mutation
4. Replace the two individuals with worst fitness with the two new offspring.
5. Repeat step 1 to 4

See also Figures 7, 8,11 and 13 which describe the flow-chart and show a diagram of the system.

3.1 Symbolic regression

Symbolic regression is the procedure of inducing a symbolic equation, function or program able to fit given numerical data. Genetic programming is ideal for symbolic regression and most GP applications could be reformulated as a variant 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
```

These input/output pairs or fitness cases are consistent with the function below:

```
f(x) = x*x+x
```

This very simple example would, in our register machine language, look as:

```
a = x*x;

y_out = a+x;


or:


a=x+1;

y_out = x*a;
```

Here the input and the expected output both consist of a single number, but in many cases symbolic regression is performed with vectors specifying the input/output relation of the sought function. In the examples below the input vector has more then 10 components and the output vector has in some cases two outputs.

The fitness used to guide the system during evolution is often some kind of error summation of the expected values versus the actual values produced by an individual program.

4 The Khepera Robot

Our experiments were performed with a standard autonomous miniature robot, the Swiss mobile robot platform Khepera [17]. It is equipped with eight infrared proximity sensors. The mobile robot has a circular shape, a diameter of 6 cm and a height of 5 cm. It possesses two motors and 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. They emit infrared light, receive the reflected light and measure distances in a short range: 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 a workstation, with data and commands communicated through the serial line. Alternatively, the controlling algorithm is cross-compiled on the workstation and down-loaded to the robot which then runs the complete system in a stand-alone fashion. At present, we have two version of the systems. One where the controlling GP-system is run on the workstation, and another where the system is downloaded and run autonomously on the micro-controller of the robot.

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

Figure 3: The Khepera Robot (diameter 6 cm)

The robot has several extension ports where peripherals such as grippers and TV cameras can be attached. Figures 3, 4 and 5 show the robot, its sensor placement and the sensor range in its environment.

5 Training Environment

The environment used for the obstacle avoiding task is about 70 cm $\times$ 90 cm. It has an irregular boarder with different angles and four deceptive dead-ends in each corner. In the large open area in the middle, movable obstacles can be placed. The friction between wheels and surface is low, 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.

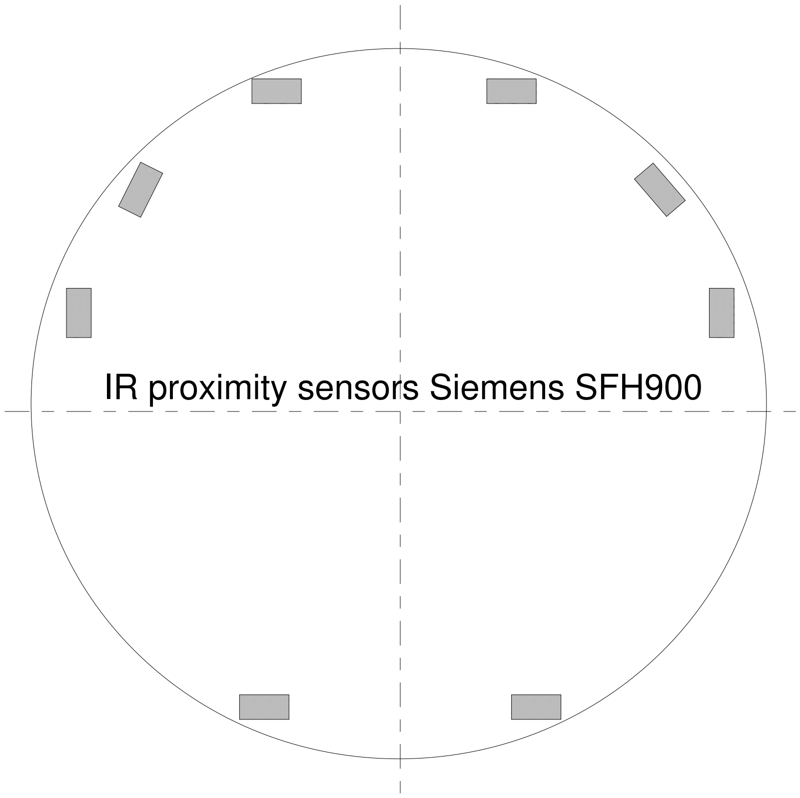

Figure 4: Position of the IR proximity sensors.

6 Objectives

The goal the GP systems in our experiments is to evolve obstacle avoiding behavior in a sense-think-act context. Both our systems operate in real-time and aim at obstacle avoiding behavior derived from real noisy sensorial data. The sensorial data come from eight proximity sensors mounted on the circular platform. (See [22], [16], [27], [3] for a discussion of this problem domain.)

Symbolic regression is, as mentioned above, the procedure of inducing a symbolic equation fitting given numerical data. Genetic programming is ideal for symbolic regression and most GP applications could be reformulated as variants of symbolic regression. In the obstacle avoiding application we would like to approximate a function that takes the sensor values as input vector and returns an action in the form of a vector of two motor speeds:

$$f(s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8) = \{m_1, m_2\} \tag{1}$$

This vector models the simple stimulus-response behavior of the robot. Our first attempt was to evolve the function directly through interaction with the environment [21]. The second, more efficient

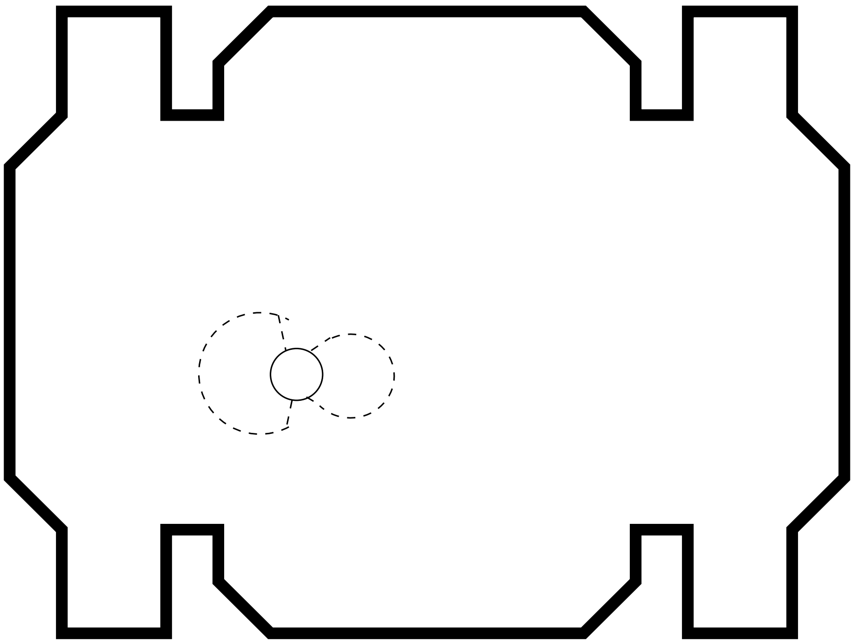

Figure 5: The approximate sensor range of the Khepera robot

approach reported here evolves a simulation or world model, instead, giving the relation between motor speeds, sensor input and fitness.

$$f(s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, m_1, m_2) = fitness \tag{2}$$

This is the central idea we shall discuss in this paper. It is memory-based in that a sensory-motor state is "associated" with a fitness, that might be termed "feeling".

6.1 Fitness Calculation

The fitness function defining 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 pain it experiences. In order to keep the robot from standing still or gyrating, it has a positive contribution to fitness, called pleasure, as well. It receives pleasure from going straight and fast. Both motor speed values minus the absolute value of their difference is thus added to the fitness.

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 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 of zero to 15. The fitness value can then be

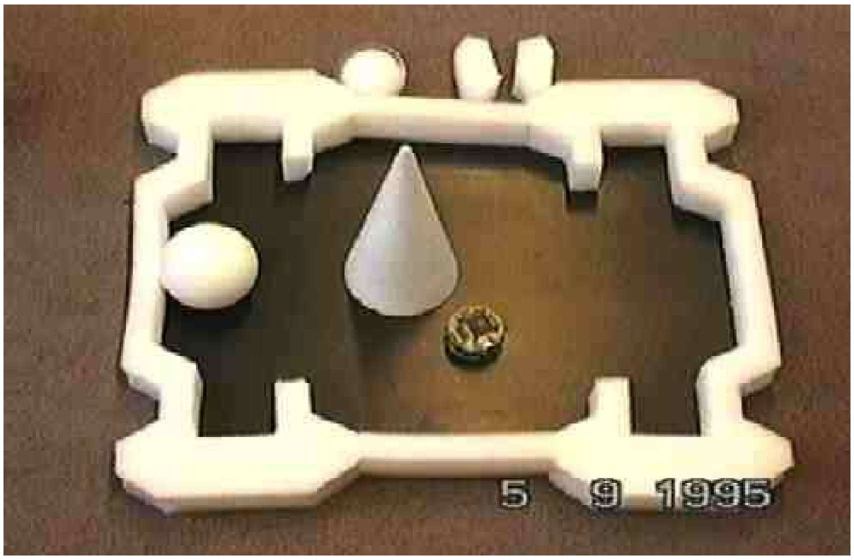

Figure 6: The Training Environment

expressed more formally as:

$$f = \sum s_i + \mid 8 - m_1 \mid + \mid 8 - m_2 \mid + \mid m_1 - m_2 \mid \qquad (3)$$

7 A First Approach

The early method tried to evolve the controlling function (Equation 1) directly and fitness was calculated from the current event. The evolved programs were true functions and no side-effects were allowed.

The learning algorithm had a small population size, typically less than 50 individuals. The individuals used the eight values from the sensors as input and produce two output values which were transmitted to the robot as motor speeds. Each individual program did this manipulation independent of the others and thus stood for an individual behavior of the robot when it was invoked to control the motors. Table 1 gives a summary of the problem and its parameters according to the conventions used in [15].

The modules of the learning system and the execution cycle of the GP system are illustrated in Figure 8. In the earlier, as well as in the new approach, each individual is thus tested against a different real-time fitness case. This could result in "unfair" comparison where individuals have to maneuver in situations with very different possible outcomes. However, our experiments show that over time this probabilistic sampling will even out the random effects in learning and a set of good solutions survive.

Table :	
Objective :	Obstacle avoiding behavior in real-time
Terminal set :	Integers in the range 0-8192
Function set :	ADD, SUB, MUL, SHL, SHR, XOR,
	OR, AND
Raw and standardized fitness :	Pleasure subtracted from pain value
	desired value
Wrapper :	None
Parameters :	
Maximum population size :	50
Crossover Prob :	90%
Mutation Prob :	5%
Selection :	Tournament Selection
Termination criteria :	None
Maximum number of generations:	None
Maximum number of nodes:	256 (1024)

Table 1: Summary of parameters used during training.

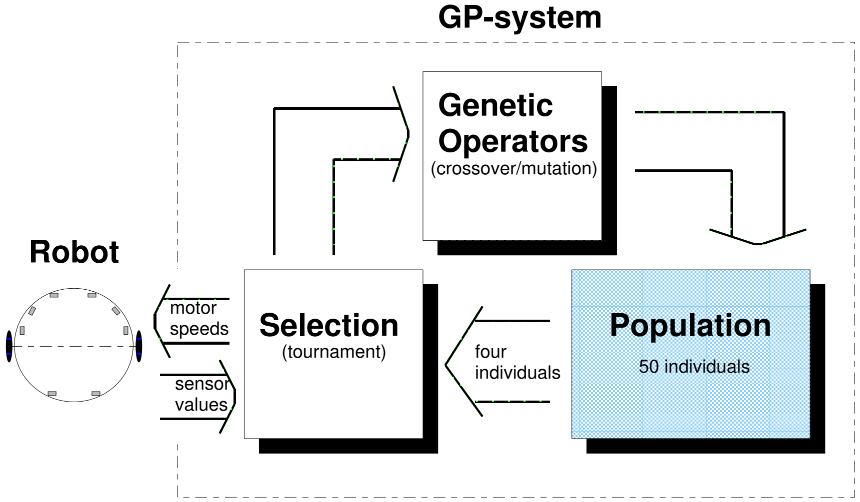

Figure 7: Schematic View of the Control System.

7.1 Results with non-memory approach

Interestingly, 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 more and more infrequent. 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.

After about 40-60 minutes, or 120-180 generation equivalents, the robot has learned to avoid obstacles in the rectangular environment almost completely. It has learned to associate the values from the sensors with their respective location on the robot and to send correct motor commands. In this way the robot is able, for instance, to back out of a corner or turn away from an obstacle at its side. Tendencies toward adoption of a special path in order to avoid as many obstacles as possible can also be observed. Figure 9 shows how the number of collisions per minute diminishes as the robot learns and the population becomes dominated by good control strategies.

Despite only processing results for less than 0.1% of the CPU time, the method competes well with other paradigms. It is for instance 100 times faster than a related evolutionary approach evolving Neural Network controllers on the same robot platform [8].

The moving robot gives the impression of displaying a very complex behavior. Its behavior resembles

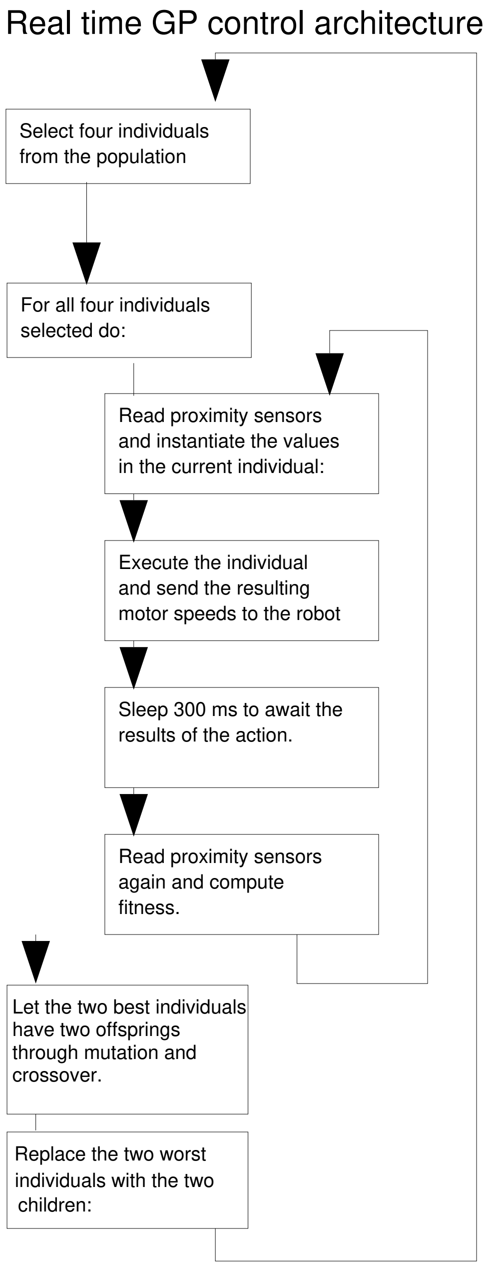

Figure 8: The execution cycle of the GP control architecture

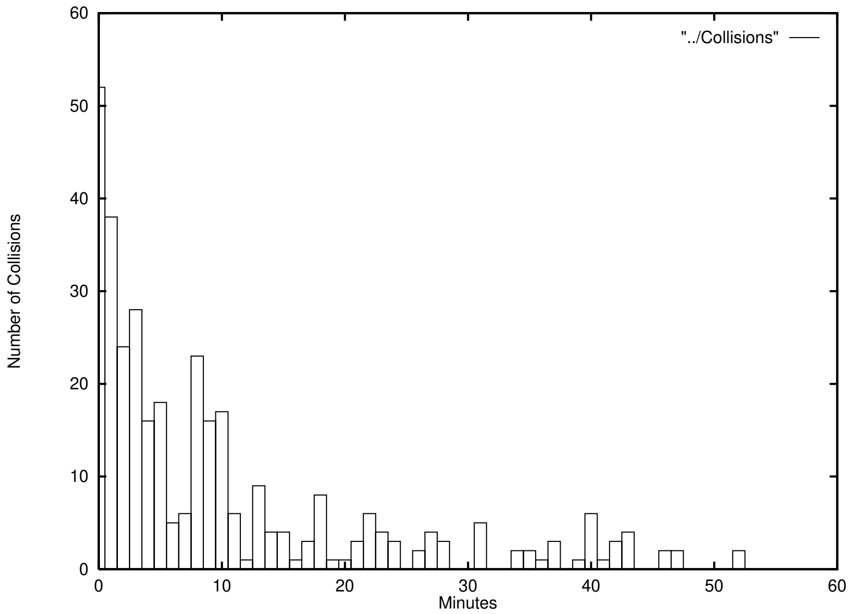

Figure 9: The number of collisions per minute in a typical training run with the environment in Figure 6.

that of a bug or an ant exploring an environment, with small and irregular moves around the objects, see Figure 14.

8 The memory-based GP control architecture

The memory-based control architecture consists of two separate processes. One process is communicating with sensors and motors as well as storing events into the memory buffer. The other process is constantly trying to learn and induce a model of the world consistent with the entries in the memory buffer.

We call the former process the *planning* process, because it is involved in deciding what action to perform given a certain model of the world. The latter process is called learning process, because it consists of trying to derive a model (in the form of a function) from memory data.

Figure 10 system gives a schematic illustration of the architecture of the control system. It contains six different major components:

1. The robot with sensors and actuators.

2. The memory buffer storing event vectors representing events in the past.

3. The GP system trying to evolve a model of the world which fits the information of the event vectors.

4. The fitness calculation module.

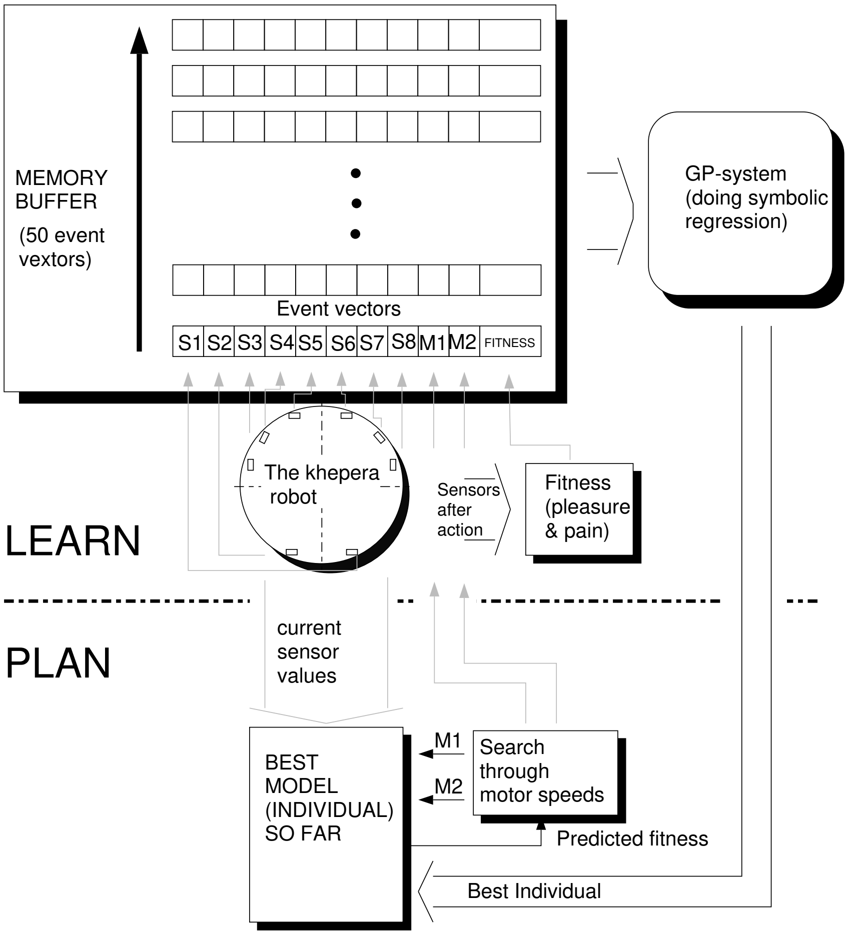

Figure 10: Schematic view of the memory based control architecture.

5. The currently best induced individual model.

6. A search module that tries to find the best action given the currently best world model.

Each component is communicating with the two main processes.

8.1 The Planning Process

The main execution cycle of the planning process has several similarities with the execution cycle of the simple genetic control architecture described in Section 7, see Figure 11.

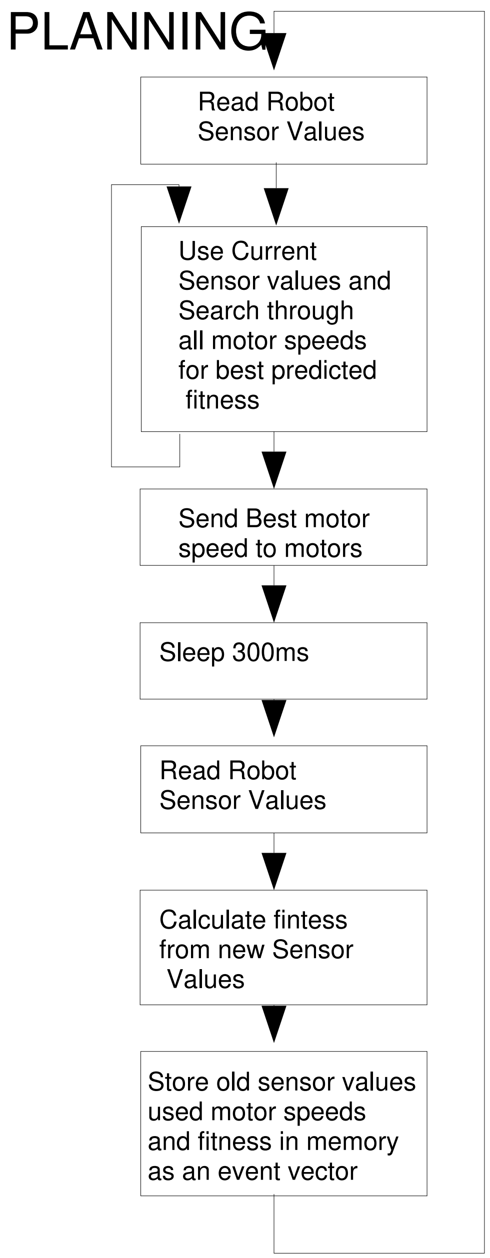

Figure 11: The execution cycle of the planning process.

It is the planning process which has actual contact with the robot and decides what action should be performed next. It operates according to the best model of the world supplied by the learning process. The process has three main objectives. It should communicate with the robot, find a feasible action and store the resulting event.

The loop in this process starts with reading all eight infrared proximity sensors. These values are used to instantiate the corresponding variables in the currently best world model. The next objective is to find a favorable action given the current sensor values. In the obstacle avoiding task used here the possible actions are 16 different motor speeds for each of the two motors. Each motor has 8 speeds forward and 7 backwards and a zero speed. Combining all alternatives of the two motors, there are 256 different actions altogether to choose from. This comparatively small figure means that we can easily afford to search through all possible actions while the world model provides us with a predicted fitness for each of them. The induced model in the form of a computer program from the learning process can thus be seen as a simulation of the environment consistent with past experiences, where the robot can simulate different actions. The action which gives the best fitness is remembered and sent as motor speeds to the robot.

If we would have an agent with so many possible atomic actions that exhaustive search becomes infeasible, then we could use a heuristic search method to find an action that gives good predicted fitness. We would, for instance, have another genetic programming system evolving a stepwise plan which optimizes fitness according to the currently best world model.

In order to get feed-back from the environment the planning process has to sleep and await the result of the chosen action. The planning process sleeps 300 ms while the robot performs the movement defined by the motors speeds. This time is an approximate minimum in order to get usable feed-back from changes in the sensor values. Thus, the main operation of the planning process is the sleeping period waiting for feedback from the environment and it, therefore, consumes less then 0.1 % of the total CPU time of the system.

After the sleeping period the sensor values are read again. These new values are used to compute a new fitness value. This fitness value is stored, together with the earlier sensor values and the motor speeds as an event vector. The event vector consists of 11 numbers; the eight sensor values, the two motor speeds and the resulting calculated fitness. This vector represents what the agent experienced, what it did and what the results were of its action. The memory buffer stores 50 of these events and shifts out old memories according to a schema described in Section 8.3.

It is then the responsibility of the learning process to evolve a program that simulates the environment as good and as consistent as possible with respect to the events in the memory buffer. As we will see below, this can be done by a straight-forward application of symbolic regression through genetic programming.

8.2 The learning process

The objective of the learning process is to find a function or a program which will predict the fitness of an action, given the initial conditions in form of the sensor values:

$$f(s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, m_1, m_2) = fitness \tag{4}$$

In most cases we have used an additional state variable as part of the program. This is simply a memory cell allowing the possibility to use side-effects in evolved solutions. This feature significantly improves performance in the complex environment used throughout these experiments:

$$f(s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, m_1, m_2, state_t) = \{fitness, state_{t+1}\} \tag{5}$$

Each event vector stores an instance of the values in equation 4 – the sensor values, motor speeds and the resulting fitness.

Figure 12 gives an illustration of interactions between the GP system and the memory buffer in the learning process.

8.3 Giving the System a Childhood

Our first approach to managing the memory buffer when all 50 places had been filled, was to simply shift out the oldest memories as the new entries come in. However, we soon realized that the system then forgot important early experiences. We found out that early mistakes, made before a good strategy was found, are crucial to remember in order to not evolve world models that permit the same mistakes to be done again. Hence, we gave the robot a childhood – an initial period whose memories were not so easily forgotten. The childhood also reduces the likelihood that the system displays a strategy sometimes seen even in humans – it would only perform actions which confirmed its current (limited) world model. Another important factor for successfully inducing an efficient world model is to have a stimulating childhood. It is important to have a wide set of experiences to draw conclusions from. Noise is therefore added to the behavior in the childhood to avoid stereotypic behavior very early in the first seconds of the system's execution. As long as experiences are too few to allow for a meaningful model of the world, this noise is needed to assure enough diversity in early experiences. The childhood of the system is defined as the time before the memory buffer is filled which takes about 20 seconds.

9 Results

The memory based system quickly learns the obstacle avoiding task in most individual experiments. It normally takes only a few minutes before the robot displays a successful obstacle avoiding behavior. The obvious reason for the speed up using memory can be identified in the flowchart of the algorithms.

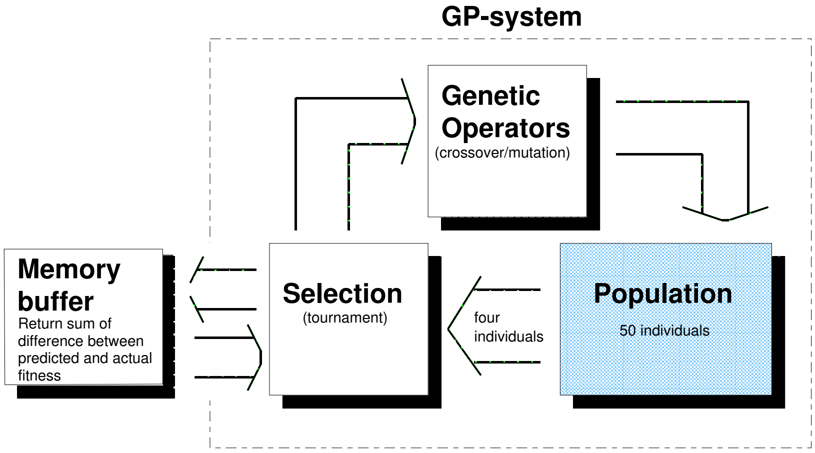

Figure 12: The GP system in the learning process

Note the difference between Figure 8, 11 and 13. There is no "sleeping" period in Figure 13 which means that the genetic programming system can run at the full speed possible for the CPU. This results in a speed up of more then 2000 times in the GP system. On the other hand, there is now a more complex task to learn. Instead of evolving an ad-hoc strategy for steering the robot, the system now has to evolve a complete model of relationships between the eight input variables, the two action variables and the fitness. This forces us to increase the population size from 50 individuals to 10,000 to ensure robust learning, compare Table 2. The system still has to wait for the robot to collect enough memory events to draw some meaningful conclusions. Yet the speed up with memory exceeds a factor 40 which makes it possible for the system to learn a successful behavior in less 1.5 minutes on average. All in all this means that the behavior emerges 4000 times faster than in similar approaches [8].

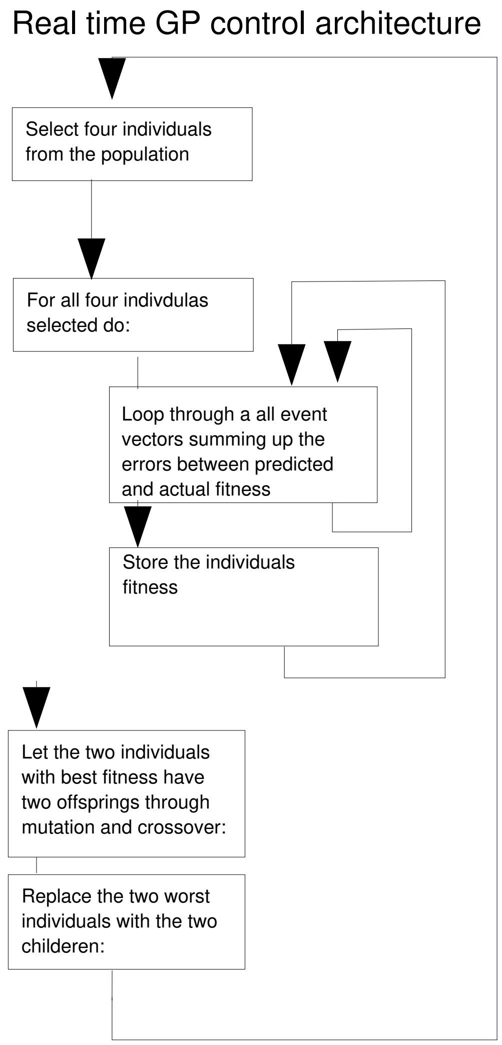

Figure 13: The GP system in the learning process

Objective :	Symbolic regression of environment simulation
Terminal set :	Integers in the range 0-8192
Function set :	ADD, SUB, MUL, SHL, SHR, XOR,
	OR, AND
Raw and standardized fitness :	The sum taken over 50 fitness cases
	of the absolute value of difference
	between the actual and desired
	value predicting the fitness.
Wrapper :	None
Parameters :	
Maximum population size :	10000
Crossover Prob :	90%
Mutation Prob :	5%
Selection :	Tournament Selection
Termination criteria :	None
Maximum number of generations:	None
Maximum number of nodes:	256 (1024)

Table 2: Summary of the parameters used by the GP system in the learning process

The behavior of the robot is very different between the two systems discussed here. The system without memory behaves in a very complex way and gives the impression of a small bug which randomly runs around avoiding obstacles, but with little overall strategy, see Figure 14.

The memory system, on the other hand, displays a set of very “thought through” behaviors. The robot always displays a clear strategy and travels in straight lines or smooth curves. Some of the behaviors evolved show an almost perfect solution to the current task and fitness function, see Figure 15.

The robot usually demonstrates a limited set of strategies during evolution in our experiments. Some of the emerging intelligent strategies are illustrated in Figure 15 and might be described as belonging to different behavioral classes (ordered according to increasing success):

1. *The straight and fast strategy*: This is the simplest ”intelligent” behavior. The induction process has only seen the pattern arising from the pleasure part of the fitness function. The model of the robot and its environment thus only contains the relationship expressing that going straight and fast is good. The robot consequently heads into the nearest wall and continues to stand there spinning its wheels. This strategy sometimes emerges right after the childhood when the noise is removed and the system is solemnly controlled by inferences from the induced model, see Section 8.3.

2. *The turning behavior*: The second simplest strategy is based on the experience that turning often improves fitness. The robot starts spinning around its own axis and does avoid all obstacles but also ignores the pleasure part of the fitness rewarding it for going straight and fast.

3. *The dancing strategy*: This strategy uses the state information in the model and navigates to the open space where it starts to move in an irregular circular path avoiding obstacles. Most of the time the robot moves around keeping a distance to the obstacles big enough to avoid any reaction from its sensors. If this strategy worked in all cases it would be nearly perfect because it keeps obstacles out of reach of the sensors and the robot is totally unexposed to pain. In most cases, however, the robot wanders off its patch and comes too close to an obstacle where it then is unable to cope with the new situation and experiences collisions.

4. *The backing–up strategy*: This is the first effective technique that allows the robot to avoid obstacles while moving around to some extent. The path the robot travels is, however, very limited and it is not the kind of solution we would prefer.

5. *The bouncing strategy.* 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 minimum speed change in the robot's path.

6. *The perfect or nearly perfect strategy*: The robot uses the large free space in the middle of the training environment to go straight and fast, optimizing the pleasure part of the fitness. As soon as the robot senses an object it quickly turns 180 degrees on the spot and continues going straight and fast. This strategy also involves state information because turning 180 degrees takes several events in the robot's perception and that cannot be achieved without states.

Most of our experiments displayed a simple behavior very early after the childhood, just to realize a more successful pattern a few seconds later and changing its strategy correspondingly. The change in strategy was always accomplished by a new best individual and fitness value displayed by the GP algorithm.

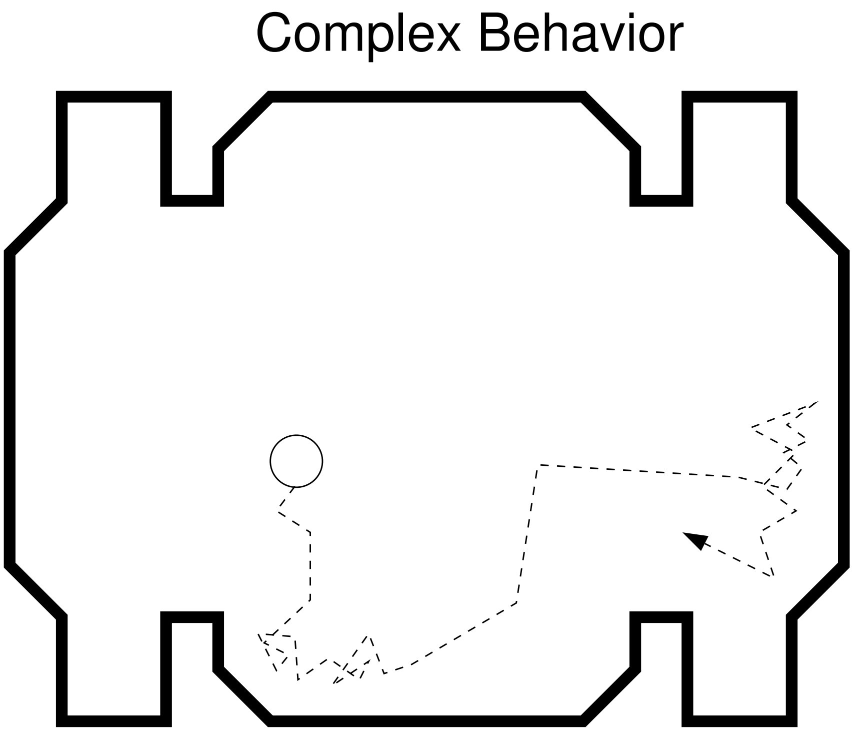

Figure 14: Behavior of the Non-Memory System

Table 3 reports the results of 10 evaluation experiments with the memory bases system. The results were produced by timing the robots behavior in 10 consecutive experiments. In each experiment the robot was watched for 20 minutes before the experiment was terminated. Each time the behavior changed was noted. The table gives the number of the experiment, the strategy displayed when the experiment was terminated and the time when this strategy first appeared.

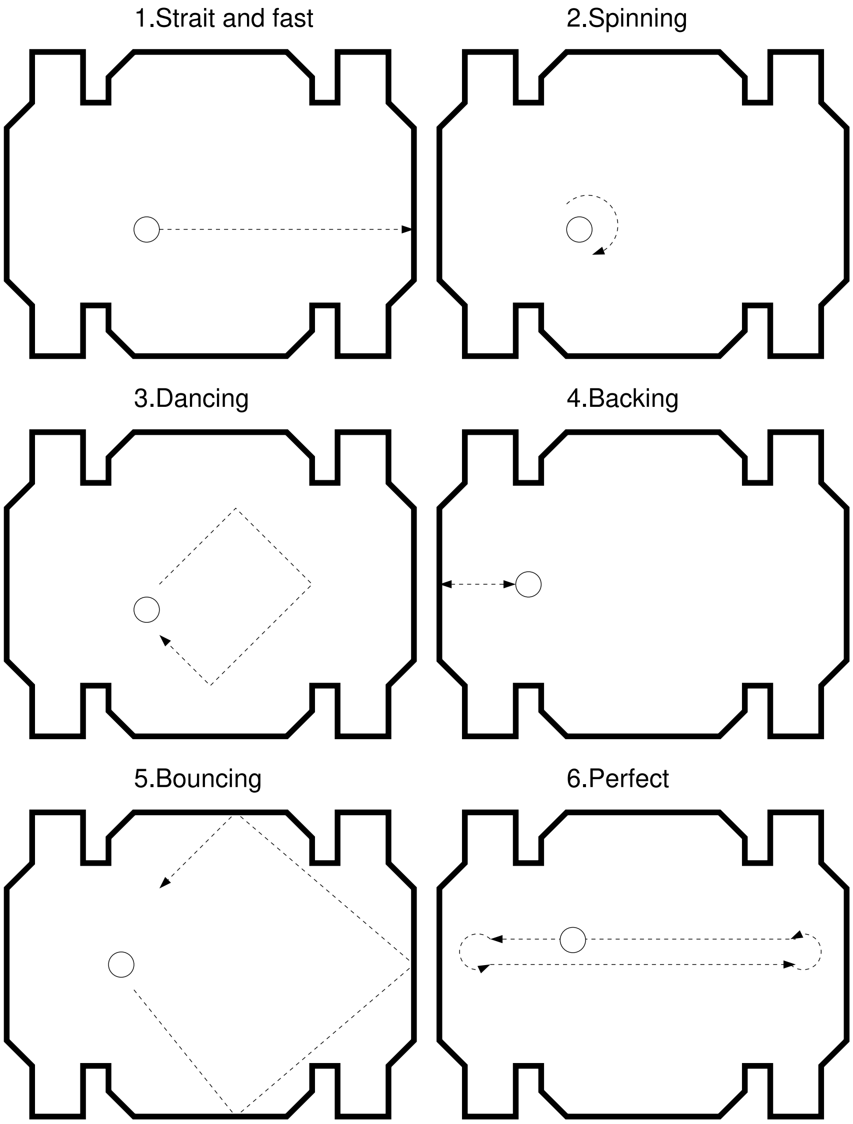

Figure 15: Different common strategies that evolves in the obstacle avoiding behavior with the memory bases learning method.

9.1 The Autonomous System

It is not completely evident what really constitutes an autonomous agent. Some would argue that the autonomy is a property of the controlling algorithm while others would argue that physical autonomy is needed.

In order to try the fully autonomous paradigm, we have ported a special version of the systems to

Run number	Result	Time (minutes)
1	perfect	1.5
2	perfect	1.5
3	backing	0.5
4	perfect	3.0
5	perfect	2.0
6	perfect	2.0
7	perfect	1.5
8	perfect	2.0
9	perfect	1.0
10	dancing	0.5

Table 3: Results of 10 consecutive runs with the memory system.

the micro-controller. It is possible to download these systems via the serial cable to the robot. With the accumulators switched on, the robot can then be disconnected from the workstation and can run fully autonomous. The Motorola 68331 micro-controller then runs the complete GP learning 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 1KB, in our present setup. The complete system without memory consists of 50 individuals and occupies 82KB which is well within the limits of the on-board system. The more complex system, learning from memory, has to use a smaller population size than the desired 10000 individuals. This results in less robust behavior with a more frequent convergence to local optima such as displayed by the first strategies in Figure 15.

In either case, it is demonstrated herewith that the compactness of the compiling GP system enables relatively powerful solutions in weak architectures such as those used in embedded control.

10 Future Work

As next steps we would like to investigate other tasks such as wall–following, and different kinds of navigation. The method is flexible in the sense that the only module necessary to change when adapting to a new task is the fitness function.

We would also like to evaluate the use of our approach with autonomous systems which have a wider set of possible actions. In such systems it would be infeasible to use exhaustive search to find the best action according to a world model. Handley has previously demonstrated the feasibility of GP for evolution of plans for simulated robots [11]. With a real robot and the memory based system – the planning system could incorporate its own GP system to evolve a suitable plan optimizing the outcome given the currently best world model.

Another possibility is to use the evolved simulation as environment when evolving a steering function such as the one in Section 7. This would have several advantages, for instance, it would give an ordering measurement involving us to pick the current best steering function and let it control, the agent.

Finally, exploration of different strategies of "active learning" [5] is warranted. The decision which memory entries to keep and which ones to discard will have a profound influence on the resulting world model.

11 Summary and Conclusions

We have demonstrated that a GP system can be used to control an existing robot in a real-time environment with noisy input. The evolved algorithm shows robust performance even if the robot is lifted and placed in a completely different environment or if obstacles 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 [20].

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 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 devices. We have demonstrated that the use of memory with a real robot and a GP based control system could speed up learning by a factor of 40. Furthermore, the strategies evolved with the memory based system display a smoother, less chaotic behavior, undisturbed by the internal dynamics of the GP system. The memory-based system could also be expected to scale up better because training times are not directly related to the dynamics of the agent and its environment, but instead almost completely depend on the difficulty of the induction problem in the application domain.

Acknowledgment

One of us (P.N.) acknowledges support by a grant from the Ministerium für Wissenschaft und Forschung des Landes Nordrhein-Westfalen under contract I-A-4-6037-I

References

[1] Agre P.E. and Chapman D. (1990) What are plans for?. In *Robotics and Autonomous Systems* Vol. 6, Nos. 1,2, Elsevier Science Publisher B.V. The Nederlands.

[2] Banzhaf, W. (1994) Genotype - Phenotype Mapping and Neutral Variation, A case study in Genetic Programming. In *Proc. 3rd. Int. Conf. on Parallel Problem Solving from Nature*, Jerusalem, Y.Davidor, H.P. Schwefel, R. Maenner (Eds.), Springer, Berlin.

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

[4] Cliff D. (1991) Computational Neuroethology: A Provisional Manifesto, in: *From Animals To Animats: Proceedings of the First International Conference on simulation of Adaptive Behavior*, Meyer and Wilson (eds.), MIT Press, Cambridge, MA

[5] For a recent report, see, e.g. Cohn, D. and Lewis, D. (1995). Working notes from the AAAI-95 symposium on Active Learning. MIT, Cambridge, MA, 1995.

[6] Cramer, N.L. (1985). A representation for adaptive generation of simple sequential programs. In *Proceedings of an International Conference on Genetic Algorithms and Their Applications*, pp183-187

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

[8] Floreano D., F. Mondada, (1994) Automatic Creation of an Autonomous Agent: Genetic Evolution of a Neural-Network Driven Robot, in *From Animals to Animats III: Proceedings of the Third International Conference on Simulation of Adaptive Behaviour*, D. Cliff, P. Husbands, J. Meyer and S.W. Wilson, Eds., MIT Press-Bradford Books, Cambridge, MA.

[9] Fogel, L.J., Owens, A.J., Walsh, M.J.,(1966) Artificial Intelligence through Simulated Evolution. Wiley, New York

[10] Friedberg, R.M. (1958) A Learning Machine - Part I,*IBM Journal of Research and Development* IBM, USA 2(1), 2-11.

[11] Handley S. (1994) The automatic generation of Plans for a Mobile Robot via Genetic Programming with Automatically defined Functions, in: *Advances in Genetic Programming*, K. Kinnear, Jr. (ed.), MIT Press, Cambridge, MA

[12] Harvey I., Husbands P. and Cliff D.(1993) Issues in evolutionary robotics, in: *From Animals To Animats 2: Proceedings of the Second International Conference on simulation of Adaptive Behavior*, Meyer and Wilson (eds.), MIT Press, Cambridge, MA

[13] Holland, J. (1975) *Adaption in Natural and Artificial Systems*, Ann Arbor, MI: The University of Michigan Press.

[14] James W. (1890) The principles of psychology Vol.1. Originally published: Henry Holt, New York1890.

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

[16] Mataric M.J.(1993) Designing Emergent Behaviors: From Local Interactions to Collective Intelligence, in: *From Animals To Animats 2: Proceedings of the Second International Conference on simulation of Adaptive Behavior*, Meyer and Wilson (eds.), MIT Press, Cambridge, MA

[17] Mondada, F., Franzi,E., and Ienne, P. (1993) Mobile robot miniaturization. In *Proceedings of the third international Symposium on Experimental Robotics*, Kyoto, Japan.

[18] Nordin J.P. (1994) A Compiling Genetic Programming System that Directly Manipulates the Machine-Code, in: *Advances in Genetic Programming*, K. Kinnear, Jr. (ed.), MIT Press, Cambridge, MA

[19] Nordin J.P. and Banzhaf W. (1995a) Evolving Turing Complete Programs for a Register Machine with Self-Modifying Code, in: *Proceedings of Sixth International Conference of Genetic Algorithms, Pittsburgh, 1995*, L. Eshelman (ed.), Morgan Kaufmann, San Mateo, CA

[20] Nordin J.P. and Banzhaf W. (1995b) Complexity Compression and Evolution, in *Proceedings of Sixth International Conference of Genetic Algorithms, Pittsburgh, 1995*, L. Eshelman (ed.), Morgan Kaufmann, San Mateo, CA

[21] Nordin P. and Banzhaf W. (1995) Genetic Programming controlling a Miniature Robot, in: *Proceedings of AAAI-95 Fall Symposium on Genetic Programming*, November 10-12 1995, MIT, Cambridge, MA

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

[23] Reynolds C.W. (1994) Evolution of Obstacle Avoidance Behavior, in: *Advances in Genetic Programming*, K. Kinnear, Jr. (ed.), MIT Press, Cambridge, MA

[24] Schwefel, H.-P. (1995) *Evolution and Optimum Seeking*, Wiley,New York

[25] SPARC International Inc., (1991), The SPARC Architecture Manual, Menlo Park, CA

[26] Syswerda G. (1991) A study of Reproduction in Generational Steady-State Genetic Algorithms, in: *Foundations of Genetic Algorithms*, Rawlings G.J.E. (ed.), Morgan Kaufmann, San Mateo, CA

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