Learning Robot Behaviors by Evolving Genetic Programs

Kwang-Ju Lee
Artifial Intelligence Lab (SCAI)
Cognitive Science Program
Seoul National University
Seoul 151-742, Korea
kjlee@scai.snu.ac.kr

Byoung-Tak Zhang
Artificial Intelligence Lab (SCAI)
School of CSE & Cognitive Science Program
Seoul National University
Seoul 151-742, Korea
btzhang@scai.snu.ac.kr

Abstract

A method for evolving behavior-based robot controllers using genetic programming is presented. Due to their hierarchical nature, genetic programs are useful representing high-level knowledge for robot controllers. One drawback is the difficulty of incorporating sensory inputs. To overcome the gap between symbolic representation and direct sensor values, the elements of the function set in genetic programming is implemented as a single-layer perceptron. Each perceptron is composed of senory input nodes and a decision output node. The robot learns proper behavior rules based on local, limited sensory information without using an internal map. First, it learns how to discriminate the target using single-layer perceptrons. Then, the learned perceptrons are applied to the function nodes of the genetic program tree which represents a robot controller. Experiments have been performed using Khepera robots. The presented method successfully evolved high-level genetic programs that control the robot to find the light source from sensory inputs.

1. Introduction

Robot control programs can be considered as hierarchical structures of basic behavior modules. When designing this structure, it is impossible or very difficult to dertermine the hierarchical structure of the control program and its size in advance. Genetic programming (GP) provides a powerful tool to design robot controllers of varying complexity and shape. Moreover, basic modules and hierarchical association between them can be easily represented by function and terminal structure of genetic program.

Several attempts have been made to control robots using GP. Koza [1] used GP to evolve a control program for artificial ants foraging food in grid environment. Zhang and Cho [2] devised the fitness switching method to evolve coordinated collective behaviors, such as herding and box-pushing, among several agents in a simulated grid environment.

The genetic programs used in the above approaches are composed of symbolic expressions to represent high-level decision rules and plans. However, these approaches are not appropriate to real world situations due to their ignorance of detailed real-world sensor inputs and motor control outputs.

In this paper, we present a method for bridging the gap between symbolic expressions and direct sensor values from the hardware robots. Instead of hand-coded symbolic expressions, a single-layer perceptron is used as elements of the function set in genetic programs. Each perceptron is composed of senor information in its input layer and a decision node in its output layer. This approach provides the robot with a learning ability as well as a method for finding proper sensor combinations and associated parameters to determine particular real-world situation. Genetic proramming is then applied to organize proper high-level strategies for achieving a certain task.

The perceptrons are first trained and then the genetic programs are evolved using the learned perceptrons. Since evolution takes a very long time and the initially created programs, which usually exhibit low fitness value, may cause damage to robot hardware when directly controlled, we used a Khepera simulator developed by Michel [3] which well imitates the real world for evolving proper computer programs. After evolving in simulation, we transferred the evolved controller to the real Khepera robot and observed the robot learning proper behavior rules.

The paper is organized as follows. Section 2 surveys the related work. In Section 3 we present the genetic program architecture used in evolving robot controllers. Section 4 describes experimental setup. Section 5 reports experimental results. Section 6 discuss our results and further work.

0-7803-6456-2/00/$10.00 ©2000 IEEE

2. Related Work

Mobile robot behavior can be considered as a result of mapping a certain stimulus into an appropriate response. In order to learn a certain behavior, the robot must perceive some stimulus and find its proper interpretation. This interpretation could be one of its past experiences, its predefined situations, preconditions of if-then rules or just a vector composed of some sensor values [5]. To endow a mobile robot with an ability to adapt to a dynamic and unpredictable environment, the domain specific knowledge should be kept as small as possible since well-suited knowledge or strategies in one domain may be useless in another domain. It should also be tried to keep the basic modules as primitive as possible. All these processes should be achieved within the limit of build-in equipment, of the robot itself.

While keeping behaviors as simple as possible, coordination architectures of simple behaviors are needed. Whithin the behavior-based paradigm, there are several coordination architectures [5]. One of them is the subsumption architecture [7]. In this architecture, complex behavior modules subsume simpler behavior modules, and coordination occur via inhibition and suppression among modules while keeping hierarchy and priority between behaviors. Lower behavior modules do not care about higher modules nor do they know about it. Though this architecture has been well coordinated behavior to be successful in producing, it has some drawbacks in learning and adaptation because of its hardwired hierarchical topology and difficulty of design and incorporation of new behavior module.

Another well-known architecture is the schema-based architecture [5]. In the schema-based architecture, behavior coordination is simply a vector summation. The output vector of all active behavior modules contributes to some degree to the robot's global motion. As a special form of a schema-based coordination, an action selection mechanism determines which behavior is performed by activation levels of each behavior modules.

Though above coordination mechanisms show strengths as a reactive system, there are much room for improvement. Because they do not impose internal representations and a sophisticated planning scheme with relatively simple computation, a behavior-based system has strength in general applicability to a dynamically changing environment and in real time interaction. But their ability is relatively confined to simple task such as obstacle avoiding, landmark finding, and simple box-pushing in obstacle-free environments [9]. To achieve a more complex task, the robots needs such functionalities as hierarchical association between behaviors, sequential ordering of several behavior modules, internal representation of a given world or past experiences, and, as the most essential feature, learning ability to adapt to a dynamic, unpredictable, and vast world. Of course, these additional capabilities should not hurt inherent merits of the behavior-based paradigm.

Genetic Programming paradigm [1] provides a powerful tool for automatically learning behavior coordination mechanisms. Genetic programming is an automatic programming method that finds the most fit computer programs by means of natural selection and genetics [1]. Since the computer programs in genetic programming are usually represented as trees or LISP S-expressions, symbolic knowledge can be easily represented. Since the genetic program consists of arbitrary function nodes and terminal nodes, genetic programming is generally applicable. Besides being different from simple genetic algorithms(GAs) which are usually encoded as a fixed bit string, genetic programming can represent hierarchical relationships between modules. Sequential action ordering can also be represented in genetic programs by defining proper functions. This feature is very useful in some situation, since it is somtimes necessay to produce well-ordered sequential actions.

Several attempts to control robots using GP have been made. Koza [1] used GP to evolve a control program for artificial ants foraging food in grid environment. In this approach, the criteria for judging the environment and a set of actions that a robot can take are predefined in a symbolic manner. Though interesting in the high level planning of robot behaviors, this approach is not appropriate to real world situations due to its ignorance of detailed sensing and action module implementation. Greg [8] has used GP to co-evolve robot controllers. In this approach, a robot reacts directly to the signals of sensors without judging its current state. This has the advantage that the robots can take actions very fast, but the long-term behavior may be poor due to its lack of action planning. Zhang and Cho [2] devised a fitness switching algorithm to evolve collective behavior, such as herding and box-pushing, among several agents in a simulated grid environment. Although this approach has demonstrated successful evolution of emergent collective behavior performing relatively complex tasks – avoiding obstacles and box-pushing in coordinatd group motion – from primitive behaviors, it has a pending question of transferring the simulation result to a real robot hardware [2]. Lee, Hallam, and Lund [9] have proposed a developing method for behavior-based controller using genetic programming. They deploy GP to evolve behavior primitives and apply evolved behavior primitives to a predesigned control structure resembling of the subsumption architecture of Brook's. Though they evolve primitive behavior modules and take an *arbitrator* module to control two primitive behaviors to overcome the lack of action planning, the prespecified overall control structure may impose constraints on the flexibility of strategies evolved.

Table 1: Function and terminal symbols used in GP trees.

	SYMBOL	MEANING
Function set	IOL, IOR, IOF, IOB	If Obstacle is located Left/Right/Front/Back
	ITL, ITR, ITF, ITB	If Target is located Left/Right/Front/Back
	PROG2	Execute the left subtree and then the right subtree
Terminal set	TL, TR	Turn Left/Right
	MF	Move Forward
	RM	Random Move
	ST	STop

3. Neurogenetic Programs

Genetic programming(GP) is an automatic programming method that finds the most fit computer programs by means of natural selection and genetics. It starts with initial population of randomly generated computer programs, and these computer programs undergo adaptation by applying genetic operators to approach the most fit solution for given problem space [1]. Each individual program in the population is usually represented as *trees* or LISP S-expressions. The tree consists of elements from the function set and the terminal set appropriate to the problem domain. Typically, terminal symbols provide values to the GP program while function symbols perform operation on their input, which are either terminals or output from other functions. For our work, functions denote sensing of environments and are implemented by perceptrons. Terminals denote actions to be taken. Table 1 shows the function and terminal symbols used in our genetic programs. An illustrative example of the genetic program for mobile robot control is shown in Figure 1.

When the computer programs are created, each individual in the population is run so that its fitness is measured. In our work, each computer program is evaluated in terms of how well it performs the given task in the particular environment. Then computer programs are selected in proportional to their fitness. Offspring programs are produced from the selected programs by applying genetic operators such as reproduction, crossover, and mutation. The offspring population replaces the old population.

The adaptation mechanism of GP is similar to that of genetic algorithms, except their representations. Individuals of GP are usually represented as trees or LISP S-expressions. But GA chromosomes are usually represented as fixed-length bit strings. For many problems, the most natural representation for a solution is a hierarchical computer program rather than a fixed-length bit string. The size and the shape of the hierarchical computer program that will

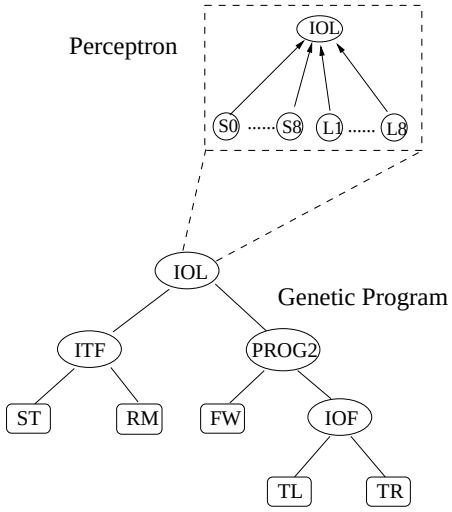

Figure 1: An example of a robot control program represented in a GP tree. The tree denotes the control strategy: 'If Obstacle is located Left then do [If Obstacle is located Front then Stop else Random Move] else do [Move Forward] and [If Obstacle is located Front then Turn Left else Turn Right]'. Function nodes are represented as circles and implemented as perceptrons as shown in the box.

solve a given problem are generally not known in advance, and it is difficult, unnatural and constraining to represent hierarchical computer programs of dynamically varying sizes and shapes with fixed-length bit strings [1].

Robot control programs can be constructed by a hierarchical structure of basic functional modules. When designing this structure, it is impossible or very difficult to have complete information about the given environment in advance. Thus, the distingushing feature of GP in evolving tree structures of dynamically varying size and shape can be very useful to design robot controllers. Moreover, basic modules and hierarchical association between them is more easily represented by function nodes and terminal nodes structure of GP trees rather than GA bit strings.

In our work, each function node represents a state-decision rule such as 'if the obstacle is located to the left'or 'if the target object is located behind the robot'. These symbolic expressions are useful for designing a high-level planner, but realization in hardware needs more tedious and complicated work. Usually these expressions are implemented as a program module consisting of hand-coded if-then rules based on unreliable heuristics of the designer. Since this method is weak and inflexible to the change of the environment and it is difficult to find proper parameters, we adopt single layer perceptrons as state-decision rules instead of using hand-coded state-decision rule. This approach provides the robot with learning ability as well as a method to find proper sensor combinations and associated parameters. The perceptron consists of 16 input nodes which take nor-

malized sensor input values and a single output node which reports whether the representing condition is satisfied. In this way, the function nodes can learn to determine the particular state in given environment. Terminal nodes are interpreted as action command such as 'Go forward', 'Turn right', or 'Stop'.

4. Experimental Setup

4.1 The Khepera Robot

The Khepera robot has 8 infrared sensors for proximity measurement, 8 light sensors for measuring ambient light, and two motor-drived wheels for movement. Figure 2 shows the robot equipment in the Khepera simulator [3]. The simulator has the same equipments as those of the real Khepera robot.

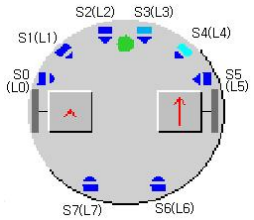

Figure 2: The Khepera robot in the Khepera simulator.

The 8 infrared sensors detect the proximity of objects in front of it up to 5 cm by reflection of infrared rays. Each sensor returns a value ranging $0 \sim 1023$, where 0 means that no object is perceived while 1023 means that an object is very close to the sensor. These sensors can also measure the level of ambient light around the sensor. It returns a value between $0 \sim 525$, where 0 corresponds to maximum brightness, 525 to maximum darkness. Robot movements are carried out by controlling left and right motors, which can take value within the range of -10.0(back) $\sim$ +10.0(forward). To approximate real environments, the simulator adds random noise of $\pm 10\%$ to the proximity sensor output and $\pm 5\%$ noise to the light sensor output. Also, motor amplitude and direction commands are realized with distortion of $\pm 5\% \sim \pm 10\%$.

4.2 Learning and Evolution

As mentioned above, table 1 shows the function and terminal symbols used in our genetic programs. All function nodes are implemented as single-layer perceptrons except the PROG2 element. Each perceptron consists of 16 input units and 1 output unit. All input units are fully connected to the output unit. The 8 input units take 8 proximity sensors (in Figure 2, S0 $\sim$ S7) and normalize $0 \sim 1023$ sensor output values to $0.0 \sim 1.0$. The other 8 input neurons take 8 light sensors (in Figure 2, L0 $\sim$ L7) and normalize sensor output values to $0.0 \sim 1.0$.

The output neuron's activation function and weight update rule is as follows:

$$o(\vec{x}) = \vec{w} \cdot \vec{x} \tag{1}$$

$$\vec{w} \leftarrow \vec{w} + \eta(t - o)\vec{x}, \tag{2}$$

where o is the output value, $\vec{w}$ is the weight vector, $\vec{x}$ is the input vector, t is the target value, and η is the learning ratio of 0.1.

The robot learns perceptrons first. Examples consist of sensor data, target value (if the condition is met then +1.0, else -1.0) and are sampled arbitrarily in the simulated environment. About 1000 examples were used in perceptron learning.

After the perceptron learning phase is completed, the procedure of evolving robot control programs begins. A robot control program is represented as a GP tree, which is composed of function nodes and terminal nodes. Maximum depth of trees is 10. Each function node denotes a situation decision rule, and each terminal node denotes an action command. When perceptrons complete learning, they play the role of function nodes in the GP trees. The evolutionary process starts by selecting one individual from the initial population. The selected control program is executed for a given number of move steps and then the fitness value of the current individual is evaluated by the following formula:

$$F(i) = w_1 \frac{C_i}{S} + \frac{S - w_2 H_i}{S}, \tag{3}$$

where S is the step number per life cycle(= 2000 steps), C_i is the number of times the robot collides with the obstacles, H_i is the number of times the robot approaches the target object, and w_1, w_2 are the weight parameters. These parameters are fixed as $w_1 = 1.0$, $w_2 = \frac{S}{10}$ through experiments.

The fitness measure consists of two terms: the first term is the collision ratio during a life cycle, the second is the *hit* ratio which reflects how often the robot achieves its goal, i.e. how often the robot reaches the target object without collision. Since we want better individuals to have lower fitness values, the number of hits should be subtracted from step counts. Weight parameters are multiplied to increase the convergence speed.

After evaluation, another control program in the population is selected and the robot is controlled by it. After all the individuals have been evaluated, we select the best 50% of the population by the uniform ranking selection method, and apply them to genetic operators such as reproduction, crossover, and mutation to create offspring for next generation. Reproduction, crossover, and mutation rates are 0.1, 0.8, and 0.1, respectively. The population size is 100 and the best individual from generation is always retained in the next generation (elitism).

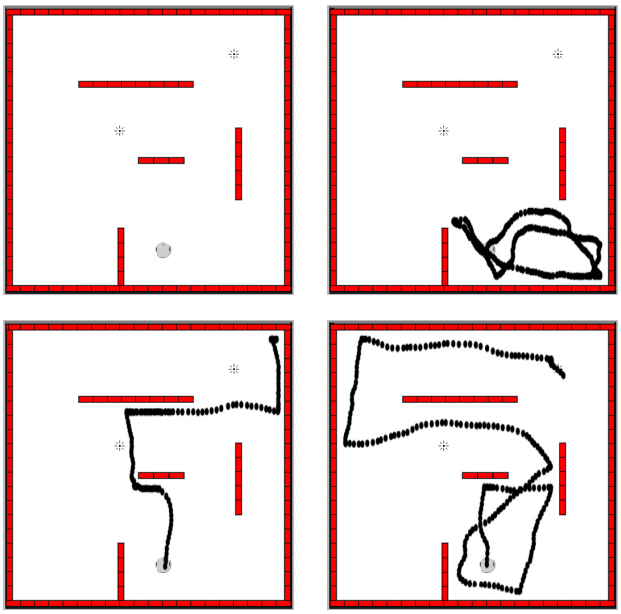

Figure 3: Illustrative snapshots of the simulation environment (top left), the trajectory of the best individual in the first generation (top right), the best individual in the 50th generation (bottom left), and the final best individual (bottom right).

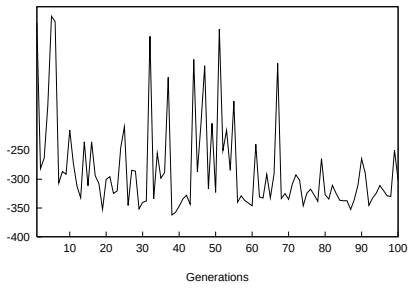

Figure 4: Fitness of the best individual at each generation.

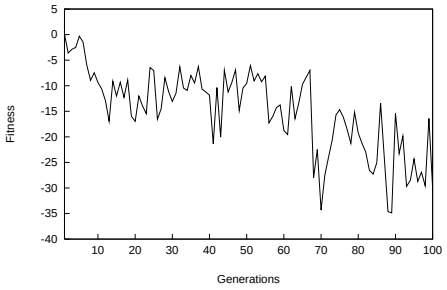

Figure 5: Average population fitness at each generation.

Throughout the perceptron learning and the evolutionary process, the same simulation environment has been used. The environment consists of surrounded walls as obstacles and two lamps as targets in the rectangle dimension of 1m $\times$ 1m.

After evolution, we transferred the evolved program to the Khepera robot in real world. The size of the experimental environment is identical to the simulation environment, 1m $\times$ 1m. However, the configuration is slightly different. Major troubles in real world experiments are ambient light. Contrary to the simulation environment where the ambient light source is only the inserted lamps (the target in the learning phase), the real environment is full of light sources such as the sun and indoor lights. Though we had much trouble with such ambient lights, we could not intercept the lights because we had to record the experimental procedure with a video camera to analyse the behavior of the robot. Thus the threshold of the perceptron was needed to be adjusted to draw a proper decision.

5. Experimental Results

The environment and some illustrative trajectory snapshots of the robot during the evolutionary process are shown in Figure 3. The change of fitness and performance during the evolutionary process are shown in Figure 4, 5, 6.

For the first 100 generations, both of the average fitness and the best fitness were improved (Figures 4 and 5). As you can see, from the 10th generation the fitness value of the best individual reaches as low as that of the final generation, yet showing a heavy fluctuation. Though the best individual was retained at each generation, the same control program can exhibit different fitness values between the trials because the fitness measure depends on the performance of it, not on the structure of the control program itself. However, from about 70th generation fluctuation decreases and the fitness value becomes stable, resulting the convergence of most individuals in the population to a stable state.

The average number of hits also increases steadly for the first 100 generations, while the collision count shows no significant change (Figure 6). Thus the ratio of hits to collision frequency steadly increased.

Figure 7 shows the environment in which the real Khepera robot has been operated with its trajectory controlled by the evolved program. The distal position changes to the target are presented along time (Figure 8). The target object is a lamp located in top–left corner. The trajectory shows how the robot find its way to the target.

6. Conclusion

We designed a method for evolving robot control programs using genetic programming. Our GP architecture provides a good robot behavior coordinator which can easily represent hierarchical symbolic rules between the basic modules with ability to adapt to a given environment. Instead of using predefined symbolic expressions as the func-

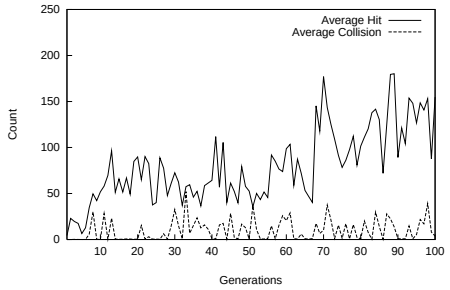

Figure 6: Change of the average number of collisions and the average number of hits.

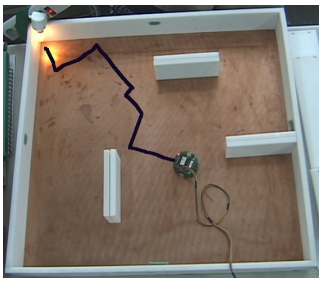

Figure 7: A trajectory of Khepera controlled by an evolved neurogenetic program.

tion set in GP trees, we used single-layer perceptrons as the elements of the function set during evolution. It provides a method to bridge the gap between symbolic representation used in the control programs and the sensor outputs of the robot hardware.

Evolution was carried out on the real-value based Khepera simulator, which is more realistic and easier to transfer results to the real robot hardware than the simple grid-based simulator. Evolved results showed that genetic programs could find behavior rules which were suitable for performing the given task. Then, we transferred the evolved program to the real Khepera robot. Though the simulation environment and the real environment were different, the robot could avoid obstacles and find the target to reach. Since the evolved control programs are composed of simple basic modules, they exhibited real-time performance, which are inherent strengths of behavior-based robot. These results demonstrate that our architecture has a potential for implementation of behavior-based adaptive robots in the real world in learning proper behavior rules with little domain-specific knowledge.

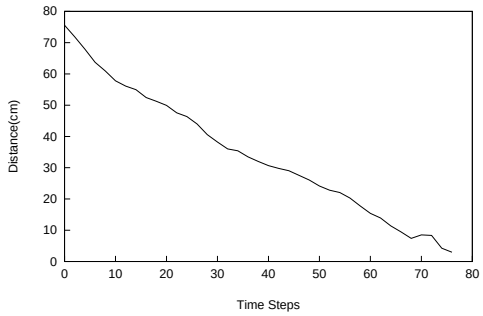

Figure 8: Distal position change along time (averaged over 5 trials).

Future work includes co-evolving the function set and the control programs and devising method for evolving more complex behaviors by concatenating several evolving control program modules.

Acknowledgments

This research was supported by the Korea Science and Engineering Foundation (KOSEF) under grant 981-0920-107-2.

References

[1] J.R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, The MIT Press, Cambridge, MA, 1992.

[2] B.T. Zhang, and D.Y. Cho, "Coevolutionary Fitness Switching: Learning Complex Collective Behaviors Using Genetic Programming", *Advances in Genetic Programming III*, The MIT Press, Cambridge, MA, 1998, pp. 425–445.

[3] O. Michel, "Khepera Simulator Package version 2.0: Freeware Mobile Robot Simulator Written by Oliver Michel", http://www.epfl.ch/lami/team/michel/khep-sim/index.html, 1996.

[4] D. Floreano, and F. Mondada, "Evolution of Homing Navigation in a Real Mobile Robot", *IEEE Transactions on Systems, Man and Cybernetics*, 26(3):396–407, 1996.

[5] R.C. Arkin, *Behavior-Based Robotics*, The MIT Press, Cambridge, MA, 1998.

[6] R.A. Brooks, "Intelligence without Representation", *Artificial Intelligence*, 47:135–159, 1991.

[7] R.A. Brooks, "A Robust Layered Control System For a Mobile Robot", *IEEE Journal of Robotics and Automation*, vol RA-2(1):14–23, 1986.

[8] M. Greg, "Using Co-evolution to Produce Robust Robot Control", *Late Breaking Papers at the 1997 Genetic Programming Conference*, J.R. Koza (editor), Stanford Bookstore, 1997, pp. 141–149.

[9] W.P. Lee, J. Hallen, and H.H. Lund, "Applying Genetic Programming to Evolve Behavior Primitives and Arbitrators for Mobile Robots", *The First NASA/DoD Workshop on Evolvable Hardware*, 1997, pp. 501–506.