Visual Obstacle Avoidance Using Genetic Programming: First Results

Martin C. Martin
Carnegie Mellon University
mm@cmu.edu

Abstract

Genetic Programming is used to create a reactive obstacle avoidance system for an autonomous mobile robot. The evolved programs take a black and white camera image as input and estimate the location of the lowest non-ground pixel in a given column. Traditional computer vision operators such as Sobel gradient magnitude, median filters and the Moravec interest operator are combined arbitrarily. Five memory locations can also be read or written to. The first evolved program is now controlling the robot.

When constructing a system, engineers typically practice iterative design, namely instantiating a design, evaluating it, and then modifying it in light of the evaluation. In the current work Genetic Programming can be seen as automating this process by iteratively improve the architecture of the system in fundamental, previously unplanned ways. The system described here successfully navigates in the hallways outside the lab.

1. Introduction

Computer vision in unstructured environments, such as a typical office environment, is notoriously difficult. Different methods have their strengths and weaknesses, and no one method is universally better or worse than the alternatives. To find a method that works empirically for a particular environment, I use Genetic Programming to evolve arbitrary expressions that combine the results of traditional computer vision operators over a window. The expressions return an estimate of the distance to the nearest obstacle in a given direction. The expression is evaluated for six different directions, and a separate hand-written program uses these estimates to steer the robot.

As in any field of research, one can find threads in the literature by following the evolution of a single idea. The idea that most inspires this work started with Ian Horswill's Ph.D. thesis on Polly the Robot. Polly gave simple tours of the seventh floor of the MIT AI lab, which had a textureless carpeted floor. Obstacles, or at least their boundaries, could therefore be detected as areas of visual texture. The system had problems with other carpet patterns, or even sharp shadows. Liana Lorigo extended this work by assuming the bottom of each image represented floor, and searched for areas with different colors or texture than the floor. However, an object near the robot could confuse it. Iwan Ulrich and Illah Nourbakhsh took the floor to be part of a previous image that had since been traversed. The work reported here is a first step towards automatically repeating and generalizing this thread.

1.1. Previous Work

Evolutionary Robotics

Evolutionary Robotics is a new field that uses simulated evolution to produce control programs for robots. Recent work can be found in [7] and [20]. Unless otherwise mentioned, the work reported here has been validated by running it on a real robot, not just in simulation. Most work uses bitstring Genetic Algorithms to evolve recurrent neural nets for obstacle avoidance and wall following using sonar, proximity or light sensors, e.g. [8, 17, 16]. Significantly, recurrent neural networks are considered much harder to train than feed forward networks, since gradient information typically isn t available. Evolutionary Computation doesn t use gradient information, and therefore even exploratory, toy problems use recurrence.

Nordin et al. [21] use a Genetic Programming variant that directly manipulates SPARC machine language. They use symbolic regression to predict the goodness of a state 300 ms in the future, based on the current sensor readings and action. For obstacle avoidance, the goodness is simply the sum of the proximity sensors, plus a term to reward moving quickly in a straight line. To choose a direction, the robot runs the best individual for all possible actions with the current sensor readings. The action with the highest estimated goodness is chosen. They use a population size of 10,000 and find that, in runs where perfect behaviour developed, it developed by generation 50.

Most work evolves in simulation, with the best individuals then run on a robot in the real world. Reynolds [24] has pointed out that without adding noise to a simulation, EC will find brittle solutions that wouldn t work on a real robot. Jakobi et al. [8] discovered that if there is significantly more noise in the simulation than on the real robot,

new random strategies become feasible that also don t work in practice.

As far as I know, no one has explored the reverse, i.e. whether entire classes of solution won t be found because they don t work in simulation. For example, dead reckoning error may be different on carpet then on a hard floor, so one possibility is to try to distinguish between them based on, say, their visual appearance. If this difference in error isn t modeled in the simulation, such a solution will never be found. Once evolutionary robotics gets beyond a basic stage, programs evolved in simulation may miss many subtle solutions.

As well, to my knowledge, no one has tried to simulate CCD camera images, either using standard computer graphics techniques or morphing previously captured images. The Sussex gantry robot [2] uses a CCD camera, but the images are reduced to the average brightness over three circles. These are significantly easier to simulate than a full CCD image, especially when the only objects are pure white on a black background. Smith [25] simulated a 16 pixel one-dimensional camera with auto iris on a robot soccer field. The 16 pixels were actually derived from 64 pixels; Smith doesn t say how. This is an important step, but again much easier than simulating a CCD image at, say, 160 x 120 pixels or above.

A few research groups perform all fitness evaluations on the real robot. Floreano and Mondada [4] evolve recurrent neural networks for obstacle avoidance and navigation from infrared proximity sensors. It takes them 39 minutes per generation of 80 individuals, and after about 50 generations the best individuals are near optimal, move extremely smoothly, and never bump into walls or corners. Naito et al. [19] evolve the configuration of eight logic elements, downloading each to the robot and testing it in the real world. Finally, the Sussex gantry robot [2] mentioned earlier has used evaluation on the real robot. They used a population size of 30, and found good solutions after 10 generations.

The closest work to that reported here was done by Baluja [1], who evolves a neural controller that interprets a 15 x 16 pixel image from a camera mounted on a car. The network outputs are interpreted as a steering direction, the goal being to keep it on the road. Training data comes from recording human drivers.

In summary, Evolutionary Robotics has used low bandwidth sensors, such as sonar or proximity sensors, presumably to cut down the amount of information to process. There are typically less than two dozen such sensors on a robot, and each returns at most a few readings a second. However, much traditional work in computer perception and robotics uses video or scanning laser range finders, which typically have tens to hundreds of thousands of pixels, and are processed at rates up to 10 Hz or more. Evolutionary Robotics has much to gain by scaling to these data rich inputs.

In addition, most Evolutionary Robotics has designed algorithms for simplified environments that are relatively easy to simulate. While evaluating evolved programs on real robots is considered essential in the field, those environments are typically still tailored for the robot. The current work attempts to evolve algorithms to interpret video of an unmodified office environment in real time, to help a robot wander while avoiding obstacles.

Visual Obstacle Avoidance

Somewhat surprisingly, there have only been a handful of complete systems that attempt obstacle avoidance using only vision in environments that weren t created for the robot. Larry Matthies group has built a number of complete systems, all using stereo vision [14]. They first rectify the images, then compute image pyramids, followed by computing sum of squared differences , filtering out bad matches using the left-right-line-of-sight consistency check, then low pass filter and blob filter the disparity map.

Their algorithm has been tested on both a prototype Mars rover and a HMMWV. The Mars rover accomplished a 100m autonomous run in sandy terrain interspersed with bushes and mounds of dirt [13]. The HMMWV has also accomplished runs of over 2 km without need for intervention, in natural off-road areas at Fort Hood in Texas [15]. The low pass and blob filtering mean the system can only detect large obstacles; a sapling in winter, for example, might go unseen.

Ratler [10] used a stereo vision algorithm to do autonomous navigation in planetary analog terrain. After rectification, the normalized correlation is used to find the stereo match. The match is rejected if the correlation or the standard deviation is too low, or if the second best correlation is close to the best. Travelling at 50 cm/sec over 6.7km the system had 16 failures, for a mean distance between failures of 417m. No information on failure modes is available.

David Coombs group at NIST has succeeded with runs of up to 20 minutes without collision in an office environment [3]. Their system uses optical flow from both a narrow and a wide angle camera to calculate time-to-impact, and provide feedback that rotationally stabilizes the cameras. Reasons for failure include the delay between perception and action, textureless surfaces, and hitting objects while turning (even while turning in place).

Liana Lorigo s algorithm [11, 12] assumes the bottom of the image represents clear ground, and searches up the image for the first window that has a different histogram than the bottom. This is done independently for each column. If the ground is mostly flat, then the further up the

image an object is, the further away it is. The robot heads to the side (left or right) where the objects are higher up.

Failure modes include objects outside the camera s field of view, especially when turning. Other failure modes are carpets with broad patterns, boundaries between patterns, sharp shadows, and specularities on shiny floors.

Ian Horswill s algorithm [5, 6] is similar to the above. It assumes that the floor is textureless, and labels any area whose texture is below threshold as floor. Then, moving from the bottom of the image up, it finds the first non-floor area in each column, turning left or right depending on which side has the most floor.

The system s major failure mode is braking for shafts of sunlight. In addition, it cannot break for objects it has seen previously but doesn t see now. Textureless objects with the same brightness as the floor also cause problems, as does poor illumination.

Ulrich and Nourbakhsh [27] took the floor to be the part of a previous image that had since been traversed.

Illah Nourbakhsh has used depth from focus for robot obstacle avoidance [22]. Three cameras, focused at different distances (near, middle and far), image the same scene. Whichever image is sharpest is the most in focus, so the objects are roughly at that distance. Actually, the images are divided into 40 windows (8 across and 5 down), which are treated independently, giving an 8 by 5 depth map.

In hundreds of hours of tests, the robot has avoided stair cases as well as students, often running down its batteries before a collision. However, failure modes include areas of low texture and tables at the robot s head height.

2. Experiments

All experiments were performed on the Uranus mobile robot, in the Mobile Robot Lab at Carnegie Mellon University. The robot has a three degree of freedom base with dead reckoned positioning. While forward/backwards motion and turning in place are fairly accurate (~ 1% error), sideways motion isn t (about 10-20% error, significant rotation). For sensing it uses a b/w analog video camera and a ring of 24 sonar sensors. Processing was done by an off board 700 MHz Pentium III computer running BeOS.

The work to date has taken place in a hallway whose most problematic features are glossy, textureless grey walls which often confound local depth estimation techniques such as stereo, optical flow and depth from focus.

2.1. The Evaluation of Learned Programs

One possible method of evaluating a learned program is to run it on a simulated robot in a virtual environment. The simulation could proceed faster than real time and doesn't require the constant supervision and resetting by hand that experiments with real robots often do. Many experiments with sonar and proximity sensors proceed in this way.

However, simulating CCD camera images to the fidelity required here is difficult, to say the least. Also, the creation of a simulation with noise levels and characteristics similar to those found in the real world is a time consuming and difficult task. Finally, as mentioned in the Previous Work section, certain subtle solutions may be possible in the real world, but not in simulation. Evolving in simulation may make the problem harder than need be, or just different.

In other words, simulations are necessarily different from the real world. The best evolved programs may not work in the real world, and the best program for the real world may do poorly in simulation. The task of constructing and refining a simulation to minimize these problems could prove interesting and valuable, but was not the approach chosen here.

Another option is to evaluate the algorithms by running them on the real robot. The robot could use the algorithm in question to navigate, stop when it hit something, and then travel back to where it started. When heading back, it could cheat by using a map of the space or additional sensors such as its sonar ring. Even so, this is very slow and requires a person to supervise the robot. The research groups that have attempted this use population sizes of less than 100 individuals, whereas GP typically uses sizes of 2000 to 10,000. Each evaluation on the robot takes a good fraction of a minute, and is prone to getting stuck, can t be done while its batteries are charging, can t be done at night, etc. As a rule of thumb, evaluations should take a second or less on average. This may actually be practical using a number of small, fast, reliable robots, but not with our beloved, lumbering Uranus.

For these reasons, the evolution is done off-line. Before the simulated evolution, the robot is run and vision and dead reckoning data is collected. During evolution, each evolved program is evaluated by executing it with the collected data as input, and comparing its output to a hand constructed correct answer. The output is interpreted as a distance to the nearest object, rather than a steering command directly, since this representation is more closely related to the input. The input, a set of greyscale values, may be so distantly related to the steering commands that the mapping is impossible to learn with current techniques and resources. With the success reported here, predicting the steering direction directly is an exciting next step.

2.2. System Description

The considerations of the previous section lead to the following setup. The robot is run autonomously with a

Table 1: Functions and Terminals of the Window Iteration Branch

root	`iterate-horizontal`, `iterate-vertical`
rectangle sizes	`r22`, `r23`, `r32`, `r33`, `r24`, `r42`, `r44`, `r55`, `r26`, `r62`, `r36`, `r63`, `r66`, `r77`, `r28`, `r82`, `r38`, `r83`, `r88`, `r2020`
arithmetic	`*`, `+`, `%`, `-`, `sqr` and random constants
parameters	`x-obstacle`, the horizontal pixel location in which to find the obstacle; `area`, the area of the window in pixels; `image-max-x` (319): `image-max-y` (239); `first-rect`, one if this is the first rectangle of the iteration, zero otherwise; `x` and `y`, the center of the rectangle in pixels.
flow control	`prog2`; `prog3`; `break`, with halts the execution of the branch, returning immediately without any more iterations; `if-le`
memory	`set-a` ... `set-e`, `read-a` ... `read-e`
image statistics	average and average-of-squared over the window: raw, truncated median, median corner, Sobel magnitude, and four directional Moravec interest operators.

hand-coded algorithm that uses sonar to avoid obstacles. During this online data collection run, camera images are continuously recorded. Then, during the offline learning run, simulated evolution evolves programs to estimate obstacle distance from each image. Finally, the best evolved programs are used to control the robot during online obstacle avoidance.

To collect images that are representative of what the cameras might see during that final stage, the robot collects data while avoiding obstacles under sonar. While obstacle avoidance under sonar is considered easier than under vision, it still took many attempts to get a working system. The method that proved most successful determines speed based on proximity to the nearest object, and determined direction of travel by fitting lines to points on the left and right sides of the robot. More details can be obtained from the author.

Each evolved program takes as input the image and the horizontal position of the column it must estimate. It returns a single number, the vertical position, in pixels, of the first (i.e. lowest) non-ground pixel. It s run on six different columns per image, on each of 75 images in the training set, for a total of 450 fitness cases. The fitness is the sum of the absolute differences between the returned values and ground truth. The absolute differences were capped at twenty.

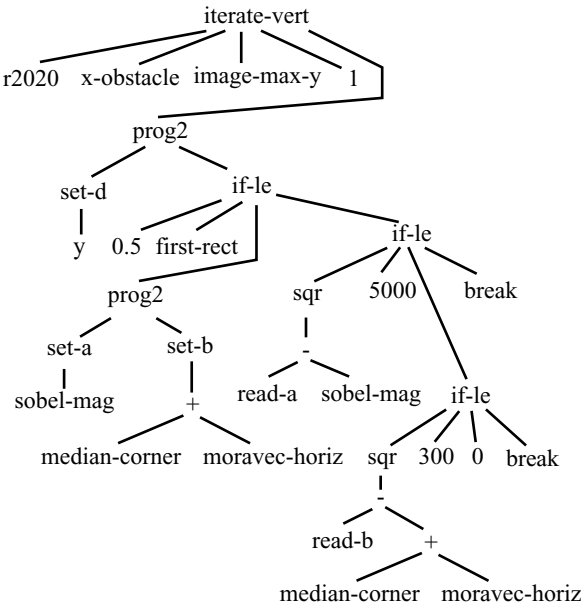


Figure 1: An example window iteration branch, used in a seed for the first generation of all runs.

The particular form of simulated evolution used here is Genetic Programming [9]. To give GP a little more structure, iterated window branches are included. These branches evaluate an evolved expression whose terminals include image statistics over a small window. The window is moved one pixel at a time, either horizontally or vertically, evaluating the expression at each location. They can read and write to five real valued registers, similar to Teller and Veloso s work with PADO [26]. The complete list of functions and terminals in the window iteration branch is shown in Table 1. An example iterated window branch, using a Lorigo style algorithm, is shown in Figure 1.

The return values of these expressions are discarded, so the register values are the only values left after execution. Each program has three window iteration branches, for a total of fifteen register values. These values are provided to the result producing branch, which must use them to estimate the location of the lowest non-ground pixel.

A Koza style tableau is shown in Table 2. The best evolved program is then run on the robot, and a hand written program converts the estimated object locations to speed and direction commands.

2.3. Experimental Setup

First, the camera was calibrated using the system described in [18]. The camera was then mounted on the robot, pitched 31 degrees down from horizontal. 75 images were collected, each 320 by 240 pixels, during a

Table 2: Koza Style Tableau

Objective	Given an image and a horizontal position within it, return the first non-ground pixel.
Architecture of individuals	Three *window iteration* branches and one result producing branch.
Terminal set for result producing branch	random constants, `x-obstacle`, `image-max-x`(319), `image-max-y` (239), `mem0a` ... `mem0e`, `mem1a` ... `mem1e`, `mem2a` ... `mem2e`
Function set for result producing branch	+, -, *, %, sqr, if-le
Fitness cases	Six columns in each of 75 images. 450 total.
Raw fitness	The sum, over the fitness cases, of the absolute value of the difference between the pixel location estimated by the evolved program and the hand created ground truth. If a difference was greater than 20, it was replaced by 20.
Standardized fitness	Same as raw fitness
Hits	The number of fitness cases for which the absolute difference was less than 2.0.
Parameters	101 generations, population size of 2000, tournament selection size 7, ramped-half-and-half with min size 6 and max size 9.

7.5 m data collection run. These images became the training set. The run was through a hallway and included two turns. Ground truth was then assigned by hand using a simple GUI. Typical images and ground truth are shown in Figure 2.

During offline learning, the images were first rectified to conform to an ideal perspective projection, and cropped to a horizontal field of view of 83 degrees using the above calibration information. The results of the operators were precomputed at every pixel, then the genetic programming run was started.

Offline learning was performed on a dual 700 MHz Pentium III, and evaluation times varied widely, but averaged approximately 725 msec. The time for a simulated evolution run also varied widely, averaging approximately 20 hours.

Figure 2: Training data, with ground truth indicated.

All genetic programming runs were seeded with the individual from Figure 1. That is, one of the 2000 individuals in generation zero has the window iteration branch show there, plus a result producing branch that simply returns the d register of that branch. This individual was designed as an example for explaining the approach and not intended to be run. The set of operators and their combination was chosen arbitrarily for their explanatory power without any thought as to how well they would work in practice. It was later used to test the system, and the thresholds were determined interactively at that time. The remaining individuals were created randomly, using the ramped-half-and-half method from [9].

After the offline learning, a hand written navigation algorithm used the estimates to decide speed and direction to travel. The best evolved algorithm was run on twelve columns in the image, twice as many as used in training. To better navigate around nearby obstacles, the camera was tilted further down, to 39.5 degrees from horizontal.

The navigation algorithm classifies estimates as either near (requiring an immediate halt), medium (slow to 2/3 speed to avoid collision) or far (avoid them before they become a problem.) This case based approach is inspired by the Property Mapping approach of Nourbakhsh [23]. If any of the middle four estimates are in the lower fifth of the image, or either of the two readings outside are at the bottom, then the object is considered near and the robot immediately halts. Otherwise, it looks for objects within four feet to the left and the right of where the robot is and where it would be if it continued straight. To convert pixel height in the image to real world distances, it assumes that the floor is flat, and that the non-ground object touches the floor. If an object is sighted, on either the left or the right, a line is drawn through the readings on each side, and the robot turns to run parallel to the lines.

If there are no objects near or in medium-sides, the algorithm looks for objects straight ahead within the far

boundary. Objects there cause it to respond by turning left or right, towards the largest gap. If all areas are clear other than far-sides, a line is fit to the side with the closest readings, and if the line is converging with the robots center line, the robot turns to move parallel. Finally, if there are no objects anywhere within the robot s field of view, it simply moves straight.

This algorithm was created by hand using traditional iterative design, and is still far from optimal. It is a natural application for simulated evolution, which is likely to do significantly better.

2.4. Results and Discussion

The seed individual performed better than any other individual in generation 0, although not by much. Of the 450 fitness cases (6 columns in each of 75 images), only two of them were within two pixels of the ground truth, the criterion for a hit. Its fitness was 14 pixels absolute error on average, where individual errors were limited to 20 pixels. By design, it fails in the many cases where the ground wasn t visible at the bottom of the column.

Twenty runs were completed. In all but one of these, the best-of-generation individual on the last generation did better than the seed. The average fitness (lower is better) of these twenty individuals was 7.43 pixels absolute error, the average number of hits (higher is better) was 111 out of a possible 450.

Interestingly, the three best best-of-run individuals had the same result producing branch as the seed, and iterated vertically in the desired column from bottom to top, just like the seed. This means the result producing branches only used one register from one of the three window iteration branches. Therefore, these best-of-run individuals have essentially the structure of Ian Horswill s and Liana Lorigo s systems.

The best individual from all runs had a standardized fitness of only 2.42 pixels absolute error per column, got 272 hits (i.e. 60.4% of estimates within 2 pixels), and had 587 nodes. After simplification (which didn t change the estimates it computes), 228 nodes remained. The rectangle size had been reduced from 20 by 20 pixels to 8 by 8. The first rectangle branch was considerably expanded, and after the first rectangle it only looked at the Sobel magnitude, i.e. the gradient.

The solution generalizes surprisingly well on the same camera in the same hallway. While its sensitive to the height of the camera, it is relatively insensitive to the pitch and the horizontal location of the column in the image. With the camera set to automatic gain it also provides acceptable results over a wide range of iris settings. It detects objects that weren t present during training, such as chairs or people, with about as much fidelity as it detects walls. It s also fast. Even without precomputing the image operators, the individual runs at about 10 Hz on a 333 MHz Pentium II. However, it s relatively sensitive to centimeter long pieces of metal or other small, shiny objects on the floor that produce high gradients.

With the camera fixed in one place, the algorithm produces occasional glitches, most often declaring that a pixel at the bottom of the image is non-ground when it is, in fact, ground. To stop these from causing too many panic halts, the hand written navigation algorithm filters readings by taking the minimum (highest pixel location) of consecutive estimates.

When navigating under sonar, fourteen sonar sensors for a total field of view of approximately 215 degrees, seeing well to the sides. While pitching the camera down increases awareness to the front left and front right of the robot, the area of awareness is still much smaller than with sonar, and entirely in front of the robot s base. In the reactive framework described here this makes it almost impossible to successfully navigate doorways, especially since the robot is only a few inches narrower than them. However, it performed very well at corridor following and avoiding obstacles such as people and chairs. The next revision of the system will include state, in order to ease these problems.

3. Bibliography

[1] S. Baluja, Evolution of an Artificial Neural Network Based Autonomous Land Vehicle Controller. IEEE Transactions on Systems, Man and Cybernetics Part B: Cybernetics. 26, 3, 450-463. (1996)

[2] D. Cliff, P. Husbands and I. Harvey. Evolving Visually Guided Robots. In Proceedings of SAB92, the Second International Conference on the Simulation of Adaptive Behaviour. MIT Press, 1993.

[3] D. Coombs, M. Herman, T. Hong and M. Nashman Real-time Obstacle Avoidance using Central Flow Divergence and Peripheral Flow. Fifth International Conference on Computer Vision June 1995, pp. 276-83.

[4] D. Floreano and F. Mondada. Automatic Creation of an Autonomous Agent: Genetic Evolution of a Neural-Network Driven Robot. Proceedings of the Third International Conference on Simulation of Adaptive Behavior: From Animals to Animats 3. 1994.

[5] I. Horswill, Specialization of Perceptual Processes. Ph.D. Thesis, Massachusetts Institute of Technology, May 1993.

[6] I. Horswill, Polly: A Vision-Based Artificial Agent. Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI-93), July 11-15, 1993.

[7] P. Husbands and J.-A. Meyer (Eds.), Evolutionary Robotics, Proceedings, First Euopean Workshop, EvoRobot98, Paris, France, April 1998.

[8] N. Jakobi, P. Husbands and I. Harvey. Noise and the reality gap: The use of simulation in evolutionary robotics. In Advances in Artificial Life: Proceedings of the Third European Conference on Artificial Life, 1995.

[9] J. Koza, Genetic Programming, MIT Press, Cambridge, MA. 1992.

[10] E. Krotkov, M. Hebert & R. Simmons, Stereo perception and dead reckoning for a prototype lunar rover. Autonomous Robots 2(4) Dec 1995, pp. 313-331

[11] L.M. Lorigo Visually-guided obstacle avoidance in unstructured environments. MIT AI Laboratory Masters Thesis. February 1996.

[12] L.M. Lorigo, R.A. Brooks, and W.E.L. Grimson Visually-guided obstacle avoidance in unstructured environments. IEEE Conference on Intelligent Robots and Systems September 1997.

[13] L.H. Matthies, Stereo vision for planetary rovers: stochastic modeling to near real-time implementation. International Journal of Computer Vision, 8(1): 71-91, July 1992.

[14] L.H. Matthies, A. Kelly, & T. Litwin Obstacle Detection for Unmanned Ground Vehicles: A Progress Report. 1995.

[15] L.H. Matthies, Personal communication.

[16] L.A. Meeden, An Incremental Approach to Developing Intelligent Neural Network Controllers for Robots. IEEE Transactions of Systems, Man and Cybernetics Part B: Cybernetics. 26, 3, 474-485. (1996)

[17] O. Miglino, H. H. Lund and S. Nolfi, Evolving Mobile Robots in Simulated and Real Environments. Artificial Life , 2, 417-434 (1995).

[18] H.P. Moravec, DARPA MARS program research progress, http://www.frc.ri.cmu.edu/~hpm/project.archive/robot.papers/2000/ARPA.MARS.reports.00/Report.0001.html, Januray 2000.

[19] T. Naito, R. Odagiri, Y. Matsunaga, M. Tanifuji and K. Murase, Genetic Evolution of a Logic Circuit Which Controls an Autonomous Mobile Robot. Evolvable Systems: From Biology to Hardware. 1997.

[20] S. Nolfi and D. Floreano, Evolutionary Robotics. MIT Press / Bradford Books. 2000.

[21] P. Nordin, W. Banzhaf and M. Brameier, Evolution of a World Model for a Miniature Robot using Genetic Programming. Robotics and Autonomous Systems, 25, pp. 105-116. 1998.

[22] I. Nourbakhsh, A Sighted Robot: Can we ever build a robot that really doesn t hit (or fall into) obstacles? The Robotics Practitioner, Spring 1996, pp. 11-14.

[23] I. Nourbakhsh, Property Mapping: A simple technique for mobile robot programming. In Proceedings of AAAI 2000.

[24] C. W. Reynolds, Evolution of Obstacle Avoidance Behavior: Using Nosie to Promote Robust Solutions. In Advances in Genetic Programming, MIT Press, pp. 221-242. 1994.

[25] T. M. C. Smith, Blurred Vision: Simulation-Reality Transfer of a Visually Guided Robot. In Evolutionary Robotics, Proceedings of the First Euopean Workshop, EvoRobot98, Paris, France, April 1998.

[26] A. Teller and M. Veloso, PADO: A new learning architecture for object recognition. In Symbolic Visual Learning, Oxford Press, pp. 77-112. 1997.

[27] I. Ulrich and I. Nourbakhsh, Appearance-Based Obstacle Detection with Monocular Color Vision. In Proceedings of AAAI 2000.

The Incremental Evolution of Gaits for Hexapod Robots

Gary B. Parker

Computer Science
Connecticut College
New London, CT 06320
parker@conncoll.edu

Abstract

Gait control programs for hexapod robots are learned by incremental evolution. The first increment is used to learn the activations required to generate a single leg cycle. At this level the control program is required to produce the proper sequence of pulses needed to generate smooth movement by the servos. The learning program needs to take into account the peculiarities of the servo, its mounting and the capabilities of the leg. The second increment of the learning process is used to learn the best combination of individual leg cycles to produce a gait. This part requires the learning system to choose the best leg cycles for each leg and to coordinate their movement. In this paper, we describe an application of this method to learn gaits for an actual hexapod robot. A cyclic genetic algorithm is used to learn efficient gait cycles for each leg. A genetic algorithm is used to combine these leg cycles in such a way that coordinated gaits result. Tests are conducted on the actual robot to confirm the method's viability.

1 INTRODUCTION

The generation of gaits is important for the effective use of hexapod robots. Proper gaits are needed to ensure that the robot moves quickly and efficiently. Gaits need to be custom designed specifically for the individual robot to make the best use of its capabilities. There are two main parts to gait generation; the cyclic action of the individual legs and the coordination of all the legs to make effective use of their cycles. These can be learned together by finding the sequence of concurrent movements required by all the actuators as was done in previous work [3,4,5]. Or, they can be learned separately. Learning together greatly increases the complexity so details are often lost in the abstraction necessary to keep the computations within reason. This method can produce reasonable gaits that can operate on simpler controllers, but since some detail is lost, they cannot fully exploit the capabilities of the robot. Learning the leg cycles separate from their coordination allows the system to better use each leg as long as the controllers are complex enough to handle the increased details.

Individual leg learning can take into account the capabilities of the actuators and movement constraints of individual legs. In our work, we use a robot that has servos for actuators. These servos require a pulse to designate their desired position. The pulse length for each position is distinct for each servo and is dependent on its placement during installation. A single pulse does not guarantee proper positioning since it may be asking the servo to move further then it can in the time between pulses. A sequence of pulses with small changes in pulse length is required to get rate control. This sequence of control signals needs to be repeated to get the cycle of activations required to produce a cycle of movement for the leg.

Since evolutionary computation (EC) is well suited for adapting a solution to the peculiarities of a problem, some form of EC would work well in learning what signals are needed for the leg cycle. The difficulty comes in that most forms of evolutionary computation are not naturally equipped to handle the cyclic nature of these leg cycles. One exception is with genetic programming, which can be used to evolve programs and programs can have loops. Graham Spencer [6] had some success in generating programs for hexapod gaits using genetic programming. His programs worked concurrently on all the actuators to produce gaits for hexapod robots. His programs, tested only on robot simulations, resulted in gaits that maintained sustained forward movement but could not obtain the optimal tripod gait.

Randall Beer and John Gallagher [1,2] used genetic algorithms (GAs) to develop neural network controllers for a simulated hexapod robot. In this work, the structure of the neural networks (NNs) was pre-defined and the GA learned the weights required to generate gaits. This work makes more of a division between the leg cycles and the coordination of legs. NN structures are separately defined for the leg cycles and the coordinators, but the weights are learned concurrently. The nature of the leg cycles are somewhat defined by the structure of the NNs and further

learned by the GA. The coordination is controlled by central NNs with defined structures; the weights of these are learned at the same time as the leg NN weights.

In previous work [3,5], we used Cyclic Genetic Algorithms (CGAs) to generate the sequence of primitive instructions that produced a gait. CGAs were developed to allow for the representation of a cycle of actions in the chromosome. They differ from the standard GA in that the chromosome is in the form of a circle with two tails. The tails of the CGA chromosome are provided to allow for pre and post-cycle procedures. They provide a means for completing tasks before and after entering the cycle. For gait sequence generation, the pre-cycle can position the legs in a ready to walk posture and the post-cycle can return the robot to a stable at rest posture. In our application, we used only the pre-cycle tail. The CGA genes can be one of several possibilities. They can be as simple as normal genes that represent traits of the individual or they can be as complicated as cyclic sub-chromosomes that can be trained separately by a CGA. For our purposes, the genes represent tasks that are to be completed in a set amount of time. The trained chromosome will contain the cycle of servo control pulses that will be continually repeated by the leg's controller to produce a leg cycle.

Tests showed that CGAs could produce tripod gaits on robot simulations that were transferable to an actual autonomous hexapod robot [4]. This was accomplished by creating a model with specific information taken from an individual robot. The CGA used this model to develop an optimal gait that was specific to the robot's capabilities. This gait was subsequently downloaded into the actual robot where its performance was confirmed to correspond to the performance of the model. The primitive instructions used in these experiments were not designed to take advantage of the full capabilities of the servo motor actuators. Each servo had 2 possible states; either full forward or full back for horizontal servos or full up or full down for vertical servos. Each servo was given a control pulse that would drive it to the extreme. This was necessary to accommodate the limited capabilities of the single BASIC Stamp II controller.

In this paper, we use incremental evolution to learn control for 7 controllers. One controller is used to coordinate the other 6, which are each used to control a leg. These 6 additional controllers allow the system to take advantage of the capabilities of the servos. Each leg controller controls that leg's vertical and horizontal servo. Cycles of pulses are learned using a CGA that produces individual leg cycles optimizing for time on the ground and forward movement. These individual leg cycles are then combined, using a standard genetic algorithm to produce gaits for the robot. Tests in simulation and on the actual robot confirm the viability of this method for producing gaits.

2 THE ROBOT

The robot used was the ServoBot, which is a hexapod robot that has two degrees of freedom per leg. Twelve servos, two per leg, provide thrust and vertical movement. They can be set to specific angular positions by providing a control pulse. This pulse should be repeated every 25 ms for the servo to maintain a constant position. The length of the pulse determines the position. Pulses from 20 to 2400 microseconds cover the full range of movement for each leg, although each servo is unique in its pulse to position ratios. Some may have a full down position at 20, on others it may be 80. There is the same variance in the full up position. In addition, the right and left side servos are mounted differently to ensure consistent mechanical capabilities, so in some cases the full down position is at a pulse length of 20 and in some cases it's at 2400.

The servo cannot move the leg fast enough to reach the desired position within one pulse if the differences in pulses are too much. This results in the fastest leg movement as the servo attempts to get to its desired position as soon as possible. Varying speeds of movement can be attained by incrementally changing the pulse lengths. For example, moving a leg using consecutive pulse lengths of 40, 45, 50, etc. will move the leg at a slower speed than 40, 50, 60, etc., unless, of course, the increments are already more than the servo's capability. Consecutive pulses of 40, 240, 440, etc. would probably result in the same speed as the consecutive pulses of 40, 340, 640, etc.

Control was provided by BASIC Stamp IIs, one per leg and one working as the overall controller. Each leg's stamp could take in a sequence of pulses that indicated the position of its two servos. The central stamp controller told each leg stamp when to start its sequence and if needed, when to cut short one cycle to start another in order to maintain leg coordination.

3 THE FIRST INCREMENT: EVOLVING LEG CYCLES

In order to produce leg cycles, each stamp needs a sequence of pulses to continually position its servos. This sequence must by variable in length to accommodate the differing capabilities of each leg and its servos. Fixed length chromosomes offer distinct advantages when using CGAs since like areas of each chromosome are more likely to correspond to similar tasks. In order to formulate the problem in such a way as to be able to use a fixed length chromosome, some observations of a leg cycle had to be made. Pulses within 20 microseconds of each other result in positions that are only slightly distinguishable from each other (usually within 1 mm). This level of position accuracy is sufficient for our problem, so we can represent all pulses from 0 to 2400 by the numbers 0 to 120 considering each to be in increments

of 20 microsecond pulses. This allows us to use a seven bit number to represent each pulse. It takes 14 bits to represent pulses for both servos.

Smooth movement is required by the horizontal servo, especially while on the ground. A sequence of pulses such as 100, 120, 140, 160 would move the leg smoothly from the position corresponding to 100 to the position corresponding to 160. The sequence 100, 110, 150, 160 would result in the same final position, but the movement would not be as smooth. The chromosome representation needed to be such that smooth movement would be possible for horizontal movement, but was not needed for vertical movement since vertical movement does not affect the smoothness of the robot's movement.

$$((R_1\ HP_1\ VP_1)\ (R_2\ HP_2\ VP_2)\ (R_3\ HP_3\ VP_3)\ (R_4\ HP_4\ VP_4)\ (R_5\ HP_5\ VP_5)\ \ldots\ (R_8\ HP_8\ VP_8)\ )$$

Figure 1: Leg cycle chromosome. Each gene of the chromosome was made up of three parts: repetitions, horizontal pulse, and vertical pulse.

In order to accommodate these considerations, the chromosome representation shown in Figure 1 was used. The chromosome was made up of 8 genes. Each gene consisted of 3 parts. The first was called the *repetitions*, the second was the *horizontal pulse*, and the third was the *vertical pulse*. The *horizontal pulse* and *vertical pulse* numbers were each multiplied by 20 microseconds to calculate the actual pulse width sent to the servo. The effect of the *repetitions* was different on the two types of pulse. For the *horizontal pulse* the repetitions number was used to calculate the increments required to move from the servo's last pulse length to the new pulse length. The following formula was used:

pulse increment = (horizontal pulse - previous horizontal pulse) / repetitions

This *pulse increment* was then added for *repetitions* number of consecutive pulses until the end servo pulse was at *horizontal pulse*. For example, if the *previous horizontal pulse* was 40 and the gene was (5, 60, 100) then the following pulses would be sent to the horizontal servo over the next 5 inputs : 44, 48, 52, 56, 60. *Repetitions* effected the *vertical pulses* only by telling the controller how many times to repeat this *vertical pulse*. The extra computation was not required since smoothness was only a factor for horizontal movement.

The contents of the chromosome representation were used directly by the BASIC Stamp II and upon execution it would do the calculations required to direct its two servos. An example of the resultant sequence of pulses that would be produced is shown for a shortened chromosome in Figure 2.

Genes	Horizontal Pulses	Vertical Pulses
(4 25 127)	100	127
	75	127
	50	127
	25	127
(2 55 43)	40	43
	55	43
(5 125 38)	69	38
	83	38
	97	38
	111	38
	125	38

Figure 2: Sequence of pulses resulting from example genes. The 125 from the last gene is used to calculate the increments (-25) from 125 to 25. The first of this is added to calculate the first pulse. As can be observed, the pulse of 125 is again reached and the cycle continues with smooth horizontal movements.

3.1 LEG MODEL

Each leg was represented by a simple data structure that held the information required to produce a leg cycle. Each servo's maximum throw positions were stored as x, y, coordinates. The horizontal servo's full forward position was defined as x = 0, the full back position was x = the measured number of millimeters distance from the full forward. The vertical servo had a y = 0 if it rested on the ground when all the legs were full down and the max up was y = the millimeters off the ground when the leg was fully lifted. Along with these positions the pulse required to attain each was recorded. The model data structure also included a lookup table for each servo. This table listed the corresponding leg position of 13 different pulse lengths (1,200, 400,…2400). These figures were attained by applying consistent pulses to each servo and measuring the leg's response. The final data kept in the model was the current position and pulse of each servo.

3.2 TRAINING

Evolution of a leg cycle started by taking accurate measurements of the leg's capabilities. This information was fed into the model data structure used for training. A population of 64 chromosomes (each representing a leg cycle) was randomly generated and trained for 500 generations on the model of the robot. Fitness was calculated using three factors: forward movement, down count, and smoothness. Forward movement was calculated by determining the movement generated while the leg was on the ground. To attain the maximum forward movement, the leg should be on the ground throughout the length of its effective throw. The effective throw is usually less than the full throw. As the leg reaches its extremes of movement, the distance moved per pulse reduces significantly, so in the optimal solution the leg is repositioned before it reaches its full extreme. Another facet of the leg fitness is the down count. This factor gives more fitness to leg cycles where the leg is on the ground for a high proportion of the time. A third contributor to fitness is smoothness. This is calculated for movement on the ground. Leg cycles where the horizontal movement over the ground is consistent score higher smoothness. These three fitness indicators were added together to get the total fitness. The fitness for each chromosome was used to stochastically select individuals to produce each new population.

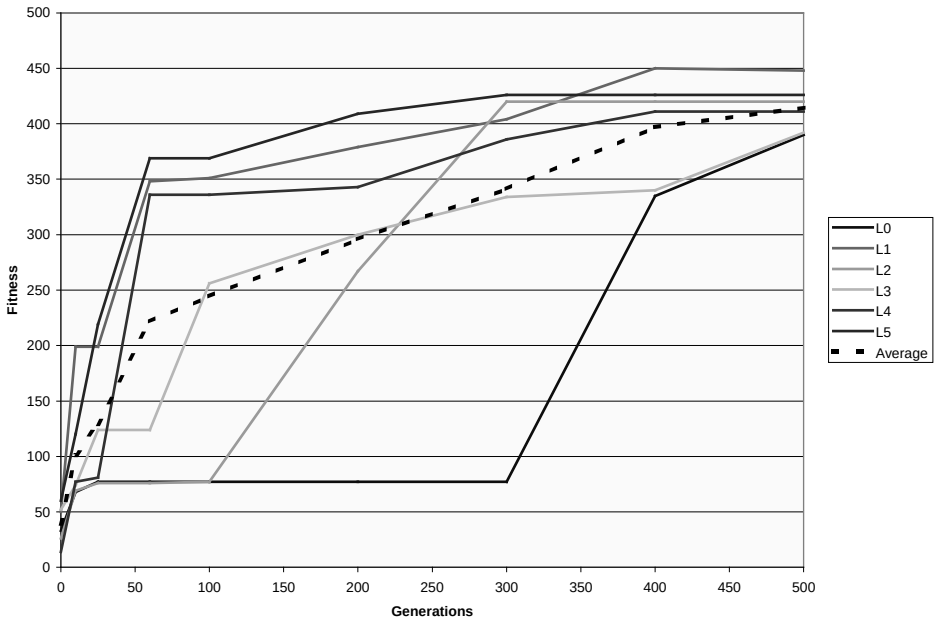


Figure 3: Single leg training for each leg.

Crossover was accomplished by randomly picking corresponding spots in the two selected parents. In the pre-cycle tail, a single point in both chromosomes was picked. In the cyclic section, since it could be considered a circle, crossover was performed at two points. The effect was to swap sections within the circle. An alternate type of crossover was a gene-by-gene crossover that performs crossover in each of the corresponding genes of the two chromosomes. Crosses could happen between the individual members of the list or within the bits of the specific numbers in the list. There were two types of mutation used and selected randomly after each recombination. In one, each gene had a random chance of

being replaced by a new completely random gene. In the other, each part of the gene had a random chance of having one of its bits flipped.

Gene-by-Gene Evaluation, a genetic operator peculiar to CGAs, was used to clean up the chromosome by randomly picking one or two individuals from the population on each set of trails and examining each gene one at a time. Genes were evaluated move-by-move by comparing the previous move fitness to the present. Genes that performed poorly in their current position were eliminated. Genes that were good in the execution of their early repetitions and subsequently dropped in the later repetitions were modified by reducing their repetitions. Genes that had zero repetitions were moved out so that only active genes were at the start of the cyclic section.

3.3 RESULTS

Training was done for 500 generations with the fittest individual chromosome saved at 0, 10, 25, 60, 100, 200, 300, 400, and 500 generations. The results of this training, done for each leg, is shown in Figure 3. Both the optimal length (number of pulses in the cycle) and content of the cycle had to be learned. Each solid line represents a leg. The dashed line is the average. Three of the 6 legs learned quickly. One of the legs was stuck for some time, with a suboptimal length, which precluded it from further growth until it evolved to a different length. At this time it also improved rapidly. The optimal lengths found for the six legs varied from 29 to 36 pulses per cycle.

Training was repeated, in preparation for the gait training discussed in the next section, but this time an additional fitness calculator was used. *Desired length* reduced the fitness if the chromosome's length was different than a predesignated desired length. A second test was performed using 5 randomly generated populations, but this time the *desired length* factor was included. The length used was 36 pulses, which was the maximum optimal length found in the previous test. The results of this test are shown in Figure 4. With pressure to conform to a specified near optimal length, all six legs grew quickly in their fitness.

The resultant leg cycles were downloaded and observed on the actual robot where they appeared to produce efficient, useable leg cycles. No mechanism was constructed to test individual legs, so actual quantitative tests were not possible until the individual leg cycles were used together to form a gait.

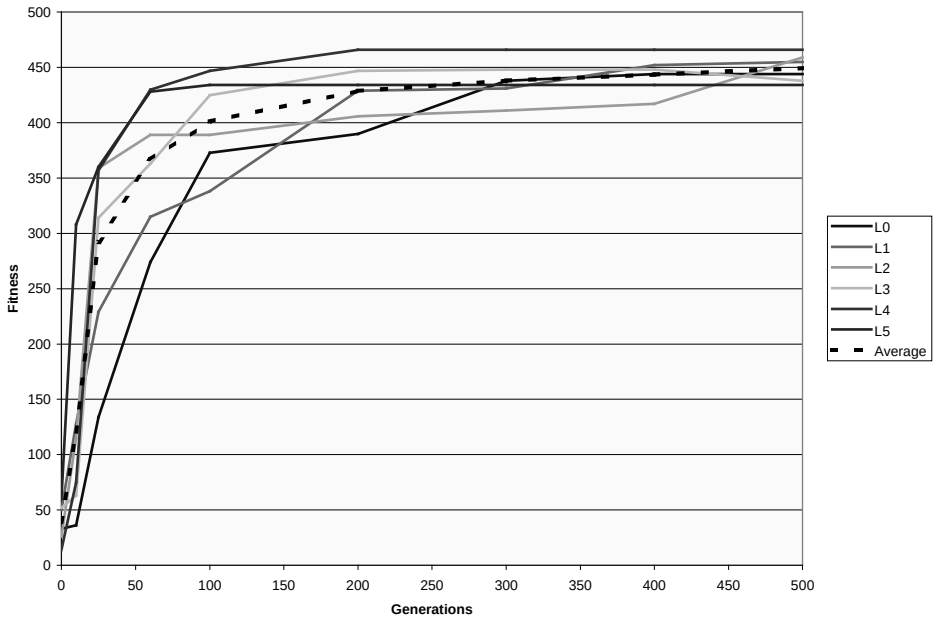


Figure 4: Single leg training with a desired length specified.

4 THE SECOND INCREMENT: EVOLVING GAITS FROM THE LEG CYCLES

A hexapod gait can be looked at as the coordination of 6 legs with each leg performing its own cycle. The proper combination of the six correct leg cycles will produce the desired gait. In addition to finding the best gait for the ServoBot using optimal leg cycles, this task also shows how to combine several cycles to form a single cyclic behavior.

4.1 EVOLVING FIXED LENGTH LEG CYCLES

Section 3 discussed the evolution of gait cycles that were optimal when they were at a desired length of 36. This length was chosen since it was the longest of the 6 no-length-restriction optimal leg cycles found previously. A set of leg cycles using a range of desired lengths would be needed to produce a gait. The gait learning algorithm would be able to choose leg cycles from anywhere in this range for each leg to come up with the proper coordination of legs. The no-length optimal should be in the center of this range but there was more likelihood that longer length leg cycles would be of use in further experimentation so the longest gait cycle length found in the no-length tests was chosen to be the middle desire length.

Each leg trained for 500 generations to learn optimal leg cycle with a desired length of 36. This population was then used to learn gait cycles with desired lengths from 21 to 52. Starting with the 36 length population, the desired length was changed to 35 and training continued for 200 generations. This continued down to a desired length of 21. Similarly, training up to 52 was done starting from the 36 length population. These learned leg cycles were stored in 6 files, which were called up when gait training began.

Gait training was done using a standard GA. The chromosome (Figure 5) was made up of 7 parts. The gait cycle length (GCL) represented the number of pulses in each gait cycle. Information for each leg included its leg cycle length (LCL) and start time (START). Each of these values is described in the next section.

```
(GCL
  (LCL START)
  (LCL START)
  (LCL START)
  (LCL START)
  (LCL START)
  (LCL START))
```

Figure 5: Chromosome used for gait training.

4.2 ROBOT MODEL

The single stamp that acted as the central controller was to coordinate the individual leg cycles. It needed to know the length (in pulses) of the gait cycle and which leg cycles to use for each leg. In addition, it needed the start time for each gait cycle. This was where the coordination took place. Upon execution the controller program would count through the total number of pulses 0, 1, 2, 3…. When the start time for each leg was researched, its leg cycle began. The central controller ensured that all the stamps executed their pulses together. When the gait cycle length was reached, the count started again at 0. When each leg's start number was reached they begin their cycle again. To simulate the effect of this on the robot, each of the leg cycles was run separately for the number of designated pulses used for training (500 in this case). They were then considered to be running simultaneously in a simulator that would determine at each pulse what the result of the 6 leg pulses would be.

4.3 TRAINING

A population of 64 randomly generated chromosomes was produced to start training, which was done for 500 generations. Each individual's fitness was calculated by determining the effect of the 6 leg-cycles running simultaneously as specified by the gait cycle chromosome. In addition to calculating the fitness produced by the legs, additional factors such as balance and drag were introduced. Balance was a determination of the robot's stability. Drag was used to penalize the fitness of the gait when the legs were on the ground but not producing thrust. Using these fitnesses, individuals were stochastically selected to be the parents of the next generation. Crossover and mutation were done both at the gene level and at the bit level as described in section 3.2.

4.4 RESULTS

The best individual at 0, 10, 25, 60, 100, 200, 300, 400, & 500 generations was stored. The results on the robot model are shown in Figure 6. Graphs of the fitness growth of the 5 distinct starting populations along with their average (dashed line) is shown. There are three things to note from this graph. The start fitness at generation 0, in most cases, is fairly high. This is because all the legs are already moving in a near optimal cycle; they just need to be coordinated. The GA quickly learns adequate coordination by 100 generations. After that, the GA works to improve this solution to find the optimal leg cycle lengths and start spots for each leg. In all 5 cases, near optimal tripod gaits are produced.

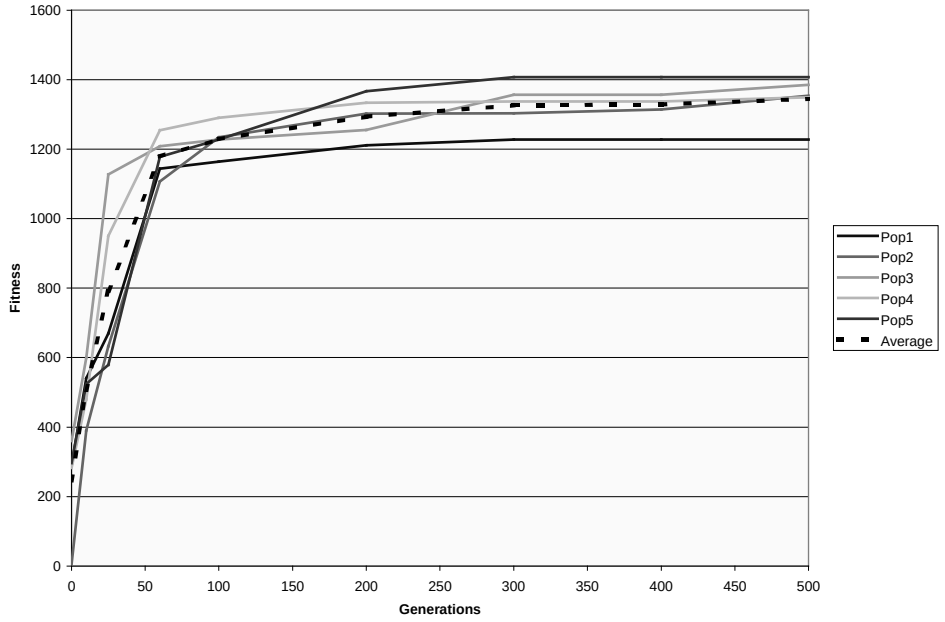


Figure 6: The results on a model of GA training to coordinate the legs.

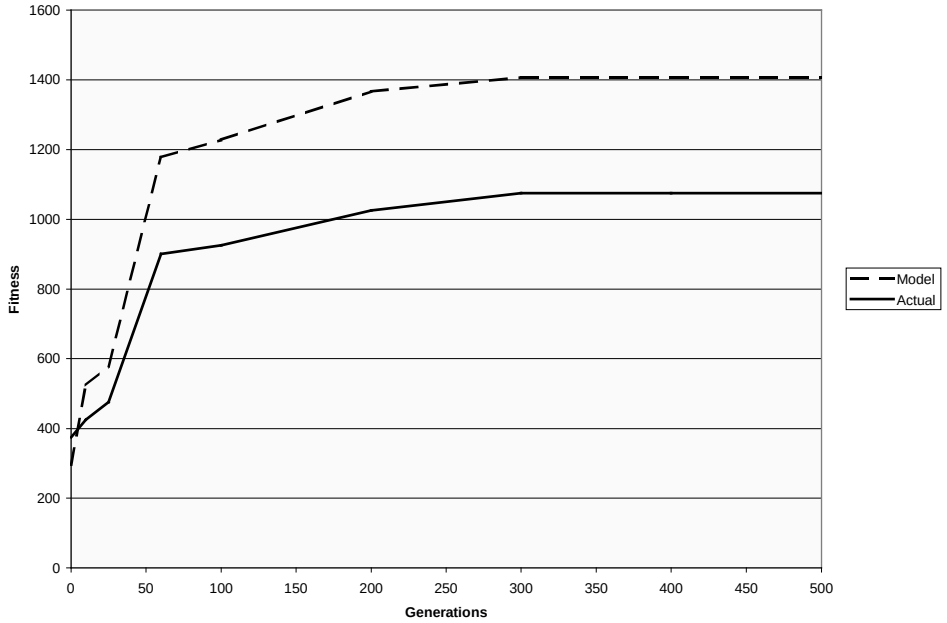


Figure 7: Comparison of the model to actual robot performance for a single population over the 500 generations of training.

Tests on the actual robot confirmed the viability of the produced gaits. Figure 7 shows the results of actual tests on the robot using the 0, 10, 25, 60, 100, 200, 300, 400, and 500 generations of one of the populations. As can be observed, the system consistently overestimates the fitness in the higher ranges. The model is purposely simple to reduce computation time and does not take into account lost speed due to slippage and actuators moving slower due to resistance. Both of these factors have more of a negative effect at high speeds. In addition, the model does not compensate sufficiently for the weight of the robot. Observations of actual tests on the robot show

that the legs need to be lifted higher to avoid some drag during leg repositioning. The gait is fast enough, however, that there is minimal time with only three legs on the ground. The result is a steady forward movement with little time wasted. Figure 8 shows a comparison of the end products of the five trials. In all cases the gait produced was a fast tripod.

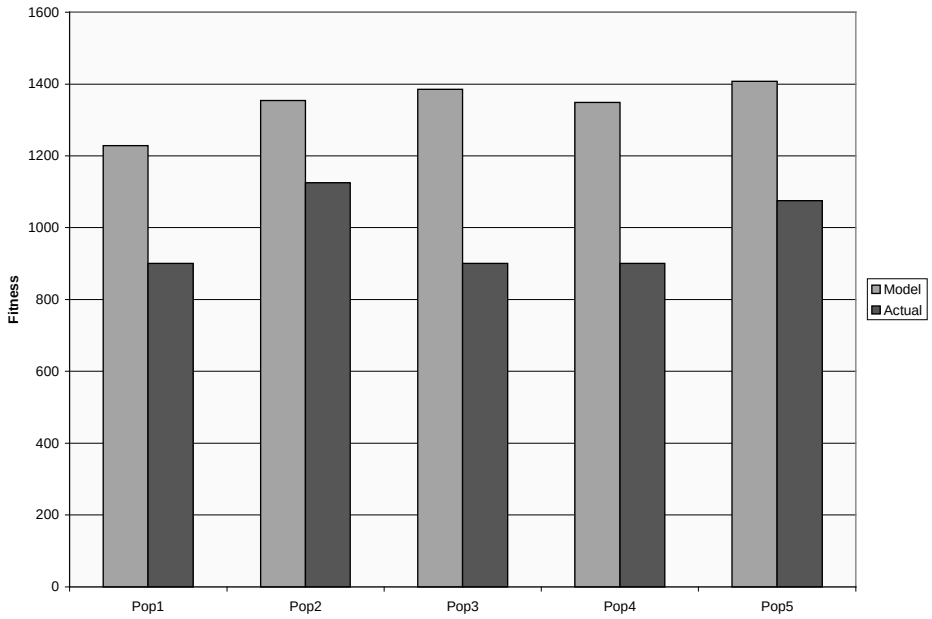


Figure 8: Comparison of the model/actual fitness for all five populations after training is complete.

5 CONCLUSIONS

Incremental evolution is an effective means of evolving gaits for hexapod robots. In the first increment CGAs can be used to generate the cycles of pulses required to produce a leg cycle for a two servo leg. Tests in simulation showed that they improve performance significantly over training and observation of the results on an actual robot confirmed the viability of the produced cycles. In the second increment, these leg cycles can be combined in such a way that their concurrent execution can produce a gait. Using a GA to coordinate the 6 leg cycles, with fitness predicated on maximum forward movement, the leg cycles can be combined to form a near-optimal gait cycle. Tests in the simulation and the actual robot confirm the viability of this method.

References

1. Beer, R. D., and Gallagher, J. C. (1992). "Evolving Dynamical Neural Networks for Adaptive Behavior." Adaptive Behavior, 1 (pp. 91-122). Cambridge: MIT Press.

2. Gallagher, J. C. and Beer, R. D. (1994). "Application of Evolved Locomotion Controllers to a Hexapod Robot." Technical Report CES-94-7, Department of Computer Engineering and Science, Case Western Reserve University.

3. Parker, G. and Rawlins, G. (1996). "Cyclic Genetic Algorithms for the Locomotion of Hexapod Robots." Proceedings of the World Automation Congress (WAC'96), Volume 3, Robotic and Manufacturing Systems. (pp. 617-622).

4. Parker, G., Braun, D., and Cyliax I. (1997). "Evolving Hexapod Gaits Using a Cyclic Genetic Algorithm." Proceedings of the IASTED International Conference on Artificial Intelligence and Soft Computing (ASC'97). (pp. 141-144).

5. Parker, G. (1998). "Generating Arachnid Robot Gaits with Cyclic Genetic Algorithms." Genetic Programming 1998: Proceedings of the Third Annual Conference. (pp. 576-583).

6. Spencer, G. (1994). "Automatic Generation of Programs for Crawling and Walking." Advances in Genetic Programming. (pp. 335-353) K. Kinnear, Jr. (ed.), Cambridge, Ma: MIT Press.

Evolution for Behavior Selection Accelerated by Activation/Termination Constraints

Eiji Uchibe **Masakazu Yanase** **Minoru Asada**
Dept. of Adaptive Machine Systems
Graduate School of Engineering
Osaka University

Abstract

This paper proposes an evolutionary approach to select appropriate behaviors for a mobile robot. We augment each behavior by adding activation/termination constraints to accelerate the evolutionary processes. The former constraints reduce the number of situations where each behavior is executable, and the latter contribute to extract meaningful behavior sequences, each of which can be regarded as one action regardless of its length. We apply the genetic algorithm to obtain the switching function to select the appropriate behavior according to the situation. As an example, a shooting task in a soccer game is given to show the validity of the proposed method. Based on the combination of the proposed architecture and GA, we can obtain the purposive behaviors. Simulation results are shown, and a discussion is given.

1 Introduction

Machine learning techniques such as reinforcement learning [8] and genetic algorithm [4] are promising to obtain purposive behaviors for autonomous robots in complicated environments. Many learning and evolutionary techniques can obtain purposive behaviors such as wall-following [2, 6], shooting a ball into the goal [1], and so on. However, if the robot has no *a priori* knowledge to obtain the complicated behaviors, it takes enormous time. Consequently, the resultant behavior seems trivial in spite of the long learning time. That is, a direct mapping from sensory inputs to motor commands is not tractable.

In order to obtain the feasible solution in the realistic learning time, a layer architecture is often introduced to cope with large scaled problems [7]. Figure 1 shows an example of layered architecture for behavior selection. In this approach, the upper layer learns the switching function to select the suitable behavior already designed or learned. Because the designed behaviors can generate purposive action under the limited situations, they can help the evolutionary computation to search the feasible solutions.

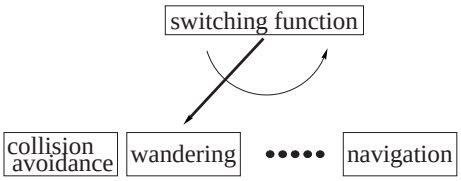
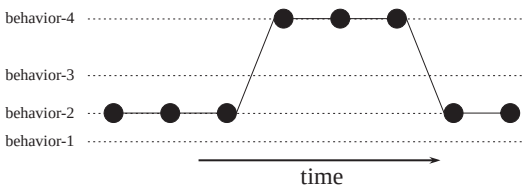


Figure 1: A layer architecture for behavior selection

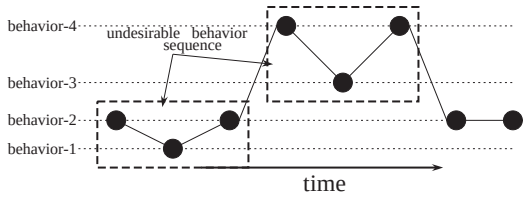
In this approach, we face with the following problems:

1. how to coordinate and switch the behaviors,
2. when to select the behaviors, and
3. when to terminate the currently executing behavior.

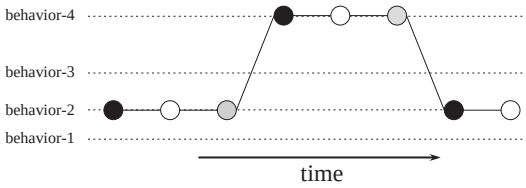
Uchibe *et al.* [9] applied genetic programming to solve the above three problems in the robotic soccer domain. However, the resultant decision tree is not represented in a compact style. In their case, the robot selected the collision avoidance although there were no obstacles near the robot. In addition, the robot did not use the given shooting behavior when it is suitable to be activated. There are two major reasons why a layered approach could not obtain the appropriate behavior sequences: (1) GP does not take account of the precondition of the given behavior explicitly, and (2) the behavior is often switched although the goal of the behavior is not achieved. The first reason prevents GP



(a) ideal behavior sequence



(b) resultant behavior sequence



(c) behavior sequence using a termination constraint

Figure 2: A timing to select the next behavior

from reducing the learning time, and the second causes the scraps of behavior sequence.

An example sequence of the selected behaviors is shown in Figure 2, where the black circle indicates the timing to change the behavior to another one. Figure 2 (a) indicates the ideal behavior sequence, whereas Figure 2 (b) the resultant behavior sequence often obtained by the learning or evolutionary approaches. In the case (a), the robot selects the same behavior for a while. On the other hand, in the case (b), the robot often switches the behaviors according to the situation. Although this can be regarded as an acquisition of the new behavior sequences instead of the given behavior, it causes enormous learning time because the robot does not make good use of the behavior given by the designer.

In order to take advantage of the behavior given by the designer, we have to consider the the precondition and the goal of the behavior. This paper proposes a behavior selection mechanism with *activation/termination constraints*. The former constraints reduce the number of situations where each behavior is executable, and the latter contribute to extract meaningful behavior sequences. We call behavior with activation/termination constraints *module*. These constraints enable the robot to modify the timing to select the next behavior as shown in Figure 2 (c), where the gray circle indicate a vague state whether a new behavior is changed or not. From the case (c), the termination constraint contributes to avoiding frequent switching behavior. It enables us to deal with heterogeneous modules[1] in the same manner. Once the module is selected, actions are executed until the module terminates stochastically based on the termination constraint. Thus, we can obtain the behavior sequence like Figure 2 (c).

The lower layer consists of multiple modules, while the upper layer selects the appropriate module according to the situation. Genetic algorithm is applied to obtain the switching function to select the appropriate module and the timing to terminate it according to the situation. Activation/termination constraints affect not the genetic operations such as crossover and mutation but the individual representation. Although we utilize standard genetic operations, we can obtain purposive behaviors owing to the activation/termination constraints within a reasonable computational time. The results of computer simulation are shown, and a discussion is given.

2 Behavior Selection with Activation/Termination Constraints

2.1 Lower layer

Suppose that the robot has L modules m_i ($i = 1, \cdots, L$). A module m_i consists of three components: a *behavior* $\pi_i : \boldsymbol{X}$ (state space) $\rightarrow \boldsymbol{U}$ (action space), an *activation constraint* and I_i a *termination constraint* T_i. There are several ways to implement the behaviors, but they must be given to the robot in advance.

The activation constraint I_i gives a set of states where the module should be executable.

$$I_i(\boldsymbol{x}) = \begin{cases} 1 & m_i \text{ is executable at state } \boldsymbol{x}, \\ 0 & \text{otherwise.} \end{cases} \quad (1)$$

If the designer gives the behavior π to the robot, it is not difficult to give the precondition of the behavior.

[1] In this paper, "heterogeneous" means the differences of time to achieve the goal of the module.

For example, the collision avoidance behavior is implemented for the mobile robot with sonars, where the designed behavior is activated only when the obstacles are detected by sonar.

Each module has one termination constraint T_i consisting of a probability to sustain the module or not. In other words, this function gives the time to continue to execute the selected module.

$$T_i(\boldsymbol{x}, t) = \begin{cases} 0 & \text{the goal of the module} \\ & m_i \text{ is achieved,} \\ 0 & t > t_{p,i}, \\ p_i & \text{otherwise,} \end{cases} \quad (2)$$

where t and $t_{p,i}$ denote the elapsed steps and the pre-specified time interval, respectively. If the robot continues to execute the same module $t_{p,i}$ steps, it is forced to stop. The robot judges whether the selected module should be terminated or sustained with probability p_i.

2.2 Upper layer

In the upper layer, the modules are switched according to the current situation. Let the value of the module m_i at the state $\boldsymbol{x}$ be $V_i(\boldsymbol{x})$. The robot selects the module of which value is the highest:

$$i^* = \arg \max_{i=1,\cdots,n} V_i(\boldsymbol{x}) I_i(\boldsymbol{x}). \quad (3)$$

Once the module is selected, then actions are executed according to the current behavior π_i until the module terminates stochastically based on T_i.

In order to approximate $V_i(\boldsymbol{x})$, we use the function expressed by

$$V_i(\boldsymbol{x}) = \sum_{j=1}^{N} \exp\left(-(\boldsymbol{x} - \boldsymbol{c}_{ij})^T \boldsymbol{W}_{ij} (\boldsymbol{x} - \boldsymbol{c}_{ij})\right), \quad (4)$$

where $\boldsymbol{c}_{ij} \in \Re^n$ and $\boldsymbol{W}_{ij} \in \Re^{n \times n}$ denote the center position and the symmetric matrix. If $\boldsymbol{W}_{ij}$ is positive definite, Eq.(4) expresses the Gaussian function.

3 Genetic Operations

In order to obtain the appropriate p_i, $\boldsymbol{c}_{ij}$ and $\boldsymbol{W}_{ij}$, we use the genetic algorithms. In GA, it is an important role to design the genetic operations such as crossover and mutation. A procedure to generate the new offspring is indicated in Figure 3.

Suppose that the robot has L modules, and each module has N parameters (p_i, $\boldsymbol{c}_{ij}$, $\boldsymbol{W}_{ij}$). Figure 4 (a) shows the chromosome of individual. We perform two types of crossover.

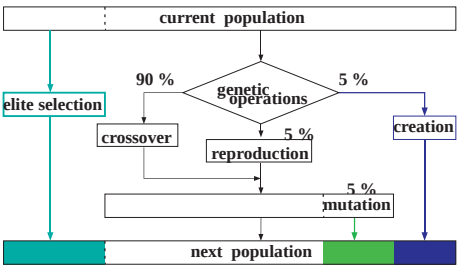


Figure 3: Flowchart of GA

Global crossover : Figure 4 (b) shows a basic idea of global crossover. For each module m_i, a pair of parameters is selected randomly from each parent. Then, we swap two parameters.

Local crossover : At the beginning, we find the parameters of which distance is minimum.

$$(j^*, k^*) = \min_{j,k=1,\cdots N} ||\boldsymbol{c}_{ij}^1 - \boldsymbol{c}_{ik}^2||.$$

1. In case of $\boldsymbol{W}_{ij}$, we perform two point crossover.
2. In case of p_i and $\boldsymbol{c}_{ij}$, we utilize BLX-α [3] based on real-coded GA. Figure 4 (c) shows a basic idea of BLX-α in a case of two dimensional vector. The BLX-α uniformly picks parameter values from points that lie on an interval that extends αI on either side of the interval I between parents. In other words, it randomly generates two children around their two parents by using uniform distribution in the hyper rectangular region whose sides are parallel to axes of the coordinate system.

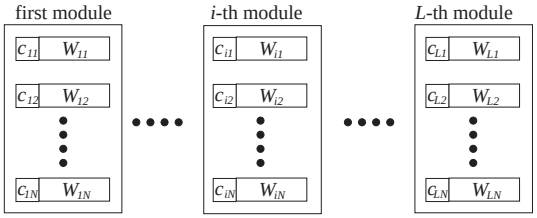
Mutation : One of the elements of $\boldsymbol{c}_{ij}$ or $\boldsymbol{W}_{ij}$ is replaced to a new random value.

Genetic operations used here does not take account of activation/termination constraints explicitly. In other words, activation/termination constraints do not help GA to search the feasible solutions directly.

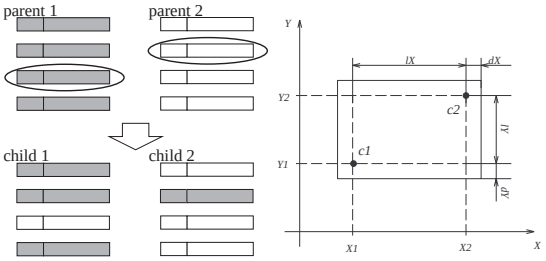
4 Task and Assumptions

4.1 Robot and Environment

We have selected a simplified soccer game as a testbed. The task for the learner is to shoot a ball into the opponent goal. The environment consists of a ball and two goals, and a wall is placed around the field except the goals. The sizes of the ball, the goals and



(a) chromosome for one module



(b) global crossover (c) local crossover

Figure 4: Crossover

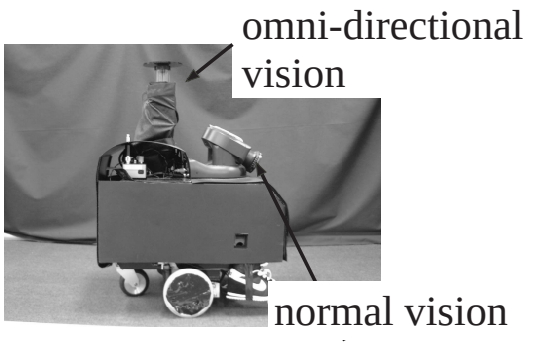
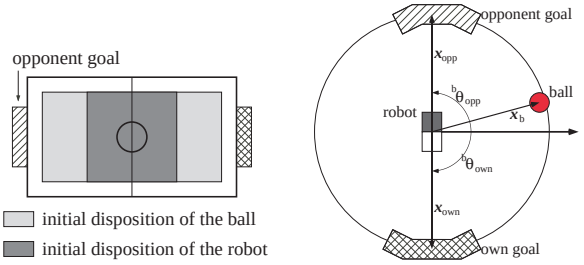


Figure 5: Our mobile robot



(a) top view of the field (b) view captured by an omni-directional vision

Figure 6: The experimental setting.

the field are the same as those of the middle-size real robot league of RoboCup Initiative [5] that many AI and robotics researchers have been involved in.

Figure 5 shows the real robot used for modelling. The robot moves around the field based on the power wheeled steering system. As motor commands, our mobile robot has two degrees of freedom. The input $\boldsymbol{u}$ is defined as a two dimensional vector:

$$\boldsymbol{u}^T = \left[\begin{array}{cc} u_l & u_r \end{array} \right] \quad u_l, u_r \in \{-1,\ 1\},$$

where u_l and u_r are the velocities of the left and right wheels, respectively. In addition, the robot has a kicking device to kick the ball.

The robot has two vision systems; one is a normal vision system to capture the front view, and the other is an omni-directional one to capture the visual information whole around the robot. The omni-directional vision has a good feature of higher resolution in direction from the robot to the object although the distance resolution is poor.

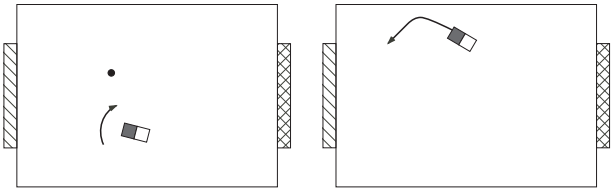
The robot observes the center positions of the ball and two goals in the image plane using two vision systems. Therefore, the number of image features is 12. A simple color image processing is applied to detect the ball and the goal areas in the image plane in real-time (every 33 [msec]). Figure 6 (b) shows detected image features to extract the information of the environment based on the omni-directional vision, where $\boldsymbol{x}_b$, $\boldsymbol{x}_{own}$ and $\boldsymbol{x}_{opp}$ are the center position of the ball, the own goal, and the opponent one, respectively.

4.2 Module Design

4.2.1 Basic Modules

We prepare five basic modules of which behavior just generates a simple action regardless of sensory information. That is, the motor command generated by each basic module is described as follows:

- m_1 : go forward
  $\boldsymbol{u}^T = [1.0,\ 1.0]$
- m_2 : go backward
  $\boldsymbol{u}^T = [-1.0,\ -1.0]$
- m_3 : stop
  $\boldsymbol{u}^T = [0.0,\ 0.0]$
- m_4 : turn left
  $\boldsymbol{u}^T = [-1.0,\ 1.0]$



(a) m_6 (b) m_7

Figure 7: Typical behaviors generated by the reactive modules

- m_5 : turn right
  $\boldsymbol{u}^T = [1.0,\ -1.0]$

These modules are always executable, in other words, for all $\boldsymbol{x}$ we set $I_i(\boldsymbol{x}) = 1$ $(i = 1, \cdots, 5)$. Since they have no explicit purpose to achieve, the termination constraints only depends on the steps. In this experiment, we set the termination parameters in Eq.(2) as $p = 0.4$ and $t_p = 150$ steps $(= 5$ [sec]).

4.2.2 Reactive Modules

Figure 7 shows typical behaviors generated by the four modules prepared in advance. In order to realize the defending behavior, we design the following four reactive modules.

- m_6 : search the ball
  The purpose of this module is to capture the ball image using the normal vision. Therefore, the robot searches the ball by turning to left or right. T_6 is set to zero when the ball is observed.

- m_7 : avoid collisions
  The purpose of this module is to avoid collisions with the wall. If the wall is not detected, $\boldsymbol{u}^T = [1.0,\ 1.0]$. This module is activated when the robot moves near the wall.

- m_8 : kick the ball
  In a case where the ball is in front of the robot and this module is selected, the robot succeeds in kicking the ball. Of course, this module has no effects when the ball is not in front of the robot. This module is activated when the ball image is captured by the normal camera.

- m_9 : shoot the ball
  The purpose of this module is to push the ball into the opponent goal. This module is activated when both the ball and the opponent goal images are captured by the normal camera. The resultant behavior is the same as that of m_1, that is, $\boldsymbol{u}^T = [1.0,\ 1.0]$. This module does not always succeed in shooting behaviors, especially when the ball position is shifted from the goal direction.

In this experiment, we set the termination parameters in Eq.(2) as $p = 0.8$ and $t_p = 150$ for the above four modules.

4.2.3 Complex Modules

We prepare a controller which makes the features on the image plane converge to the desired values. For the desired state $\boldsymbol{x}_d = [x_d\ y_d]^T$, a motor command $\boldsymbol{u}$ is computed by

$$\boldsymbol{u} = \begin{bmatrix} u_r \\ u_l \end{bmatrix} = \boldsymbol{K} \begin{bmatrix} -1 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x_d \\ y_d \end{bmatrix}, \tag{5}$$

where u_r and u_l are the velocities of the right and left wheels, respectively. $\boldsymbol{K}$ is a gain matrix. Using the controller based on Eq. (5), we prepare the following two modules.

- m_{10} : move to the defensive position
  The purpose of this module is to move to the place between the ball and the own goal. The desired state $\boldsymbol{x}_d$ is given by

$$\boldsymbol{x}_d \quad = \quad (1-a)\boldsymbol{x}_b + a\boldsymbol{x}_{own}.$$

- m_{11} : move to the offensive position
  The purpose of this module is to move to the opposite side of the opponent goal to shoot. The desired state $\boldsymbol{x}_d$ is given by

$$\boldsymbol{x}_d \quad = \quad (b+1)\boldsymbol{x}_b - b\boldsymbol{x}_{opp} \quad (0 \le b \le 1).$$

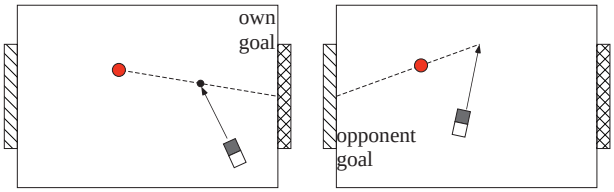
These two modules can be executed when the desired state is not achieved, that is,

$$I_i = \begin{cases} 1 & ||\boldsymbol{x} - \boldsymbol{x}_d|| \le \varepsilon \\ 0 & \text{otherwise} \end{cases},$$

where ε and $||\boldsymbol{x}||$ denote the norm of $\boldsymbol{x}$, and the small threshold, respectively. In this experiment, we set the termination parameters in Eq.(2) as $p = 0.8$ and $t_p = 150$ for the above two modules.

4.3 GA Settings

The population size is 50, and we perform 30 trials to evaluate each individual. At the beginning of the trial, the robot and the ball are placed at the dark and light gray areas, respectively shown in Figure 6 (a). One trial is terminated if the robot shoots a ball into the



(a) m_{10} (b) m_{11}

Figure 8: Typical behaviors generated by the complex modules

goal or the pre-specified time interval expires. In order to select parents for crossover, we use tournament selection with size 10.

One of the most important issues is to design the fitness measures. In this experiment, we set up four fitness measures as follows:

- f_1 : the total number of obtained goals,
- f_2 : the total number of lost goals,
- f_3 : the total number of steps until all trials end
- f_4 : the total number of ball-kicking,

In order to cope with multiple fitness measures, one simple realization is to create the new scalar function based on the weighted summation of multiple fitness measures by

$$f_c = \sum_{i=1}^{n} w_i f_i, \tag{6}$$

where w_i denotes the weight for i-th evaluation. The problem is to design the value of w_i since we must consider the tradeoff among all the fitness measures. In this experiment, we use the adaptive fitness function [10] to decide the weights. Based on this method, the weights are modified considering the relationships among the changes of the four evaluations through the evolution process.

5 Experimental Results

In order to show the validity of the proposed method, we perform the following four experiments; (1) without activation/termination constraints, (2) with termination constraints, (3) with activation/termination constraints, and (4) proposed method. In cases of (2) and (3), a probability p_i for each module is fixed while the probability based on (4) is obtained by the learning method. Figure 9 shows the averaged scores during

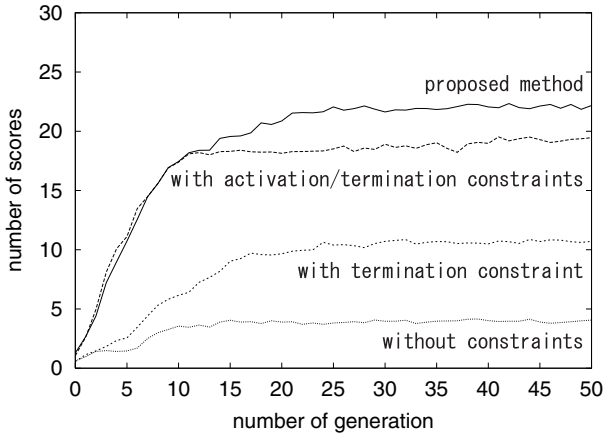


Figure 9: Average of scores

the evolutionary processes. Since we perform 30 trials to evaluate one individual, the maximum value of the averaged score is 30.

5.1 Simulation Results

Without activation/termination constraints

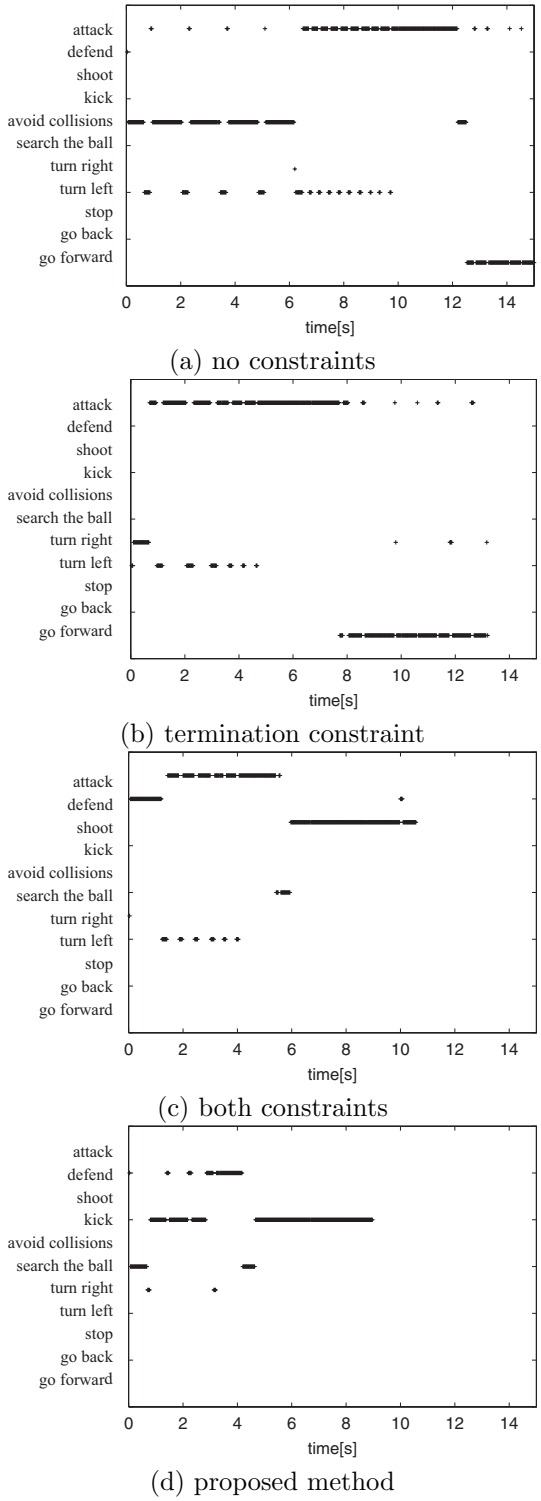
Figure 9 shows that this approach did not fulfill the goal of shooting behavior. Figure 10 (a) shows the transition of the selected modules. Since the robot selects the new module in real time (every 33 [msec]), the robot changed the module frequently, especially from six to ten seconds. At the beginning, the robot selected the avoiding module m_7 although the robot is not located near the wall. In this case, the robot utilized m_7 to approach the ball since this module generated the backward action when the ball is not observed. Then, the two modules m_{11} and m_4 are selected frequently from six to ten seconds. As a result, the robot failed to shoot the ball into the goal until the pre-specified time interval expired.

With termination constraints

This approach caused the successful shooting behavior, and took shorter learning time than the case of no constraints described in the above. However, this approach took longer time to evolve than the case with both constraints. Figure 10 (b) shows the transition of the selected modules. In this experiment, m_1 was selected to shoot the ball into the goal instead of m_9.

With activation/termination constraints

Figure 10 (c) shows the transition of the selected module. Until six seconds, the robot used three modules



(a) no constraints

(b) termination constraint

(c) both constraints

(d) proposed method

Figure 10: Sequences of the selected modules

Figure 11: Example behavior sequences based on the proposed method

m_4, m_{10}, and m_{11} to go back to the offensive position. In this situation, the pre-defined behavior m_{11} can not succeed in moving to the offensive position since this behavior is implemented by a local linear feedback controller. After the robot moved to the front of the ball, the robot succeeded in shooting the ball into the goal.

Proposed method

Figure 9 shows that this approach took the shortest learning time to obtain the shooting behavior, and got best scores. In this method, the basic module, for example, m_5 (turn right) was terminated quickly. One of the successful behaviors based on the proposed method is shown in Figure 11, where the numbers in the figure represents the elapsed time (second).

The learning processes of the cases (c) and (d) are almost same as shown in Figure 9. In this experiment, the probability p_i did not converged because the optimal probability depends on the switching function.

5.2 Real Experiments

We show a result to demonstrate how the proposed method works. We transfer the result of computer simulation to the real robot. A simple color image processor (Hitachi IP5000) is applied to detect the ball and the goal area in the image in real-time (33 [msec]).

Figure 12 shows an example sequence of obtained behavior in the real environment. Because of the low image resolution of the omni-directional vision system, the robot sometimes failed to detect the objects at a

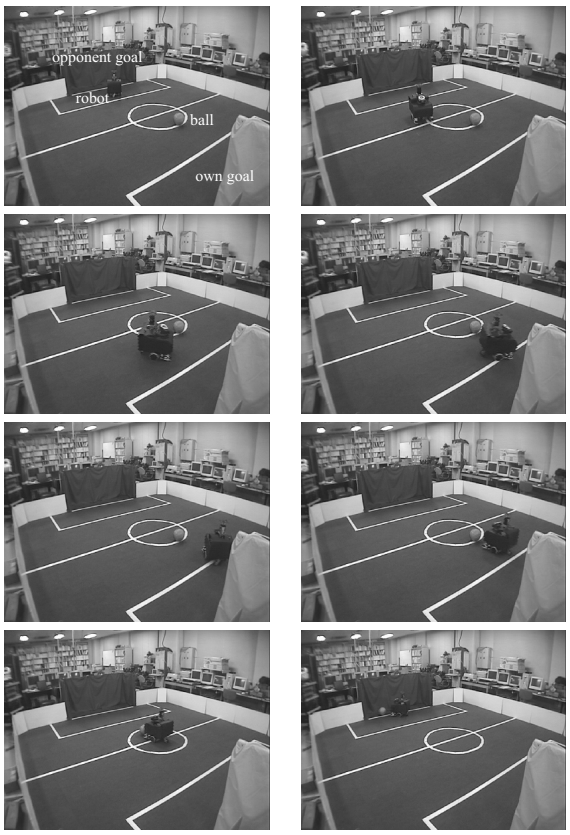


Figure 12: Obtained behavior in a real environment

long distance. In this case, the module could not be performed appropriately. In spite of those troubles, our robot could accomplish the given task.

6 Discussion and Future Works

This paper presented an architecture for behavior selection with activation/termination constraints. We applied the proposed method to a soccer situation, and demonstrated the experiments on a simulated robot.

In the current version of our method, the genetic operations are applied only to learning the upper layer, that is, $\boldsymbol{c}$ and $\boldsymbol{W}$. One interesting extension is to learn the appropriate termination constraints since it should be better to set the appropriate T according to the situation.

As future work, we will apply the proposed method to co-evolution for cooperative behavior acquisition in the context of RoboCup.

Acknowledgements

This research was supported by the Japan Society for the Promotion of Science, in Research for the Future Program titled Cooperative Distributed Vision for Dynamic Three Dimensional Scene Understanding (JSPS-RFTF96P00501).

References

[1] M. Asada et al. Purposive Behavior Acquisition for a Real Robot by Vision-Based Reinforcement Learning. *Machine Learning*, 23:279–303, 1996.

[2] M. Ebner and A. Zell. Evolving a behavior-based control architecture – From simulations to the real world. In *Proc. of the Genetic and Evolutionary Computation Conference*, pages 1009–1014, 1999.

[3] L. J. Eshleman and J. D. Schaffer. Real-Coded Genetic Algorithms and Interval-Schemata. In *Foundations of Genetic Algorithms 2*, pages 187–202. 1993.

[4] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison–Wesley, 1989.

[5] H. Kitano, ed. *RoboCup-97 : Robot Soccer World Cup I.* Springer Verlag, 1997.

[6] J. R. Koza. *Genetic Programming I : On the Programming of Computers by Means of Natural Selection.* MIT Press, 1992.

[7] P. Stone. *Layered Learning in Multiagent Systems.* The MIT Press, 2000.

[8] R. S. Sutton and A. G. Barto. *Reinforcement Learning.* MIT Press/Bradford Books, March 1998.

[9] E. Uchibe, M. Nakamura, and M. Asada. Cooperative and Competitive Behavior Acquisition for Mobile Robots through Co-evolution. In *Proc. of the Genetic and Evolutionary Computation Conference*, pages 1406–1413, 1999.

[10] E. Uchibe, M. Yanase, and M. Asada. Behavior Generation for a Mobile Robot Based on the Adaptive Fitness Function. In *Proc. of Intelligent Autonomous Systems (IAS-6)*, pages 3–10, 2000.