Evolving Efficient Sensor Arrangement and Obstacle Avoidance Control Logic for a Miniature Robot

Muthukumaran Chandrasekaran[1], Karthik Nadig[1] and Khaled Rasheed[2]

[1] Institute for Artificial Intelligence, University of Georgia, Athens, GA, USA
mkran@uga.edu, kvnadig@uga.edu
[2] Computer Science Department, University of Georgia, Athens, GA, USA
khaled@cs.uga.edu

Abstract. Evolutionary computation techniques are being frequently used in the field of robotics to develop controllers for autonomous robots. In this paper, we evaluate the use of Genetic Programming (GP) to evolve a controller that implements an Obstacle Avoidance (OA) behavior in a miniature robot. The GP system generates the OA logic equation offline on a simulated dynamic 2-D environment that transforms the sensory inputs from a simulated robot to a controller decision. The goodness of the generated logic equation is computed by using a fitness function that maximizes the exploration of the environment and minimizes the number of collisions for a fixed number of decisions allowed before the simulation is stopped. The set of motor control decisions for all possible sensor trigger sequences is applied to a real robot which is then tested on a real environment. Needless to say, the efficiency of this OA robot depends on the information it can receive from its surroundings. This information is dependant on the sensor module design. Thus, we also present a Genetic Algorithm (GA) that evolves a sensor arrangement taking into consideration economical issues as well as the usefulness of the information that can be retrieved. The evolved algorithm shows robust performance even if the robot was placed in completely different dynamically changing environments. The performance of our algorithm is compared with that of a hybrid neural network and also with an online (real time) evolution method.

1 Introduction

Evolutionary robotics is a widely researched methodology that uses evolutionary algorithms to design controllers for autonomous robots. Different examples of applications of evolutionary robotics and approaches to develop controllers for mobile robots were introduced by Nolfi et al. [4]. Robotic controllers have been previously evolved using dynamic recurrent neural nets [1, 2]. Several experiments have also been performed that directly use genetic programming to evolve controllers that implement an obstacle avoidance behavior [3, 5, 6]. The efficiency of such controllers and the impact on their performance, however, depend on the sensor configuration present in the robot. Optimal sensor placement is essential to ensure maximal useful information retrieval from the surroundings in which the robot is placed. However, increasing the number of sensors alone does not guarantee a better performance. The usefulness of the information brought in by an additional sensor must also be taken into account. For instance, if sensors are placed very close to each other, there would be redundancy in the information that was retrieved by the sensor module. Not only would this redundancy render the information useless, but it would also raise economic issues. Therefore, when designing a sensor system, one must attempt to find those combinations of sensors that would maximize the information brought by those sensors while minimizing the possible cost. Previous work in optimizing sensor placement using genetic algorithms has been reported [7] where they propose a method to design a cost optimal sensor system that ensures a certain degree of diagnosability in an instrumentation system domain. In this paper, we use a GA to design an efficient sensor module that is later applied to a real robot with an objective to avoid obstacles while favoring movement in the forward direction and maximization of the area covered by it. The robot placed in the simulated environment also incorporates the resulting sensor configuration.

Previous experiments with genetic programming and robotic control have been performed with a simulated robot placed in a simulated environment. Problems with performing experiments on only simulated environments are several. Real environments are dynamic and uncertainty exists everywhere. There are also an infinite number of real-life training situations that simulated environments fail to capture. Hence, testing on just simulated settings could result in specialization in certain types of environments and failure to evolve a behavior to generalize unseen types. Previous experiments have also been done on real robots trained in real-time [5]. However, since the OA algorithm was evolved in real-time, the robot would function under a good control strategy, only hours after the robot was placed in the environment. The robot would be subjected to several collisions as the training progresses even in a controlled environment. Such a robot would either, require a high RAM and a large ROM containing a small operating system with multi-tasking capabilities in order to be able to implement the entire algorithm in real time in a stand-alone fashion, or, require the controlling algorithm to be run on a workstation with data and commands communicated through a serial line. In our experiments, we use a real robot (with actual sensors) that was trained in a realistically simulated, wholesome dynamic environment using a GP to evolve the OA algorithm in the simulated environment where a simulated robot, whose sensor and constructional designs accurately resemble that of our real robot, was placed. The final sensor arrangement used in both our simulated and real time experiments is the outcome of the GA used to design the configuration.

2 The MK$_{alpha}$ Robot

We constructed a real robot containing real sensors for the purpose of testing our offline OA algorithm in the real world. The MK$_{alpha}$ robot is built on an Arduino platform constituting an ATmega328 IC. It is equipped with four infrared distance sensors with a detection range from 2 to 10 cm. The dispersion angle for the sensors is negligibly small. The mobile robot has a circular shape, a diameter of about 10 cm and a height of 10 cm. It consists of two 5V DC motors, an on-board power supply for the Arduino board and the sensor module. The four infrared sensors are initially uniformly distributed around the circumference of the robot. A modular design is used for the motor driver and sensor units. This design enables repositioning sensors as per the configuration evolved by the GA. Some simple real time experiments showed that the sensor arrangement output by the GA was better than the initial sensor arrangement in terms of the number of collisions for the same controlling algorithm. This controlling algorithm code is written and compiled on an open-source Arduino environment and then uploaded on to the I/O board via a serial port.

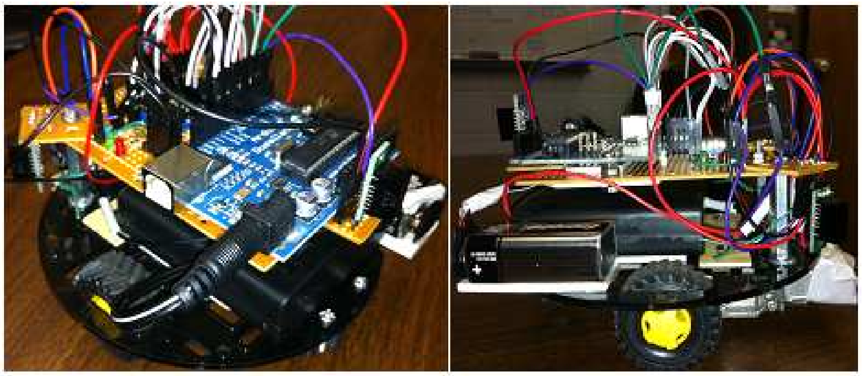


Fig. 1. The MK$_{alpha}$ Robot

The ATmega328 microcontroller can operate on an external power supply of 6 to 20 volts. It possesses of 32 KB of flash memory for storing code (of which 2 KB is used for the boot loader). It also contains 2 KB of SRAM and 1KB of EEPROM. It has 14 pins that can be used as an input or output operating at 5V. Figure 1 shows the fully functional MK$_{alpha}$ robot including the sensor arrangement.

3 Genetic Algorithm for Sensor Placement

The goal of the GA is to find an optimal sensor arrangement around the robot given its actual physical configuration and application-specific biases. Our OA robot finds pleasure in moving forward and fast. It also favors a sensor arrangement, that maximizes the area covered by the sensors and minimizes the cost (by using only as many sensors as necessary in order to ensure no redundancy occurs in the information retrieved). The GA takes as input, the actual dimensions of the robot and infrared sensors and the maximum number of sensors available. It computes the maximum number of possible sensor slots based on the dimensions of the robot and the sensors. The GA also takes as input, the desirability of moving in a particular direction. Thus, for our specific application, since moving forward is favored, larger weights are arbitrarily assigned to forward-facing regions of the robot. Figure 2 shows how the robot was divided into 8 *regions of interest* and their corresponding weights. These weights indicate the degree of desirability of moving in the corresponding direction which should in turn proportionately increase or decrease the likelihood of finding a sensor in some slot in that region of interest. The blue contour in the figure shows the desirability curve for application-specific sensor arrangement. It is assumed that each region is uniformly distributed and spans over an angular distance of 45 degrees. For example, the *forward* region of interest is given a large weight of 0.7. This means that the robot strongly desires movement in the *forward* direction and in order to ensure safe passage, ideally, a sensor must be placed in some slot in this region. The genotypes or the individuals constituting the initial population are

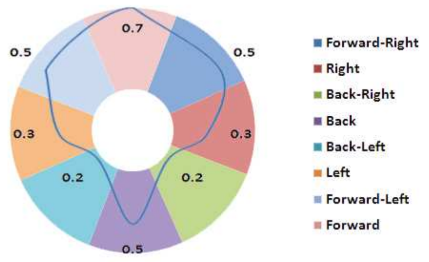


Fig. 2. Regions of interest and their corresponding weights indicating a desirability to move in that direction. The blue contour around the robot shows the desirability curve whose radial dimensions measure the degree of desirability

represented as bit strings containing one gene each for all available slots around the robot where a sensor unit could be assembled. A “1” in the genotype denotes the presence of a sensor in the corresponding slot. The number of 1’s in the genotype will be less than or equal to the maximum number of sensors available to us. For instance, assume that the robot has a diameter d = 10 cm and the size of the sensor unit l = 2 cm, the genotype will contain $(\pi \times d) / l = 15$ slots (genes). An example genotype would look like [1,0,1,0,0,0,0,1,0,0,0,0,0,1,0]. This denotes the presence of 4 sensors in the 1^{st}, 3^{rd}, 8^{th} and 14^{th} slot. Each slot occupies an angular distance of (360 / *no. of slots*) = 24 degrees. Assuming the midpoint of the first slot is at zero degrees, the 4 sensors are positioned at angles of 0, 48, 168 and 312 degrees with respect to the midpoint of the first slot. We then generate a set of n (=100 for our experiment) 8-bit strings, where each bit denotes one of 8 regions of interest. If the bit string has a “1” in some position, it would indicate the desirability of the robot to move in that direction and the likelihood of getting a “1” would depend on the weight corresponding to that region. The fitness of the individual is evaluated using the following rubric: Traverse through each gene in the individual,

1. If the gene is a “1” and the corresponding region has a “1”, award 2 points
2. If the gene is a “1” and the corresponding region has a “0”, award -0.5 points
3. If the gene is a “0” and the corresponding region has a “1”, award -0.5 points
4. If the gene is a “0” and the corresponding region has a “0”, award 1 point

The fitness would be the average of the total points received by each gene in the individual for n regions of interest divided by the total number of 1’s in the individual (= the number of sensors present in the current configuration). Mathematically, it would be represented as follows,

$$F(genotype) = \frac{\sum_{r=1}^{n} TotalPoints_r/n}{TotalNumberOfOnesInGenotype} \quad (1)$$

We implemented a steady state GA with a ternary tournament selection methodology, 1-point crossover and scramble, swap, insert or inversion mutation chosen randomly. We used a replace-worst strategy with the termination set at 200,000 fitness evaluations. A summary of the GA parameters used are shown in Table 1. Incidentally, the best sensor module design output by the GA happened to be the one used in the example above.

Table 1. Summary of GA Parameters

Representation	Binary (bit string)
Selection	Tournament (Size = 3)
Crossover	1-Point (P_{Xover} = 0.8-1.0)
Mutation	Insert, Inversion, Scramble or Swap (P_{Mut} = 0.9)
Replacement	Replace Worst
Termination	200,000 fitness evaluations
Population Size	200

4 Genetic Program for Motor Control

The inspiration for this experiment comes from [5] where control strategies were evolved in real time on a real robot using a GP. The design of the algorithm allowed for flexibility in the percept and control design. In this section, we present a GP that trains a realistic robot in a simulated dynamic environment to evolve a controller to enable the robot to navigate around obstacles minimizing collisions while exploring the environment it was placed in. The simulator design mimicked real-time scenarios in the best possible way, thereby allowing the controller, which was evolved offline, to be implemented on a real robot. The simulated environment generates digital percepts and expects a command response to process the next move. This response is generated by evaluating the percepts using an equation which transforms the percept data to motor control commands, such as Move Forward, Turn Right, Turn Left and Move Backward, represented as numeric values ranging from 0 to 3 respectively. Genetic Programming techniques were used to evolve the equations that calculate the motor control commands. This equation takes the sensor values as the input vectors and returns an integer representing a motor control command.

$$f(s_1, s_2, s_3, s_4) = m, \text{ where } m \in [0,3]$$

Each individual in the population represents the entire solution described as an equation tree. The individual was initialized by generating two types of random equation trees. They were either *complete* trees, where the equations used all of the available variables from the input, or *partial* trees, where the equations used only a subset of the input variables. The input variables represent the values received by the robot as percept inputs. Twenty percent of the population was populated with partial trees and the remaining eighty percent with complete trees. An example of such trees is shown in Table 2. Numbers within square brackets represent the index of input variables used. In the example above, the complete tree uses all four of the inputs available to it, whereas, the partial tree takes three, out of four, inputs. The expressions use any of the operators available to it from a predefined operator set. The operator sets were classified as binary and unary operators. The unary operator set contains the unary minus (−) and the bitwise-not (˜) operators. The binary operator set was further classified as bitwise, arithmetic and relational operators. The initial population of individuals consisted of only bitwise and/or arithmetic operator. The relational operators were later added to test the performance. In order to avoid this equation from becoming really large, equations were limited to contain only a fixed number of operators. The equation trees consist of two types of nodes - the branch nodes (that take values from the operator sets) and the leaf nodes (that contain indexes of the input variables used or some constant integer values). The index of the input variables correspond to the percept inputs and are available as a zero based array. The results of the Boolean relational operations are evaluated as numerical values, by substituting TRUE with integer 1 and FALSE with integer 0. This allows relational operators to be evaluated as numeric values and hence generate the commands for the motor control.

Table 2. Equation Trees

Complete Tree :	((([1] + [3])ˆ([2] | [0])) + (([0]ˆ[1])ˆ([0] + [1])))
Partial Tree :	((([1] + [2]) ∗ [0]) &[1])

Both crossover and mutation operators are used in the evolution of equation trees. Sub-tree crossover was used with a crossover probability that was proportional to the depth of the tree. Multiple mutation operators are used with a low probability of mutation along with an inverse relation to the fitness of the individual. The mutation operator was applied to the binary and unary nodes. Index and constant nodes are mutated by selecting random input variable indices or random integer values to replace the corresponding gene in the original genotype, respectively. Sub-tree mutation with extremely low mutation probability was also included. The probability of mutation was also made proportional to the depth of the tree. A randomly generated short sub-tree was used to replace an existing sub-tree at the selected branch node. The GP was configured to run with a steady state scheme. Tournament selection was used to select parents. The generated child was added to the population and then the individual with the worst fitness was removed from the population. A fixed population size was used and random seeding was done if no improvement in fitness was seen for a long time. The fitness of the individual was calculated by averaging the simulation results from 10 runs of the robot within a large pre-defined maze-like environment containing strategically placed obstacles to mimic real world settings. The MK$_{alpha}$ robot's objective was to maximize the area covered by it while minimizing the number of collisions, for a fixed number of steps.

We designed the simulator to allow the control of the position and orientation of the robot, its step size, angular step size, sensor error probability, actuator error probability, faulty triggers, number of sensors and sensor positions. Simulation runs were commenced by placing the robot at a randomly selected point within the movable area in the maze. The orientation of the robot was also randomly selected. The robot would then be assigned a fixed quota of steps it can use to move around. Each percept evaluation is considered as a step irrespective of any manifestation of change in the state of the robot. The environment within the simulator is probabilistic, and hence introduces perception errors in the sensors. At each step, the simulator generates the required values and applies it to the equation, and evaluates the next move. Data such as the number of collisions, number of critical decisions, area covered, encountered cases, etc., are recorded at each step. The results are averaged over 10 runs and the area covered and number of collisions are used as a part of a multi-objective evaluation of the individual.

5 Hybrid Neural Controller

An evolved neural controller was developed and its performance was compared to that of the GP. The neural controller is represented as an array of real valued weights. The controller was designed to accept sensor inputs and generate motor control commands. Arithmetic and/or uniform crossover and swap and/or random mutation were used in a steady state GA scheme to train the neural network. The controllers were evolved using identical parameter settings as that used for evolving the equation trees of the GP thereby allowing a fair comparison.

6 Experiments

In the first phase of experimentation, a realistically simulated MK$_{alpha}$ robot, controlled using the OA behavior evolved by the GP, was tested in the simulated environment (described earlier). First, we tested the performance of the robot based on the average fitness attained after 600 decision steps for the sensor configuration evolved using our GA, against a manually designed configuration with uniformly distributed sensors. We show the difference in fitness while the GP was run for 1000 generation equivalents. Next, we compared the performance of the robot, when its OA control strategy was evolved using a GP and using a hybrid neural network (neural network whose weights are evolved using a GA). We illustrate the difference in fitness while the controllers trained for 1000

Table 3. Summary of GP and Hybrid NN Parameters

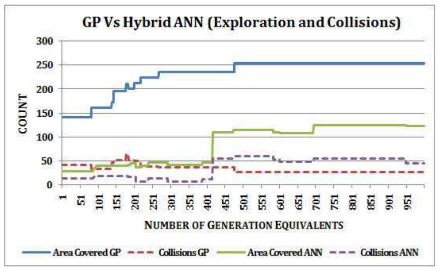
Parameters	Generic Programming	Hybrid NN
Objective	OA Behavior	OA Behavior
Terminal Set	Integers (0 to 3)	Integers (0 to 3)
Function Set	AND, OR, XOR, ADD, SUB, MUL, SHL, SHR, NOT	AND, OR, XOR, ADD, SUB, MUL, SHL, SHR, NOT
Relational Set	LT, GT, EQ, LTE, GTE, NEQ	LT, GT, EQ, LTE, GTE, NEQ
Representation	Equation Trees	Weights (Float array)
Population Size	50	50
Crossover Prob	0.95	0.95
Mutation Prob	0.05	0.05
Selection	Tournament (5%-10%)	Tournament (5%-10%)
Max Generations	None	None
Limits	200 Function Nodes	20 Hidden neurons
Termination	None	None
Fitness	Weighted Multi-Objective	Weighted Multi-Objective

generation equivalents. We also show the difference in the number of generations taken to achieve a particular fitness for each controller.

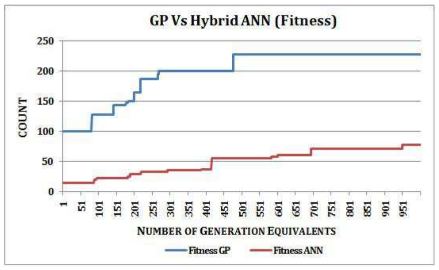
In the second phase of experiments, the objective is to compare our offline GP controller with the online counterpart developed by Peter Nordin and Wolfgang Banzaf [5]. They tested their GP on the Khepera robot which is half the size of the MK$_{alpha}$ with twice the number of sensors. The Khepera robot was trained on two environments - a rectangular box (30cm $\times$ 40cm) and a more complex larger environment (70cm $\times$ 90cm) with irregular borders and loose objects placed inside it. In order to conduct a fair comparison, we designed and tested the MK$_{alpha}$ robot on similar environments but twice as big to account for its larger size. The results of the Khepera robot are taken as it is from [5] for comparison.

7 Discussion of Results

The first phase of experiments show an exploratory behavior for the simulated robot as the fitness function of the GP favors a strategy that maximizes exploration. Since, the robot carried out in-place turns, only movement in the forward or backward directions brought about a change in the area covered. Thus, the desirability of the robot to favor movement towards forward facing regions was used as bias to evolve a sensor arrangement so as to better enable this exploratory behavior. It was seen that the GA-evolved sensor arrangement performed better when compared to the manually designed uniform sensor arrangement. As the control logic allowed for more area coverage, the robot was exposed to greater chances of collisions. During the first few generations, while the robot explores the map, it keeps colliding with different obstacles in its path, but as time goes on the number of collisions becomes more and more infrequent even as a greater area of the map was explored as seen in Fig. 3 (a).



(*a*)



(*b*)

Fig. 3. Comparison between GP an Hybrid Artificial Neural Network (ANN). (*a*) Shows exploration and collisions(*b*) Shows average fitness. Note that each data point is the average of 10 simulation runs.

The GP based individual shows a maximum area covered of 254 given a quota of 600 decision steps. The reason for this small ratio could be one or more of the following - (1) in an attempt to over train for realistic situations, the robot was trained in a congested and dynamically changing environment. (2) The robot makes in-place turns which are counted as steps while the area covered remains unchanged. (3) In order for the robot to maneuver around a wall, it takes an average of 3 to 4 decision steps because the robot was programmed to make 10 to 20 degree turns. So, the environment had to be sampled at least 3 times for the robot to make a 60 degree in-place turn.

From Fig. 3 (*b*), we observe that GP was able to generate a better control strategy than an evolved neural controller for equal number of generations. Due to large selection pressure during parent selection, the algorithms show a faster convergence rate. Since a minor change in the equation tree could drastically alter the fitness level of the individual, with careful design, evolving an effective equation could improve the performance of the robot.

The GP-evolved offline controller was tested on the MK$_{alpha}$ robot placed on the complex environment and the number of collisions that occurred every minute was recorded over a period of 60 minutes. These results were superimposed on the results obtained when an online GP-evolved controller was tested on the Khepera robot (Figure 7 in [5]). The online algorithm evolved a comparably good control strategy only 40 to 60 minutes after the training commenced while the offline GP converged much faster. An even slower convergence may be expected if the robot was trained in real-time in a more complex environment. We also observed only a small advantage in the online controller even when the converged control strategy was used. Also, considering the fact that the size of the Khepera robot is half of that of the MK$_{alpha}$ and that it contains twice as many sensors, one wonders if such economically and computationally expensive online controllers are really worth the small advantage in performance.

8 Conclusion and Future Work

We have demonstrated that a GP system can be used to evolve a good control strategy for a real robot, offline. The goodness of the control strategy depended largely on the percept inputs to the GP which in turn depends on the sensor arrangement around the robot. We used a GA to evolve an efficient sensor arrangement. The evolved controller showed robustness even if the robot was placed in a completely different environment or if the obstacles were moved around. We also showed the advantages of evolving the controller offline as opposed to evolving one in real time in terms of the robot design configuration, hardware capabilities of the on-board microcontroller and time taken to achieve a good control strategy.

Future work includes additionally evolving the speed of the motor. For instance, the speed of the motor could be automatically increased if no obstacle was detected for several steps. Robot designs that have multiple motors have to be manually aligned accurately as they are used for in-place rotational and linear movements, which is a painfully tedious process. This could also be done automatically by evolving the right motor speeds to enable precision movement. The control strategy could be extended to incorporate analog, or a combination of analog and digital sensors and allow a GA to choose an optimal combination of the various types of sensors and their arrangement on the robot.

References

1. Cliff, D.: Computational neuroethology: a provisional manifesto. In: Proceedings of the first international conference on simulation of adaptive behavior on From animals to animats. pp. 29–39. MIT Press, Cambridge, MA, USA (1990)
2. Harvey, I., Husbands, P., Cliff, D.: Issues in evolutionary robotics. In: Proceedings of the second international conference on From animals to animats 2 : simulation of adaptive behavior: simulation of adaptive behavior. pp. 364–373. MIT Press, Cambridge, MA, USA (1993)
3. Koza, J.R.: Genetic programming: on the programming of computers by means of natural selection. MIT Press, Cambridge, MA, USA (1992)
4. Nolfi S., Floreano D., M.O..M.F.: How to evolve autonomous robots: different approaches in evolutionary robotics. In: Artificial Life IV. pp. 190–197. MIT Press/Bradford Books (1994)
5. Nordin, P., Banzhaf, W.: A genetic programming system learning obstacle avoiding behavior and controlling a miniature robot in real time. Tech. rep., The University of Dortmund (1995)
6. Reynolds, C.W.: Evolution of obstacle avoidance behavior: using noise to promote robust solutions, pp. 221–241. MIT Press, Cambridge, MA, USA (1994)
7. Spanache, S., Escobet, T., Trav-massuys, L.: Sensor placement optimisation using genetic algorithms. In: Proceedings of the 15th International Workshop on Principles of Diagnosis, DX-04. pp. 179–183 (2004)