Using Genetic Programming to Evolve Robot Behaviours

Christopher Lazarus and Huosheng Hu

Department of Computer Science
University of Essex, Wivenhoe Park,
Colchester CO4 3SQ, United Kingdom
Email: clazar@essex.ac.uk, hhu@essex.ac.uk

Abstract

This paper presents the application of genetic programming (GP) to the task of evolving robot behaviours. The domain used here is the well-known wall-following problem. A set of programs were evolved that can successfully perform wall-following behaviours. The experiments involving different wall shapes were designed and implemented to investigate whether the solutions offered by GP are scalable. Experimental results show that GP is able to automatically produce algorithms for wall-following tasks. In addition, more complex wall shapes were introduced to verify that they do not affect our GP implementation detrimentally.

Keywords: Evolutionary Robotics, Genetic Programming, Wall-following.

I. INTRODUCTION

Traditionally, researchers have programmed robots by explicitly programming rules and designing control structures that includes knowledge concerning the environment where the robot will be placed. On the other hand, Evolutionary Robotics (ER) is a new strategy in robot design and control, which takes its inspirations from natural evolutionary processes. The robot designers specify the traits or behaviours that are desired, by encoding the fitness function for the population to evolve. There is no need to program precisely what a robot needs to do.

ER is concerned with the use of Evolutionary Algorithms (EA) to solve various problems in the field of robotics. EA is a branch of the stochastic search techniques that also includes Simulated Annealing. Nature has provided many insights and inspirations to researchers over the years. The basic tenet of 'survival of the fittest' is one way of describing a process where the fittest members of a population usually manage to reproduce successfully and their desired traits are carried forward via their offspring to the next generation. In nature, the ability of creatures to live long enough to reproduce is the driving force behind the evolution. From the computer science perspective, the mechanism of evolution is the key aspect and the researcher is free to devise and implement suitable objectives for the evolution. The implementations involve both software and hardware and have been applied to robot control and design. These attempts are generally to find ways of making robots more adaptive.

This paper describes an attempt to evolve a robot controller that is capable of performing a wall-following behaviour. GP is selected as the method to do this because it is not deterministic. In other words, it does not need to know the environment that it has to operate in a priory. Instead, we need to devise fitness evaluators to guide the evolution of programs towards the required task. Specifically, we would like to investigate whether the addition of complexity in terms of wall shape would cause the GP algorithm to take a longer time to converge on a successful solution.

Since the seminal publication of Koza's volume on GP [9], the interest in GP research has increased. Many researchers are finding new ways of using GP to solve various types of problems. Koza described in his book that if the evolved GP program consists of all the necessary programming constructs, that is if the programs are Turing complete, theoretically GP could generate any program to solve a particular problem. This means that in terms of machine learning applications, GP is the superset. However, since GP typically considers hundreds of programs per population, there is time penalty to be considered. Nevertheless, with computer processing speeds increasing all the time according to Moore's law, GP performance has improved dramatically compared to ten years ago. There are also parallel processing implementations of GP to increase its performance. It comes as no

Proc. TIMR 01 - Towards Intelligent Mobile Robots, Manchester 2001.
Technical Report Series, Department of Computer Science, Manchester University, ISSN 1361 - 6161.
Report Number UMCS-01-4-1, http://www.cs.man.ac.uk/cstechrep/titles01.html

surprise that GP is being applied to many different types of problems. A number of typical GP applications include prediction, classification, image and signal processing, design, trading, robotics, and artificial life [10].
In this paper, a brief description of GP is shown in section II. A brief account description on works related to our experiments and our experiment objectives are described in section III. Section IV explains the settings and parameters of our experiments. Experimental results are presented in section V. Finally, section VI gives a brief conclusion and future work.

II. GENETIC PROGRAMMING

GP is used for automatic generation of computer programs and is a subset of Genetic Algorithm (GA) [5][9]. Although GP uses the same process as GA to search for solutions, GP is used to evolve computer programs instead of evolving parameters. Populations of programs are evolved in competition and finally the most optimal program is considered for the desired task. Traditionally researchers use the LISP language for GP implementation since its intrinsic capability of manipulating trees. LISP [9] also has a built-in interpreter and the *eval function* that can be used for interpreting each of the evolved programs for evaluation of fitness. However, LISP is very slow comparing with other compiling languages. Other languages that are used in implementing GP are machine code format of a RISC processor [1], C [12] and also C++ [4]. These do not represent the complete taxonomy of GP language implementation. For more detail, please refer to Langdon [10].

The program to be evolved has to be represented in the computer as data structures. The choice of data structures is dependent on the language being used. The data structure is also determined by the GP operators and the performance gains being considered for the type of computers and the operating system used. As an example, in a LISP implementation the data structure of choice is expression trees. This is because the LISP language is adept in dealing with lists of functions. When we implement GP in C language, we have many more options for considering the type of data structures. We could represent the programs as expression trees or graphs using pointers or we can use indexed memory employing the use of pointer tables. One advantage of using linked lists with pointers for representing the expression tree is that we could allocate memory needed during runtime. However, this presents a problem of "garbage collection". This means that allocation and de-allocation of memory could cause memory fragmentation and thus slows down the GP process. Fixed memory location such as arrays do not encounter this problems but then we have the inefficiencies of having to allocate memory at compile time where the extra memory space allocated might not be used at all. Another feature that must be noted is the amount of runtime memory available to the GP software.

The choice of parameters at the start of the GP run affects the performance of the GP implementation. One such parameter is the size of the population to be used. The larger the population size, the more variety of programs could be represented. However, this leads to performance penalty since it takes longer to evaluate the programs at each generation. Another important parameter is the probability of applying any of the operators used. For example, the rate of mutation determines how often the mutation operator is applied to an individual. A high rate of mutation will cause more variety to emerge within the population but it would unnecessarily remove fit individual from the gene pool. There are no hard and fast rules concerning the choice of parameters. Usually, researchers use and refer to parameters presented by other researchers who have performed experiments and have empirical results. Others use a problem specific choice via trial and error.

The population of programs at the start of the GP run has to be created and initialised. Typically, a random set of programs is created using the available functions and terminals. It is clear that the performance of a specific GP run is dependent on the initial fitness of the population. If the created population has a very low fitness value then it would take the GP more time to "discover" or evolve optimal programs. A population that happens to have a good fitness value at the start of the GP run could converge at an optimal solution earlier. There are three main ways of creating an initial population.

- The first method is called the "full" method. A tree creation depth is decided beforehand. Then the functions are chosen at random to create the tree. When the predetermined tree depth reaches, a terminal is chosen and the tree creation is complete. This has the effect of creating trees that has balanced nodes. However, one caveat is that there are fewer varieties of trees that could be created.
- The second method is the "grow" method. The choice of functions or terminal is random. When a function is selected, the tree creation continues. The tree creation stops when a terminal is chosen. This method could create a high variety of trees depth. Nevertheless, some researches feel that this method introduces too many variations and might produce a low fitness value of initial population.

- The third method is called the "ramped half and half" method. This method is a combination of the "full" and "grow" method. Half of the population is created using the "full" method and the other half is created using the "grow" method. This is essentially a way to find a balance between the first two methods and proved popular with many GP researchers.

Individuals in the population need to be selected for reproduction into the next generation. There are several methods used for selection. One method is called the fitness-proportional selection. This was the selection method of choice for many researchers. This method was first used in a GA implementation. Another method is the truncation method. This method comes from the Evolutionary Strategies (ES) field. The last two methods are the ranking and tournament selection method. For a detail explanation of these selection methods please refer to Banzhaf [1].

III. RELATED WORKS AND PROJECT GOAL

In the robotics field, GP is typically used to generate robot control programs. Although GP may not be the panacea for producing robot control programs, yet the possibility of having complex control programs automatically generated for a robotics researcher is a tantalising promise. There is also the possibility that a control program could be automatically generated according to the operating environment. This is an attractive option for making the robot adaptive. However, until recently, many attempts by researchers typically involve simulation because of the long GP run time. That is one of the reasons why more and more researchers are considering evolving a robot's control program in real time.

The wall-following domain has been the subject of many previous researchers. Mataric used subsumption architecture [11] for the original experiment. Subsequently Koza used the LISP language to evolve the wall-following behaviour by employing GP [9]. Ross *et al*. built upon the work of Koza by adding automatically defined functions (ADFs) in their GP implementation [13]. Dain [3] investigated the use of multiple fitness cases by using differently shaped rooms to evolve a robust wall-following algorithm. There is obviously a lot of encouraging results for comparison. Attempting to develop a GP program is not a task to be taken lightly. Incorrect implementation of GP methods might give erroneous and misleading results. Daida *et al.* [2] described the difficult process of gaining duplicable results from papers written by GP researchers. More often than not, published papers usually only gives details of experimental results and not the details of the implementation. This often leads to problems for other researches in trying to duplicate the results presented in the papers.

There is a number of GP software that has been developed since the publication of Koza's book [9]. Earlier implementations favour the LISP language due to its built-in facility for manipulating and evaluating expressions. Subsequently a port for the C language was done by Punch and is named lil-gp [12]. A C++ version was also developed by Singleton called GPQUICK [14]. GPC++ is another C++ version and was developed by Fraser [4]. Currently Banzhaf [1] is experimenting with a machine code version for evolving real time robot control using the Khepera robot.

The aim of our research is to gain an understanding and to develop an evolutionary method of producing a robot control program for an autonomous mobile robot. GP is selected as the method to be used because of the nature of the representation of the problem, which are computer programs. As described in section IV, there are many known works involving the use of GP to evolve robot controllers [1][3][13]. We have selected the well-known wall-following problem as a starting point for our project. Using the experience derived from these experiments, we hope to eventually expand our experiments to deal with real robots. We have produced a GP program that was able to evolve the control program of a wall-following robot. In doing so, we needed to know whether GP would be robust enough to allow for scalability. Thus, in further experiments, we have introduced slightly more interesting walls for GP to work with. Extrusions on the wall are increased incrementally from one experiment to another.

The design and choice of functions and terminals are very important to the GP implementation. The functions and terminals must be capable of describing the desired solutions. In other words, a solution could be found from a combination of these functions and terminals. For example, if the desired solution is a program that adds two numbers together, there should be a function that returns the product of two arguments. A terminal should always represent the leaf of a tree. A terminal does not return any values back to the calling program. The design of the fitness function is important in determining whether the GP can converge into the desired solution. Most GP uses single fitness functions. However, some researchers also consider program size and execution time as a measure of a program's fitness. A measure of the program complexity based on the depth and width of the program tree could also be considered in determining the program's fitness. Usually, a GP run stops when the fitness measure is achieved or is satisfactorily close to an optimal. However, a second stopping condition is included to ensure that the GP run stops if it takes too much time. Normally this is in the form of the number of generation the GP is to be performed. There is no specified "good" generation number. It is up to the researcher to choose the number of generation, which is detailed in the next section.

IV. EVOLVING WALL FOLLOWING BEHAVIOURS

The experiment was conducted to find out what are the effects that can be seen from the results of the GP runs when more complex wall shapes are introduced. The experiment involves a robot moving in a world that consists of a 16 x 16 cells map. The robot has eight sensors that detect any obstacle placed in cells all around it. The outer cells represent the walls of the room and cannot be occupied by the robot. The inner cells adjacent to the walls signify the cells where the robot is able to pick up fitness points. The robot is free to occupy the rest of the cells. For each iteration within a generation, the robot is randomly placed within the allowable cells. In order to have a repeatable experiment, we may place the robot on any allowable cell at our discretion instead of being randomly placed. The experiments were run subsequently on five different wall types. Each of the wall type is added with an extrusion starting with none for the first wall and ending with four extrusions for the fifth wall (see Figure 1). These experiments were conducted to see whether the addition of complexities on the walls would affect the GP algorithm adversely. This may show the scalability of the GP algorithm for this problem domain.

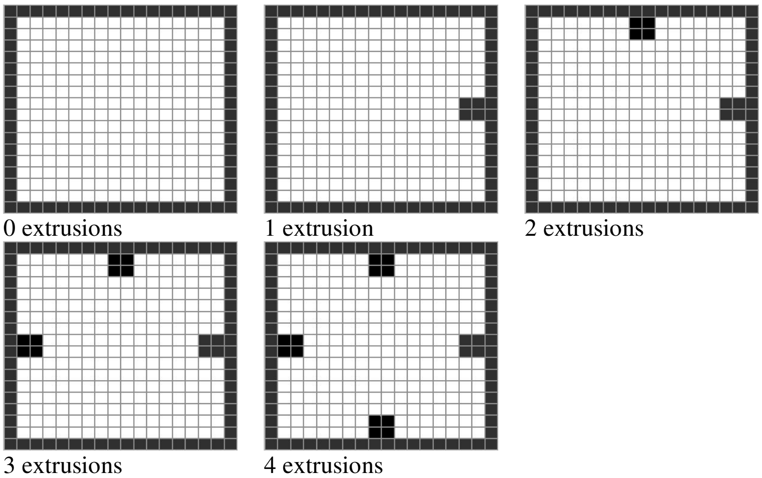

Figure 1: Five different wall types used in the experiments

In our GP implementation, the sensor terminals, connective functions and action terminals are defined here:

- The sensor terminals are *n, ne, e, se, s, sw, w*, and *nw*. A sensor terminal returns 0 when there is no obstacle located at the particular sensor cell location around the robot and returns 1 if there is an obstacle.
- There are four connective functions as follows:
  and(x, y) = 0 if x = 0; else y
  or(x, y) = 1 if x = 1; else y
  not(x) = 0 if x = 1; else 1
  if (x, y, z) = y if x = 1; else z
- The action terminals include
  north moves the robot one cell up in the cellular grid
  east moves the robot one cell to the right
  south moves the robot one cell down
  west moves the robot one cell to the left

These action terminals do not use or return any values to the calling program. However, they have effects that move the position of the robot. The execution of any of these action terminals causes the evaluation of the program to terminate. However, a terminal must return control to the calling program within the program tree. Therefore, we implemented the criterion for this application by specifying a termination flag when program control is passed to one of the action terminal for the first time. Subsequent calls to these terminals would be ignored and thus have no effect on the robot's movement.

Objective	To evolve a wall following robot behaviour for walls with increasing complexity	
Primitives	Actions	north, east, south, west
	Sensors	n, ne, e, se, s, sw, w, nw
	Connectives	if, and, or, not
Fitness Cases	10 iterations of 60 program evaluations. Room 1: raw fitness = 60 * 10 = 600 Room 2: raw fitness = 64 * 10 = 640 Room 3: raw fitness = 68 * 10 = 680 Room 4: raw fitness = 72 * 10 = 720 Room 5: raw fitness = 76 * 10 = 760	
Selection	Tournament of 7	
Hits	The number of time the robot moves into cells placed near the walls of the room. The robot is not awarded a fitness point for moving into a cell already visited	
Parameter	Population=1000 G=100 10 <= Program size <= 100	
Success Predicate	Room 1: 600 hits Room 2: 640 hits Room 3: 680 hits Room 4: 720 hits Room 5: 760 hits	

Table 1: Table for the wall-following experiments

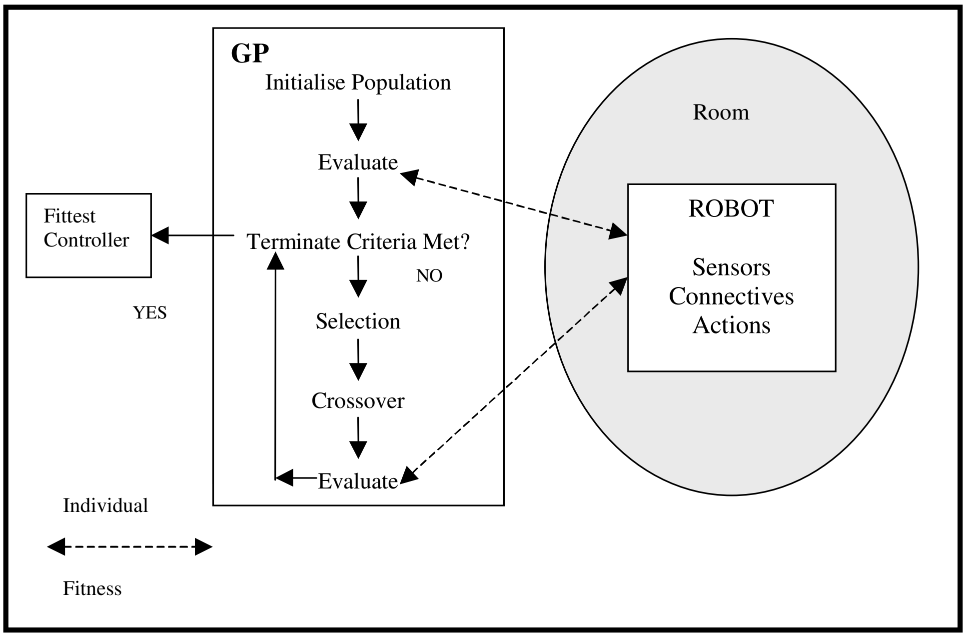

Figure 2: Simulation Architecture

The maximum generations for each GP run is set to 100. The population is constant, and is assumed with a population size of 1000 individuals. The maximum nodes created for each individual are set to 100, and the maximum depth of each individual tree is set to 10. The chosen method for initialising the tree structure for generation 0 is a combination of the "full" and "grow" method, which is called "ramped half and half" method. This means that half of the trees are generated using the "full" method and half using the "grow" method. The genetic operator used is crossover with no mutation. The parents' selection method is tournament selection with a size of 7. The termination criterion is met when 100 generations have passed or when the fitness hits are equal to specific hits criteria for each room type (see Table 1), whichever comes first.

V. INITIAL EXPERIMENTAL RESULTS

GP was able to evolve control programs for all of the room types. Nevertheless, the results of the experiments in terms of number of generation to successfully evolve the wall-following behaviour are counter-intuitive. We have expected that due to the increased complexity of the environment, the room with more extrusions should be more challenging for the GP algorithm. In actual fact, the initial results that we have collected show that this initial assumption is only true for the GP runs of room type 0, 1 2, 3 and 4. For room type 5, the solutions converged within fewer generations than both room types 3 and 4. This could be because the room types 3 and 4 were "stuck" at local minima before finally converging to a solution. If we look at the graph in figure 3, we could see that the plots for the standardised fitness of room 4 and 5 fell suddenly from a stable but not optimal fitness values to the optimal solution. The experiments' result so far does not reveal any cause for us to suspect that the deformation of the room wall in this way affects our GP implementation. We can deduce that the shape of the rooms selected might not be "hard" enough for the GP algorithm to be stymied. At this stage, the main influence on the results comes from the well-known problem of local minima. However, the GP algorithm managed to evolve programs that converged at the optimal requirements in all cases.

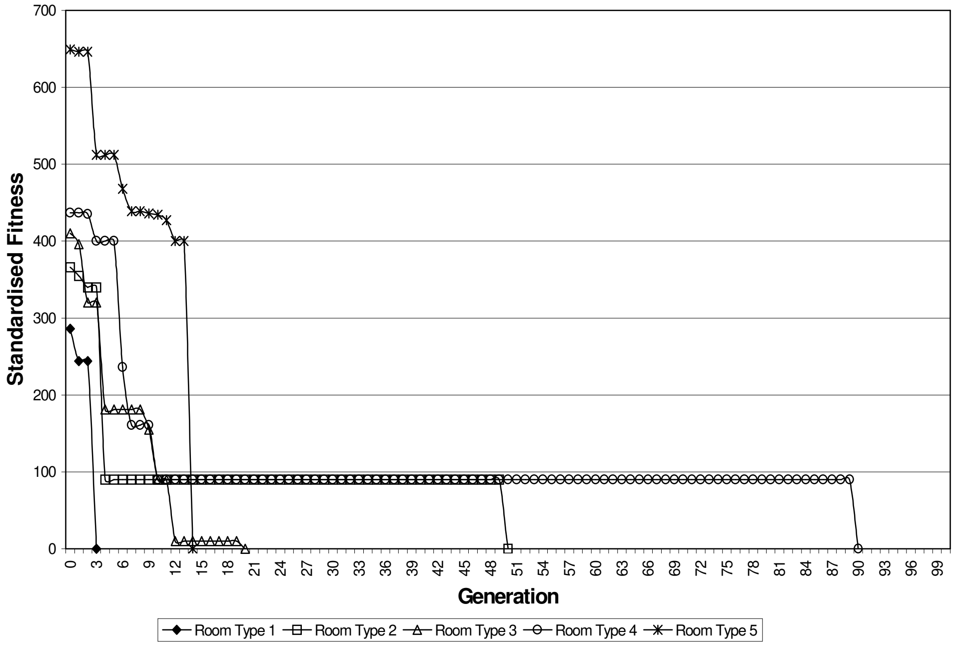

Figure 3: Comparison between 5 GP runs on different room types

Further investigation into the source of the local minima problem reveals an insightful cause. It seems that the GP algorithm was having difficulty in evolving a controller for a particular navigation problem. The problem seems to be at the point where the robot approaches a corner of the wall and then proceeded to approach the next extrusion without following the wall into the corner (see figure 4). That path is the easiest path for the GP algorithm to evolve rather than having to navigate to a corner whereby the robot needs to navigate another corner when it comes to the second extrusion on the wall. This usually happens when the individuals in the population have reached a relatively high fitness. This seems to suggest that the population is affected by this problem later on in the evolutionary process when the population is saturated with highly fit individuals that have evolved this type of controller. When we look at the GP run for room type 3 in figure 5, and plot the fitness graph for the best individual with the mean of the population, we can see more clearly that most of the population has converged on a local optima. Towards the end of the GP run, we see that the fittest individual was able to escape the local optima and the graph climbed. However, the graph for the *generation average fitness* did not go up, which shows that most of the individuals within the population was still trapped in the sub-optimum solution. The result shows that it is difficult but not impossible for the GP algorithm to escape the local minima.

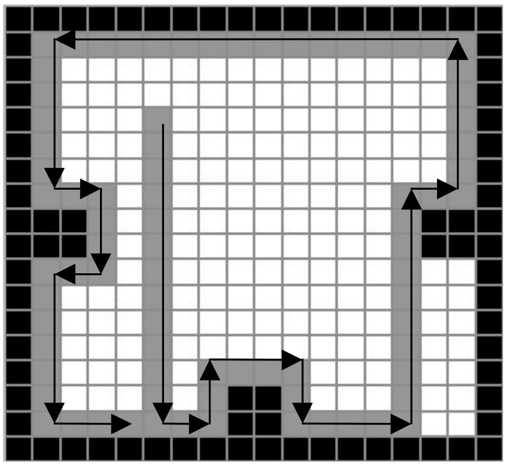

Figure 4: An Example of the Local Minima Problem

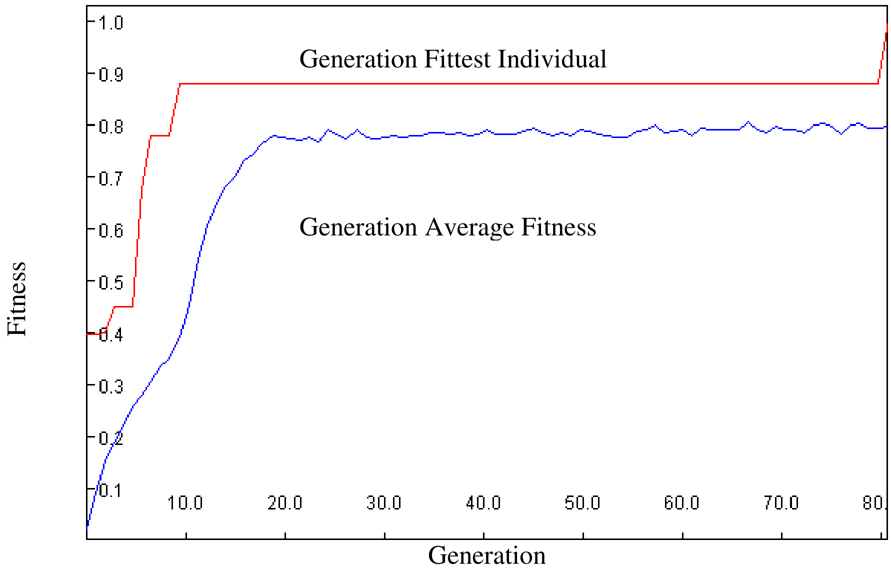

Figure 5: GP run for Room Type 3

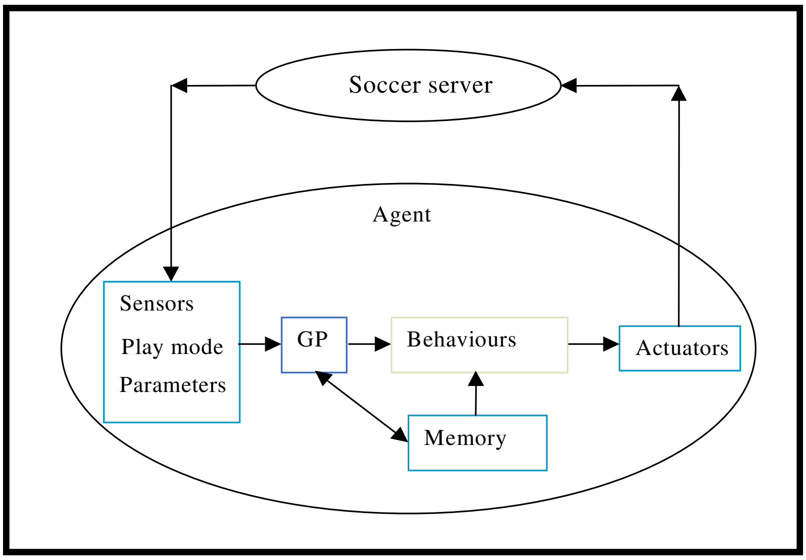

Figure 6: Augmented Soccer Agent Architecture

VI. CONCLUSIONS AND FUTURE WORK

Our wall-following experiments show that GP was able to evolve wall-following behaviours for the robot to navigate five different types of rooms successfully. The results also indicate that even with four extrusions being added to the walls of the room, GP did not have much difficulty in evolving wall-following behaviours for the robot. In these experiments, the expected outcome of the GP algorithm needing longer evolutionary time for more complex wall shapes did not appear. However, we noted the problem of local minima appearing during the GP run of two of five room types. Therefore, in our future implementations, careful consideration would be placed on the identification and avoidance of the local minima problem.

Based on the experimental results described above, we are currently enhancing our GP implementation for evolving a good goalkeeper agent in the RoboCup domain [6][7]. Figure 6 shows our initial proposed architecture for incorporating the GP module into our Essex soccer agent architecture [8]. Our final goal is to evolve an entire team of soccer playing agents. The applications of GP in the RoboCup domain are several magnitudes more difficult than our current experiments involving the wall-following behaviours. It is essential for us to investigate further whether GP could be scaled to meet such a challenge.

REFERENCES

[1] Bhanzaf W., Nordin P., and Olmer M., Generating Adaptive Behaviour for a Real Robot using Function Regression within Genetic Programming. Koza JR, Deb K, Dorigo M, Fogel DB, Garzon M, Iba H and Riolo, Rick L., editors. 35-43 1997: Proceedings of the 2nd Annual Conference, Stanford University, 1997.

[2] Daida J., Ross S., McClain J., Ampy D. and Holczer M., Challenges with Verification, Repeatability, and Meaningful Comparisons in Genetic Programming. Koza JR, Deb K, Dorigo M, Fogel DB, Garzon M, Iba H et al., editors. Genetic Programming 1997: Proceedings of the Second Annual Conference. San Francisco, CA: Morgan Kaufmann Publishers, Inc., 1997.

[3] Dain RA., Developing Mobile Robot Wall-Following Algorithms Using Genetic Programming. Applied Intelligence 1998; 8(5):33-41.

[4] Fraser AP., GPC++: Genetic Programming in C++. http://www.cs.cmu.edu/afs/cs.cmu.edu/project/airepository/ai/areas/genetic/gp/systems/gpcpp/0.html, 1995.

[5] Goldberg D.E., Genetic Algorithms in Search, Optimisation, and Machine Learning. Reading, MA: Addison-Wesley, 1989.

[6] Hu H. and Gu D., Reactive Behaviours and Agent Architecture for Sony Legged Robots to Play Football, International Journal of Industrial Robot, Vol. 28 No. 1, ISSN 0143-991X, January 2001.

[7] Hu H., Kostiadis K. and Liu Z., Coordination and Learning in a team of Mobile Robots, Proceedings of the IASTED Robotics and Automation Conference, ISBN 0-88986-265-6, pages 378-383, Santa Barbara, CA, USA, 28-30 October 1999.

[8] Kostiadis K., Hunter M. and Hu H., The Use of Design Patterns for Building Multi-Agent Systems, IEEE Int. Conf. On Systems, Man and Cybernetics, Nashville, Tennessee, USA, 8-11 October 2000.

[9] Koza JR., Genetic Programming: On the Programming of Computers by Means of Natural Selection. Cambridge, Massachusetts: MIT Press, 1993.

[10] Langdon WB., Genetic Programming and Data Structures: Genetic Programming + Data Structures = Automatic Programming! Kluwer Academic Publishers, 1998.

[11] Mataric MJ., A Distributed Model for Mobile Robot Environment-Learning and Navigation. AI-TR-1228. MIT Artificial Intelligence Laboratory technical report, 1990

[12] Punch B., Genetic Programming System. http://garage.cps.msu.edu/software/lil-gp/lilgp-index.html. 2000.

[13] Ross S. J., Daida J. M., Doan, C. M., Bersano-Begey and McClain, J. J., Variations in Evolution of Subsumption Architectures Using Genetic Programming: The Wall Following Robot Revisited. Koza JR, Goldberg DE, Fogel DB, Riolo RL, editors. 191-199. 1996. Cambridge, MA, The MIT Press. Genetic Programming 1996: Proceedings of the 1st Annual Conference, Stanford University, July 28-31, 1996

[14] Singleton A., GPQUICK: Simple GP system implemented in C++. http://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/areas/genetic/gp/systems/gpquick/0.html, 13-2-1995.