Term Project Report for AOSS 499 Directed Study
A GP-GA Hybridization System Using a Parasitic Symbiosis Model for Evolving Wall Following Robot with Parsimonious Sensor Set

Shuangyu Shawn Chang
Faculty Adviser: Jason M. Daida
The University of Michigan
April, 1997

Abstract
Previous work has demonstrated that GP can evolve successful solutions to the wall following robot problem. One observation shows that optimal solutions seem to need only a subset of the twelve available sensors. Based on a parasitic symbiosis model, a GP-GA hybridization was created to produce robots with limited number of effective sensors. GP is used for robot evolution as in previous works; GA is used to evolve parasites that will infect GP robot individuals based on a tag-matching mechanism and knock out some of the infected robot's sensors specified by a mask in the parasite's chromsone. Experimental data from 61 trials shows that this symbiosis model produced robots with much more parsimonious sensor sets but better performance than robots evolved without parasite infections.

Introduction
The wall following robot is a well known problem in GP, and previous work has demonstrated that GP can evolve successful solutions to this problem, given a small function and terminal set. One interesting observation in [Ross] shows that optimal solutions seem to need only a subset of the twelve available sensors. If sensors are expensive components of a robot, it would become very cost-effective to employ such lean individual with a few sensors but having comparable or even better performance than the all-sensor equiped individuals. The [Ross] also shows that the sub-optimal solutions, which are the majority of the solution space, used a wide range of sensors. These two results suggest us that if we have a way to control the number of sensors used in the general population of GP robot individuals, we might be able to find more optimal solutions and converge faster to individuals with only a few sensors. Thus, one primary goal of this project is to achieve efficient usage of sensors and produce more parsimonious robot with comparable or even better performance by some means of controlling the usage of sensors.

Genetic programming is an adaptive and evolutionary process. The solution space is very large and the optimal solution pattern is not very predicatible. It is a very difficult problem to find a good static mean to limit the sensor usages, since this problem itself must have a very large solution space and it is not quite independent of the host robots. Therefore, it is conceivable to also use an adaptive method to find a good mean to limit the sensor usage. As we now have two co-existing adaptive processes, the evolutions of the individual robot and the sensor usage control method, we can model them as a parasitic symbiosis [Daida 1]. In this model, the GP robot individuals are hosts, and the sensor usage control method is the parasite. Note that in this parasitic symbiosis model, deviating from the conventional defination of parasites, our parasites do not neccessarily benefit at the cost of the host robots. Rather, our intention is to see that the parasites survive and grow fitter without being aware of the fitness

condition of the host robots, and host robots become more parsimonious on the sensor usage and perform the same as or better than that before the infection of parasites. Thus, we are looking for a special parasitic phenomenon, in which both the hosts and parasites are benefiting each other unintentionally. This may seem unnatural, but in fact, it's perceivable that parasite can be beneficial. For example, as more parasites hit a body, the body will develop a more robust immune system that will fight against harmful parasites; also, as parasites alter some of the physical structure or behavior of the hosts, the host population becomes more heterogeneous and more likely to survive an epidemic since only some individual will die, while other homogeneous population is prone to extinct upon a single disease. There is an attached condition to the two exmples above, which is that the infection has to be under a certain threshhold so that the host population will not be wiped out entirely. This is often referred to as "optimal brain damage" [Russel].

To model such a parasitic symbiosis, a general scheme is developed to represent the parasite with a mask and a tag. The mask is used to decide which sensors of an infected robot individual are knocked out; and the tag is used to decide the mapping of the parasites and the robot hosts, i.e. which parasite infects which robot. Several mapping scheme of the parasites and robot hosts have been proposed during the early development stage. A parasite can infect a randomly chosen individual; a parasite can infect a robot with a certain fitness rank (ordered by performance scores) that matches the parasite's tag; or, a parasite can infect an individual with a certain inherent characteristic, such as the robot program tree size, number of sensors used, etc. If we are to map a parasite to a robot randomly, the parasite population will not be able to evolve effectively based on its interaction with the robot population since random mappings are not likely to become an intrinsic characteristic to be passed on generationally. A mapping based on robot fitness rank is also not desireable as it imposes a too strong association between the host health, which is indicated by the robot performance, and the fitness of the parasite. As mentioned previously, a parasite should not concern about hosts' fitness conditions regarding to the decision of which host to infect. Such an awareness of the parasite would result in a biased selective infection of the hosts based on some extrinsic health condition, which is not very justifiable in nature. Rather, we would like to see a parasite infects a certain kind of robots distinguished by some inherent characteristics regardless the robots' fitness scores. In our implementation, we use the presented sensor pattern in the robot's program tree as the mapping target for a parasite's tag. This will be explained in detail.

Having two different species in a symbiosis, it is possible to simulate the evolution processes of the parasites and the robots with different techniques. It's been demonstrated that the GP is an effective way to evolve the wall following robot individuals. However, for the parasite individuals, GA can be a better technique, since both the mask and tag can be represented with simple binary strings of certain lengths. It is also very interesting to show that two interacting symbionts can indeed be modeled with two different evolution systems. Thus, it would be more evident to say that the principles are the deciding factor regardless of implementation methods. It might also produce some interesting examples of GP-GA hybridization useful for other subjects.

Scopes and Limitations

The works described in this report are continuation of previous works done by the UM-AC-ER-Animats group. The wall following robot implementation follows descriptions in [Koza]. Minor modifications are described in [Ross]. The GP robot code is based on a C implementation[Daida 2] of the original *LISP* code by Koza. The GP kernel is *lil-gp* [Zongker] by Michigan State University. The GA kernel is *LIBGA* [Corcoran] by the University of Tulsa.

Modifications have been made to the random number generator code in both *lil-gp* and *LIBGA*. See Appendix A for a detailed explaination.

The coding and testing are done on the Sun SparcStations running SunOS 5.5.1 (Solaris).

Implementation

The wall following robot has 12 sensors, each covering an area of 30'. The room has an irregular shape with protrusions on the south and east walls. The objective of the wall following robot problem is to evolve a computer program that allows a simulated robot to move along the perimeter of the room. Table 1, taken from [Ross], shows the key elements of problem specific code in the GP robot implementation (before the consideration of parasite infection).

Table 1: Wall following robot problem specific code

Terminal Set Twelve sonar measurements[S00, S01..S11]; Derived minimum of measurements [SS]; minimum safe distance and preferred edging distance from wall [MSD, EDG]; Primitive motor functions [MF, MB, TR, TL]

Function Set If-Less-Than-Or-Equal-To macro [IFLTE(arg1,arg2,arg3,arg4)] (i.e., IF(arg1 <= arg2) then arg3, else arg4); Connective function [PROGN2(arg1, arg2]] (i.e., eval arg1 return eval arg2)

Fitness Function Fitness cases: one fitness case Koza's irregular room with an initial starting location near the middle of the room(13.8, 13.8) 1 and an initial facing direction of south(270°)

Hit: robot touches a 2.3 square foot tile along the wall of the room

Raw fitness: number hits in 400 time steps

Standardized fitness: total number of wall tiles minus the number of hits

Success predicate: 56 out of 56 hits

The genetic algorithm implementation of the parasite is done by representing each parasite with a 24-bit chromsone. The first 12 bits in a chromsone is the tag, and the next 12 bits is the mask as shown in Figure 1.

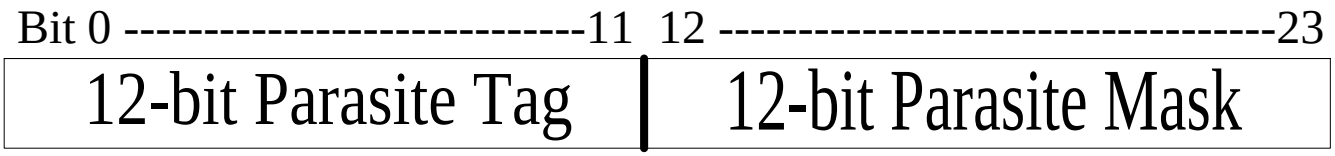

Figure 1: GA parasite tag and mask representation

The 12-bit sensor mask determines which of the 12 sensors of an infected individual are knocked out, where a bit of 1 means knock-out, and a bit of 0 means sensor intact. The 12-bit tag determines which GP individual is to be infected by this parasite. Each individual in a GP population is parsed and a 12-bit tag is generated in the following way:

If sensor X is present in a GP robot individual's program tree one or more times, bit X of the tag is set to 1; otherwise, bit X is set to 0, where X is between 0 and 11.

Then, the tag in each GA parasite is compared against the just computed tag of each GP robot individual. If there is a match, the GP individual is set to be infected, and evaluated for number of hits with sensor knock-out effect. And, we add the number of 1-bits in the GA sensor mask to a cummulative sum. This sum is used as an adjustment for GA fitness evaluation in the next GA evolution. Thus, GP hits do not affect GA fitness computation directly. However, a GA parasite that infects more GP individual will get higher fitness score.

If a GP robot individual has been infected by multiple GA parasites, or by the same or different GA parasites multiple times, we use the maximum number of hits among all those evaluations as the final hits for this GP robot individual.

When evaluating a GP robot individual, a copy of good sensor readings is saved before the knock-outs take effect. This saved copy is used in determining if a "move" (MF, MB) should be made, i.e. if the robot will bump into the wall. The readings for the knocked out sensors can be set to an arbitrary but consistent constant, zero in our experiment. Using the saved copy of sensor readings is to enhance the performance of GP robot individual. Since in the GP robot code, the MF and MB (move forward and move backward) functions each uses 6 of 12 sensor readings to determine if the move can be made without jepardizing the robot to bump into a wall. This checking is done by comparing each of the 6 forward or backward sensor readings with a minimum safe distance (MSD), and it's considered a bump will happen if any one of the readings is less than the MSD. When sensor knockout takes effect, it is very likely no move is ever made since the knocked out sensor reads zero always, so the GP performance is likely to be very poor. So according to the saved copy of sensor readings, if the robot is too close to the wall, a move will not be made and the robot will stand still. If the robot does not change position for a certain number of consecutive steps, the current run terminates to save computational time. This scheme actually simulates that the robot is really "blind" when making a MF or MB move. It tries to make the move, and if it bumps into a wall, it stays at its original position. This scheme does not violate the claim of not using the knocked out sensors, since if we do not use the saved copy of readings, we would need to compute how far we are to the wall according to the robot's current location, and that result would be just the same as using the saved copy of sensor readings, except more computational effort is required.

Before we can run the program, several parameters have to be set in the *ga-test.cfg* for GA and *input.file* for GP. Once the executable is invoked, the GP and GA will be initialized according to the options set in those two configuration files. Individuals in the initialized GA pool are copied to a global structure containing the GA parasite masks and tags, and a variable to keep track of the cummulative sum mentioned previously. This global structure is then accessed by the GP during its evolution process. When infection occurs, GP individuals are re-evaluated with sensor knock-out effect and the infecting GA parasites' cummulative sums are updated. After a specified number of runs of GP, five generations in our case, GA is invoked. The global structure is then used again to pass back to GA each parasite's cummulative sum. This cummlative sum is used in evaluating a GA parasite's fitness in the following way:

GA parasite fitness = Number of 1-bits in mask * Cummulative sum / FITNESS_SCALE

The FITNESS_SCALE is a constant used to scale down the final fitness score for each individual proportionally since the variations in fitnesses of parasites are too large. The objective of GA is to maximize this computed parasite fitness score. The cummulative sum of a GA parasite individual is

used only once during the first generation of a GA evolution process. After the GA finishes its specified number of generations, we again copy the GA parasite pool to the global structure and run the GP codes.

The code has been partially verified by implanting some GP robot solutions from previous works, and same final results have been obtained with the implanting non-effective dummy GA parasite.

Running Statistical Experiments

Due to the time and resource limitation, only two experiments with 61 runs each have been conducted. All the test runs use same parameters except the random number seeds. Only non-ADF (no automatically defined function) architecture were tested. Table 2 shows the GP parameters used in the experiments.

Table 2: GP Parameters used in statistical Experiment I & II

Population Size:	1000
Maximum Generation:	57
Initial Population Generation:	Ramped Half-and-Half
Maximum Initial Depth:	6
Maximum Depth after Crosssover:	17
Selection Method:	Fitness-Overselect
Probability of Crossover:	70%

Experiment I was run allowing the GA parasites to infect GP robot individuals and use the sensor knock-out effects in GP fitness evaluation. Table 3 shows the GA parameters used in Experiment I.

Table 3: GA Parameters used in statistical experiment I

Population Size:	4000
Maximum Iterations:	400
GA Type:	Steady-State
Selection Method:	Rank-Biased with bias=1.8
Crossover Rate:	1.0
Mutation Rate:	0.0
Replacement Method:	By rank

To produce control data, Experiment II was run not allowing GA parasites to infect GP individuals. Testing runs were performed distributedly on a number of Sun SparcStation 20 and Ultra. GNU gcc compiler was used to compile the C codes. Run time varies between 12 minutes and 8 hours for test runs in Experiment I, and between 8 minutes and 20 minutes for test runs in Experiment II. The great variation in run time is due to the low running priority when some test run processes were running in the backgroud of some busy machines. Mathematica was used in visualizing solutions. Thanks to the previous works in the Mathematica visulization code done by the AC-ER group.

Results and Discussion

The statistical data experiment has given encouraging results in five major areas: robot performance, parasite fitness, robot fitness modality, robot tree size, and number of effective sensors used in a robot. Successful solutions evovled in Experiment I are shown in Appdendix B. Additional experimental data and graphs in a spreadsheet format is presented in Appendix C.

Robot Performance

Table 4 presents the performance results of both two experiments. As shown in the table, the number of successful individuals, number of individuals scored in top 10% of possible score range, and number of individual scored in top 25% of possible score range, all have at least about 50% improvements in Experiment I over Experiment II. The performance results in Experiment II are consistent with the reported performance of non-ADF experiment of the wall following robot in [Ross]. The performance improvements are significant. If the test run results are proven valid, they are strong enough to show that the GA parasite and GP robot symbiosis model is beneficial to the robot hosts in term of fitness.

Table 4: Performances of Experiment I & II

Experiment	Trials	Successes	Success Rate	Numbers of Ind. Scored over 50 (top10%)	Number of Ind. Scored over 41 (top 25%)	Average Score
I	61	8	13.11%	20	38	45.49
II	61	5	8.20%	13	26	43.15

Parasite Fitness in Experiment I

The final scaled fitness score of the best GA individual in each trial has been examined. The fitness scores range from 0.064 to 112.65 with a mean of 36.52 and a median of 24.60. No significant pattern or trend has been found in these fitness values. Future works may look into the association between GP robot performance and corresponding GA parasite fitness score.

Robot Fitness Modality

Figure 2 shows the histograms of the number of hits of the best-of-run GP robot individuals in both Experiment I and II. The horizontal axis of each histogram refers to the number of hits. The vertical axis of each histogram refers to the frequency of runs whose best-of-run individual scored the specified number of hits.

From the two histograms in Figure 2, one can see a clear modality shift from Experiment II to Experiment I. In Experiment II, there is a prominent grouping around 34 to 43 hits. But, in Experiment I, there is no such a strong grouping pattern anywhere on the histogram, and number of runs in the low hits area are leveled off and more number of hits appear in the high hits area compared to Experiment II. This change in modality suggests us that with the presence of GA parasites, a previous sub-optimal modality has been reduced, and trials tend to produce best-of-run robots that have high hits.

Robot Tree Size

The size of the program tree for a GP robot individual can be best described with two parameters, the number of nodes in the tree and the depth of the tree. Table 5 summarizes the two parameters of the best-of-run individuals in both Experiment I and II. The result has shown that, on average, robots produced by GP with GA parasite infections tend to have a few more nodes than the robots produced without parasite infection. However, the depth of the tree for an average robot produced with parasite infection effect is slightly smaller than that of non-infected robot. Combining the two observation, we can conclude that the infected robot tend to be a little chubbier than the non-infected. Another observation shows that robots with very small trees usually have lower number of hits, but this is not always true.

Table 5: Robot tree sizes of best-of-run individuals in Experiment I and II

Experiment	Average Number of Nodes	Minimum Number of Nodes	Maximum Number of Nodes	Average Depth	Minimum Depth	Maximum Depth
I	194.7	9	649	9.4	2	17
II	161.1	15	166	9.8	2	17

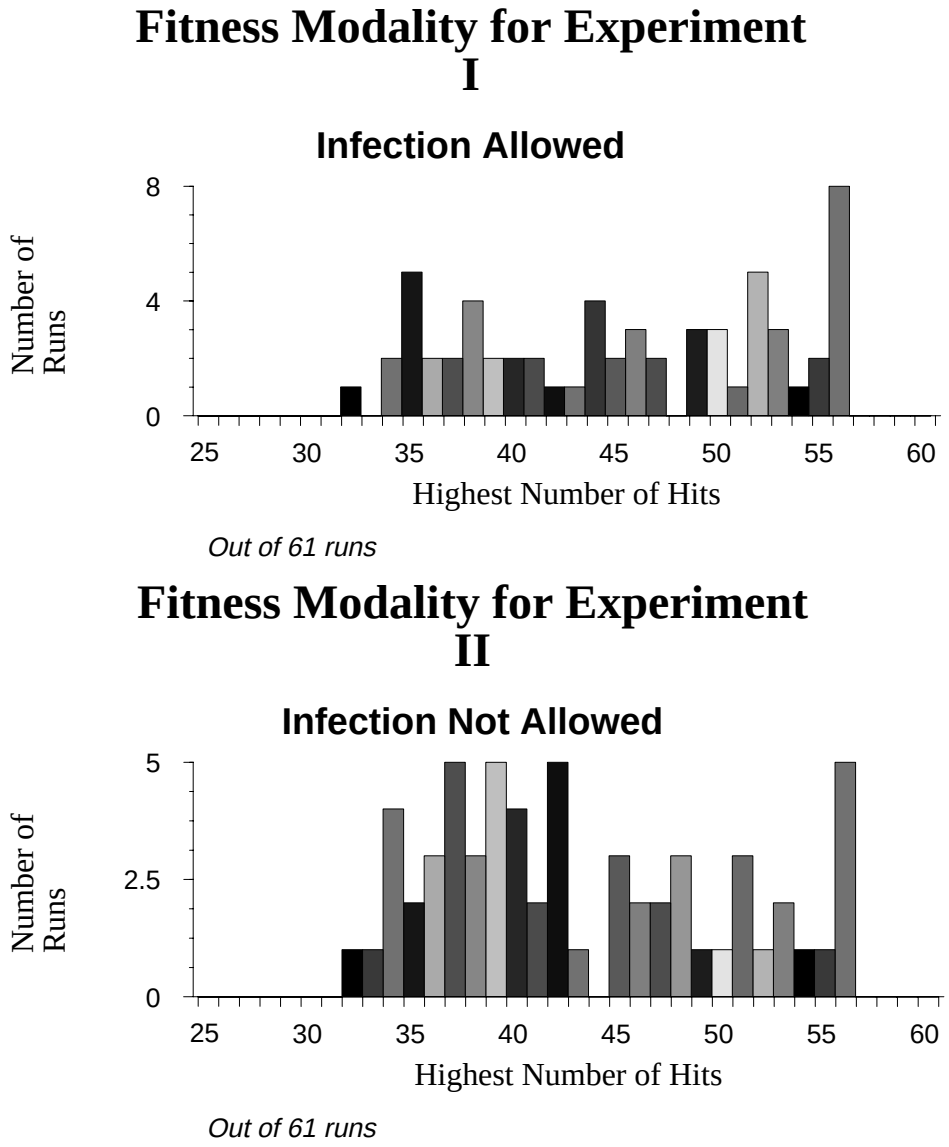

Figure 2: Hits histograms.

Number of Effective Sensors

One of our primary goals of doing this project is to find a way to produce parsimonious robots with only a few working sensors but being able to perform the same as or even better than regular robots. Thus, the results described in this category is very important in evaluating the project. Table 6 presents the results obtained in measuring the number of effective sensors for each best-of-run individual in both Experiment I and II. Note that in Experiment I, the number of effective sensors is calculated by applying the parasite mask to the corresponding infected robot, and counting only those sensors present in the robot's program tree and not having been knocked out by the parasite. The number of effective sensor for an Experiment II robot individual is simply the number of sensors present in the robot's program tree. It is obvious that the best-of-run robots produced in Experiment I are generally much leaner than their counterparts in Experiment II in terms of number of effective sensors. There is a noticeable but weak pattern showing that extremely parsimonious robots tend to have fewer hits than those equipped with an around-average number of effective sensors. However, robot individuals with large number of effective sensors do not appear to outperform the rest robots significantly. Another observation worth mentioning is that most of the robots produced in both Experiment I and II tend to retain sensor S02 and S03 in their program trees. This is because many functions use the readings of S02 and S03 as the return value to have closure.

Table 6: Number of effective sensors for each best-of-run robot individual

Experiment	Average Number of Effective Sensors	Minimum Number of Effective Sensors	Maximum Number of Effective Sensors
I	6.5	2	12
II	10.3	3	12

Figure 3 shows a histogram of the number of effective sensors for each experiment. The horizontal axis of each histogram refers to the number of hits. The vertical axis of each histogram refers to the frequency of runs whose best-of-run individual has the specified number of effective sensors.

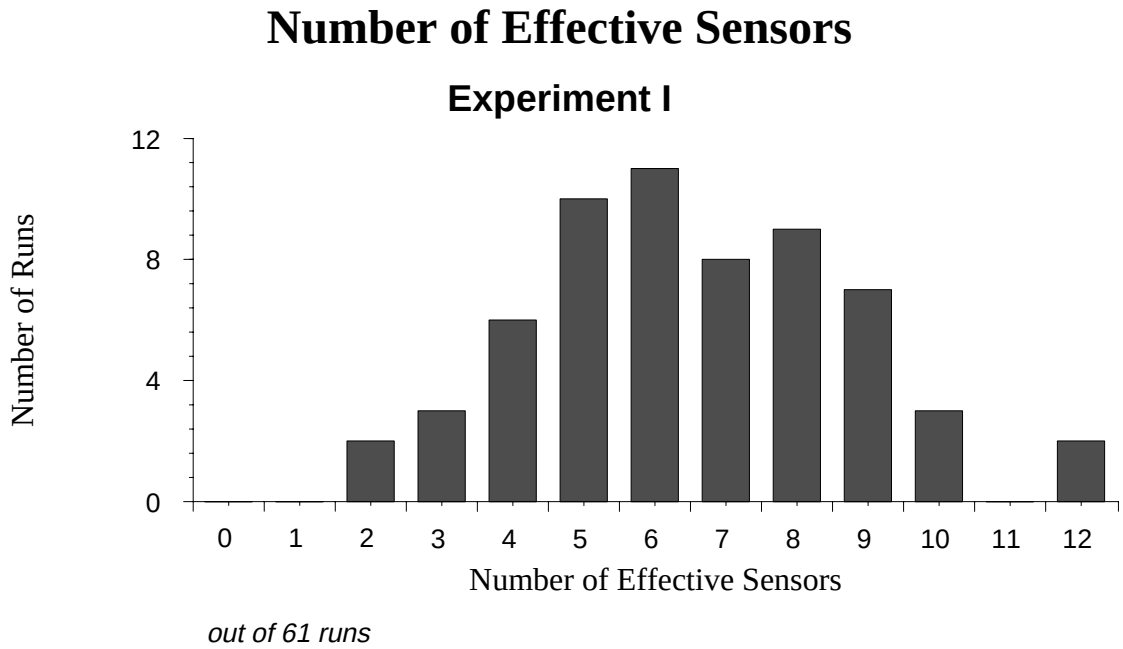

Figure 3a: Histograms of Number of Effective Sensors for Experiment I

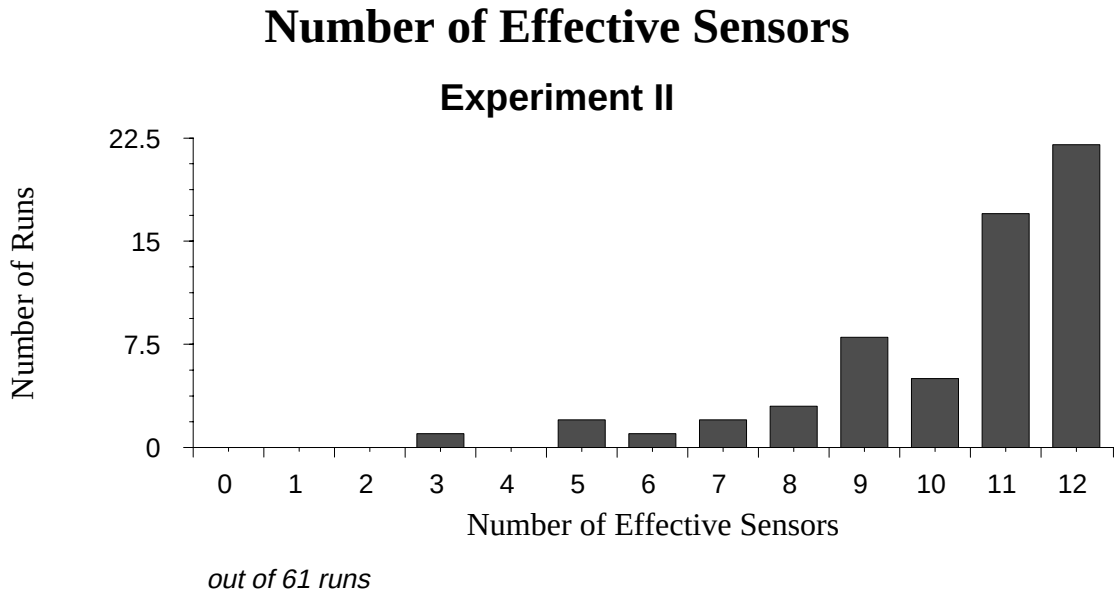

Figure 3b: Histograms of Number of Effective Sensors for Experiment II

Conclusions

This project was aimed at finding a way to evolve wall following robots with efficient usage of sensors while having comparable or even better performance by some means of controlling the usage of sensors. While trying to achieve this goal, a number of encouraging results have been obtained:

- A method for controlling the usage of robot sensors is found, which uses a parasitic symbiosis model to represent the robots as host symbionts and sensor controlling masks as parasites.
- A GP-GA hybridization system is successfully created and implemented, where GP is used to evolve wall following robot individuals and GA is used to evolve parasite masks and tags to infect host GP robots for sensor knock-out effects.
- A set of controlled statistical data experiments are completed, and the results have shown that robots evolved under the infection of sensor-knocking parasites have significantly better performances, more optimal fitness modality, slightly bigger program trees, and much leaner effective sensor sets, than their counterparts evolved without the infection of parasites.

Future work recommendations:

- Further validation on the correctness of the assumptions and implementations described in this report and actual program source codes.
- More experimental trials to obtain a larger set of data.
- Create new fitness evaluation conditions, such as new room configuration, or different initial position or direction for the robot, to test the robustness of the robots produced under infection.
- Investigation on the GA parasite evolution process, association between host performance and parasite fitness, and fine tuning of GA parameters to achieve optimal parasites that benefit both hosts and parasites themselves.
- Experimental trials on different architectures with ADFs, and similar trials using *ran2* and *ran3* random number generators.

Bibliography

Corcoran, A. L. and R. L. Wainwright, "LIBGA: a User-Friendly Workbench for Order-Based Genetic Algorithm Research," to appear in the *Proceedings of the 1993 ACM/SIGAPP Symposium on Applied Computing*, ACM Press: New York, 1993.

Daida[1], J.M., C.S. Grasso, S.A. Stanhope, and S.J. Ross, "Symbionticism and Complex Adaptive Systems I: Implications of Having Symbiosis Occur inNature," to appear in Evolutionary Programming V: *Proceedings of the Fifth Annual Conference on Evolutionary Programming*, Cambridge: The MIT Press, 1996. pp. 177−186. Invited Paper.

Daida[2], J.M., S.J. Ross, J.J. McClain, D.S. Ampy, and M.R.Holczer, "Challenges with Verification, Repeatability, and Meaningful Comparisons in Genetic Programming," to appear in Genetic Programming 1997: *Proceedings of theSecond Annual Conference.* J.R. Koza, et al. (eds.). San Francisco: Morgan Kaufmann Publishers, 1997.

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

Ross, S.J., J.M. Daida, C.M. Doan, T.F. Bersano-Begey, and J.J. McClain, "Variations in Evolution of Subsumption Architectures Using Genetic Programming: The Wall Following Robot Revisited," to appear in Genetic Programming 1996: *Proceedings of the First Annual Conference*. J.R. Koza, D.E. Goldberg, D.B. Fogel, and R.L. Riolo (eds.). Cambridge: The MIT Press, 1996. pp. 191-199.

Russell, S. and P. Norvig, *Artificial Intelligence - A Modern Approach.* New Jersey: Prentice Hall, Inc., 1995. pp. 379.

Zongker, D. and W. Punch, lil-gp code C, http://isl.cps.msu.edu/GA/software/lil-gp/

Appendix A. Problems with Random Number Generators and Solutions

During code implementation of the project, I have encountered problems with random number generators included in the *lil-gp* kernel. Following is a description of the how the problem was spotted, analyzed, and resolved.

The problem with the random number generator implemented in the *random.c* file of the GP robot codes was first noticed when a "bus error" occurred during a GP run. The cause of the "bus error" was found to be the use of a system clock generated random number seeds. One of such numbers is "857874808", and there are many more numbers would cause the same problem.

To verify that the problem was caused only by the random number generator, an independent program was created by using the *random.c* file and a main function that seeds the random number generator, repeatedly calls the *random_double()* function, and prints out the obtained "random number".

The result of this test shows that the numbers generated by the random number generator was getting bigger and bigger (they were supposed to be between 0 and 1 exclusively), and eventually running out of range of double precision floating point values on the host machine and produced a NaN (Not a Number).

Then, the implementation in random.c file was compared against the Knuth's *ran3* (subtractive method) function in C, which was believed to be a correct implementation. The comparison shows there are a number of differences between the two implementations:

1 MBIG and MSEED are different, where Knuth's uses MBIG = 1000000000,MSEED = 161803398 and *random.c* uses MBIG = 10000000.0, MSEED = 1618033.0
2 Knuth's uses "long" for some data variables, and *random.c* uses "double".
3 Two other lines are not entirely equivalent, which should not cause any crash, but the sequence of random numbers generated will be different.

In the next step, I typed in the Knuth's implementation and ran the same test as mentioned above with the ran3(). When "857874808" was used as seed, the Knuth's implementation did not have any problem.

From all the facts above, I determined that the problem in the *random.c* implementation was caused primarily by two things:

1 It uses "double" instead of long.
2 MBIG is too small.

The following was my explanation of the problem, for which I do not have a concrete proof:

By looking at the codes, you'll see that there is a subtraction operation "mj=ma[inext]-ma[inextp];" followed by "if (mj<0) mj+=MBIG" in both of the two implementations. It's entirely possible for the difference "mj" to be negative or positive with an absolute value greater than MBIG. In either case, the value returned by the random number generator "mj/MBIG" will be out of the range of (0,1). Even though this is true for both implementation, the Knuth's "*ran3*" implementation almost never produces bad result because it uses "long" for "mj","ma[]". On Sun Sparc machines, "long" is 4 bytes, so the greatest positive value represented by a "long" can be about 2.15*10^9, which is about 2.15 times as the MBIG. If the "mj" in the above subtraction overflows the range of the long integer, it will wrap around. This effect, combined with the adjustment of adding MBIG to negative "mj", almost ensures that the result returned by the random number generator will be in the range of (0,1). (But this is not always the case, which I'll explain.)

On the other hand, the random.c implementation suffers from its "double" and small MBIG value. Since "double" on most machines has an incrediblly large range, so it will not wrap around

when its value gets bigger (but still within the range). When a negative "mj" value is big, adding MBIG will not help much. Therefore, it could be very often that the value returned by the random number generator is out of range (0,1), and even becomes diestrous large.

If the above explanations are rational, one should expect that even the Knuth's *ran3* could produce bad result sometimes. Then I ran a series of exhaustive test on the *ran3* trying to find such a bad seed. I first tried from 1 and up to 3,000,000, and testing first 1000 random numbers generated with every seed. This test didn't produce any bad result.

Then I tried very large numbers as seed, 1500000000 and up, testing first 1000 random numbers generated with every seed. Very soon, a bad result came up.
Using "1500002115" as seed, the 54th random number generated by ran3 is -0.034557. This is clearly out of range (0,1). There are also many more numbers that will generate less than 0 or greater than 1 result.

So, at the time, I had the following recommendations for solving the random number generator problem:

- Do not use the *random.c* implementation in the *lil-gp*.
- Use Knuth's *ran3* implementation, but only with seed that are not too big, i.e. not close to MBIG. Hence, the number "seconds" generated by the system *time()* function might be too big to use.

A few days later, I received the following message regarding the random number generator problem:

Numerical Recipes Current Bug Reports

This file lists known or suspected bugs that were reported or discovered after the deadline for the current release of *Numerical Recipes*. Not all the reports listed here are fully validated, so this listing should not be relied on as definitive. All the entries here will be further investigated before the next release. However, users of the current release who encounter bugs may wish to see if their bugs are already in this listing and, if so, whether they have additional information that may be useful for a fix. (If so, we encourage email reports to "bugs@nr.com".) Note that this file does not include bugs already fixed in the current release. If you want information on those, look at these upgrade patch files.

Version 2.06 (including 2.06h):

1. ran3 will behave improperly if it is seeded with a value of idum whose magnitude is greater than 161803398. This has affected users who have tried to seed it with the system clock. The fix is to rewrite the two lines that initialize mj so that mj is initialized to a positive quantity less than MBIG.

According to this message, I have made a simple patch to the *ran3* code and implemented it in the *random.c* file. Also, I have implemented another random number generator *ran2* as an alternative. Both functions are now in the *random.c* as part of the GP-GA robot code.

Appendix B. Successful Solutions of Experiment I