A Discussion on Generality and Robustness and a Framework for Fitness Set Construction in Genetic Programming to Promote Robustness.

Tommaso F. Bersano-Begey
The University of Michigan Artificial Intelligence Laboratory
1101 Beal Avenue Ann Arbor, Michigan 48109-2106
tombb@engin.umich.edu

Jason M. Daida
The University of Michigan Artificial Intelligence and Space Physics Laboratory
1101 Beal Avenue Ann Arbor, Michigan 48109-2106
daida@eecs.umich.edu

ABSTRACT

A significant problem in Genetic Programming consists in ensuring the robustness or generality of the evolved code (its ability to work correctly on never before seen data). We examine approaches attempted so far, and propose a different solution, based on multiple training sets (to discriminate between possible interpretations), augmentation and refinement, which improves on both task specification and distribution of fitness. We then present some preliminary results on its application to a fairly canonical GP problem (wall-following behavior) that has been previously shown to be very brittle. Finally, we discuss issues in using additional information about the generality of individuals.

1. Introduction

In evolving an algorithm, Genetic Programming (GP) can only guess what the actual purpose and required behavior of the algorithm should be. In fact, an essential part of evolving code with genetic programming is to set up training or fitness cases (i.e., the environment in which to test the population) and a fitness function to implicitly encode such information, to "train" and shape the evolution so that only the solution to the intended problem will thrive.

This, however, is often non-trivial, and a perfect scoring evolved individual might turn out to be brittle or too specific (e.g., perhaps just avoiding the required task, or too specific to the data on which it was evolved), especially if the fitness cases or function were poorly designed.

Unfortunately, it is not clear how to properly design fitness sets and fitness functions, and even when a program has been found to lack robustness, it is not clear how to modify the evolution set-up to promote robustness.

This paper discusses previous attempts to tackle these problems, propose a new approach to the problem as well as guidelines to improve the design of fitness cases and fitness functions, and discuss the advantages and significance of such approach. The rest of this paper is organized as follows. In Section 2, we define what we mean by robustness and generality. Then, in Section 3, we discuss previous attempts to promote robustness of code and attempt to analyze their strengths and weaknesses. In Section 4 we introduce the concept of using multiple examples, or 'tasks' to specify a general concept and promote robustness. We then discuss the advantages of our approach and further discuss implementation issues (e.g., using a fitness vector, etc.), suggest the type of problems in which this approach is better suited, and its application for designing fitness functions to further promote robustness. In Section 5 we apply our ideas to a problem which has been shown to be extremely susceptible to brittleness and discuss our results. Finally, in Section 6, we present our conclusions and discuss future work and unresolved issues.

2. Robustness and Generality

We should first specify exactly what we mean by "robustness", by providing the definition which we use throughout this paper. The robustness (as opposed to brittleness) of an individual is its ability to work correctly on data other than the one it was generated on. Given this definition of robustness, the term is actually equivalent to the concept of generality, and we thus use the terms interchangeably throughout this paper. In order to introduce some of the issues which are essential in obtaining generality, we introduce the following related anecdote:

"A popular story in the neural-network community tells of scientists in a military project trying to train a neural network to classify images as containing either tanks or trees. The story goes approximately as follows: the scientists present 20 pictures to the neural network, 10 with tanks and 10 with trees alone. After a sophisticate pre-processing of the images, these are fed in the neural-network and, after considerable training, the neural-network is able to classify each image correctly. However, when this is later tested on other images, the network seems to classify every image as trees, even when it contains a tank. After careful study, the scientists finally resolve the mystery: in all the images used in the training,

those containing trees were always taken in broad daylight, while those containing trees were always taken in a darker setting! Thus, the network had simply learned how to distinguish between light settings rather than recognizing the presence of tanks."

After this example, the significance of choosing unambiguous training cases should be clearer. Unfortunately, the same problem can easily apply to evolutionary computation. As discussed earlier, in fact, the environment and fitness functions are the only way to specify (through implicit encoding) what behavior should be evolved. And in both neural-networks and evolutionary computation the input data (or environment) can be ambiguous and thus accommodate multiple behaviors (many of which are likely to be unwanted, e.g., classifying light from dark settings).

3. Approaches to Robustness

Perhaps with the exception of finding correlation, and rare cases of very specific algorithm discovery (i.e. Koza's symbolic regression, [1], the programs which evolve from a GP run are useful only in relation to their ability to behave correctly under any input combination. If a program were to apply merely to the fitness cases on which it evolved it would be necessary to evolve a new program for every different input combination (just like it is done for linear regression).

Furthermore, to create a set of fitness cases, it is necessary to know the solution for every fitness case beforehand, as a reference for assigning fitness to individuals. Therefore, a successful program would simply find what is already known, or merely a correlation between the fitness cases. For these reasons, the Genetic Programming paradigm is often used in the attempt to produce robust individuals, which are able to perform correctly under any allowed input combination beyond those which they were trained on.

This attempt, however, has not always been successful, producing in many cases (depending in great part on the domain) brittle individuals which only worked on the data on which they were evolved [3]. For this reason, there has been much effort in trying to find a general, domain-independent technique to promote robustness.

Here we analyze the main approches for promoting robustness.

3.1 Fitness Function Approach (Size)

Modifying the fitness function has been shown to have some degree of success in increasing the robustness of evolved programs. While evolving a sorting algorithm, Kinnear noted a connection between size and generality, and found that modifying the fitness function by adding the inverse size to the original score yielded programs which were both smaller in size and more general [2]. In [8] the connection between size and generality of the solution is further discussed and exploited.

In trying to understand this phenomena, a simple approach is to imagine the shorter possible solution which scores perfectly, call it S'. In S' every statement of the code is necessary to obtain a perfect score, so any shorter solution would not be able to score perfectly. However, any solution longer than S' might either be equivalent to S' or contain S', and in its additional length there is an increasing probability of contain additional unnecessary statements, which might be specific of the given environment and thus not general (for an example of this, consider the encapsulating IF statement discussed in the analysis in section 5) or simply wrong. Thus, the class of solutions longer than S' can potentially score perfect and contain many irrelevant statements which happen to work for the given data, but would have increasing probability of misbehaving on other data. On the other hand, the class of solutions which are shorter than S' would not be able to obtain perfect score, and would thus be selected out.

However, depending on the domain and representation choice, it might turn out that S' is itself very brittle, in which case its selection over the longer robust solution would not be a desired event.

The reader can easily verify this possibility by considering an instance in which this is true (e.g., doing linear regression on a set of points which are common both to y = x and, say, y = abs(x)+e^(abs(x)-x) which are equivalent for positive values of x).

Another interesting point is that this connection between size and robustness of the solution seems to be a form of the Ockham's (or Occam's) Razor, a general principle of inductive learning that essentially states that the most likely hypothesis is the simplest (or shortest) one that is consistent with all observations. This same principle has been previously applied with success to GP in [7].

3.2 Fitness Cases Approach (Number)

The use of fitness cases in promoting robustness of code can be further divided according to the level of specification. There is, in fact, one trivial solution to this problem, which is to make the fitness set exhaustive, much like a truth table for logic functions. In that case, the concept of robustness no longer applies and a perfect score guarantees correct behavior in all cases. Although the latter method has been employed before [\cite Koza GP I, even-parity], it has obvious draw-backs.

Firstly, the number of possible input combinations is often too big to be tractable. Furthermore, the solution to all input combinations might not be known and, if the solution to every input combination is already known, then a look-up table often eliminates the need for the evolution of an algorithm.

Another more common approach has been to under-specify the problem through the fitness cases, and merely consider robustness as a probabilistic outcome (which also raises the question of how to test for robustness after evolution). However, depending on the particular problem domain, brittle solutions might be much easier to find than robust ones, and if fitness cannot be used to determine which solution is better, the probability of finding a robust solution might be small.

A third solution is to increase the number of fitness cases and improve their selection in the attempt to raise the probability of finding a robust solution. While this method would certainly reduce the number of perfect scoring

brittle solutions, and thus promote the evolution of more robust ones, there has been no general guideline or method for optimizing this selection. Providing such guideline is one of the focuses of this paper (see section 4).

Fitness Cases Approach (Noise)

An additional approach which does not rely on the selection or number of fitness cases has been proposed by Reynolds. His work applied to the evolution of a corridor following behavior, for which he introduced the use of noise in the sensor readings which the program uses as terminals. Although its first attempts were unsuccessful [4a], he was later able to demonstrate the validity of this approach in producing robustness

"The solutions discovered by this process are simple and robust. It appears that noise in fitness testing discourages strategies that are brittle, opportunistic, or overly complicated."[4b]

However, this conclusion is valid only if the premise that a robust program is more resistant to noise than a brittle program is true. The latter premise might not be true for all domains, although what could be generalized is the fact that weaknesses of brittle code can be exploited to promote the evolution of more general code.

A comparably similar approach is the use of dynamic fitness sets, consisting in submitting only a selected subset of all the fitness cases at any time (at random). This approach can be considered similar to Reynolds for it selectively modifies (here eliminates) part of the fitness set and therefore damages over-specific individuals which rely on a subset of fitness cases for their score. Again, however, nothing prevents a brittle individual to perform equivalently to a robust individual under this setting (and, especially if a perfect scoring brittle individual is shorter and less complex than a perfect scoring robust individual, the brittle individual would likely be produced and thus selected first).

3.3 Co-Evolution

Co-evolution is another technique which can be used to improve the generality of evolved code. A simple example of this would be, for instance, a robot learning obstacle-avoidance co-evolving with a room which changes its configuration. While this approach has been successfully applied to some problems [6], it still presents several problems.

Firstly, it is not always possible or clear how to co-evolve the environment to produce results. Take the problem of classifying pictures of tanks from trees; would the co-evolution need to create new realistic pictures of tanks and trees which are different from the existing ones? Aside from perhaps being able to produce some sort of noise (see previous section) over the input, this would seem highly unlikely.

Secondly, even in the case of the obstacle-avoidance behavior, this works well in simulation, but it might be harder to implement with real robots training in a real room. Thirdly, the co-evolution might have unwanted equilibrium points or oscillations, in which the parties do not produce the most general and complex behaviors, but rather oscillate between very specialized and simple ones.

Finally, the complexity of evolving rooms and robot behaviors can be much larger than needed, since perhaps a much smaller subset of room configurations is sufficient to uniquely specify (or recognize) the correct behavior from unwanted ones. Even if any number n of components were independent, co-evolution would still have to search through the space of all combinations of them (O(X^n)) as opposed to searching them independently (O(nX)).

4. An Alternative Approach: Using Multiple ‘Tasks’ to Reduce Ambiguity:

In his theory of forms, Plato noted that ideas such as a circle or a line do not exist in reality, but their essence can be learned or known by observing a series of objects, imperfect representations of the pure concept from which the concept or form can be reconstructed. If all such examples share nothing more than the attribute of being circles or lines, then their meaning would be correctly learned.

So the basic idea is that a single ideal fitness set might not be easy to produce, and any single fitness set is likely to be ambiguous and contain superfluous and misleading information (as discussed previously, see section 2). However, assuming that each example carries a set of possible interpretations or attributes, the intersection of examples which contain significantly different sets of such interpretations carries a significantly smaller number of interpretations, and there is a set of such examples whose intersection is the one single desired interpretation.

It is unlikely to find one unique fitness case which can only be solved by one solution, just like it is unlikely to find a context or object to which only one word applies. In this respect, the relation between the fitness cases and the information they wish to convey is similar to that between several examples or manifestations of a concept and the pure concept itself.

4.1 Scalability and Iterative Refinement:

The main idea is dividing a complex task or environment into sub-components. For instance, for Koza’s wall-following robot, we might want to make the task more complex by adding round corners and other additional details. But this might be problematic because it is hard to integrate too many additional details into a fitness set (in this case the walls of a room). Also, a single room might contain coincidences which might be exploited accidentally by the individuals (e.g., move only any of the wall is closer than 5ft., which would be brittle in a larger room, for instance). My solution is to use a loop execution on separate fitness sets (in the case of the robot, use separate, simpler rooms instead of a complex one).

This not only simplifies the creation of more complex fitness cases, but also simplifies the analysis of the performance of individuals and possible causes of brittleness. Unwanted interactions due to distinct elements combination is also reduced, and number of tests for n justifiably distinct elements is reduced from Xn to nX (e.g., test the ability of an individual to turn left and right corners without having to incorporate different rooms with

possible combinations of left and right turns). Also, whenever an individual is found to fail on a given fitness set, this same fitness set can be added to the vector of separate fitness sets to create an individual which is forced to be robust in both situations (simplifying the task of refining fitness sets iteratively), where the chances of two very different fitness sets being solved by the same program by mere chance are much lower.

4.2 Fitness case selection

The need of an exhaustive fitness set can be eliminated by observing the function and terminal sets and ensuring that any wrong solution would fail in at least one fitness case of each subset.

The terminal and function sets restrict the number of possible solutions. For instance, in the case of linear regression, if we use a terminal and function set similar to that used by Koza [1] (cartesian coordinates, with the addition of +/-sqrt) then four points are enough to uniquely specify a circle centered in the origin (as long as they are placed at the intersection with each axis), while any number of points less than four could be satisfied by a number of different solutions. Notice, however, that four points do not constitute an exhaustive fitness set (which would consist of all the points in the circle). Yet, if correctly chosen, they are sufficient to guarantee that a perfect scoring individual uniquely describes a circle.

4.3 Sub-Tasks and Subsumption:

Another additional potential advantage is the fact that humans can often break-down complex tasks into sub-tasks (and this is also valid in teaching a complex subject). So, while we might not know the actual solution to a task, we might be able to identify simpler independent sub-tasks. This can be used to reduce the complexity of the fitness set and to provide additional information on the relative performance of individuals. For instance, the original set-up used by Koza for evolving wall-following behavior was originally used by Mataric in order to show subsumption of behaviors (by dividing the complex task of wall-following into: approach, align, stroll, etc.)

To give an example, in the case of the wall-following robot, we could identify the behavioral subtasks of reaching a wall, walking along a wall and turning corners.

Structural components can be identified and promoted through fitness cases as well. In the case of the robot, we could think that the solution should contain at least (TL), (TR), (MF) or (MB), and some form of conditional statement IFLTE.

There is a danger in simplifying a problem by isolating its separate components, which consists in the loss of Gestalt information (the whole is more than the sum of its components). Although the latter is a significant concern, it can be often compensated for, by using insights on the search space or by using both separated components and their combination and including them in the fitness set, or by ensuring that all the useful information that comes from the combination of components is conveyed one way or another.

Furthermore, the advantages of such separation can be very significant. Three points are of particular importance:

Figure 1. Classification problem: Assuming sensors can only return boolean values for size, shape, weight, color. The classifier is asked to be able to learn how to recognize roundness (the trivial solution is ***C****) by training over the fitness cases of different objects and their classification (expected solution) w.r.t. roundness. The +Check, –Check, AND, and OR are used to see if the fitness set is unambiguous and find which additional cases make it unambiguous.

legend: (different properties represented here as symbols)

B big **C** circular **H** heavy **R** red

- not big **-** not circular **-** not heavy **-** not red

Object	Solution	**+ Check**	**- Check**
BC-R	Yes		1101
B-H-	No	1110	
-C-R	Yes		0101
-C--	Yes		0100
--H-	No	0110	

Test ambiguity			
AND		0110 (brittle)	0100 (ok)
OR		1110 (brittle)	1101 (brittle)

Include			
---R	No	0101	
-CH-	Yes		0110

Test ambiguity			
AND		0100 (ok)	0100 (ok)
OR		1111 (ok)	1111 (ok)

1. It might be easier to produce a robust fitness set for each component than it would be to produce one for the combination of all components.

2. The same Gestalt information might otherwise suggest relationships where mere coincidences are present, perhaps rewarding programs for finding such 'relations', thus promoting the evolution of brittle programs.

3. Separating a problem into its sub tasks can convey more information on the potentials of each program. (It tells not merely how bad or good a program performed, but also tells where and what are the program's weak and strong points). This makes possible to use a fitness function which breaks the problem down into sub-tasks and takes them into account (just like a student studying for the GRE might benefit more from knowing the separate scores in each subject rather than knowing just the cumulative score).

Figure 1. shows a classification problem: recognize which objects are circles given the means to perceive color, size, shape and weight.

4.4 Additional Information for the Fitness Function:

The fitness function can also be improved by using a fitness vector instead of a single fitness score. The entries in the vector can be scores from distinct tasks or from subtasks. The separate scores in the vector can be averaged (thus losing information on the score distribution)

or recombined in other ways to preserve such information, which can then be used in the evolution to place additional selection pressure toward robustness (a score of [5/10,5/10] might be preferable than [10/10,0/10] because is more general). Taking the minimum of the vector is an easy way to obtain a single score useable in existing GP kernels, yet retaining part of the additional information. Another possibility is to take the Nth root of the product of the N scores.

This fitness function has advantages and disadvantages. The main advantage is the potential to increases the evolving pressure toward general solutions since the beginning of a run.

However, it has two main disadvantages: robust solutions might be a product of two very brittle individuals, evolution of which would be discouraged by this fitness function. Furthermore, it does not account for the syntactical value of brittle programs (see section 6).

4.5 Checking ambiguity in fitness cases:

A simple test to check the robustness of fitness sets consists of creating a table in which rows are the fitness cases and columns are the different concepts which can be associated to the fitness cases. Each entry is either a 1 or a 0 depending on whether a fitness set does or doesn't express a given concept. As we discussed in section 4.2, the terminal and function sets (or more generally, the perceivable basic concepts) restrict the number of possible solutions.

A fitness set is robust if the AND results in a zero for all columns except the one representing the correct concept and the OR results in a 1 for all columns. In the case of classification, two tables are necessary: to classify the presence, and the absence of a concept. The fitness cases must be split accordingly. The problem illustrated in figure 1 shows how knowledge on what is perceivable (here color, size, shape and weight) can be used to disambiguate a fitness set.

The above fitness set is not complete, since light (or not heavy) and circle both solve all fitness cases. Thus, 'light' and 'circle' could be confused or taken to be synonyms. To fix this, an object which would contradict the latter conclusion (i.e. a heavy circle –CH-) must be added to the fitness set. Similarly, this fitness set might mislead anybody to think that a circle is an object which is not simultaneously red and heavy. This can be corrected by substituting the H heavy object with a R red object and labeling it as not circular.

This method isolates the desired concept completely by ensuring that the only thing that all fitness cases labeled 'yes' have in common is the fact that they are all circles, and vice-versa. Thus, the smallest fitness set needed to communicate the concept of circle would be:

Object	Solution	+ Check	- Check
BCHR	Yes		1111
-C--	Yes		0100
B-HR	No	1111	
----	No	0100	

This would seem ideal, but has two draw-backs. The first is that it might not always be possible to join all desired properties or concepts together, and the second is that it does not leave much room for partial credit and thus for a gradual growth.

Figure 2. Wall-following behavior robust w/respect toposition and orientation. legend: (different properties represented here as symbols)
P (x=12.1, y=9.2, a=*) **O** (x=*, y=*, a=270) **W -** **-** (x=13.6, y=13.6, a=*) **-** (x=*, y=*, a=0) **- -**

Object	Solution	+ Check	- Check
WPO	-	111	-
WP-	-	110	-
W-O	-	101	-

Test ambiguity		
AND	100 (ok)	-
OR	111 (ok)	-

Generation **6**: standardized fitness=**2/3** and 166/3 hits (55/56, 56/56, 55/56).

(IFLTE SS (MF) (IFLTE S00 MSD (IFLTE SS (MF) (IFLTE S00 MSD (TR) (TL)) S06) (TL)) S06)

Generation **22**: standardized fitness=**0** and 56 hits (56/56, 56/56, 56/56).

(IFLTE SS (MF) (IFLTE S00 MSD (IFLTE S00 MSD (IFLTE SS (MF) (IFLTE S00 MSD (TR) (TL)) S06) (TL)) (IFLTE S00 MSD (TR) (TL))) (IFLTE SS (TR) (IFLTE S00 MSD (TR) (TL)) S06))

Because of its size, it does not create enough variety of examples which could otherwise raise the chance of a more easily treatable fitness case. Thus, we might consider that a greater number of distinct fitness cases might make it easier to extract the right concept, the same way that a student might learn more easily if a greater number of examples of the same concept are given.

Note that this technique can also help detect inconsistent, misleading or noisy fitness sets.

5. Application to a Sample Problem: Wall-Following Behavior

This very same method can be applied to any other problem, rather than just classification ones, by thinking of each problem in terms of lexical definitions or behaviors. However, in order to use this method to produce more robust code, it is necessary to identify which elements might be misleading.

To demonstrate how this method for selecting fitness cases can be applied to produce robust code which would otherwise be brittle, we used the problem of the wall-following robot [1], a fairly canonical problem which is not usually considered a classification problem.

In particular, this task has been shown to often produce very brittle solutions [3], in a similar way to what was observed by Reynolds for the similar problem of obstacle-avoidance behavior [5].

"... programs were found to be quite 'brittle'. ... They will not even work in the same obstacle course if any of the vehicle's initial conditions (position and orientation) is slightly perturbed." [4a]

In the case of the wall-following problem, a robot is asked to use information from its 12 sensors to guide it along the walls of a room.

However, brittle code is produced if the robot 'memorizes' certain aspects of the room, rather than extracting more general rules, and uses correlation which might be mere coincidences. So, for instance, it might not start moving unless the walls in front of it are closer than those on its back.

As a disclaimer, I would like to point out that I used a simulated wall-following behavior because it is a somewhat intuitive problem which has been shown to be brittle. Aside from this, I am not making any other claims such as relations with actual (not simulated) evolution of controllers for robots.

Just to give an idea of why some of these individuals are brittle with respect to initial position and orientation, we examined numerous such brittle individuals, and a fairly common instance was an individual which had a perfect score in a given room with a given initial position, but would not score any points in the same room when the initial position was changed, because the essential program (the S' we described in section 3) was encapsulated in an IF statement which was based on the initial position. Clearly, the same program would not have survived if the training set consisted of two or more trials, where the initial position was changed (it becomes less and less probable to evolve such an IF statement which would be irrelevant or a mere coincidence, and yet be present in both cases).

We have decided to use a set of fitness cases to produce robustness with respect to both initial position and orientation. For this, we considered a room as a single fitness case, and we included the two concepts of position and orientation as binary entries in the table described above.

It should be noted that perhaps much other misleading information is eliminated by this very same fitness set. For instance, by changing orientation but leaving position intact, almost all sensor readings change, so that the likelihood of maintaining misleading coincidences between them is reduced. However, if any other obvious misleading relation were to be included (ex: room symmetry or orientation parallel to walls), it would be easily eliminated with the addition of other fitness cases or a more careful selection of the present ones.

we used exactly the same room for all three fitness cases (the same used by Koza). The fitness was then computed by taking the average between the score of each fitness case. All parameters are as specified by Koza [1], with the exception of the number of generations and the number of individuals, which were, respectively, 25 and 2000 (where Koza used 50+ generations and 1000 individuals).

Although the results we obtained were very preliminary (we only executed few runs with these parameters, so that a meaningful analysis of the likelihood and frequency of obtaining a solution are not yet possible), we were able to find a perfect scoring individual as early as in generation 22 (where Koza did not obtain a perfect scoring solution until after the 50th generation). An almost perfect individual (scoring 55 and 1/3 out of 56) was found as early as in generation 6.

These best individuals were then tested in the same room with several different initial conditions (position and orientation). All best-of-run individual were extremely robust to a change in both position and orientation, and exhibited the same basic behavior (the robot would move forward until it reached a wall, then proceeded by moving along the walls).

An extra amount of time (up to 1/20th of the total time) was allowed to some individuals, since their initial position would place them much further from the wall in front of them.

A significant detail is that many such robust evolved individuals are shorter in comparison to their brittle counterparts. Figure two shows the code for the 2 individuals mentioned above (again, refer to [1] for an explanation of functions and terminals; in short, MoveForward, MoveBack, IFLessThanorEqual, TurnRight, TurnLeft, Sensors 0-11 and ShortestSensor):

6. Discussion and Future Work:

After a more clear or robust set of fitness cases has been produced, the fitness function can be modified to extract and exploit additional information in order to improve the selection of individual programs.

In the case of the robot, for instance, it is now possible to select an individual based on whether they are more general or more specific by looking at how the score is distributed between the 3 trial. We can even infer whether a program is robust with respect to position, orientation, etc. by grouping similar position or orientation fitness cases. Even within the same fitness cases, it is possible to group tiles into meaningful groups (i.e. corners, straight walls, etc.) to know strong and weak points of an individual program.

In order to make the best use of this additional information, however, it is important to predict how a modified fitness function might affect the GP evolution of a program.

The main issue is: can GP take advantage of this additional information?

For instance, there is no clear mapping between generality of parents and that of their offspring. So, for instance, a very brittle individual might essentially contain the correct robust solution, making it extremely valuable and allowing its offspring to potentially inherit a complete robust solution in one step (and in fact we shown just such a case in the previous section, namely the encapsulating IF

statement code). At this point, without knowing how such individuals come to existence and what the tradeoffs in overly penalizing brittle individuals are, this question remains unanswered, although we plan to explore it in future work.

For this task it is helpful to identify three components which can promote or inhibit the search for a solution. These components are:

1. the evolution of building blocks,

2. the misleading action of high scoring brittle programs (which lack both syntactical and conceptual value) and

3. the syntactical value of conceptually wrong or brittle individuals.

1. Partial or building blocks were first proposed by Koza [1] and their role in the GP evolution process has been examined by several authors and has lately been increasingly questioned. An important distinction is that between genotypical (based on structure) and phenotypical (based on behavior) building-blocks. Since it is still not clear whether the building-block hypothesis has validity, we consider its possible implication in my discussion. By grouping the fitness cases according to meaningful and more basic sub-tasks, in fact, it is possible to promote the evolution of individual sub-tasks, and by noting which individuals solve which sub-task, it is conceptually possible to modify the fitness function to promote the recombining of complementary building blocks. These observations, however, are bound to fail if the sub-tasks yield to incompatible building blocks (or if the building-block-hypothesis is false) and might not, therefore, apply to all GP problems.

2. It is conceivable that a confusing data set might steer the GP toward an incorrect local optima which is easily obtainable and from which the GP would not be able to escape. In fact, if the probability to find such a program is high, it might be found in great numbers and before any better scoring solution can be found. Thus, such incorrect solution will be involved in most cross over reproductions, which might prevent the GP from finding a more complex but correct result.

3. It is also important to realize that the behavior of a program is not the only factor in determining its value. The GP cross-over and mutation, in fact, does not act on concepts or behavior but rather on syntax, so that a conceptually wrong individual might be one atom away from becoming right. Another important factor in this respect is the presence of introns, as described by Angeline, which are parts of a program which are inactive, but which might become active after cross-over. Introns do not modify behavior, but can play an important role in the GP process.

These factors are important, and thus we plan to examine them in more details in future work, by tracing relations between parents and offspring to find possible traces (or absence) of either genotypical or phenotypical building blocks and their role in evolution.

we have also written a version of wall following code in C (for speed, compared to LISP) which can flexibly represent any room configuration as a matrix of blocks (using a ray-tracing algorithm for the sensors), and plan to evolve robust behaviors of wall-following and obstacle avoidance with respect to room shape and perform extensive numbers of runs to determine the likelihood and frequency of solution, measure their robustness, and compare these to single-task runs.

7. Conclusion:

The analysis of previous techniques gives many insights on the dynamics of the evolution of robust code, but none of the previous approaches addresses the fundamental issue of creating unambiguous fitness sets. The problems of using a simple fitness set is illustrated in evolving a wall-following behavior. To show how the proposed technique can be used to produce robust non-exhaustive fitness cases, a robust wall-following behavior was evolved.

The results indicate that non-exhaustive fitness cases can be organized to produce reliable robustness with respect to a given attribute. For this reason it can often be advantageous to consider a solution (or cast a problem) in terms of a classification (of a behavior, function, etc.) even when this is not the most obvious one (e.g., the wall-following and obstacle-avoidance behaviors) so that we can apply the described techniques to detect noise, promote more general solutions and iteratively improve training sets.

While the proposed technique also produces additional information related to the generality of individuals, it is not clear how this information can be used successfully by the fitness function for selection and cross-over, and further research on the topic is needed.

Acknowledgments

We appreciate the contributions from the following people: E. Durfee, W. Birmingham, P. Angeline, U.-M. O'Reilly. We also thank these other individuals for their assistance: R. Bertram, C. Grasso, J. Polito, S. Stanhope and A. & M. Bersano-Begey.

Bibliography

[1] Koza, J. R. (1992) Genetic Programming: on the Programming of Computers by Means of Natural Selection, ISBN 0-262-11170-5, MIT Press, Cambridge, Massachussetts.

[2] Kinnear, K. E. Jr. (1993a) "Generality and Difficulty in Genetic Programming: Evolving a Sort", in Proceedings of the Fifth International Conference on Genetic Programming, K. E. Kinnear Jr, Ed. Cambridge, MA: MIT Press.

[3] Ross, S.J., J.M. Daida, C.M. Doan, T.F. Bersano-Begey, and J.J. McClain, "The Wall Following Robot Revisited," 1996. Genetic Programming 1996: Proceedings of the First Annual Conference, July 28-31, 1996, Stanford University, Cambridge, MA:MIT Press

[4a] Reynolds, C. W. (1994a) "Evolution of Obstacle Avoidance Behavior: Using Noise to promote Robust Solutions", in Advances in Genetic Programming, K. E. Kinnear Jr., Ed. Cambridge, MA: MIT Press.

[4b] Reynolds, C. W. (1994b) "Evolution of Corridor Following Behavior in a Noisy World", in SAB-94

[5] Reynolds, C. W. (1993) "An Evolved, Vision-Based Behavioral Model of Coordinated Group Motion", in From Animals to Animats 2: Proceedings of the Second International Conference on Simulation of Adaptive Behavior (SAB92), Meyer, Roitblat and Wilson editors, MIT Press, Cambridge, Massachussetts, pages 384-392.

[6] Andreas Ronge and Mats G. Nordahl. 1996. Genetic Programs and Co-Evolution Developing robust general purpose controllers using local mating in two dimensional populations. In Voigt, Hans-M., Ebeling Werner, Rechenberg Ingo, and Schwefel Hans-P (editors). *Parallel Problem Solving from Nature IV, Proceedings of the International Conference on Evolutionary Computation.* Berlin, Germany: Springer Verlag. Pages 81-90.

[7] Byoung-Tak Zhang and Heinz Muhlenbein. 1993. Genetic Programming of Minimal Neural Nets Using Occam's Razor. In Stephanie Forrest (editor). *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93*. University of Illinois at Urbana-Champaign: Morgan Kaufmann. Pages 342-349.

[8] Justinian Rosca. 1996. Generality Versus Size in Genetic Programming. In Koza, John R., Goldberg, David E., Fogel, David B., and Riolo, Rick L. (editors). *Genetic Programming 1996: Proceedings of the First Annual Conference*. Cambridge, MA: The MIT Press. Pages 381-387.