Development of a Minimal Information Line Following Algorithm using Genetic Programming

Eric Berger

PO box 11650

Stanford, CA

eric.berger@stanford.edu

(510) 918-7038

Abstract: This paper describes a method for developing algorithms that can perform a given task using a minimum of information. Using the framework of genetic programming, with a fitness partially determined by both program complexity and information required, this technique allows automatic discovery of the most efficient ways to both obtain and use information in a given domain. When combined with a set of interactive detectors that allow the program to access any arbitrary information about the environment, this results in an algorithm which automatically discovers and uses the optimal detectors for the specific problem at hand. The technique is illustrated with the problem of following level-sets of mathematical functions.

Introduction

Traditionally, when genetic programming is used to develop algorithms, the problem formulation specifies what information is to be made available to the individuals in the population. For some problem domains, where the relevant information is easy to obtain and can be clearly identified, this is an appropriate paradigm in which to operate. In other types of problem domains, however, the choice of what information to provide the algorithm with is a difficult one, however it is a choice which can have a big impact on the utility of the final product. One place where this problem occurs is in environments that are not discrete, but have an infinite number of potential variables to be measured. Although measuring a given quantity at all points or on a fine grid covering the area can be very computationally expensive, limiting the algorithm to only have access to a small number of the potential variables unnecessarily increases constraints and drastically reduces the potential for finding a near-optimal solution.

Level-Set Curves

The specific problem this paper will address is the problem of following level-set curves on mathematical functions. This can be reduced to the problem of following zero curves by shifting the target function up or down. The exact problem statement is as follows: given a function f(x, y) which is equal to zero along some line in the x-y plane, and given a starting point along that line, find a set of points which lie spaced along that line and as close to the zero-curve as possible while checking the value of the function at a minimal number of locations. The individual programs do this by using an abstraction of a bug crawling along the line, represented by a position and a heading. Individuals can turn, move forward and laterally, and read the value of the function at points relative to their own position and heading.

Methods

To explore the possibilities for reducing information use in this problem, I implemented it as a genetic programming problem using ECJ 8. The tableau for the first experiment was as follows:

Objective:	Develop a program capable of following the zero-curve of a linear function (Ax + By) starting in any arbitrary orientation

Terminal set	The only terminal I used was the number 1
Function set	+, -, *, % were implemented in the usual way, with % returning 1 on divide by zero "If" tested whether its first argument was greater than 0, and used that to decide between its second and third arguments. "Trim" capped the absolute value of its first argument at the absolute value of its second argument (e.g. (trim 1 2) = 1, trim(2, 1) = 1, trim(-2, 1) = -1. "Move" took two arguments, and used them as forward and lateral components of motion "Turn" took one argument, and turned by that amount "Read" took two arguments, a forward and a lateral distance, and returned the value at that position relative to the current position and heading
Fitness cases	Functions were generated by randomly picking A and B from [-1, 1] ten times for each individual. The individual would move for 20 time steps, at which time fitness would be evaluated.
Fitness	Fitness was
Parameters	Population size = 1024
Termination Criteria	Perfect solutions aren't possible, so I always run until the max number of generations

The major steps to implementing a genetic programming problem are:

- Constructing the terminal set. For this problem, the constant 1 is the only terminal necessary to reach any solution. Because of the presence of other mathematical operators, any constant can be constructed from combinations of 1 and other mathematical functions. The goal of not including arbitrary constants was to encourage the system towards using the results from the read function more directly, and not just evolving solutions which were tailored to take a specific path, regardless of the results of the read function.
- Constructing the function set. There are two main components of the function set: The functions which were for manipulating the data, and the functions which interacted with the world to read the value of the target functions and to turn and move the current location and heading. The functions to turn, move, and read were all fairly straightforward to decide on. The one decision was whether to have the coordinate system be relative to the current position and heading, or relative to a global coordinate system. Using relative positions was clearly a superior choice for several reasons. First, using relative positions would mean that the program would not have to keep track of its position, because it would always be at (0, 0) and facing at heading 0. Secondly, using relative positions the algorithm would not be affected by the location in coordinate space of the curve it was trying to follow, which seemed to me to be a good idea. Third, and perhaps most importantly, if the coordinate system was global, even a program which learned to follow a given path would not be able to follow that same path in another location. The number of different cases would increase hugely, and the training would probably require a much larger training set. In order to deal with these issues, the system uses a relative coordinate system for all functions.

  The second component of the function set, for manipulating the data, is fairly standard. The addition, subtraction, multiplication, and protected division are sufficient to approximate almost any mathematical function arbitrarily well. The trim function was provided to allow programs to prevent unstable behavior, and the if was provided mainly to allow programs to reduce the number of times they were checking the value of the function, where possible.
- Determining the fitness criteria. An ideal solution would have to meet multiple criteria, and so the fitness function had to balance those different considerations. The chief factors were the proximity of the path taken to the zero-curve, the distance traveled, the size of the program, and the number of times the value of the function was checked. Avoiding the individual that didn't move or turn as a local optimum which would trap initial populations required ensuring that the distance traveled has a bigger effect on the fitness than any of the other three factors, when distance was 0. This is because for a non-moving individual achieving error of 0, minimal size (3), and no reads is not difficult, and is in fact accomplished by the individual (+ 1 1). Making raw fitness proportional to distance traveled eliminated this problem. After that, multiplying by factors that penalized error, size, and number of reads put pressure on the population to reduce all of those quantities. These multiplicative factors were all of the form (1 / (1 + A * (quantity)), where A could be varied to affect the relative importance of the specific quantity in the final fitness measure.
- Setting parameters. I picked a large population, M = 1024, because that would provide more genetic variability in the initial population, and would allow more possibilities to be explored simultaneously. Exploring large numbers of potential solution at once also made the possibility of independently discovering multiple techniques and combining them more realizable. Because of the large population size, runs of 21 generations

were generating almost perfect individuals, so short runs were sufficient and there was no reason to set the max number of generations any higher than 21. Because of the way that training sets were randomly generated each generation, there was no real possibility of overfitting, so because 21 generations was finding good solutions quickly it seemed to be a good choice.

- Finding termination criteria. Because it is hard to determine what a perfect individual is in a continuous problem domain such as this one, I let all of my runs go until they met the maximum number of generations.

Structure of Individuals

To make this problem more concrete, here is a sample individual: "(move(1 1))". This individual, on each time step would move one unit forward and to its right, tracing out a straight line. Its fitness would probably be quite low for any given function, because it does not use the read command, and so has no way of following, or even being affected by, a zero-curve. If the function happened to be f(x, y) = x – y, then it would do well, because it would be moving along the correct path, however it will always move down that path regardless of the function chosen. A more interesting sample individual is shown here: "(add (turn(read 1 0)) (move 1 0))". This program would read the value one unit ahead of it, would turn by that amount, and would then move one unit forward. Adding the values returned by these two operations wouldn't have any effect, however the add works to allow us to sequentially execute two commands without having the overhead of defining new functions to do so. If the line was fairly straight and the function was positive on the right side and negative on the left side of the line, then this would result in fairly intelligent behavior. If the current heading were two far to the left then (read 1 0) would return a negative value, causing a turn to the right, and it would similarly correct for being too far towards the left. The heading correction would tend to lead to a fairly low error, assuming that the line was fairly straight, and since the individual would always keep moving forward, has a small size, and only calls read once, we could expect quite a good fitness value from this case.

Fitness

The fitness of an individual is measured by checking its performance against a number of different target functions. For each function, the position and heading are set, and then the individual is allowed to move for twenty time steps. The following information about the path and the individual is collected, and used to calculate fitness.

- Distance is the total distance traveled, summed over all functions the individual has been tested on.
- Error is the sum, taken over all points on the path, of the square of the function whose zeros we are trying to find, also summed over all function the individual has been tested on.
- Size is the total number of nodes in the tree
- Reads is the total number of times the individual actually checked the value of f(x, y), which is not always equal to the number of times that read appears in the code, because of the if statement.
- The parameters A, B, and C help control the contribution each factor makes to the overall fitness, as well as the target value for each.

The specific fitness measure used is of the following form: raw fitness = distance * 1 / (1 + A * error) * 1 / (1 + B * size) * 1 / (1 + C * reads). To convert this raw fitness to a standardizes fitness, we just use standardized fitness = 10000 – raw fitness. This becomes adjusted fitness of 1 / (1 + standardized fitness). This leads to very low adjusted fitness for most individuals, which allows competitive selection pressures at the beginning, but lets good solutions quickly become dominant.

Generality

Using this function set and the operators of crossover and a small amount of mutation on top of fitness proportionate selection, it is reasonable to expect that most, if not all algorithms for solving this problem efficiently will be discovered. Within the framework of moving a point, the move and turn operators provide all of the options. The read function is fully general, allowing the program to check the function value at any point, and to do it an at an arbitrary number of locations. The other functions provided allow for the construction of polynomial approximations to any mathematical functions, and the conditional allows for selective evaluation of different program branches.

Crossover should be satisfactory for the reproduction, and lead to a realization of the function set's potential, because with a large population and fairly limited function set, all of the functions should be represented in individuals at any step in a given run. The problem also lends itself to the recombination possible through the use of crossover, since using multiple methods to line up with the curve will just contribute to greater accuracy and robustness of the algorithm. Because of mutation, and because even a single good individual can quickly become dominant, even if the population converges to a local optimum, it should break free of that peak fairly quickly and continue approaching the global maximum fitness.

Equipment and Running Time

The experiments themselves generally took about 5 to 10 minutes each on a dual processor 1.3 GHz machine. Without the selection pressure towards small programs, the complexity tended to grow hugely and the run times were much larger. Despite that, the final fitness and rate of convergence were often better with the penalty for large programs.

Results

Many of the runs resulted in quite fit individuals with relatively small trees, and without consulting the value of the function in many places. The best examples of this were the programs to follow lines and large circles. In those applications, with small curvatures, once the heading and position are correct, the adjustments to be made are small and a fit individual can cover a lot of distance in each time step while still staying remarkably close to the target curve.

I will look in detail at how one run, approximating a sin curve. The progress towards a solution in terms of fitness was fairly steady, however by looking at the paths taken a clear pattern emerges. Every generation, the current best solution would be trimmed down a little smaller, and made a little more accurate, bumping up its fitness. Every few generations, however, a new individual which went significantly farther would develop. These individuals were usually not a precise or as parsimonious, and so over the next several generations these new rough solutions would have to be gradually refined. Figure 1. shows best of generations paths for some of the key generations, and shows the pairing between the individuals which made a jump in distance and the individuals in the generation after them, who were able to maintain that increased performance while not sacrificing accuracy.

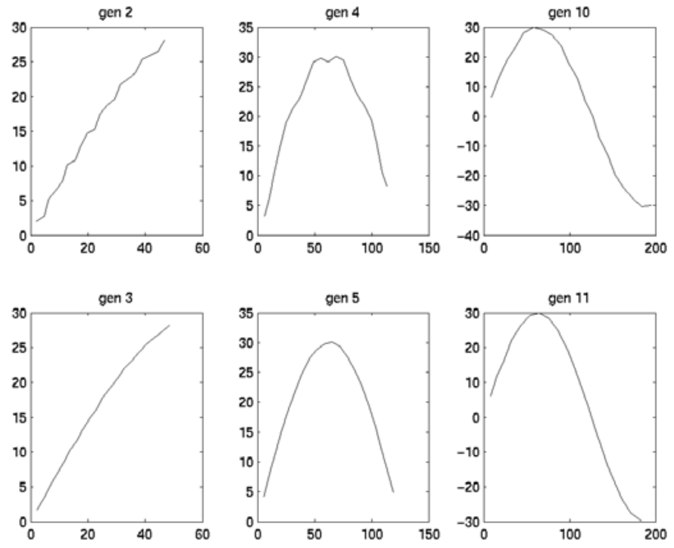

Figure 1

Although runs on just straight line training sets often developed individuals that could not follow lines with any significant curvature, I found that individuals which were trained on either the circular or sin curve data behaved well on other curves also. The most robust individual that I discovered was the final product of a run combining all three types of curves. Although it was reasonable complex, it had good accuracy moving along all types of curves, and was precisely correct on straight lines and curves with low curvature, up to distances as small as 10^ -13.

One of the most interesting results came from examining the effect of the parsimony pressures that I used. I tried to develop solutions based on training data which consisted of circles of radius varying from 500 to 2000. Without the pressure for simple solutions, very good individuals would emerge around generation 20. They would, however, be gigantic and take a long time to run. After I added in the parsimony and information pressures, the solutions began to be significantly less complex, yet equally good solutions were still emerging by generation 20. This indicates that in situations where obtaining information is costly, genetic programming has shown an ability to limit its information use without significant deterioration to its ability to find solutions.

Conclusions

The results of this project indicate that genetic programming is capable not just of effectively combining provided information, but of determining which information is the most useful for the task at hand. When coupled with the fact that fitness measures provide an easy way to optimize your solutions with respect to the cost of information, this is a strong argument in favor of using genetic programming in real world situations. It is an especially good candidate for any environment in which obtaining the information to make a decision is a problem of comparable complexity to that of interpreting the data. Even while obtaining data in this way, individuals are clearly able to reach a very high level of competency, as was evidenced by the runs that ended up reaching a level of accuracy of less than one part in a billion drift off of the target line over the course of the movement. This demonstrates that the problem of following a solution curve in a function efficiently is a problem well suited to genetic programming, and that use of a detector that enables the program to dynamically select the data that is gathered is an effective way of dealing with a continuous problem domain like this one.

Future Work

It would be instructive to try hand-designing detectors for this environment in order to see how effective the automatically discovered ones were by comparison. Additionally, I would like to evolve programs to do this same task, but with memory in which to store variables. Because this would give the program more freedom over its information flow, I would hypothesize that genetic programming with variables would be able to design algorithms that were still more efficient in terms of memory usage. Finally, designing a system to find good starting points, given a function, would make this a much more useful tool. The combination of an ability to find zeros of a function with the ability to follow the zero-curves efficiently could be a valuable method of obtaining numerical estimates of solutions to complex equations without ever having to solve them explicitly, or to do an enumerative search through the domain.