Genetic Programming with Guaranteed Constraints

Colin G. Johnson
Computing Laboratory
University of Kent at Canterbury
Canterbury, Kent, CT2 7NF, England
e-mail: C.G.Johnson@ukc.ac.uk

Abstract: *Genetic programming is a powerful technique for automatically generating program code from a description of the desired functionality. However it is frequently distrusted by users because the programs are generated with reference to a training set, and there is no formal guarantee that the generated programs will operate as intended outside of this training set. This paper describes a way of including constraints into the fitness function of a genetic programming system, so that the evolution is guided towards a solution which satisfies those constraints and so that a check can be made when a solution satisfies those constraints. This is applied to a problem in mobile robotics.*

Keywords: genetic programming, constraints, safety, mobile robotics, verification

1 Introduction and Background

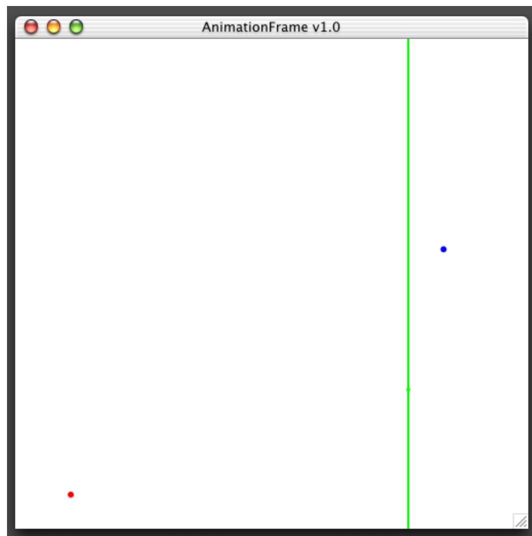
1.1 Genetic Programming

Genetic programming [1, 7] is the application of genetic algorithms [3, 8] to the creation of program code. A set of programs are generated at random, tested on a set of training data, and their fitness for the desired task assessed by measuring their performance on those training data against some success metric. The better programs are selected from this set and a new set formed by combining the programs together in some randomized fashion which swaps routines between the programs, and small changes (mutations) are made to the programs. This process is then repeated starting from this new set and iterated until a good solution is found and/or the population stabilizes.

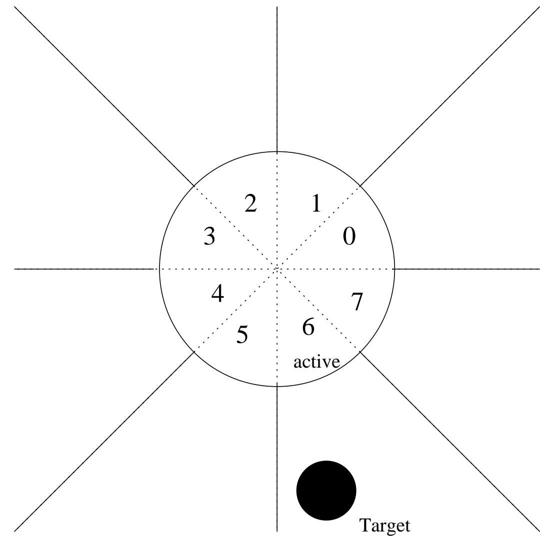
Clearly this cannot be carried out in an arbitrary fashion. The canonical way to ensure that the programs generated are executable is by swapping over components in a LISP parse tree [7]. Since this method was devised many other related techniques have been developed. In this paper I shall use the term "genetic programming" (GP) in a broad sense to refer to any application of genetic algorithms to the development of program code.

1.2 Trusting Genetic Programming

One problem with genetic programming is that the results cannot be formally trusted. At the end of a successful GP run the output is a program which performs on the training set, however there is no guarantee that it will perform adequately outside that set. In traditional programming this is not always a problem, because the program has been devised by a human programmer who has some knowledge of the problem which is to be solved, so testing is a confirmation that the problem has been adequately represented in the program logic. However in GP the final program will often be incomprehensible to human programmers so *post hoc* verification of its



(a)



(b)

Figure 1: The experimental setup.

conformation to specification is difficult.

1.3 Towards Trustable GP—Guaranteed Constraints

One approach to solving this problem is to carry out some analysis of the program to determine whether the program satisfies certain properties. Several analysis techniques, such as static analysis [9] and model checking [2] provide ways to verify that properties of programs hold across all possible execution paths. The idea which is exploited in this paper is to use the output from such techniques as a component of program fitness; in the example below a component of fitness calculated using such an analysis is used alongside a fitness measure (for a different aspect of the problem) measured by execution of the programs on a training set. The benefit of this is that the end result, if the GP run is successful, is a program which works and has certain formally guaranteed properties. More details of this general approach are given in my recent papers [5, 6] and in a recent paper by Huelsbergen [4].

2 A Case Study from Robotics

To demonstrate the above ideas a case study is given using a simulated mobile robotics problem.

2.1 Problem Details

The problem concerns the motion of a mobile robot reacting to two other moving objects in its environment. The environment is a square, 200×200 units in area. Within this a robot, represented by a point, moves under the control of the program evolved using the algorithm to be described. There are two other objects in the environment, which move around according to a random choice of direction each turn. This will simply be referred to as "the robot" below. The first is another point-robot, which shall be referred to as the *target* robot; the second is a barrier parallel to the y-axis. A snapshot from this is given in figure 1a.

The task for the main robot is to get to and remain as close as possible to the target robot. However the robot must always remain to the right of the barrier. This provides a problem which has both a data-driven component (remain as close as possible to the target) and a fixed

constraint (remain to the right of the barrier). It is assumed that this constraint is a hard constraint, i.e. a solution which does not satisfy this is not acceptable.

The simulation works as follows. A test run of the program consists of 1000 discrete timesteps. Between timesteps the three objects in the environment make a move. At the beginning of the timestep the robot senses its environment. It has 8 sensors to sense the target (figure 1b), and a sensor which detects the shortest distance to the barrier.

The programs being evolved consist of a sequence of statements. The statements are each self-contained, e.g. there are no loops or `if` statements which in turn contain other loops or `if` statements. Each statement consists of either an unconditional piece of arithmetic (incrementing or decrementing either the x or y position of the robot), or the same type of statement conditional on an `if` statement which takes a condition on one of the sensor values or the distance to the barrier.

An individual in the population consists of a program containing 20 such statements. The population consists of 50 individuals. The population evolves by a generational genetic algorithm. Crossover is implemented using uniform crossover at the statement level, and mutation changes the calculation carried out in the statement or the condition in the `if` statements. Selection is roulette wheel selection with an elitist strategy ensuring that the top solution is migrated from one solution to the next. The robot begins each run at the position $(120, 100)$.

The fitness has two components. The first component is of traditional form, consisting of a measure based on the training data. At the beginning of each timestep the (euclidean) distance d from the robot to the target is measured, and the fitness calculated as follows:

$$fitness = \begin{cases} 0.0 & \texttt{if } d > 50.0 \\ 50.0 - d & \texttt{if } d \leq 50.0 \end{cases}$$

These fitnesses are summed over the 1000 timesteps of the run, and these summed over the 50 training cases.

The second component of the fitness ascertains whether the robot is always to the right of the barrier. This is done by carrying out an *interval analysis* of the change in the x position of the variable. An interval analysis consists of tracking a pair of numbers through the program, which represent the maximum and minimum values which the quantity can take at that current point in the program. This is set initially to a conservative approximation to the initial state of the quantity, and as each line is looked at in turn, the values are updated to represent the possible state *regardless* of what other data values were present.

In this example the analysis tracks the (dx_{min}, dx_{max}) values which represents a conservative approximation to the maximum and minimum values which the change in x can take. The initial value is $(0, 0)$—at the beginning of the program no change has been made. A simple arithmetic statement applies that calculation to both values; for example if the first statement in the program is `x++;`then the interval changes to $(1, 1)$. When a conditional statement is met then the interval is expanded to contain the outcome of executing either state. So for example if the next statement is

```
if (sensor[3] is active)
  {
    x++;
  }
```

then the interval becomes $(1, 2)$, because either the condition is true in which case the value can increase by one, or it is not in which case the value stays the same.

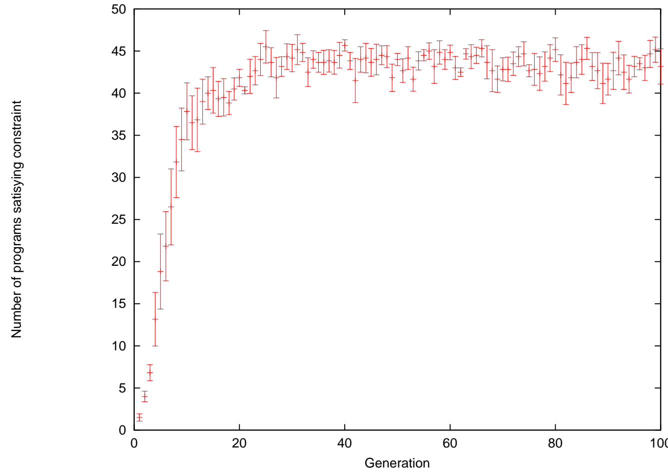


Figure 2: The number of solutions which satisfy the constraint, per generation

Two such interval analyses are done. One tracks the maximum and minimum x values in the case where the robot is "close" to the barrier, i.e. when it must move rightwards next turn or else there is some probability of it hitting the barrier; the second case is when the robot is elsewhere.

To generate the fitness the guaranteed minimum x distance which the robot moves when it is close to the barrier is examined for each program. If this value is greater than 5.0 then the constraint is considered to be satisfied, because the maximum the barrier can move in a turn is 5.0. This influences the GA in two ways. Firstly the fitness of any program which is guaranteed to satisfy the constraint is multiplied by three. Secondly the fact that the program satisfies the constraint is stored, and no program is reported to the user as successful unless it satisfies the constraint. To encourage the search towards solutions which satisfy the constraint, programs for which the guaranteed minimum is in $[0.0, 5.0]$ also get a fitness multiplier (an intermediate value of 1.8 was used).

This constraint analysis also affects the elitism strategy. Instead of simply choosing the program with the highest fitness, the fittest program which satisfies the constraint is chosen. Only if no programs in the population satisfy the constraint is the program with the absolute highest fitness chosen.

The training data are generated by making a random move of the barrier and the target each turn. The barrier moves with a speed between 0 and 5 units per timestep, whilst the target moves with a constant speed of one unit per timestep. The target starts at $(100, 100)$ and the barrier at $y = 100$.

2.2 Experimental Results

This section contains experimental results; these results are drawn from six experimental runs of the above system. Note that the figures given below are for the "raw" fitness value before the multipliers due to the constraint analysis are applied. Error bars give the standard error.

The first set of statistics (figure 2) show the number of population members which satisfy the constraint at each generation. It can be clearly seen that this increases with successive generations. After a number of generations the algorithm is exploring almost solely within the space of solutions which satisfy the constraint.

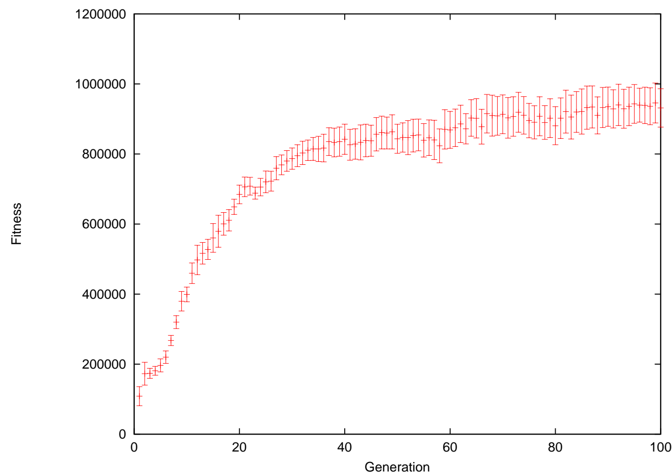


Figure 3: The average fitness of the constraint-satisfying solutions on the target-following task.

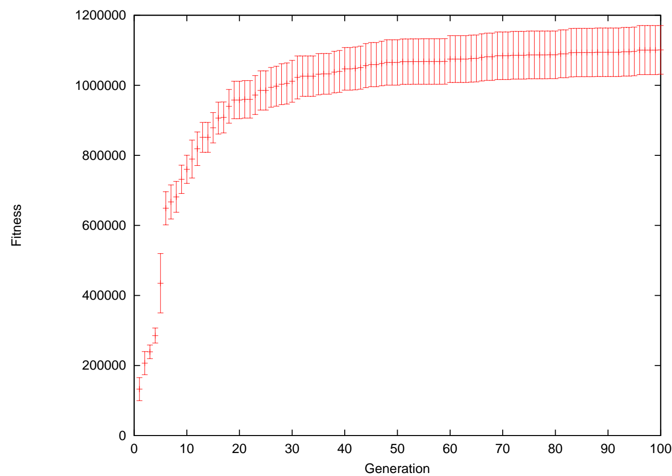


Figure 4: The fitness of the best of the constraint-satisfying solutions on the target-following task.

The second set of statistics show the performance of the programs generated by the algorithm on the task of keeping close to the target robot. The first graph (figure 3) gives the fitness across the set of those programs which satisfy the constraint; programs which do not satisfy the constraint are ignored. The second graph (figure 4) gives the best fitness per run, and the final graph (figure 5) gives the "best of the best", i.e. the best performance across all of the runs.

The maximum fitness attained is 1195559. By way of comparison the fitness of a basic hand written solution on the same test case was 910091; the best evolved solution is therefore somewhat better.

3 Conclusions and Future Work

This paper has presented a way of incorporating guaranteed constraints into the fitness function of a GP system. The effectiveness of this on a simple test problem from mobile robotics has been demonstrated. The next stage in working on this is to develop other applications for this idea. There are a number of more sophisticated tasks in mobile robotics which could be approached using this technique: one idea is to have multiple robots sharing a working area, yet guaranteeing

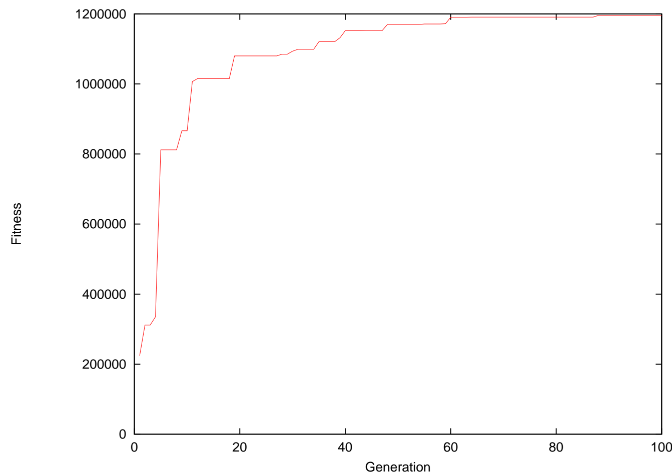


Figure 5: The best fitness over all runs of the constraint-satisfying solutions on the target-following task.

that they are able to make use of an appropriate part of the area to carry out a particular task, or the idea of guaranteeing that a robot with react in a certain way to certain stimuli. Ideas in other domains are suggested in [5].

References

[1] W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone. *Genetic Programming: An Introduction*. Morgan Kaufmann, 1998.

[2] E. M. Clarke Jr., O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[3] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, 1989.

[4] L. Huelsbergen. Abstract program evaluation and its application to sorter evolution. In *Proceedings of the 2000 Congress on Evolutionary Computation*, pages 1407–1414. IEEE Press, 2000.

[5] C. G. Johnson. Deriving genetic programming fitness properties by static analysis. In J. Foster, E. Lutton, C. Ryan, and A. Tettamanzi, editors, *Proceedings of the 2002 European Conference on Genetic Programming*. Springer, 2002.

[6] C. G. Johnson. What can automatic programming learn from theoretical computer science? In X. Yao, Q. Shen, and J. Bullinaria, editors, *Proceedings of the 2002 UK Workshop on Computational Intelligence*, 2002.

[7] J. R. Koza. *Genetic Programming : On the Programming of Computers by means of Natural Selection*. Series in Complex Adaptive Systems. MIT Press, 1992.

[8] M. Mitchell. *An Introduction to Genetic Algorithms*. Series in Complex Adaptive Systems. Bradford Books/MIT Press, 1996.

[9] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.