

SAND-2002-3147  
Unlimited Release  
Printed September 2002

# Developing Maneuvering Behaviors for a Glider UAV Using a Genetic Programming Model

R. J. Pryor  
Computational Biology and Evolutionary Computing Department

Dianne Barton  
Critical Infrastructure Surety Department

Sandia National Laboratories  
P.O. Box 5800  
Albuquerque, NM 87185-1110

## Abstract

This report describes the methodology for using a genetic programming model to develop maneuvering behaviors for an unmanned aerial vehicle (UAV) that is also a glider. The use of glider UAVs for surveillance and information-gathering operations has become increasingly important in defense and national security applications. Through an evolutionary process similar to that found in nature, the genetic programming model generates a computer program that when downloaded onto a glider UAV's on-board computer will guide the vehicle to find regions of lift for staying aloft while accomplishing an information-gathering task. This report discusses various approaches to developing behaviors using genetic programming and presents the results achieved.

Intentionally Left Blank

# Contents

Introduction.....	7
Motivation.....	7
Relation to Previous Work .....	8
Genetic Programming Model .....	8
Program Representation.....	9
Problem Definition.....	10
Definition of Functions and Terminals.....	12
Tree Generation.....	13
Fitness Evaluation.....	14
Creating the Next Generation.....	15
Selection Operator.....	15
Reproduction Operator.....	16
Crossover Operator .....	16
Mutation Operator.....	16
Solution Procedure .....	17
ASCII Tree Representation.....	19
Calculation Results .....	20
History of a Simulation.....	22
Summary .....	24
Future Direction .....	25
References .....	26

## Figures

Figure 1. An example tree. ....	10
Figure 2. Example initial conditions for UAV problem. ....	11
Figure 3. Illustration of the crossover operator.....	17
Figure 4. Solution procedure used.....	18
Figure 5. ASCII file example.....	19
Figure 6. Four tree structures considered. ....	20

Figure 7. Illustration of a typical tree found in this study. .... 22  
Figure 8. Results of a typical simulation calculation..... 23

**Tables**

Table 1. List of Functions and Terminals..... 12  
Table 2. Calculation Parameters..... 24

# Developing Maneuvering Behaviors for a Glider UAV Using a Genetic Programming Model

## Introduction

Developing maneuvering behaviors for an unmanned aerial vehicle (UAV) that is also a glider is an important technical challenge. This type of vehicle is being considered for a variety of defense and national security applications, such as surveillance and information-gathering operations. The glider UAV's importance is based on the vehicle's ability to stay aloft indefinitely because it does not require fuel. And because the glider UAV is quiet, it could also achieve some degree of stealth. The challenge comes about because the software program used to maneuver or steer the glider UAV would have to know how to find regions in space where the vehicle could obtain lift, to remember the locations of these regions, and to account for possible movement of these regions in time. The software program would have to perform all of these operations while looking about the surrounding space gathering information.

Operationally, it is envisioned that many glider UAVs, from hereon referred to simply as UAVs, would be deployed to complete a given information-gathering task. Each UAV would have on-board electronics, including a small computer, a ground-positioning system, an altimeter, communication equipment, and an obstacle detector. To enable steering, each UAV would have the normal flight controls such as ailerons, a rudder, and elevators. Although the deployed UAVs would behave autonomously, each would communicate with other UAVs during the task.

## Motivation

Two factors motivate this work. The first is the desire to see whether we can create the controlling UAV software automatically, that is, without having to write the software ourselves. While employing a person to program the UAV application would certainly be possible, the task would be difficult and time consuming. To create the controlling UAV software, we use a genetic programming model. The primary output of the genetic programming model is the application source code, which could be compiled and downloaded onto the UAV to provide instructions to the UAV to accomplish its tasks. The second factor motivating this work is the desire to see whether the program that is developed by our genetic programming model exhibits any novel approaches to solving this problem. In other words, would the genetic programming model discover more efficient ways of gathering information and finding lift regions than a human programmer would?

## Relation to Previous Work

This work extends the work done by Pryor on developing behaviors for tracking robots described in *Developing Robotic Behavior Using a Genetic Programming Model*. In a typical robot-tracking problem, robots are initially distributed randomly in a field and given the task of locating a source that is emitting some kind of signal (smell or sound). A robot's behavior program provides instructions to its motor control system to move to the source location while navigating around obstacles that lie in the robot's path. As in the current UAV application, a genetic programming model was used to create the robot's controlling computer program.

The UAV problem is more difficult than the robotics problem because more objectives are required to accomplish the task. In the robotics problem, there was a single objective: to get to the source of the signal. The UAV problem, on the other hand, has four objectives. First, the UAV must get to the lift region. The vehicle is initially given only enough altitude to reach the lift region. Second, the UAV must gain sufficient altitude to accomplish its mission. Third, the UAV must make its flight to gather information. And last, the UAV must return to the lift region before crashing into the ground.

In the sections that follow, we describe the methodology employed for the UAV problem in a logical step-wise fashion. The goal is to provide an overview of the whole process so that the reader has a good understanding of how the problem of developing the UAV's maneuvering program was solved.

## Genetic Programming Model

Genetic programming is one of many types of genetic algorithms that use evolutionary or adaptive processes to solve practical problems. Holland's pioneering book *Adaptation in Natural and Artificial Systems* provides a general framework for such analysis. Many books have since been written on genetic algorithms, with Goldberg's *Genetic Algorithms in Search, Optimization, and Machine Learning* ranking among the best. The most informative source on the theory of genetic programming is Koza's *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. This book is very well written, provides an excellent bibliography, and fills in much of the detail not provided in this report.

So what is genetic programming? Genetic programming is a methodology. When a computer program employs this methodology, it produces as output the source code of another computer program. This source code can then be compiled and executed. Unlike most computer programs, a human programmer does not write these programs. Instead, the programs are said to evolve in a biological setting, with rules of natural selection and survival of the fittest playing an important part in their evolution.

Evolution occurs in discrete steps called generations. A generation is composed of a population of individuals, each of which is a complete computer program. The size of the population can vary depending on the problem; however, hundreds, if not thousands, of programs are typical. Some programs (individuals) will be very effective at doing the prescribed task—some will not. Each program is scored for applicability, and its fitness is given a numerical score. The higher the fitness, the better the individual. The goal is to evolve the very best program that solves the problem of interest. In this application of genetic programming, we are trying to find the program for the UAV.

The solution strategy is to improve upon these evolving programs by creating successive generations of more fit individuals. To create the next generation of individuals, the genetic operators of selection, reproduction, crossover, and mutation are used. The purpose of selection is to choose an individual from the current population. In general, this individual will be better than most, but may not be the very best. Reproduction moves a selected individual directly into the next generation. Crossover uses the selection operator twice to select two parents from the current population that will be mated in some way to form an offspring that will be placed in the next generation. Mutation uses the selection operator once to choose an individual that will be mutated (changed) in some way and then placed in the next generation. The four genetic operators are discussed in more detail in later sections of this report.

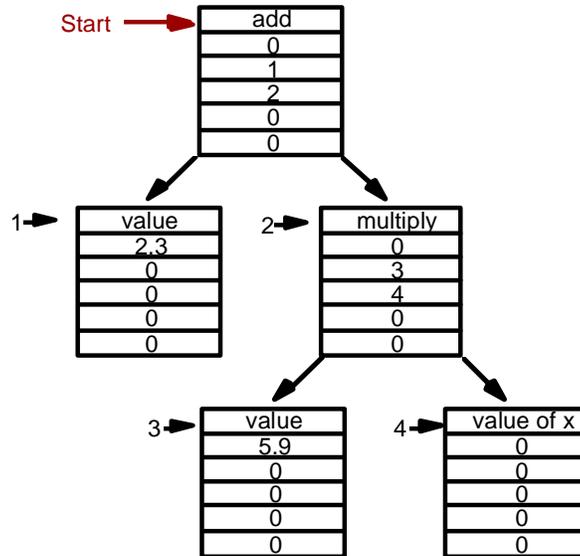
The evolutionary calculation described above proceeds across many generations until a single individual is found that meets the convergence criterion. This controlling program is then saved for use by the UAVs.

## Program Representation

This section describes how the individual programs are represented within the genetic programming model. The representation should allow complete flexibility in defining programs, yet it must also ensure that the performance of the genetic operations is not too cumbersome. A tree-like structure best meets these requirements.

The basic building block of a tree is called a *node*, with all nodes in the tree having the same fixed structure. The first element of a node specifies the node type, which can either be a function or a terminal. A *function node* performs a mathematical or boolean operation and generally has branches (nonzero pointers) that point to other nodes. The number of branches depends on the *kind* of function, e.g., add, subtract, multiply. A *terminal node* normally returns a value, does not have any branches (all pointers are zero), and terminates that section of the tree. Other elements within a node are a value position and pointers to other nodes.

Consider the example tree shown in Figure 1.



**Figure 1. An example tree.**

This tree has five nodes and is three levels deep. The tree is evaluated by starting at its root, or top, and working downward until a terminal node is reached. A terminal node returns a value that is then processed upward in the tree.

To evaluate the example tree, we begin at the first node denoted by “Start,” which is a function node whose kind is specified as "add." This kind of function node points to two other nodes that will return values that will be summed by the add node. At pointer 1, there is a terminal node that returns a constant value of 2.3. At pointer 2, there is a function node whose kind is "multiply." This node points to two other nodes: one (at pointer 3) is a value node that returns the value of 5.9, and the other (at pointer 4) is a value node that returns the value of a global variable  $x$ . These two values will be multiplied by the multiply node, which will return the resultant to the add node above it. The tree is equivalent to the expression

$$y = 2.3 + 5.9x,$$

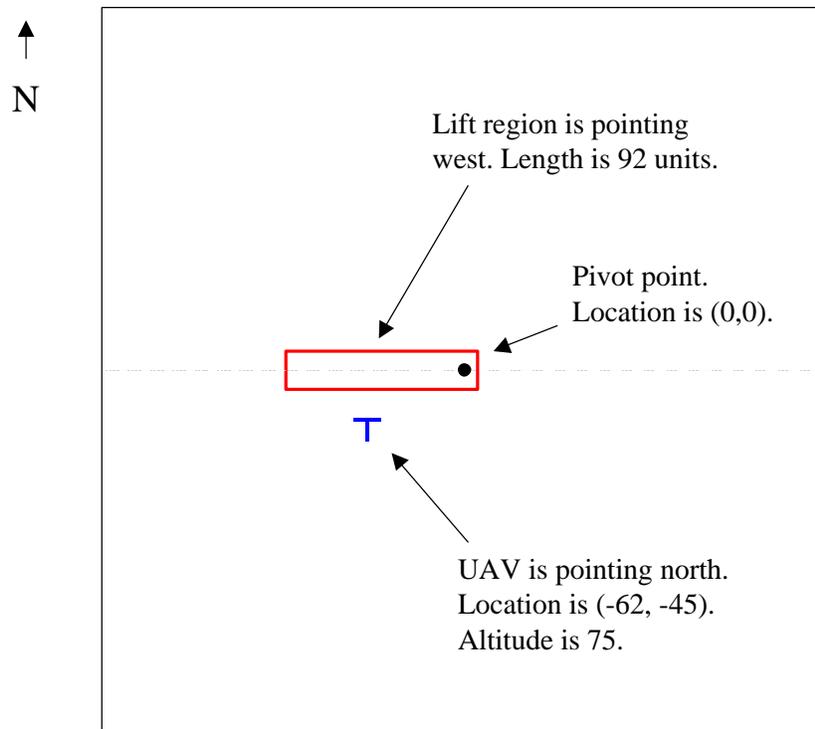
where  $y$  is the value returned by the root node at the top of the tree. The tree used in the UAV program has many more function and terminal types than the example tree and is also much larger than the example tree.

## Problem Definition

In the process of evolving the optimum behavior, the UAV will be given about a thousand problems to solve. At the start of each problem, the UAV will be placed on a grid at a random ground-level position with a random initial direction. The grid is three-

dimensional  $(x, y, z)$  with the first two dimensions,  $x$  and  $y$ , presenting the UAV's ground-level coordinates. The third dimension,  $z$ , represents the UAV's altitude, which can range from zero for ground level to a maximum height of 250 units. The ground-level space is square, with coordinates ranging from  $-300$  to  $+300$  units on each side. The initial direction of the UAV can be facing north, east, south, or west. The UAV's altitude is set so that there is just sufficient height for the vehicle to reach the lift region before striking the ground. The direction and size of the lift region, like the initial position and direction of the UAV, are randomly set. The lift region pivots about the center of the coordinate system and can be pointed in any of four directions—north, east, south or west. The length of the lift region ranges from 80 to 100 units; and its width is 10 units, with 5 units on either side of the pivot point. The initial position of the UAV is no more than 75 units from the pivot point. At each time step, the UAV moves one unit in the direction that it is facing. If the UAV is in the lift region, it gains one unit of altitude per time step. If the UAV is outside the lift region, it loses one unit of altitude per time step. The UAV has 600 time steps to complete the problem (mission).

Figure 2 illustrates an example configuration at problem start-up. The T-figure represents the UAV, the rectangle represents the lift region, and the large dot denotes both the center of the ground coordinate system and the pivot point for the lift region. The coordinates of the UAV, the size and direction of the lift region, and the altitude of the UAV are also shown.



**Figure 2. Example initial conditions for UAV problem.**

## Definition of Functions and Terminals

This section defines the set of functions and terminals that are used by our genetic programming model. It is important to note that our selection of a set of functions and terminals is not unique, nor is there any theory that indicates whether any one set is better than another. The only way to determine the effectiveness of a set is to try it out and see whether it works.

The set of 23 functions and terminals used in our UAV application are listed in Table 1. There were 7 functions (Index 1–7) and 16 terminals (Index 8–23). It should be noted that all nodes return a value, even if the nodes only direct movement of the UAV.

**Table 1. List of Functions and Terminals**

<b>Index</b>	<b>Name</b>	<b>Description</b>	<b>Number of Pointers</b>
1	ROOT	Root node. Returns the value from below unchanged. The number of pointers depends on the tree structure considered.	1–5
2	ADD	Returns the sum of its two branches.	2
3	SUBTRACT	Returns the difference of its two branches.	2
4	MULTIPLY	Returns the product of its two branches.	2
5	DIVIDE	Returns the division of its two branches. A test is made to ensure that a division by zero is not possible.	2
6	IFGTEQ	This is a conditional node. Branch 1 is evaluated, and if greater than zero, this node returns the results of branch 2; otherwise, it returns the result of branch 3.	3
7	STORE REGISTER	Returns the value of its single branch but also stores this value in the register specified in the node header.	1
8	RETURN X-POSITION	Returns the value of the UAV's <i>x</i> -position.	0
9	RETURN Y-POSITION	Returns the value of the UAV's <i>y</i> -position.	0
10	RETURN DIRECTION	Returns the UAV's direction. A value of 1 is north, 2 is east, 3 is south, and 4 is west.	0
11	RETURN ALTITUDE	Returns the UAV's altitude.	1
12	RETURN LIFT	Returns a value of 1 if the UAV is in the lift area; otherwise, it returns a value of 0.	0

13	TURN NORTH	Turns the UAV toward the north. Returns a value of 1.	0
14	TURN EAST	Turns the UAV toward the east. Returns a value of 1.	0
15	TURN SOUTH	Turns the UAV toward the south. Returns a value of 1.	0
16	TURN WEST	Turns the UAV toward the west. Returns a value of 1.	0
17	TURN HOME	Turns the UAV toward the center of the coordinate system (pivot point).	0
18	RECALL REGISTER	Returns the value of the register specified in the node header.	0
19	RETURN VALUE	Returns the value stored in the node header.	0
20	RETURN ADF1	Calls ADF1 and returns its value.	0
21	RETURN ADF2	Calls ADF2 and returns its value.	0
22	RETURN ADF3	Calls ADF3 and returns its value.	0
23	RETURN ADF4	Calls ADF4 and returns its value.	0

Automatically Defined Functions, or ADFs, are function calls that can be made from within the main tree. ADFs are discussed below in the section named "Calculation Results."

## Tree Generation

From previous discussion you may recall that the genetic operators are used to create the next generation of behavior programs (trees) from the current generation, but you may be wondering how the first generation was created. The answer is that the first generation of trees is created randomly. To generate a tree, a recursive function is called that needs to know only the current level or position in the tree. We define level 1 as the top of the tree, or root. Level 2 is the level just below the root. Level 3 and the remaining levels follow.

The level number is initially set to zero. When the recursive function is called, it first increments this level number by one and then checks on its value. If the level number is equal to one, the recursive function inserts a ROOT node, defines a single new node, sets a pointer to the new node, and then calls itself pointing to the new node.<sup>1</sup> On all other

---

<sup>1</sup> For some tree structures, more than one pointer may be defined.

levels, the recursive function selects randomly from the functions and terminals, defines the appropriate number of new nodes and pointers, and then calls itself for every new node.

Three parameters are used to control the size of a tree. The parameter `MINTREESIZE` specifies the minimum number of levels in a tree, while the parameter `MAXTREESIZE` specifies the maximum number of levels in a tree. These parameters are used in the following way. If the level number is less than `MINTREESIZE`, the recursive function will only select from the *function* node kinds, e.g., add, subtract. This constraint ensures that at least one more level will be added. If the level number is equal to `MAXTREESIZE`, the recursive function will only select from the *terminal* node kinds. This constraint ensures that no more levels are added to this part of the tree because terminal nodes do not have any branches. For all other level values, a random selection is made. The parameter `MAXNODES` specifies the maximum number of nodes in a tree. When a tree gets too large, it is deleted and a new tree is created.

## Fitness Evaluation

The most important aspect of the genetic programming methodology is the evaluation of tree fitness. Fitness provides the selection pressure that drives the programs to the desired behavior. The fitness of each tree in a population is evaluated independently. The evaluation involves testing a tree against a set of problems that have been created randomly, as previously discussed. After all problems have been run, a fitness score for the tree is computed.

A problem description consists of the initial position, direction, and altitude of the UAV, and the direction and size of the lift region. We select these parameters using a random number generator. Once these initial conditions are specified, a simulation calculation is performed. The UAV moves step-wise according to instructions provided by the behavior program. There is a finite number of steps in a simulation, and during each step the UAV is permitted to turn to a given direction, to go straight ahead, or to turn in the direction of the pivot point (home). If the UAV ends a time step inside the lift region, its altitude is incremented by one; otherwise, its altitude is decremented by one. After the prescribed number of time steps has elapsed, `MAXSTEPS`, the maximum distance from the pivot during the simulation, is then recorded. When all of the problems have been run for the current tree, an average maximum distance is computed. The fitness for the tree is set to this average, subject to penalty conditions:

$$\text{Fitness} = \text{Average Maximum Distance.}$$

The maximum distance for a problem is set to zero if the UAV's altitude ever became negative, or if the flight ended with insufficient altitude for it to reach the lift region before

striking the ground. To prevent trees from becoming exceedingly large, we reduce the tree fitness by a slight amount depending on the number of nodes in the tree:

$$\text{Fitness reduction} = \alpha [\text{number of tree nodes}],$$

where  $\alpha$  is a small constant.

The maximum fitness a tree can obtain is 215. This assumes that the UAV takes into account the variation in the size of the lift region, which ranges from 80 to 100 units long, and that the maximum altitude of the UAV is 250 units. The best tree we found yielded a fitness of 214.23.

## Creating the Next Generation

The operators of reproduction, crossover, and mutation are used to create the next generation of individuals. Each operator works independently of the other operators, and the usage of each operator is determined by its assigned probabilities. The reproduction, crossover, and mutation operators use the selection operator to determine the individuals in the current population that will be acted upon.

The population size POPSIZE of each generation remains the same. To create the next generation, a loop over the necessary individuals is started. Within the loop, a random number is drawn and three probabilities are compared. The probability PROBRP determines the percentage of times within the loop that reproduction is used. Likewise, PROBCR and PROBMU determine the percentage of times that crossover and mutation are used, respectively. The sum of the three probabilities is 1.0. From the random number and the three probabilities, an operator is selected that then creates one new individual in the next generation. When the required number of individuals is created, the loop is terminated.

### Selection Operator

The selection operator is used to identify individuals in the current population for reproduction, crossover, or mutation operations. A tournament algorithm is used,<sup>2</sup> which is easy to implement and is relatively fast. The algorithm works in the following way. NTOUR individuals are randomly selected from the population. The fitness values of these individuals are compared, and the individual with the highest fitness is the winner of the tournament, that is, the one selected. If two individuals are needed, as in the case of crossover, the tournament is repeated. Note that the parameter NTOUR affects the distribution of individuals selected, and thus increasing its value moves the distribution

---

<sup>2</sup> Several different algorithms were considered, such as roulette wheel selection, but the tournament algorithm worked well and it was fast.

toward more elite individuals in the population. For example, if the value of NTOUR is equal to the population size, only the most fit individual will be selected. Reducing the value of NTOUR improves diversity by allowing more individuals to take part in forming the next generation.

## **Reproduction Operator**

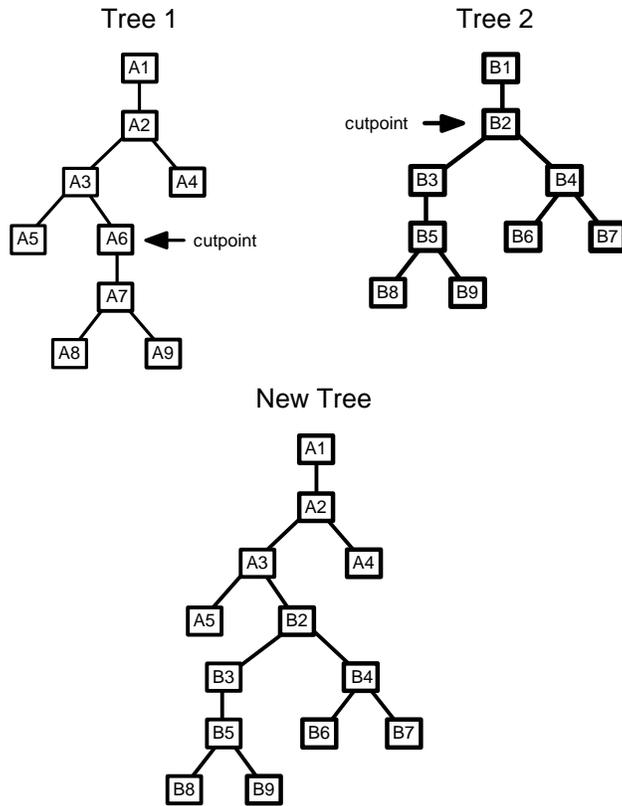
The reproduction operator is the simplest of the operators used. Reproduction makes an exact copy of a selected individual in the current generation and places it into the next generation. The fitness of this individual is saved so that the fitness value will not need to be recomputed.

## **Crossover Operator**

The crossover operator is slightly more complicated than the reproduction operator. Crossover is done in three steps. First, the selection operator is called twice to select two individuals from the current population. Next, for each selected individual, a cutpoint node is randomly selected among the nodes of its tree. Finally, the new tree is created by removing the cutpoint node and all nodes below it from the first tree and replacing them with the cutpoint node and all nodes below it from the second tree. Figure 3 illustrates this last step. The new tree, which was created by splicing together two trees in the current population, is then placed in the next generation.

## **Mutation Operator**

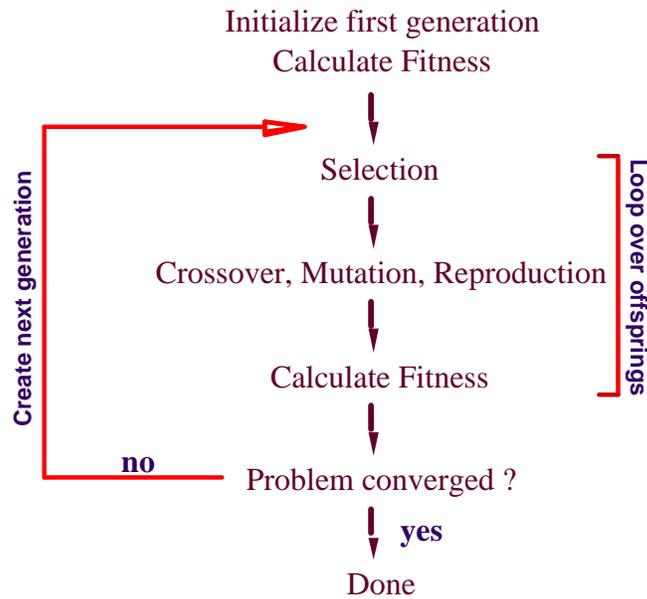
The mutation operator removes part of an existing tree and replaces it with a randomly generated new part, using the same recursive function that created the first generation. A tree is selected from the current population, and a cutpoint node is randomly selected among its nodes. That node and all nodes beneath it are then removed. Subsequently, the recursive function is called at the cutpoint location to generate a new part of the tree. The modified tree is then placed in the next generation.



**Figure 3. Illustration of the crossover operator.**

### Solution Procedure

Figure 4 illustrates the basic solution procedure. The recursive function discussed previously creates the first generation. The fitness of each individual in the population is then determined, followed by the initiation of a loop over generations. Within the generation loop, the next generation is created and the fitness of each individual within that generation is calculated. A test is then made to determine whether any individual meets the convergence criterion (a value greater than 214). If an individual is found, the loop terminates and the calculation ends. If no individual is found, the calculation continues.



**Figure 4. Solution procedure used.**

The calculations were done on the CPLANT massively parallel computer at Sandia National Laboratories. Running on the parallel computer required some slight modifications in the basic solution procedure to allow sharing of the “best” tree among processors. The number of processors allocated to a calculation varied, with the average being about 64. A processor is identified by its number, whose range is 0 to the number of processors allocated minus 1. In our implementation, each processor had the same genetic program and ran independently of the other processors. At the end of each generation, each processor would determine the best tree in its population and send it to processor 0 where the globally best tree would be determined. The send would only occur if the local best tree had a larger fitness than the global best it had received earlier. Processor 0 would then broadcast the new globally best tree to all processors. Accordingly, each processor would then decide whether the globally best tree would be employed in creating the next generation, using an algorithm that depended on the generation number and its processor number. If a processor decided to use the globally best tree, the processor inserted that tree into the current generation to replace the trees that had the smallest fitness. In this way, the globally best tree would be included in all of the genetic operations that produced the next generation.

The reason for not using the globally best tree all the time is to maintain diversity in the entire population. The convergence rate is proportional to a measure of the diversity. If all processors used the globally best tree in each generation, it would not be long before all of the trees would look much the same, and the rate of improvement would be reduced.

Processor 0 performs two additional tasks. It writes an ASCII representation of the globally best tree to disk for restart purposes and simulation studies, and it also writes an

equivalent C source-code version that can be used in simulators that cannot use the ASCII tree. The ASCII representation is discussed further in the next section.

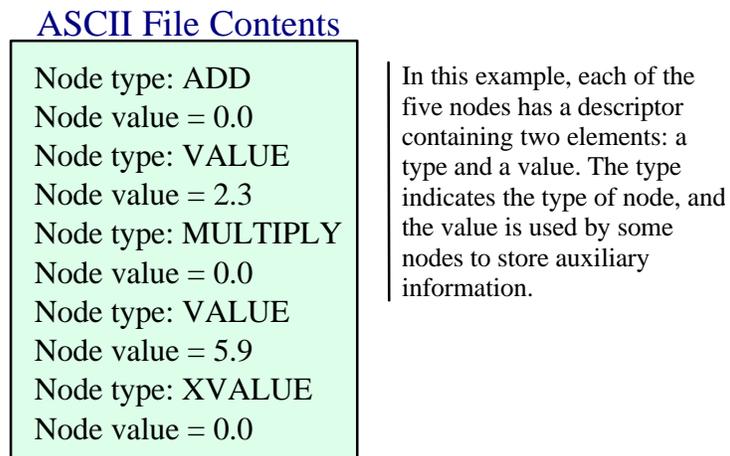
In a companion SAND report titled *An Evaluation of the Convergence Properties of a Parallel Genetic Programming Method*, Barton and Pryor give a detailed discussion of the parallel genetic programming parameters used in the UAV study.

## ASCII Tree Representation

The genetic programming calculation could be restarted by reading in the globally best tree from a previous run. We stored this tree to a disk file in an ASCII format. This same file can be read by a simulator for validation and investigative studies or used to create an executable program on an actual glider UAV. Because this ASCII file is somewhat novel, it is presented here.

The interpretation of the ASCII file assumes that a given genetic programming function has a fixed number of pointers. We write the file node by node, starting at the top of the tree and moving downward. During this downward traversal, the pointers are resolved from left to right. When a terminal is reached, the next left-most pointer of the function directly above is then resolved. If that function has no pointers remaining, we move to the next left-most pointer of the function above. We continue in this fashion until all of the tree nodes are written. Reading the file to recreate the tree is done in the same way, resolving nodes from top to bottom and pointers from left to right. If a value needs to be stored, as in the case of the value node, the node descriptor can contain the value.

An example of an ASCII file for the tree shown in Figure 1 is given in Figure 5.

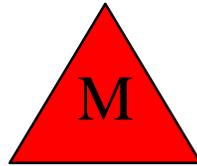


**Figure 5. ASCII file example.**

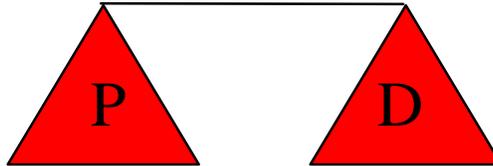
## Calculation Results

In computing the UAV's behavior, we considered four different tree structures. These structures are illustrated in Figure 6. The triangles represent trees containing hundreds of nodes and tens of levels deep. An example of how any one of these trees might appear is shown in Figure 7, which follows the discussion of the tree structures.

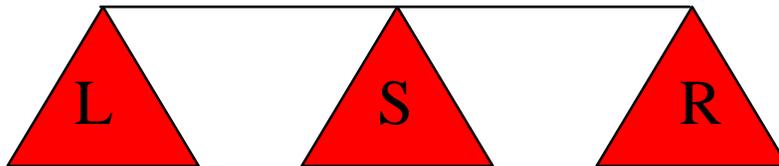
Tree structure A



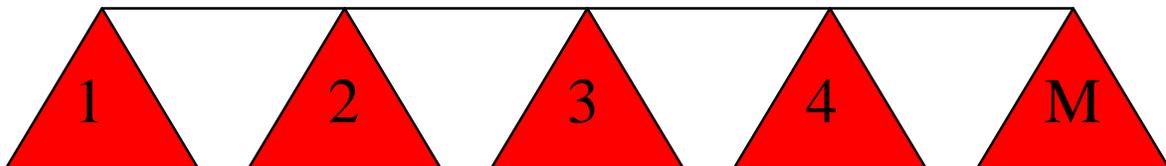
Tree structure B



Tree structure C



Tree structure D



**Figure 6. Four tree structures considered.**

For all tree structures, we ran 10 independent cases on the CPLANT computer. Each case used 64 processors, and each processor contained a population of 2000 trees (POPSIZE). The maximum tree size was set to 1500 nodes, and the maximum number of

levels was set to 100. The number of registers used for storing information from one time step to another was set to 20. The maximum number of generations was set to 300. The mutation probability (PROBMU) was set to 0.1, and the crossover probability (PROBCR) was set to 0.8, leaving the reproduction probability (PROBRP) equal to 0.1.

Tree structure A. This tree structure contained only a main tree (M). It contained no ADFs (automatically defined functions). The tree would be exited when a turn command terminal was executed. The best fitness achieved was 192.32. This tree contained 1105 nodes and was 93 levels deep. The tree had problems resolving the variation in size of the lift region. The UAV would return to the lift region before it needed to, thus not effectively using all of its altitude to gain distance from the pivot point.

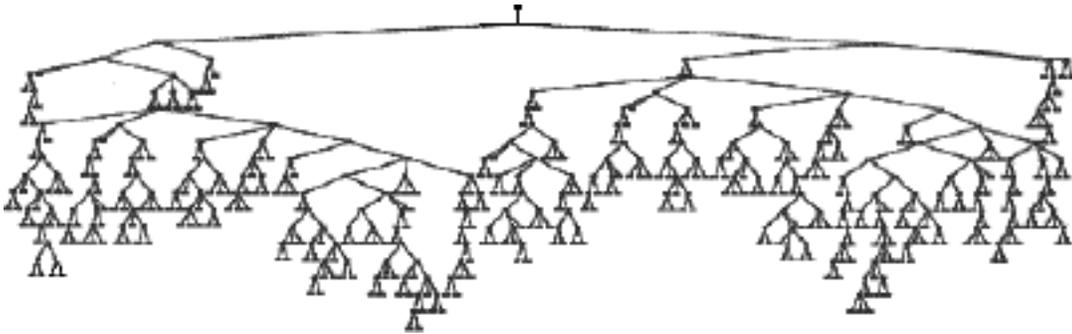
Tree structure B. This tree structure used two trees. At each time step, the ponder tree (P) would be called, followed by the decision tree (D). The ponder tree could read all of the environmental data (UAV positional and lift parameters) and could also read from and write to the registers; however, the ponder tree could not execute any turn commands. The decision tree could not read any of the environmental data, but it could read and write to the registers and execute the turn commands. Communication between the trees occurred through the registers. Neither the ponder tree nor the decision tree used ADFs. The intent in using two trees like this was to separate some of the behavior logic from the processing of environmental data so that learning would proceed more efficiently. We did see an improvement over structure A, as the best fitness achieved was 205.04. Like structure A, structure B contained a large number of nodes. Together, the ponder and decision trees had a total of 1232 nodes.

Tree structure C. This tree structure used three trees. The turn commands were turned off and no ADFs were used. At each time step, the three trees were evaluated. The tree with the largest output would indicate whether the UAV should turn left (L), go straight ahead (S), or turn right (R). Each tree could read from and write to the registers and had access to the environmental data. Of the ten runs, the best fitness achieved was 183.05. Like structure A, this structure had problems resolving the size of the lift region. Also, we observed that one direction of the lift region was generally not handled as well as the other three directions. The three trees were about equal in size, with the total number of nodes averaging about 1100.

Tree structure D. This tree structure used four ADFs (1–4) and a main tree (M). ADFs are callable programs that are similar to function calls in normal programming. ADFs could only be called from the main tree, i.e., one ADF could not call itself or another ADF. The main tree and the four ADF trees had access to the registers and the environmental data. The ADFs and the main tree were constructed using the same algorithm and were treated as branches off the root node. Crossovers could mix ADFs and main trees from different members of the population.

This structure gave nearly perfect results. The best fitness found was 214.23. For this tree, the total number of nodes was 1401, and the number of levels was 59. For all of the

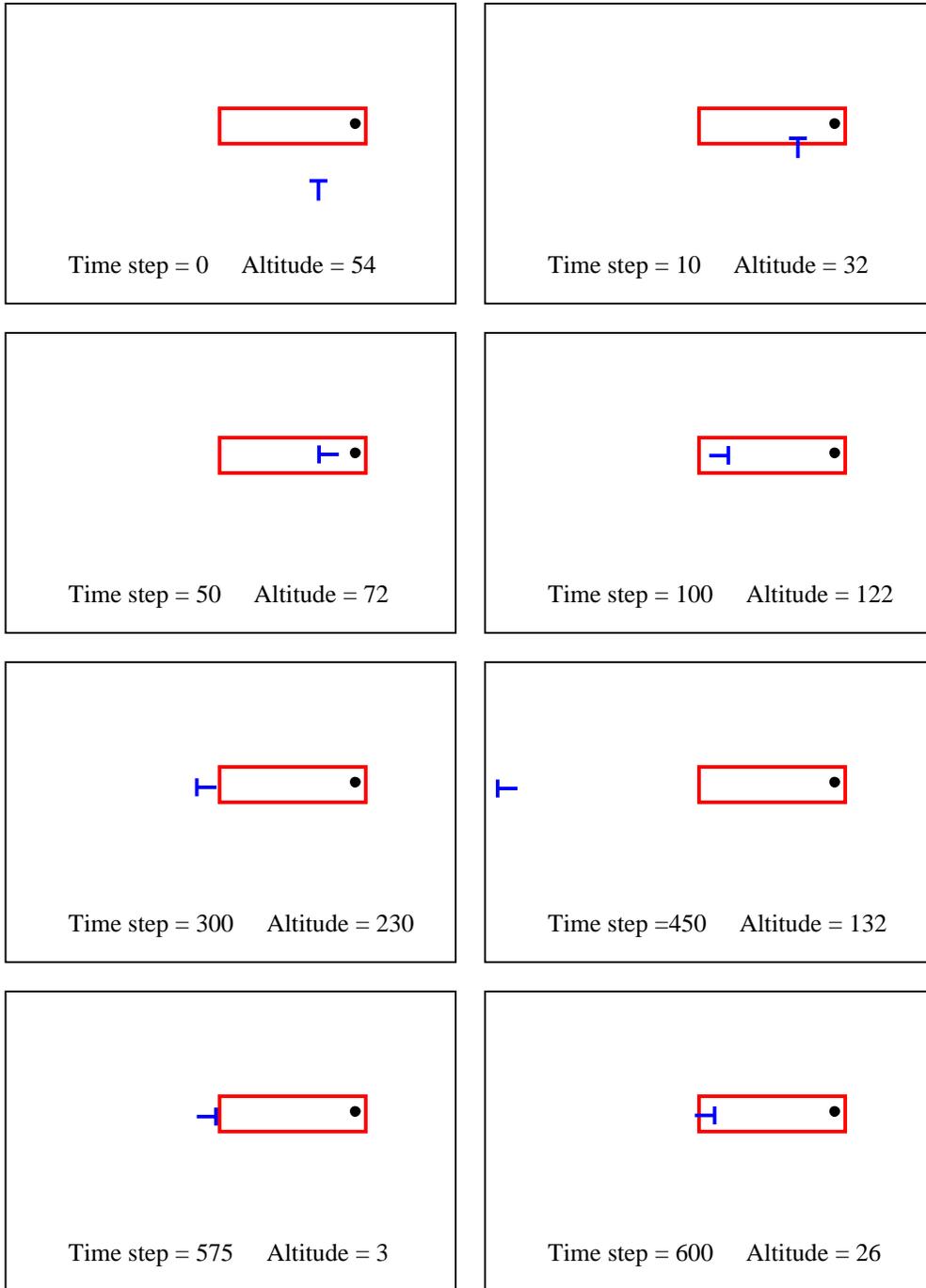
runs with this structure we had set the Fitness Reduction constant  $\alpha$  to zero. When we set the constant to 0.0005 for this tree on a restart calculation, its size decreased to 487 nodes and 26 levels with no decrease in the (uncorrected) fitness. Clearly, the tree had a lot of tree branches and nodes that were not being used. We also noted that one of the four ADFs was never called.



**Figure 7. Illustration of a typical tree found in this study.**

### **History of a Simulation**

To highlight characteristics of the implementation, we have included a brief history of one simulation. In this simulation the lift region was pointing north. Its size was 83 units. The initial position of the UAV was  $(-40, -32)$ , and its initial altitude was 54 units. Figure 8 shows this history at eight different steps during the transient.



**Figure 8. Results of a typical simulation calculation.**

Table 2 lists the parameters used in the calculations of the sample simulation.

**Table 2. Calculation Parameters**

<b>Parameter</b>	<b>Description</b>	<b>Value</b>
FLDDIM1, FLDDIM2	Dimensions (width, length) of the field used in the simulations	(600, 600)
MINTREESIZE	Minimum number of tree levels	8
MAXTREESIZE	Maximum number of tree levels	100
MAXNODES	Maximum number of tree nodes	1500
MAXALT	Maximum UAV altitude	250
ALPHA	Penalty function constant, $\alpha$	1.0e-6
MAXSTEPS	Number of simulation steps performed per problem	600
PROBSIZE	Number of problems run for fitness calculation	1200
POPSIZE	Number of trees on each processor of CPLANT	2000
NTOUR	Number of individuals taking part in tournament selection	4
PROBRP	Reproduction probability	0.1
PROBCR	Crossover probability	0.8
PROBMU	Mutation probability	0.1
NODEX	Number of Paragon processors—range (min, max).	(100, 1800)

## Summary

The genetic programming model successfully produced a behavior for a glider UAV. The best behavior was one that employed ADFs (tree structure D), and its fitness was very close to the theoretical maximum. The computations were performed on Sandia's massively parallel CPLANT computer at reasonable computing costs.

We have found that genetic programming models offer two advantages over more traditional methods for determining UAV behavior. The first, and perhaps most important benefit, is that new and sometimes novel solutions to problems are found—ones that we might not have considered. This occurs because we do not constrain the solution. In genetic programming we provide the tools (functions and terminals) and a goal to reach. The genetic program determines the best way to solve the problem. The second benefit is that the solutions appear to be more robust because many problem conditions and variations are investigated in the solution process.

We developed two simulators, one for the Macintosh and one for the PC, and placed them on our web site. Both simulators use tree structure D for their behavior. The

simulators can be found at <ftp://ftp.cs.sandia.gov/pub/rjpryor>. This site also contains other papers on this subject.

## **Future Direction**

The work on using genetic programming for UAV behavior is far from complete. More complex situations than the ones treated in this report are likely to be encountered as UAVs enter actual field operations. For example, decisions beyond movement, such as collision avoidance and operations in adverse conditions, must be considered. From our experience in developing this application, we believe that the genetic programming model can handle these conditions.

We are now beginning to use genetic programming in war-gaming applications. The challenge is reproducing the behavior of soldiers in battlefield operations and the coordination of troops needed to accomplish a mission. For example, genetic programming models for war-gaming applications need to take into account that individual soldiers are part of a command structure and thus act not only as individuals, as do the UAVs in this study, but they also cooperate as a group to achieve the command goal.

## References

Goldberg, David E. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading, MA: Addison-Wesley, 1989.

Holland, John H. *Adaptation in Natural and Artificial Systems*. Ann Arbor, MI: University of Michigan Press, 1975.

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

Pryor, Richard J. *Developing Robotic Behavior Using a Genetic Programming Model*. SAND98-0074. Albuquerque, NM: Sandia National Laboratories, 1998.

Barton, Dianne, and Pryor, Richard J. *An Evaluation of the Convergence Properties of a Parallel Genetic Programming Method*. Albuquerque, NM: Sandia National Laboratories, (Draft report, September 2002).

## Distribution

1	MS	0321	W. J. Camp, 9200
1	MS	0188	C. Meyers, 1030
1	MS	1170	R. D. Skocypec, 15310
1	MS	0316	M. D. Rintoul, 9212
1	MS	0451	J. E. Nelson, 6515
1	MS	0785	D. L. Harris, 6516
1	MS	0318	G. S. Davidson, 9200
1	MS	0318	M. Boslough, 9212
40	MS	1109	R. J. Pryor, 9212
40	MS	1109	Dianne Barton, 6515
1	MS	0165	N. Singer, 12640
1	MS	9018	Central Technical Files, 8945-1
2	MS	0899	Technical Library, 9616
2	MS	0612	Review and Approval Desk, 9612, For DOE/OSTI