SANDIA REPORT
SAND2003-2928
Unlimited Release
Printed October 2003

Developing Close Combat Behaviors for Simulated Soldiers Using Genetic Programming Techniques

Mark J. Schaller and Richard J. Pryor

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94AL85000.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865)576-8401
Facsimile: (865)576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.doe.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800)553-6847
Facsimile: (703)605-6900
E-Mail: orders@ntis.fedworld.gov
Online order: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2003-2928
Unlimited Release
Printed October 2003

Developing Close Combat Behaviors for Simulated Soldiers Using Genetic Programming Techniques

Mark J. Schaller and Richard J. Pryor
Evolutionary Computing and Agent-Based Modeling
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-1110

Abstract

Genetic programming is a powerful methodology for automatically producing solutions to problems in a variety of domains. It has been used successfully to develop behaviors for RoboCup soccer players and simple combat agents. We will attempt to use genetic programming to solve a problem in the domain of strategic combat, keeping in mind the end goal of developing sophisticated behaviors for compound defense and infiltration. The simplified problem at hand is that of two armed agents in a small room, containing obstacles, fighting against each other for survival. The base case and three changes are considered: a memory of positions using stacks, context-dependent genetic programming, and strongly typed genetic programming. Our work demonstrates slight improvements from the first two techniques, and no significant improvement from the last.

Table of Contents

Introduction.....5
Simulation Training.....5
Genetic Programming.....6
Program Morphology in Genetic Programming.....7
Genetic Operations.....10
The Soldier Problem.....11
The Fitness Function.....12
The Soldier Design.....13
The Soldier GP Function Set.....15
Advancements.....18
Developing a Superior Fitness Function.....18
Enhancement of Strategic Functions.....19
Contextual Genetic Programming (CGP) 22
Genetic Operations with CGP.....23
Manually Defined Contexts.....25
Results.....27
Accomplishments and Future Research.....29

Developing Close Combat Behaviors for Simulated Solders Using Genetic Programming Techniques

Introduction

Consider that modern combat situations differ significantly from those known and researched by military strategists from previous generations. Adversaries are more likely than ever to use sophisticated weapons and communication technologies, and modern battlefields resemble open fields no longer. To maintain America's military strength, our armed forces need to prepare themselves for combat in unique environments that resist the application of antiquated combat techniques. More than ever before, it is important for our soldiers to train in a variety of environments against intelligent adversaries who use a variety of technologies. Simulated combat, where soldiers compete against agents in virtual or augmented environments, is one available tool for providing that necessary training.

Simulation Training

Simulation has many compelling advantages over real-life war gaming. A training exercise in a simulated environment can be replayed multiple times with little additional cost. Virtual environments permit training supervisors to review extremely detailed log files, which could lead to a better understanding of trainees' errors. Also, training supervisors can effect major structural changes in combat environments without concern for the cost of physically reconstructing a training environment, and in much less time.

Simulation also entails different costs than traditional wargaming. In addition to the cost of purchasing and caring for the hardware required for realistic combat simulations, there would be a high cost in developing and maintaining the software that creates the simulation. There would also be a significant cost in developing realistic, intelligent simulated adversaries. Sandia National Laboratories is currently engaged in research that will help mitigate these costs. One instance is the UMBRA system, which provides a physical simulation environment that can bring together models that vary tremendously in their size and complexity. Another research effort, principally directed by Rich Pryor in organization 9216, Evolutionary Programming and Agent Based Modeling, seeks to develop a system to automatically produce simulated soldiers with intelligent combat behavior by using genetic programming and other evolutionary programming techniques.

This paper describes the most recent efforts of Rich and myself over the last year. We set out to build a system that develops sophisticated behaviors for soldier agents

automatically. Because genetic programming has been successfully used in the past to create intelligent agents in similar domains, such as the RoboCup softbot competition (Luke, 1998), and has proven successful in previous experiments involving agents that must prioritize and balance several goals (Pryor, 2002), we decided to use genetic programming techniques in our system. In this paper, we will document the processes used to create and refine that system, and our accomplishments; it will also describe the directions we would like to take the research, and what we believe should be possible for such a system to do.

Genetic Programming

Genetic programming is a maturing methodology that has attracted the interest of hundreds of researchers in artificial intelligence and evolutionary algorithms. Well-developed descriptions of genetic programming can be found in previous papers in this sequence (see Pryor, 1998 and 2002), or in any of several books. The seminal work in genetic programming, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, was written by John Koza, the methodology's inventor, and is extremely accessible and informative. We will include only the briefest of summaries here, in order to provide a reference against which the reader may compare the changes we made.

A program that employs genetic programming repeatedly creates collections, called populations, of programs. These populations will hold anywhere between a few dozen to hundreds of thousands of individual programs. Each time a genetic program (or GP) creates a population, it replaces its previous population. Each iteration of this process is called a generation. With the exception of the programs in the GP's first population, which are randomly generated, all the programs in a population are constructed out of fragments of programs from the previous population, in a way that is similar to how the genetic material from a generation of a biological species is generally recycled from the genetic material of its preceding generation. This analogy is the root of the convention that computer scientists use when they refer to genetic programming and similar methodologies as evolutionary, or genetic, algorithms. (There is some delicacy in these naming conventions, for while some may refer to genetic programming as an instance of a genetic algorithm, the phrase "genetic algorithms" refers to an ideologically similar but distinct methodology.)

The real world contributes to natural selection by prohibiting individuals in a species from reproducing and thereby prohibiting their genetic material from contributing to subsequent generations of the species. Similarly, a genetic program evaluates the individual programs in its populations and, when constructing a new population, prefers to use fragments of code from individuals that evaluated well over fragments of code from individuals that evaluated poorly. As long as an individual program's performance in these evaluations is based on its code, this process will lead to the proliferation of well-evaluating programs in the GP's population.

The evaluation function applied to the programs reflects some problem the user would like to see solved. For instance, if the user would like to evolve a program to

control a UAV, and direct it to survey an area, the evaluation function—more commonly called the fitness (i. e., survival of the fittest) function—might simulate the flight of a UAV over some landscape, and return the amount of ground it surveyed. (It should also return whether the UAV crashed.) Populations from later generations are likely to contain programs that instruct a UAV to cover a lot of ground, and not crash.

The primary philosophy behind genetic programming should now be clear, although several details have been glossed over so far, such as how individuals in a population can represent programs, and how those programs are executed. After all, a human programmer writes code in a programming language like C++, but it is quite unlikely that a program formed by gluing together fragments of C++ programs—even quite capable programs—will even compile, let alone perform the tasks of the previous programs, let alone perform those tasks in a better way! The standard solution to this problem, faced by many evolutionary programming methodologies, has been to encode programs in structures that are amenable to fragmentation and reconstruction. The structure used by genetic programming is the tree.

Program Morphology in Genetic Programming

In genetic programming, each individual is a tree (in the data structure sense), and each node in a tree is a function. These individuals may be called *programs* (because they are sometimes executed), *individuals* (because they are generally found in a population), *trees* (because they are structurally represented as trees), and when the desired programs are used to control agents, they may also be called *behaviors*. The nodes that make up trees may also be called *atoms* or *functions*, and a particular node's children are either called its *children*, its *parameters*, or sometimes its *branches*.

Every GP has a way of parsing, or interpreting, the trees used by its individuals. A GP needs to know what kind of atoms will be used inside its trees, and how to translate those atoms into behavior relevant to the problem the GP has been designed to solve. The set of all available functions used to build programs can be thought of as a kind of genetic alphabet. The programmer of a GP decides which functions go into that alphabet. It is hopefully evident that the choice of functions used to create programs in this manner is an extremely important decision to make while designing a genetic program. Different problems will almost certainly require different function sets. Also, the set of functions chosen by a GP programmer to solve a problem is seldom obvious or unique. Ideally, a great deal of knowledge, experience, and intuition goes into choosing a particular function set.

The number of ways a GP could parse its trees is nearly as numerous as the number of problems one could use a GP to solve. Here, I will not attempt to list all the different ways, but I will describe two problems and the ways a GP may parse its trees in order to solve them, in the hope that the descriptions will explain the variety of approaches genetic programming makes possible, and elucidate the claims already made about how a GP works.

Example One: Best-fit Functions

Imagine that you have a collection of x-y data points, but you don't know what function was used to create them. There are many techniques you could use to find the function, and genetic programming is one. To apply genetic programming, first decide what the role of the individual is: in this case, each individual shall represent a function *f* that takes a single input, *x*, and outputs a value, *f(x)*. Then, decide what the fitness function should be: in this case, for each individual *f*, it shall be the sum, for each data point (x,y), over the difference between the data point's y value and the corresponding value f(x). Finally, determine the GP's function set: in this case, one possible function set consists of the functions +, -, *, /, ^, each of which has two children, the functions sqrt(), abs(), sin(), each of which has one child, and the "function" x, and a variety of constant values between -1.0 and 1.0, none of which have any children.

In this way, each individual represents an arithmetic function on x. The way the GP parses an individual's tree is simple: the output of the program is the value returned by the node at the root of the tree, and each function's parameters are provided by the values returned by its subtrees. In this case, it may seem unreasonable to call each individual a program, because their operations are so clear and more commonly thought of as mathematical, but certainly they are programs—programs that take a single floating point input and produce a single floating-point output.

A GP defined as above would produce population after population of candidate functions, and there would be a tendency for later populations to contain individuals whose functions fit the data points better and better. If it is possible for a function to be constructed out of the mathematical operations described above to match exactly all of the data points, then it is also possible for this GP to find an exact solution to the problem. However, it might be the case that there exists a mathematical function that matches exactly the data points, but which is also impossible to construct using the operations included in the function set. In such a case, the GP would produce better and better approximations of the solution, but it would never "solve" the problem—get a perfect answer. In order to solve the problem, it would be necessary to change the function set.

In a case such as this, if we have access to the mathematical function that produced the data points, it would be clear how to change the function set. However, in more complex problems, when we cannot be certain whether a perfect solution can be reached using a particular function set, the job of changing the function set is much harder, and the reasons why should be clear. In fact, this is an important consideration that I will return to later in this paper, as it bears on the problem we have sought to solve.

Example Two: Agents in a Simple Task Environment

Let me provide another example of how a GP might operate. Consider the problem of developing behaviors for an agent in a simulated 2-d environment in which there are randomly deposited resources. Suppose that the goal of the agent is to pick up, move, and drop resources, so that after a certain amount of time, the environment has a

concentrated pile of resources instead of having them scattered around. This is a complex problem with an unobvious solution, but the rules are simple, and the capabilities of the agent are limited, and so it lends itself well to genetic programming. Let each individual be a program that directs the behavior of the agent, and let the fitness function be a measure of the concentration of the environment's resources after letting the agent act within the environment for a limited time, directed by the program of the individual being tested.

How should the function set for this problem be constructed? Well, the agent must be able to sense its environment, and act in it. Its functions should allow environmental analysis, and action. Some descriptions of sensory functions that we would expect to find in a GP used to solve this problem are:

- How many resources are right in front of me?
- How many resources am I carrying now?

Some action functions we'd expect to also find are:

- Drop all my resources.
- Pick up all the resources here.
- Move forward.
- Turn 90 degrees to the left.

In order to construct behaviors for an agent out of these functions, we would need to adapt all of these qualities so that they could be combined in a tree format. Another important factor to consider is that in standard genetic programming, one restriction on the function set is that every function must return the same data type. This is because the operations used by a genetic program to manipulate program trees do so without considering the functions they are moving. We'll describe those operations shortly, but first we need to describe how the resource-gathering problem can be expressed in the language of GP. This is especially important because our soldier problem is similar.

Here is one way of making accessible the functionality we need. Every function shall return to its parent a floating point value, so that the sensory functions mentioned above return reasonable values. By default, whenever an action function is evaluated, it shall return 0. 0, but also immediately do its action as a side effect if possible. In addition, we shall add a function, "If GT" that takes four subtrees as input, and evaluates and returns the value from its third subtree if the value of its first subtree is greater than the value from its second subtree, and evaluates and returns the value from its fourth subtree otherwise. This function allows behaviors to take actions based on conditions. Also, we could add addition, subtraction, multiplication, MAX(..), and MIN(..) functions that take multiple inputs and provide the appropriate output, and a collection of functions that return useful constant values (perhaps between 0 and 100, say, or something on the scale of the concentration of resources in the environment).

This is a very different example than the data point matching example, but fundamentally we're still using the same tools: a fitness function to evaluate the performance of programs, programs that are represented as trees, and nodes that correspond to functions that are appropriate within the problem domain. The GP operates similarly in both cases, using the same operations to build individuals out of other individuals, and evolve effective answers.

Genetic Operations

Genetic programs perform three operations to create new programs out of old programs. They are reproduction, mutation, and crossover, and operate on a single program (in the cases of reproduction and mutation) or on a pair of programs (in the case of crossover). Reproduction copies a program in whole from one population to the next. Mutation copies a program into the next population, but it also selects a subtree in that program, deletes it, and constructs a new, randomly configured, subtree in its place. Crossover selects a subtree in each program and exchanges them, putting the newly formed pair of programs into the next population. These operations preserve and reconfigure genetic material from one generation into the next.

It is important to consider the strategy of selecting which programs get operated upon. Genetic programming works by selecting better—often called fitter, in acknowledgement to the fitness function—programs more often than worse, or less fit, programs. An example of such a selection strategy is called tournament selection. When a genetic program that employs tournament selection intends to choose a program to perform reproduction, mutation, or crossover on, it considers two programs chosen at random from its population, and chooses the fitter program. This strategy makes selection biased towards fitter programs, which is desirable. Other selection strategies might sort all the programs in a population by fitness, and then use that ordering as the basis for preferred selection of fitter programs—for instance, by selecting from the top 20% of programs, 80% of the time. There are a variety of selection functions, but they all bias program selection towards fitter programs.

It should be clear by now that there are many ways to configure the methodology of genetic programming to a particular problem, and many choices that must be made when designing a genetic program. All these decisions are important, and can change the efficiency of a genetic program when applied to a specific problem. Unfortunately, there is no "best" configuration for genetic programming—one implementation may perform well on problem A and poorly on problem B, and another may do the opposite. In order to choose an implementation wisely, a genetic program's designer should take into consideration his or her specific problem definition.

The Soldier Problem

Our goal is to develop behaviors appropriate for simulated soldiers in a combat environment. An image of the simulated environment follows:

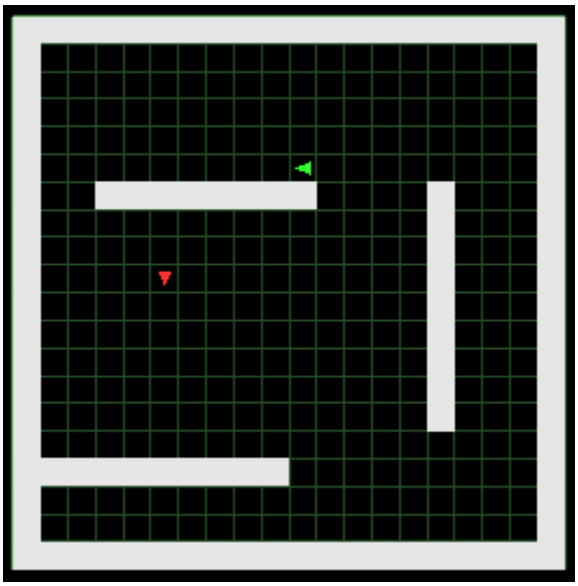

Figure 1. An Image of the Simulated Close Combat Environment

The room is a discrete, square grid, twenty units on a side, with four outlining and three interior walls. The walls obstruct the sight of the agents, and also prevent bullets from passing through them. The top of the room (as portrayed in Figure 1), by convention, is "north," and the other directions follow. Two agents play through the simulation, a "good soldier" and a "bad soldier." Each soldier occupies one non-wall grid location and faces one of eight directions. Time is discretized into steps; during a step, an agent may turn as often as it wishes and may take one action: move (forward, to either side, or backward) to an adjacent square, shoot, or do nothing. An agent that is facing a grid corner direction, such as NE, will move "farther" in a single timestep than an agent facing a direction such as N, but that effect is ignored for the sake of simplicity.

Each agent is equipped with a gun, although the skills involved in aiming it have been abstracted away. The factors involved in determining whether an agent successfully shoots its opponent are: how far apart the agents are, whether the shooter has been injured, whether the target is moving, and, naturally, whether the shooter can see the opponent. (Agents cannot shoot through walls.)

Agents can see in a 135-degree arc centered in the direction they are facing. An agent that can see its opponent can also determine the direction its opponent is facing.

An agent cannot hear another agent move; they can, however, hear another agent shoot. An agent that is shot once is considered injured, and an agent that is shot twice is considered dead. Mobility is not compromised when an agent is injured, although gun accuracy is reduced. The test ends when either agent is killed or when 100 timesteps have elapsed.

When a simulated battle begins, each agent is placed into the room in a random location, facing a random direction. If the agents are placed into the room so that they could see each other without moving (that is, if a wall doesn't separate them), then one of the agents is relocated. In each timestep, each agent may act, making their decision by analyzing the sensory information available from the current timestep, as well as any stored state that the agent might have, by executing their program. Each agent remembers whether it has seen its opponent, and if so, where the opponent can be found; also it remembers whether it has successfully injured its opponent, and knows whether it is injured. Agents also maintain additional memories, as will be described later.

In this simulation, the evolved programs always control the good soldier. A simple and effective script always controls the bad soldier. The bad soldier does not move around the playing field; it turns towards any gunshot it hears, and always fires on the good soldier if the good soldier is visible.

The Fitness Function

The goal of the genetic program is to produce a program that is sophisticated enough to win more simulated battles than it loses. Therefore, we want to use a fitness function that returns high values of fitness whenever it is passed a program that wins more often than it loses, or ties. Throughout the course of our experimentation, we developed many fitness functions, but they all were of the form:

1. Gather performance statistics from many (between fifty and a thousand) simulations.
2. Perform a function on those statistics that returns a higher value when desirable behavior, such as winning more often than losing, occurs more often, and a lower value when undesirable behavior, such as spinning in circles or getting killed, occurs more often.
3. Return that value as the program's fitness.

There are two points that we would like to make clear from what we just described. First of all, our fitness functions, even the functions that run through a thousand simulations to evaluate a single program, cannot be perfect measures of a program's fitness. Ideally, the fitness function would test every possible initial condition—and in some problems, that is possible—but in our problem, there are easily over a hundred thousand (perhaps nearly a million) initial conditions, so we settled for what we hoped would be a representative sample. Additionally, the simulation relies on some probabilistic effects, such as gunfire. These effects lead to some imprecision in the fitness function, but our intent in creating a fitness function was to measure the

approximate ability of each program, and use enough tests to reduce that measurement's imprecision while keeping the computational requirements of the evaluation low.

Secondly, because we used population sizes of between 200 and 20000 programs, and because each program might need to be tested in the simulator fifty times, and because each test in the simulator required up to one hundred timesteps, and environmental information needed to be calculated every timestep, we chose to use look-up tables for most of our environmental calculations. Tests performed during our research determined that using look-up tables speeded up the genetic program by a factor of five. Nevertheless, our optimized program is extremely computationally intensive; assuming that every program execution results in a mere hundred function calls before deciding on an action (our programs contain between 300 and 2000 functions, or nodes on the tree), and that tests last on average a mere twenty timesteps, and that only fifty tests are used to evaluate a program, and that our populations consist of a thousand programs, a billion function calls will occur every ten generations. (Note that our genetic programs typically ran for hundreds of generations; pretty soon we'll be talking about serious computational requirements!)

In the end, our most powerful genetic program is capable of producing programs that win the simulator battles between 75% and 79% of the time. Getting to this point took a great deal of experimentation, and a great deal of learning. To explain what we learned, we will describe what technologies we began with, and then examine each change we made and the results of that change.

The Soldier Design

The project began as the third phase in a multiyear effort to evaluate genetic programming as a methodology that could be used for developing soldier behaviors. The first phase investigated the use of genetic programming to produce behaviors for simple robots. A full account of that research can be found in Pryor, 1998. The second phase determined that genetic programs were capable of developing agent behaviors that pursued several goals simultaneously. For more details on that work, involving the creation of behaviors that control UAVs, see Pryor, 2002. The third phase would lead to the actual development of soldier behaviors through genetic programming.

The genetic programming system we began with came from UAV glider research. It was written in C, used some special memory allocation techniques, and could run on a multi-processor machine. In order to make the system work on the soldier problem, we changed its evaluation function so that it would create an instance of the soldier simulation instead of the glider simulation, and we changed its function set so that evolved programs could control soldier agents. We also added code that created look-up tables for the simulation, as well as a mechanism that could read and write the tables to disk, so that they wouldn't have to be calculated every time the GP ran.

One additional feature that we inherited from the UAV code had to do with the soldier's data structure. The data structure used by the UAV GP to represent the UAV included an array of registers that the behaviors could use to store and recall arbitrary

numeric data. We included that functionality in our genetic program, and augmented it by reserving the first three registers for important data: the position and orientation of the bad soldier. The simulation initializes those registers to zero at the beginning of each test. When the good soldier finds the bad soldier, the latter's positional information is stored in those registers. In other respects, those registers are treated as read-only.

The agent structure we began with looked like this:

```
struct Soldier {
      long        SoldierID;
      long        SoldierXPOS;
      long        SoldierYPOS;
      long        SoldierDIR;
      long        SoldierSTAT;
      long        SoldierACTION;
      long        SoldierSEENENEMY;
      long        SoldierSHOTENEMY;
      long        SoldierLASTMOVE;
      double      Register[NUMOFREG];
}
```

Although the memory concepts of "have I seen my enemy" are clearly boolean in nature, the variables `SoldierSEENENEMY` and the other boolean-like variables are declared as longs, and the function implementations query their state using statements such as

```
if (SoldierSEENENEMY == 1)
...
```

in order to provide for possible extensions of functionality. The variable `SoldierSTAT` described the health of the soldier agent, which could be: healthy, injured, or killed. The variable `SoldierACTION` was used by the behaviors as output; it encoded the chosen action, which could be: do nothing, shoot, move forward, move backward, move left, move right, or undo the last movement.

We also directly used the node structure from the UAV genetic program, and reused the node's data member `regIndex` in function 14, `DISTANCE IN DIRECTION`. The node structure we began with looked like this:

```
struct Atom {
    long     atomKind;
    double   atomValue;
    long     regIndex;
    long     atomHitCount;
    struct   Atom *ptr1;
    struct   Atom *ptr2;
    struct   Atom *ptr3;
    struct   Atom *ptr4;
    struct   Atom *ptr5;
    struct   Atom *nextFree;
};
```

The `atomKind` member described the type of function the node represented, the `atomValue` member held whatever internal floating-point parameter was required by the function, if any (the constant value in the case of function 25, for instance), and the Atom pointers led to the node's children, or were NULL. The `atomHitCount` member kept track of how frequently the node was evaluated in the tree, for debugging and statistical purposes.

The Soldier GP Function Set

Our initial function set included 29 functions, although some functions performed multiple tasks. 19 of those 29 functions take no parameters. A table describing all 29 functions follows.

Table 1. List of Functions Originally in the Soldier GP

Index	Name	Description	Number of Children
1	RETURN	Root node. This node is always and only found at the root of program trees. Executes and returns the value from its first child. The first child leads to the primary tree. Children 2 through 5 lead to the ADF subtrees.	5
2	ADD	Returns the sum of its two children.	2
3	SUBTRACT	Returns the difference of its two children.	2
4	IFGTZ	Child 1 is evaluated, and if greater than zero, this node evaluates and returns the results of child 2; otherwise, it evaluates and returns the result of child 3.	3
5	IF I AM HURT	If the soldier is injured, this node evaluates and returns child 1; otherwise, child 2.	2
6	IF I CAN SEE IT	If the soldier can see its opponent, this node evaluates and returns child 1; otherwise, child 2.	2
7	IF I HAVE SEEN IT	If the soldier *has* seen its opponent, this node evaluates and returns child 1; otherwise, child 2.	2
8	IF I HAVE HURT IT	If the soldier has injured its opponent, this node evaluates and returns child 1; otherwise, child 2.	2
9	IF WE SEE EACH OTHER	If the soldier can see its opponent, and its opponent can see it, this node evaluates and returns child 1; otherwise, child 2.	2
10	STORE IN REGISTER	Returns the value of its single child but also stores this value in the register specified in	1

Index	Name	Description	Number of Children
		the node header, in `regIndex`.	
11	MY X POS	Returns the soldier's X position. (1-19)	0
12	MY Y POS	Returns the soldier's Y position. (1-19)	0
13	MY DIRECTION	Returns the soldier's direction. (1-8)	0
14	DISTANCE IN DIRECTION	Returns the distance between the soldier and the closest wall in the direction specified in the node header, in `regIndex`; this specified direction is relative to the soldier's direction.	0
15	TURN LEFT	Turns the soldier 45 degrees CCW. After turning, if the soldier can see its opponent, then this function returns a value of 1. Else, 0.	0
16	TURN RIGHT	Turns the soldier 45 degrees CW. After turning, if the soldier can see its opponent, then this function returns a value of 1. Else, 0.	0
17	TURN AROUND	Turns the soldier 180 degrees. After turning, if the soldier can see its opponent, then this function returns a value of 1. Else, 0.	0
18	MOVE FORWARD	Moves the soldier forward one grid space. This function also terminates the behavior tree execution.	0
19	MOVE BACKWARD	Moves the soldier backward one grid space. This function also terminates the behavior tree execution.	0
20	MOVE LEFT	Moves the soldier left one grid space. This function also terminates the behavior tree execution.	0
21	MOVE RIGHT	Moves the soldier right one grid space. This function also terminates the behavior tree execution.	0
22	UNDO MOVE	Moves the soldier to the previously vacated grid space. This function also terminates the behavior tree execution.	0
23	SHOOT	Shoots at the opponent, if visible. If the opponent isn't visible, the soldier shoots anyway, but this only has the effect of producing sound. This function terminates the behavior tree execution.	0
24	RECALL REGISTER	Returns the value of the register specified in the node header.	0

Index	Name	Description	Number of Children
25	VALUE	Returns the value stored in the node's `atomValue`.	0
26	ADF1	Calls ADF1 and returns its value.	0
27	ADF2	Calls ADF2 and returns its value.	0
28	ADF3	Calls ADF3 and returns its value.	0
29	ADF4	Calls ADF4 and returns its value.	0

To clarify: functions that end program execution do so after effecting the appropriate side effect. Also, ADFs are important constructions that can be used by genetic programs to provide some of the advantages of code reusability. When a program is executed, execution is passed to the root node, which is always the RETURN function. It has five children, and as described above, its first child is always evaluated. However, whenever an ADF terminal function is executed, control goes to a child of the RETURN node—so that when ADF1 is executed, RETURN's second child is executed, and ADF1 returns the value returned by RETURN's second child. This has the effect of allowing the genetic program to store code for repeated use, since any ADF terminal function may be used several times in the primary subtree.

There is some finesse required to use ADFs in genetic programming. Imagine what would happen if one of the ADF subtrees included a call to another ADF subtree, which included a call to the first ADF subtree! The genetic program has to be careful to prevent recursive ADF calls from happening, or else a program's execution might never terminate. Our genetic program manages that responsibility by checking each newly constructed program for ADF calls in ADF subtrees, and replacing any that it finds with other, harmless, terminals. However, our genetic program does not distinguish between ADF subtrees and any other subtree while performing its genetic operations, so code from ADFs and primary subtrees can mix.

The last major component of the UAV GP that we changed was its fitness function. We wrote a new fitness function that rewarded individuals for doing everything that we considered valuable, and punished behaviors for doing things that we considered disadvantageous. The list of good things included: moving around the room, seeing the opponent, shooting the opponent, killing the opponent, winning the combat without being injured, and winning the combat sooner rather than later. The list of bad things included: being shot, and being killed. At first, we used a point system that rewarded each of these behaviors with different numbers of points. These numbers were:

1/20 pt for having moved around the room
10 pts for having seen the opponent
50 pts for having shot the opponent
50 pts for having killed the opponent
50 pts for having killed the opponent without being shot by it
1/2 pt for each extra moment of time at a won simulation's end
-15 pts for being shot by the opponent
-80 pts for being killed by the opponent

To find the fitness of a program, the GP would run through 120 simulations, scoring each one using the formula above, and giving the individual its average score as its fitness. When selecting programs for genetic operations, the genetic program employed tournament selection: considering two programs, chosen at random from the population, and choosing the one with higher fitness.

We ran the genetic program on two systems: a dual processor PC with a gigabyte of memory, and a CPlant cluster, where we used between four and sixteen processors. On the PC, our populations held 2000 individuals, and on the cluster, the GP evolved up to 32,000 individuals simultaneously. We discovered the best behavior on the cluster, which won its battles about 60% of the time, and won battles without being injured about 30% of the time. We did not feel that this was a success—we believed that at least a 90% success rate was possible—and so we began to consider what was limiting behavior development, and how to address those limitations.

Advancements

Developing a Superior Fitness Function

An immediate concern of ours was the fitness function we employed to score individuals. By experimenting with its formula, we eventually discovered a function that encouraged the GP to evolve better behaviors. Again, the GP ran a large number of simulations, and counted the number of times the soldier was able to find the enemy, kill the enemy, and kill the enemy perfectly. It also counted the number of times it failed to kill the enemy, and how often it died in combat.

The behavior was then assigned a fitness based on one of three formulas, designed to encourage the development of desirable behaviors in a forgiving way. In the beginning, it was most important for behaviors to move around the environment and find the enemy soldier, so if the soldier wasn't able to find its enemy more than 50% of the time, its fitness would be exactly the probability (between 0.0 and 0.5) of it locating its enemy. If the soldier was able to find its enemy more than half of the time, then its fitness also began to depend slightly on its combat ability, but still mostly on its ability to find the enemy. The formula we used for this was:

$$\text{fitness}_2 = 5 * (F_1 - F_2) + S_1 + 2 * P_1 + \text{fitness}_1$$

where

F_1 = # of times the soldier finds the enemy soldier
F_2 = # of times the soldier fails to find the enemy soldier
S_1 = # of times the soldier kills the enemy soldier
P_1 = # of times the soldier kills the enemy soldier without being hurt itself
$fitness_1$ = the probability (between 0.0 and 1.0) of the soldier locating the enemy

Finally, once a behavior could find the enemy soldier more than 90% of the time, the focus shifted to evolving strong combat ability. The final formula used in this case is:

$$\text{fitness}_3 = [300 * (P_1 - (D_1 + S_2))] + [100 * (S_1 - (D_1 + S_2))] + [10 * P_1] - [10 * F_2] + \text{fitness}_2$$

where

D_1 = # of times the soldier is killed by the enemy soldier
S_2 = # of times the soldier fails to kill the enemy soldier

This fitness function effectively guided behaviors, and we witnessed an improvement to a 65% success rate. We had hoped for much more, but began to believe that the solution that would take us to 90% had to involve a more fundamental change.

Enhancement of Strategic Functions

One of the fundamental limitations we first sought to remove was the limited long-distance planning mechanisms available to the evolved soldier behaviors. In each time step, behaviors could decide only to move one unit distance in any direction. They were, of course, capable of moving long distances, but not as the result of a single decision—a long distance movement could be built only out of a sequence of decisions to move short distances. We believed that the ability to decide, in a single calculation, to move a long distance should be available to the behaviors, and implemented a memory module, similar to the registers, that stored grid locations. It held two stacks of positions, and each could be accessed and modified through the execution of new functions that we added to the function set. The two stacks were named `IHaveBeen` and `IWannaGo`, loosely after their predicted use. The ability to store and retrieve arbitrary positions was immediately useful in the sense that we were also able to develop a powerful function that calculated the nearest position, safe from the enemy soldier, as long as the good soldier had previously seen its opponent.

We also believed that easier access to the soldier's own position would be useful, to facilitate the development of behaviors that were specific to regions of the room. That concern was addressed with the introduction of two new functions, each of which controlled tree execution based directly on the soldier's position. Finally, because of the large number of newly introduced functions that had important side effects, we introduced two functions that simply passed control to either all their children, or one of their children, chosen randomly during execution. All these new functions are included in a table, here.

Table 2. List of Functions Added to the Soldier GP during Strategic Enhancement

Index	Name	Description	Number of Children
30	DO TWO	33% of these functions (chosen at creation,	2

Index	Name	Description	Number of Children
		not during execution) will pass execution to one of its children, and return its output. The rest will execute child 1, then execute child 2, and return the maximum of their outputs.	
31	DO THREE	33% of these functions (chosen at creation, not during execution) will pass execution to one of its children, and return its output. The rest will execute child 1, then child 2, and then child 3, and return the maximum of their outputs.	3
32	IFXGT	If the soldier's X coordinate is greater than this node's `regIndex` (chosen at creation randomly between 1 and 20), execute and return the output from child 1. Otherwise, child 2.	2
33	IFYGT	If the soldier's Y coordinate is greater than this node's `regIndex` (chosen at creation randomly between 1 and 20), execute and return the output from child 1. Otherwise, child 2.	2
34	IF IWG CLOSE	If the top position on the `IWannaGo` stack is less than 2 grid spaces away, execute and return the output from child 1. Otherwise, and if the stack is empty, child 2.	2
35	IF IHB CLOSE	If the top position on the `IHaveBeen` stack is less than 2 grid spaces away, execute and return the output from child 1. Otherwise, and if the stack is empty, child 2.	2
36	PUSH MY POSITION	Pushes the soldier's current position to the top of the `IHaveBeen` stack.	0
37	PUSH TARGET POSITION	Pushes a grid space onto the `IWannaGo` stack. This space is calculated using this node's `regIndex` variable as a direction, and its `atomValue` variable as a distance, and making them relative to the soldier's position and orientation.	0
38	DISTANCE TO IHB	Returns the distance between the soldier's current position and the top `IHaveBeen` position. Returns 0.0 if that stack is empty.	0
39	DISTANCE TO IWG	Returns the distance between the soldier's current position and the top `IWannaGo` position. Returns 0.0 if that stack is empty.	0

Index	Name	Description	Number of Children
40	PUSH SAFE POSITION	If the soldier has seen its opponent, this pushes onto the `IWannaGo` stack the closest position that is visible to the soldier, and that its opponent cannot see.	0
41	TURN TO IHB	If the `IHaveBeen` stack is not empty, this causes the soldier to turn towards its top position. After turning, if the soldier can see its opponent, then this function returns a value of 1.	0
42	TURN TO IWG	If the `IWannaGo` stack is not empty, this causes the soldier to turn towards its top position. After turning, if the soldier can see its opponent, then this function returns a value of 1.	0
43	MOVE TO IHB	If the `IHaveBeen` stack is not empty, this causes the soldier to move towards its top position. This function also terminates the behavior tree execution.	0
44	MOVE TO IWG	If the `IWannaGo` stack is not empty, this causes the soldier to move towards its top position. This function also terminates the behavior tree execution.	0
45	POP IHB	If the `IHaveBeen` stack is not empty, this pops the top position off the stack.	0
46	POP IWG	If the `IWannaGo` stack is not empty, this pops the top position off the stack.	0

Although making these changes fundamentally changed the operation of the soldier behaviors, we did not see much improvement in the fitness, or level of sophistication, of the behaviors. We witnessed a slight improvement in the success rate, from 65% without stacks to 72% to 74% with them, but the results were not as dramatic as we would have liked. In part this was probably due to the broadening of the search space of programs that the GP has to navigate, because with more functions, and more node types, there are more possible trees. However, we noticed no pronounced slowdown in program evolution, so those effects were most likely not the sole cause for the mediocre gains.

We believed our problem was that we had not yet confronted the core problem preventing our GP from finding good solutions. While the older UAV GP was successful at developing behaviors that optimized towards several goals simultaneously, the soldier problem had more complicated and interrelated goals that required a variety of behaviors, and when we reflected on this fact, we thought that the GP design that solved the UAV problem might not be able to solve the soldier problem at all. We began to believe that

evolving a single program that provided effective behavior for each subtask was too difficult, but that a collection of programs, each specialized for a particular task, could, when taken together, provide the necessary amount of sophistication to make behaviors that succeeded at the soldier problem 90% of the time.

But how could a new GP, a contextual GP, work with contexts? We considered two options: describe the different contexts by hand, or let the definitions of the contexts evolve themselves. A contextual GP, trying to evolve contexts automatically, and then evolve specialized behaviors for each context, could be even more computationally expensive than a traditional GP. On the other hand, if we provide the GP with contexts designed by humans, we could decrease the computational requirements of evolving complex behaviors, but also sacrifice the useful qualities that depend on the GP discovering solutions automatically. In the end, we tried both approaches.

Contextual Genetic Programming (CGP)

We had to develop a new framework for doing genetic programming that worked with contextually specific behaviors. We introduced several new classes of objects into the GP ecosystem: genes, each of which was defined as a context and a behavior paired together, and individuals, which were collections of genes. In some tests, the number of genes per individual differed amongst individuals in a population, and in others, the number of genes per individual was fixed. In any case, we needed to develop new implementations for the familiar tasks of selection, mutation and crossover that respected genes and the contexts and behaviors that they consist of, as well as a new operation.

First, it is necessary to describe how contexts are implemented. Again, with this task, we had to experiment to find an appropriate method of implementation. We settled on a method that defined contexts using points in a high dimensional space, in which each dimension corresponded to a particular environmental or internal measurement. A context was assigned a single point in that space; at each time step, the soldier agent would decide which context it was in by calculating the distance from its state, environmental and internal, to every one of its contexts' points, and the context associated with the closest point won out. Naturally, we had to be careful in how we defined the distance metric for this space—some environmental measurements, such as coordinate position, had values that could vary between 1 and 20, and others, such as "have I been injured" and "can I see the enemy soldier", had Boolean values that could only take on one of two distinct values. To calculate the distance between points in this space, we used a metric that consisted of a sum of scaled differences, and we compensated for the varying ranges by applying scalar factors to each dimension that weighted each one roughly the same.

Behaviors did not have to change at all—the tree structure used in our previous genetic program was appropriate.

Genetic Operations with CGP

We had to change the way genetic operations affected the more structurally complex individuals in keeping with the reasons we developed our contextual approach.

Selection

The operation of selection, on the level of individuals, did not have to change. Since individuals still had their own fitness values, in order to select one individual, two individuals were pulled out of the population, and whichever one had a higher fitness rating was selected.

Crossover

Crossover, typically considered the most important tool in genetic programming, had to change to work with genes and contexts. Its early stage remained the same—first, using the selection operator, select two parent individuals. The rest, however, is different. The next task is to randomly choose one individual to be the recipient of the others genetic material. Then, within the chosen individual, choose a gene at random to be the recipient in the crossover. At this point, one gene in one individual has been chosen, so one context and its behavior tree have been identified. Next, we choose one of the genes in the other individual to be the donating participant in the crossover, but this decision can be made in one of two ways: the donor gene may be randomly chosen, independent of the first chosen gene, or alternatively, the selection of the donor gene may be influenced by the particular recipient gene. In our contextual GP, both methods are utilized, with the program randomly determining one or the other as the operation is executed.

The particular method of influence exerted by the first gene upon the second is as follows: the chosen donor gene is the one whose context's point is the closest to the point of the first gene's context. We designed this based on our (hopefully reasonable) assumption that it would be more useful to exchange code between behaviors that are trying to solve similar problems than it would be to exchange code between behaviors that may be specializing towards different tasks. The fact that two contexts have points that are close together implies that the behaviors paired with each context are trying to solve similar problems. Naturally, it is also conceivable that behaviors optimized towards different tasks could still benefit from sharing code, so both types of decisions are used, although the first occurs more often (60%).

The procedure for exchanging code between behaviors is the same as in classical genetic programming, but when we allowed the GP to evolve its contexts automatically, we also implemented a procedure to "crossover" contexts. The procedure described above is used, and in addition to receiving genetic material from the donor gene, the recipient gene's context is also changed by moving its point to the midpoint between its old position and the position of the donor gene's point. For non-continuous dimensions, the midpoint is rounded to the nearest valid value. (For Boolean dimensions, this is rather boring; if the two points involved differ for a particular Boolean value, then the final value is randomly true or false.)

Mutation

We changed mutation predictably, while taking into consideration one important factor. Mutation proceeds as follows: the selection operation is performed to choose one individual with good fitness. After that, one of two procedures take place—either a single gene is randomly chosen for mutation, or a collection of genes is chosen. The rationale for this is that smaller changes, such as those limited to a single gene, are less likely to be destructive to an individual, and more likely to realize incremental gains. However, without the ability to make several changes to several genes simultaneously, some high fitness regions of the answer space may be inaccessible to the genetic search. This phenomenon is easily illustrated by a simple example. Consider an individual with two genes, each of which can have one of two behaviors.

(State of the individual: fitness of the state)	Context X with Behavior 1	Context X with Behavior 2
Context Y with Behavior a	State 1a: Fitness 10.0	State 2a: Fitness 5.0
Context Y with Behavior b	State 1b: Fitness 5.0	State 2b: Fitness 30.0

In the table above, if we restrict the mutation so that is modifies only one gene at a time, an individual in state 1a will almost certainly never reach state 2b, although 2b is a more fit state (The indeterminacy of the selection operation could permit an individual in states 2a or 1b to survive and mutate, but this is unlikely). Therefore, it is important to permit both single-gene and multiple-gene mutation.

Regardless, when a gene is chosen for mutation, its behavior tree is always mutated, and when the GP is allowed to automatically evolve contexts, there is a small chance that the gene's context is also mutated, by moving its point a small distance in a random direction.

Specialization

Although the genetic processes described so far encompass the functionality necessary to improve extant genes, they do not permit an individual to further specialize itself and create new genes. We also developed a procedure for an individual to add a gene to its collection and refine its specialization, when we want the GP to specialize itself automatically. This process is weighed down by a number of complications, but we developed solutions that mitigate those problems.

One of the challenges in creating a new gene for an individual is that the new gene will necessarily take precedence over the extant genes within some region in measurement space, and it will almost certainly have a negative influence on the fitness of the individual, since the new behavior tree will be random and, probably, inappropriate. In order to confront this problem, we would like to use some kind of evolutionary competition to evolve a new, useful gene for an individual. One can imagine how such a competition could be held: a population of genes could be

constructed, then, the fitness of each gene could be calculated by measuring how well (or how poorly) it assists the individual in its task.

This would make sure that the newly introduced gene doesn't badly harm the specializing individual, but there's another problem: if the candidate genes permitted their contexts to evolve with their behaviors, one way of optimizing the fitness of the new gene could be to evolve its context to represent as small a volume in measurement space as possible, by mutating its context's point into a corner, or underneath an extant context's point, and thereby minimize its use in the individual! This wouldn't serve the purpose of refining the specialization of the individual, since behaviors that occur infrequently wouldn't be subject to much, if any, evolutionary pressure, and wouldn't improve over time.

We managed to use a procedure similar to the one just described, but to confront the problem of contexts drifting towards minimal use, we fixed the position of the genes' contexts in the beginning of the evolutionary competition. That is, when we generated the initial population of genes, we created a number of sub-populations of genes, each of which shared a common context. Then, in the first (approximately) hundred generations, we held the shared contexts constant, and permitted behaviors to only swap genetic material with others within their sub-population. By the end of this first stage of evolution, we had a collection of behaviors that were fit within the context they were given, and thus could withstand the genetic force that would otherwise cause them to drift into obscure regions of measurement space.

During this process, it was possible for the GP to discover that no new gene augmented the fitness of the specializing individual. This was rare early on, but as individuals became more and more specialized, it would happen more often. In such a case, the GP gave up searching after detecting that no progress was being made after a certain number of generations of trying (for example, we used a threshold of 200 generations). The specialization procedure would quit, and the original individual would survive, unchanged.

Manually Defined Contexts

When we provided contexts for the GP manually, we were able to make more human-understandable separations between contexts than when contexts evolved automatically. This kind of change to the GP is a double edged sword, in that we may be able to effect some positive refinements that are rooted in our intuitive understanding of the problem, but in doing so we might overlook factors that could be discovered by the GP on its own, and therefore hinder the development of solutions that lie outside of our intuitive understanding of the problem. Nevertheless, we felt it would be productive to experiment and see whether defining contexts manually could contribute to the GP. We tried two approaches: setting up a collection of contexts that could have been evolved using the automatic procedure described above, but that would make sense intuitively to a person trying to decompose the soldier problem, and setting up contexts in a way inaccessible to those mechanisms, but still in a human-understandable way, which was to assign a context to each room in the simulated area, as illustrated below.

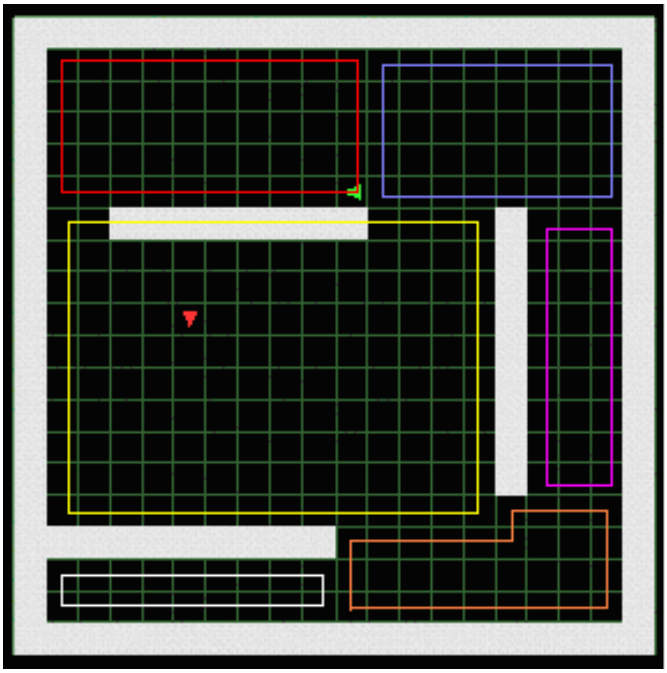

Figure 2: An image of the simulated environment, with room contexts highlighted

Figure 2 shows how we decomposed the soldier problem into the problem of finding behavior appropriate for "rooms" in the environment. This sort of decomposition would be difficult to obtain with the automated context evolution described above. In the other method, we separated the task into five stages and evolved a behavior for each stage. The stages we identified were:

- Search, when the enemy hasn't yet been found,
- Panic, when the enemy has shot at the soldier but the soldier doesn't know where the enemy is,
- Favorable combat, when the soldier can see the enemy but the enemy doesn't see the soldier,
- Normal combat, when the soldier and its enemy are face to face, and
- Retreated, when the soldier has seen the enemy, doesn't see it now, and the enemy isn't shooting at the soldier.

In these cases, when the number and shape of contexts are held constant, evolution can proceed in two ways. First, a population of individuals can evolve and compete with each other, and mutation and crossover can operate on a number of genes simultaneously. Alternatively, a single individual can be chosen, and all of its genes but one held constant, and a population of behaviors can evolve to replace the behavior in the single gene. This can be repeated for each of the genes belonging to an individual. In any case, the GP lets the behaviors change, but contexts stay the same from generation to generation. We employed both of these techniques—using the first to generate a few

high quality individuals, and then the second on each of those individuals to refine the behaviors—and then back to the first, placing all the refined high quality individuals in a large population and evolving them all together again. And so on.

Results

One of the challenges we faced while developing this portfolio of technologies was how we could compare the behaviors generated by different approaches. Some of the approaches used fitness functions that were incompatible with others, so it made little sense to compare calculated fitness directly. Instead, we experimented with fitness functions and found the ones that produced the best individuals, which we then evaluated in a separate program that thoroughly tested every individual in the same way.

The test program tested each behavior a large number of times—generally around 400,000 tests were performed for each behavior. It could evaluate a single behavior in less than five minutes. For each behavior, the test program would report a number of percentages: the probability of the soldier winning a combat without being shot at all, the probability of the soldier merely winning the combat, the probability of the soldier finding its enemy, and the probability of the soldier actually being killed in combat.

We used this test program to ultimately evaluate all the behaviors our GP's, and our contextual GP's, produced. From it, we can say with certainty that the best of the stack-based behaviors was superior to the best of the behaviors without stacks, as the best stack-based behavior rated:

Normal GP, Stack-Based Functions Implemented:
Perfect test: 30%
Successful test: 74%
Finding the enemy: 99%
Getting killed: 19%

This was better than the 65% success rating the stackless behaviors obtained.

Contextual genetic programming fared well, relative to the best non-contextual GP we tested. When contexts were predetermined, and based on a spatial decomposition, the best individual we found to evolve had statistics of:

Contextual GP, with Manually Designed Spatial Contexts:
Perfect test: 32%
Successful test: 78%
Finding the enemy: 100%
Getting killed: 30%

Unfortunately, stage-based decomposition, with predetermined contexts, reached only a best performance of:

Contextual GP, with Manually Designed Stage-Based Contexts:
Perfect test: 31%
Successful test: 70%
Finding the enemy: 95%
Getting killed: 32%

We tried two different kinds of parameter space with the automated context evolution. One kind used only the X and Y coordinate of the soldier agent, and therefore produced a spatial decomposition. Its best performance consisted of:

Contextual GP, Spatial Contexts Automatically Evolved:
Perfect test: 32%
Successful test: 74%
Finding the enemy: 99%
Getting killed: 32%

The other parameter space we tested consisted of practically all the sensory data available to the soldier. It managed a best performance of:

Contextual GP, Generic Contexts Automatically Evolved:
Perfect test: 33%
Successful test: 73%
Finding the enemy: 100%
Getting killed: 34%

Every one of these behaviors was found after running the GP, or contextual GP, over 24 hours on over 16 processors, at approximately one generation per processor per minute.

In the end, the manually designed spatial decomposition of parameter space led to the best individual we produced. However, this cannot make us discard the other approaches, because further refinements in any of them could easily elevate its capabilities beyond the watermark set by manually designed spatial contexts. All our developed techniques scored within a small band of ability—70% to 79%—a slight, yet unmistakable, improvement over the basic method.

Accomplishments and Future Research

In our research, we have demonstrated that the soldier problem is resilient to traditional techniques of genetic programming, and we have shown that progress is possible by changing the way our GP approaches the problem. Since the most impressive gain in performance was associated with manually decomposing the problem space, this would be the most natural place to extend the work described in this paper. There are several important, unanswered questions about spatial decompositions in these kinds of problems that could be the focus of future research. Some are: How different are the behaviors that specialize for different areas? What decomposition (in terms of configuration, number of contexts, and so forth) is the most appropriate for a given environment? Can spatial decomposition be automated in an evolutionary way, or are there principles on which a spatial decomposition of an environment can be formulated? Is this a psychologically plausible method of context evaluation, and is that, in any case, relevant?

The behaviors evolved in our tests so far wouldn't be very useful in training simulations. Their behavior is unsophisticated, and the game they're trained at is far less complex than what would be required to engage human participants. In order to create a GP to evolve behaviors for such an application, the problems we have faced would need to be better understood. The three most significant problems we encountered are: how to measure performance while dealing with multiple, complex, interacting goals, how to decompose a complex problem into smaller and more easily solvable sub-problems, and how to select the most useful set of functions from which behavior trees are built.

Usually, performance measurements are easy to develop, because it is easy to describe desirable behavior. While the goal of the soldier problem is straightforward, our research suggested that a straightforward fitness function was inadequate for our purpose, and that a function that rewards behaviors gradually while they slowly improve was necessary. Our final formula succeeded in encouraging individuals to develop in the direction we wanted them to go, but also might have discouraged creative solutions to the problem with its micromanaging qualities. We do not know whether simpler or more complex fitness functions lead to the development of better behaviors, only that the simple functions we tried were insufficient and the complex functions were capable.

The soldier problem is also burdened by its indeterminate gameplay, because of which some behaviors may, through no fault of their own, fail or succeed. It may be possible to develop a fitness function that adjusts a behavior's score in a particular test based on the difficulty of its encounter. Also, applying a single fitness function to a collection of genes evolved by a contextual GP may be entirely inappropriate. It could be better to have a separate fitness function for each context—somehow making evaluations contextually dependent as well as behaviors—to improve the development of specialized behaviors.

We found automated problem decomposition to be particularly difficult, and our contextual GP did not respond as well as we had hoped to our efforts to implement it.

Progress in this area will be difficult and very valuable. First, it will be necessary to determine exactly when problem decomposition is useful, something that hasn't been made completely clear yet. In the soldier problem, there are certainly several fundamentally different problems that need solving—navigation and combat are the two most significantly distinguishable processes involved, and it makes sense that a GP could be better when the problem is broken up into at least those pieces, but it isn't clear, for instance, whether retreating rapidly from battle and withdrawing carefully from battle ought to be evolved separately. Furthermore, the issue of how contexts are defined is far from resolved, especially considering the success of the spatial decomposition over the stage-based decomposition.

Several tools could be useful in pursuit of solutions to those problems. Foremost is the use of statistical methods such as factor analysis to determine the most significant distinguishing parameters, based on which a decomposition could be automatically proposed and evaluated. In human agent simulations such as the soldier problem, cognitive psychology could also assist, by describing what sensory changes are necessary for a person to shift from one frame to another, and in what cases such shifts are most useful.

Finally, designing a function set is one of the most fundamental steps in developing a GP, and makes a tremendous difference to the success of the GP. Employing functions that give individuals the ability to plan strategies and perform sophisticated analyses of their environment and memory is critical. Unfortunately, developing a rich function set, capable of dealing with a large amount of input and available decisions, is difficult, and giving the GP more functions is not always the answer. A larger function set also entails a larger and, perhaps, more difficult to navigate solution space.

Macro functions, functions that provide complex behavior in a single node, might be useful in this regard. Instead of having functions like "move forward" and "turn left", it might be better to have higher level functions such as "retreat from battle to a known safe place" and "run to the nearest wall". Expert knowledge could be used to form such a function set; in the case of the soldier problem, we may gain from talking with military strategists and trainers, and experienced soldiers, about how they analyze situations and make decisions. We might be able to combine the expertise of several individuals in the function set of a single GP, and evolve from it behaviors that represent not just the sum of each expert's knowledge, but the value of the interactions between the knowledge of several experts.

References

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

Luke, Sean, "Genetic Programming Produced Competitive Soccer Softbot Teams for RoboCup97", In Genetic Programming 1998: Proceedings of the Third Annual Genetic Programming Conference. J. Koza et al, eds. Madison, WS: University of Wisconsin, 1998.

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

Pryor, Richard J., "Developing Maneuvering Behaviors for a Glider UAV Using a Genetic Programming Model", Sandia Report: SAND-2002-3147, Sandia National Laboratories, 2002.

Distribution

1	MS	0321	W. J. Camp, 9200
1	MS	0318	J. E. Nelson, 9209
40	MS	1110	R. J. Pryor, 9216
1	MS	9018	Central Technical Files, 8945-1
2	MS	0899	Technical Library, 9616