Learning Using Chunking in Evolutionary Algorithms

Annie S. Wu
Hal Stringer
University of Central Florida
Orlando, FL 32816
407-823-5922
aswu@cs.ucf.edu, stringer@cs.ucf.edu

Keywords:
Genetic Algorithm, Chunking, Learning, Micro Air Vehicles, Simulation

ABSTRACT: *We describe a modified genetic algorithm (GA) used to evolve control rules for autonomous vehicles. Inspired by the chunking operator in SOAR, our GA attempts to identify and preserve building blocks (useful rules) from solutions that the GA has already considered and make these building blocks available to potential future solutions. Our GA's "chunking" mechanism operates on two levels. First, evolved rule sets within a current generation are evaluated to determine their salience or importance. Those rules identified as more salient are used to update a communal memory. Second, individuals within the population can access all or portions of this shared memory to augment their own component rules. An individual within the GA's population evolves both component rules and references to building block rules in memory that it wishes to incorporate in a final evaluated rule set. The resulting modified GA combines a population-wide effort to evolve highly desirable rules with the evolution of each individual's unique application of those rules.*

1 Introduction

An important challenge in building effective computer generated forces (CGF) systems is the development of simulations that depict intelligent and realistic behaviors without human intervention. Such systems may be used as training, analysis, proof of concept mission planning, and exploratory planning tools. We focus on the problem of learning how to control a team of micro air vehicles (MAVs). Such vehicles are inexpensive, disposable autonomous vehicles with basic sensors and minimum payload. As a result, control of MAVs must be either generally applicable or easily learned and re-learned for specific situations. We study the development of a genetic algorithm (GA) for autonomously learning control rules for specific situations.

One approach to modeling human behavior is to simulate the methods by which humans perform tasks. Chunking in SOAR is based on human tendencies to create chunks or single units of information out of multiple smaller, related units of information. This "chunking" mechanism significantly increases the amount of information that humans can process. We attempt to model such behavior in a GA applied to the problem of learning MAV control rules. We enhance a basic GA [1] to include a chunking-like mechanism. This mechanism attempts to identify and preserve useful rules from previously considered solutions and make those rules available to future solutions. The GA chunking mechanism is implemented with the addition of a population-wide memory unit. This memory unit provides an additional mechanism for GA individuals to communicate and exchange useful information and provides a mechanism with which the GA may identify and form chunks of epistatic building blocks (groups of rules that work together). Our GA is also modified to work with variable length individuals, thus allowing evolution of both the size and the content of the solutions.

2 Background

2.1 Genetic Algorithms

A GA is a search method based on principles from natural selection and genetic reproduction. GAs have been successfully applied to a wide range of problems, including optimization, classification, and machine learning. The specifics of how a GA is implemented vary widely but most forms have their basis in a simple GA [2] which searches a problem space for an optimal solution. Key features that distinguish GAs from other search methods include:

- A *populatio*n of *individuals* where each individual represents a potential solution to the problem to be solved. Individuals are typically binary strings. The terms *individual*, *chromosome*, and *solution* will be used interchangeably in this paper to refer to a potential solution generated by a GA.

- A *fitness function* which evaluates the utility of each individual as a solution.

- A *selection function* which selects individuals for reproduction based on their fitness.

- Idealized *genetic operators* which alter selected individuals to create new individuals for further testing. These operators, e.g. crossover and mutation, attempt to explore the search space without completely losing information (partial solutions) that is already found.

The following pseudo-code shows the steps normally performed by a GA during the search process:

```
procedure GA
    initialize population;
    while (stopping condition not satisfied){
        for (j=1 to population size)
            evaluate fitness of all individuals;
        for (j=1 to population size / 2){
            select two parents for crossover;
            crossover to produce two children;
        }
        for (j=1 to population size)
            perform mutation on children;
        copy children to next generation;
    }
end procedure GA
```

The starting population may be initialized randomly or with user-defined individuals. The GA then iterates through an evaluate-select-reproduce cycle until either a user-defined stopping condition is satisfied or the maximum number of allowed generations is exceeded. The use of a population allows the GA to perform parallel searches into multiple regions of the solution space. Operators such as crossover allow the GA to combine discovered partial solutions into more complete solutions. As a result, the GA is thought to work by searching for small building blocks in parallel, then iteratively recombining small building blocks to form larger and larger building blocks. In the process, the GA attempts to maintain a balance between exploration for new information and exploitation of existing information.

2.2 Chunking in Soar

Chunking was first proposed by Miller in [3] as a model for human memory. The idea of chunking as a learning theory was explored by Newell and Rosenbloom in [4] where they proposed the following Chunking Hypothesis:

> "A human acquires and organizes knowledge of the environment by forming and storing expressions, called *chunks*, which are structured collections of the chunks existing at the time of learning."

The basic idea behind this hypothesis is that human performance improves with practice. A production system described in [5] was developed using a 1,023-choice-reaction-time-test to investigate chunking as a machine-implementable learning mechanism. This initial chunking system created and stored new productions based on the results from previous subgoal based experiences. Results showed that chunking was possible in a machine learning environment and task performance improved over time.

Chunking was later proposed by Laird, Rosenbloom and Newell as a task-independent general learning mechanism [6] applicable beyond a mere practice environment. Soar [7], an architecture for research into the development of intelligent systems and behaviors, incorporates chunking as its single learning mechanism for a variety of diverse task and task-independent applications. For more information about the precursors and early development of Soar, see [8].

Of course genetic algorithms and Soar are very different in their abilities and implementations. At a simplistic level, Soar can be thought of as a single individual whose objective is to solve a problem through the accomplishment of subgoals and then learn from past experience to improve future performance. A GA on the other hand uses a population of individuals to search through a problem space for a best solution.

These fundamental differences make it impossible to implement exactly Soar's version of chunking within a genetic algorithm. For this reason we look to Soar for inspiration rather than replication in terms or our chunking GA. A review of the Soar literature has yielded a number of qualities inherent in Soar chunking which we wish to incorporate in our GA's chunking mechanism. Some of those qualities include:

- **Ability to chunk and store productions** – Once Soar solves a problem (accomplishes a subgoal), its chunking mechanism combines the relevant productions into a single chunk and adds the new chunk to its production memory. Our mechanism should be able to store genetic building blocks (e.g., genes) in a reduced format for later use.

- **Learning within trials and across trials:** Soar is able to use chunks from the instant they are created through the end of a single trial (goal). As long as the contents of Soar's production memory remains unchanged, those chunks are available for use in future trials. Our chunking GA should have a similar characteristic, able to use chunked building blocks during the current run or any future run for the same problem.

- **Improvement in performance over time** – For the GA, we hope to use chunking to improve the search process both in terms of the quality of solutions and the speed with which they are found.

Sharing of chunked productions is not a quality found in Soar due to its single-system architecture. For our chunking GA, we add sharing to our list of desirable qualities in order to allow individuals to access chunked building blocks already found by other chromosomes within the same population.

3 Related Work

3.1 Use of GAs to Evolve Rule Sets

Evolution of control rules for distributed MAVs has been previously documented in [9]. The goal of that work was to develop a system to learn rule sets for controlling individual MAV behavior, allowing teams of multiple MAVs to successfully perform large area surveillance. Figure 1 gives an outline of the resulting system. Each GA individual represents a complete rule set. Thus, a GA population is a sampling of possible rule sets. Each rule set is evaluated based on the performance of a MAV simulation utilizing the rule set. Both the size and the content of a rule set are evolved. The study focused on the size of final evolved solutions, i.e. the number of rules in a rule set, and found that the final evolved sizes are independent of the sizes of the initial population. Solution size is important for several reasons. Larger solutions may contain more detailed rules, but require more processing time. On an autonomous robot where resources are limited, CPU processing time is valuable. Smaller solutions are quicker to process and are more likely to contain generalized rules, but may not be able to handle all necessary situations.

GAs have also been used to evolve rule sets in a number of other systems The SAMUEL system successfully evolves complex rules for controlling autonomous robots [10]. Classifier systems evolve rule sets applied to a range of complex problems [11].

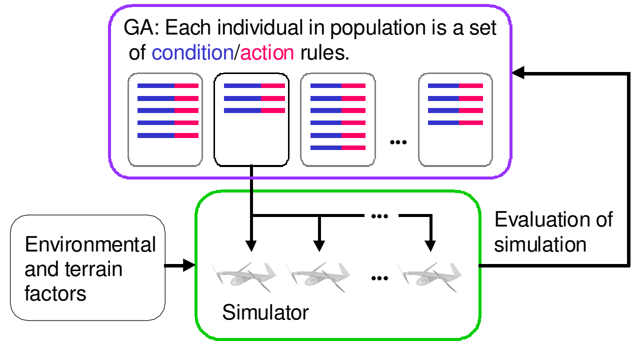

Figure 1 - GA with MAV Simulator

3.2 Memory Enhancements to GAs

The idea of memory has been explored in a range of studies in the GA literature. Memory is thought to be particularly useful in dynamically changing environments, particularly those with repeated cyclical changes, e.g. seasonal changes. In such environments, genes that define a good individual or solution may become dormant when the environment changes, but can become useful again when the cycle returns to the original environment. The ability to retain previously useful information in a memory would benefit a GA working in such environments.

The most common method of creating memory potential in a GA is to use redundant representations. Redundant representations are representations in which only a subset of the encoded information on an individual is expressed. Memory is possible because non-expressed information is not subject to GA selection pressure making it possible to retain information that may not be currently optimal. Examples of redundant representations include diploidy and dominance mapping [12][13][14][15], the messy GA [16], the structured GA [17], and the floating representation [18]. Additional studies on the dynamics of redundant representations include [19][20][21][22].

3.3 GAs with Variable Length Representation

Within evolutionary computation, variable length representations are most prominent in genetic programming (GP), a variant of the GA which directly evolves programs that vary in both size and content. Interestingly, with no bound on size, GP tends to evolve programs that are much larger than necessary, containing sections of code that are never used [23][24]. Though many early efforts focused on "editing out" these excess regions, later studies indicate that the size of evolved programs may affect their fitness and evolvability [25][26][27][28].

Although most GA's focus on fixed length chromosomes, there are several examples in which variable length genomes have been used successfully, including the messyGA [16], the SAGA system [29], the SAMUEL learning system [10], and the Virtual Virus (VIV) project [30][31]. Various studies have found that parsimony pressure is an important factor in keeping individuals manageable in size as well as in maintaining reasonable fitness. There appear to be links between the evolved length and fitness of individuals and various GA parameter settings.

4 Experiment Design

We performed numerous experiments to test the validity and usefulness or the chunking mechanism proposed in this work. The experiments incorporated three primary components: a variable length genetic algorithm, our chunking mechanism, and a MAV simulator to judge the fitness of evolved rule sets. Figure 2 shows a high level view of how these three components interact.

A GA is used to evolve a population of individuals consisting of *base* chromosomes, each representing a possible set of rules for an MAV. Periodically, the GA halts and calls the chunking operator which inspects all *base* chromosomes for valuable rules. A communal or shared memory is updated with newly identified valuable or salient rules.

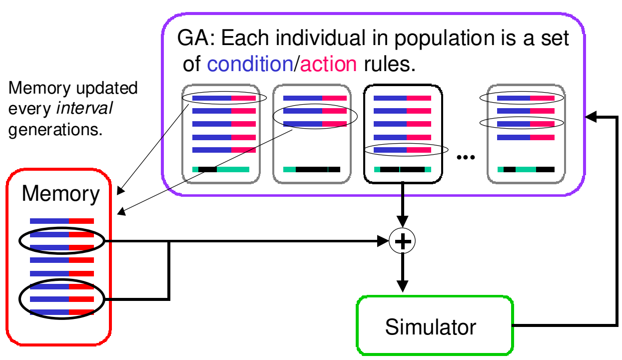

Figure 2 - GA Incorporating a Chunking Mechanism with MAV Simulator for Fitness Function

Each individual has a *memory* chromosome (not shown) in addition to its *base* chromosome. Prior to fitness evaluation, the GA combines *base* chromosome rules with rules in shared memory reference by the *memory* chromosome. This larger *augmented* chromosome is then sent to the simulator to determine the individual's fitness. Crossover and mutation of *base* and *memory* chromosomes occurs after each evaluation cycle.

4.1 The MAV Simulator

We use a MAV simulator from the Naval Research Laboratory for this study. Figure 3 shows an example of the simulator environment with five MAVs. The simulator is initialized with a number of parameters that specify the environment (the area to be surveyed) and the MAV team makeup. The simulation defines the world within which the MAVs move about and sense one another as well as landscape features, all while performing their task of surveying the region. MAVs are assumed to have sufficient energy capacity to function throughout the entire simulation although they can be destroyed if they collide with one another or go beyond the boundaries of the surveillance area.

All MAVs within the simulation are identical in configuration. Each MAV has eight range sensors, distributed around its perimeter, which indicate if anything (e.g. another MAV or a boundary) is within a pre-specified range in the direction of the sensor. Range sensors are not capable of detecting the number or distance of objects within their range. As a result, sensor data is binary: "1"' indicates that an object or boundary has been detected, "0"' indicates that no object is detected.

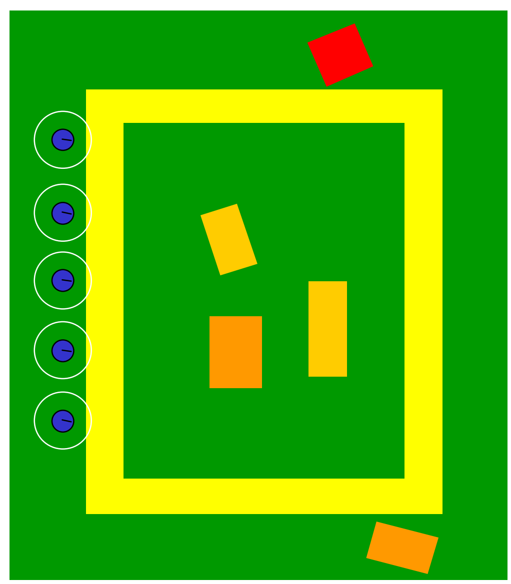

Figure 3 - MAV Simulator Area with Five MAVs

In addition to eight range sensors, each MAV also has a camera sensor that is pointed down. A camera sensor detects regions are varying interest on the ground. Interest levels are indicated by colors. Yellow regions are low interest; red regions, high interest. The evaluation of a simulation run consists of the average surveillance coverage over an entire run and the amount of surveillance over regions of high interest.

All MAVs are governed by the same augmented rule set fed to the simulator by the chunking GA. The rule set specifies which action should be taken at any time step given the sensor data. Each rule consists of a condition and an action clause. Sensor data is compared to the condition clause of each rule in the rule set. The rule with the best match is selected. If multiple rules qualify as the best match, one is chosen at random from the candidates. A selected rule fires only if the degree of match of the selected rule exceeds a given threshold. The action clause of a rule specifies the direction of the MAV's movement. A turn can be in one of the eight compass directions. Although geometrically diagonal moves (NE, NW, SW, SE) result in larger changes in position, they are assumed to take place in one time step, similar to non-diagonal moves (E, N, W, S). Every MAV performs this evaluation of rules and sensor data once in each time step and executes an action (movement in the specified direction) if the matching threshold is exceeded.

The MAV simulator, like the GA and memory has its own set of input parameters. Experiments described in this work run for 800 time steps each and incorporate five MAVs with starting positions evenly spaced along the western edge of the simulation area. The environment consists of fixed features of varying interest levels in a pattern similar to that of Figure 3. Each simulation is initialized with a different random number seed.

4.2 GA Design

A standard GA is used as the basis for our experiments. Features and parameters incorporated into our GA for all experiments include the following:

- Generations per Run = 100
- Population Size = 100 Individuals
- Representation Method = Bit String
- Initial Genes per Chromosome = 20
- Maximum Genes per Chromosome = 100
- Number of Bits per Gene = 11
- Selection Method = Fitness Proportional
- Crossover Type = 1-Point Inter-Gene
- Crossover Rate = 70%
- Mutation Type = Bit Change
- Mutation Rate = 1%
- Parsimony Enforcement = Truncation

The number of runs for each experiment varied and are given in the subsection describing our results.

4.3 Memory Design

In order to implement our proposed chunking capability, it is necessary to develop some form of memory and memory management mechanisms that can be used to augment base rules sets found by a simple GA. Such a memory mechanism must meet the following requirements:

1. The mechanism must be able to identify valuable rules (salient genes) from chromosomes within the population.
2. Rules identified as valuable must be preserved in a structure accessible by all chromosomes within the population prior to fitness evaluation.
3. Individuals within the population must be able to evolve their choice of rules from within the shared memory which provide the most benefit.
4. The memory mechanism must be updateable over the course of a single GA run.
5. Rules in memory should be replaceable in the event a better rule is found by the general population.
6. A method must be in place to add replacement rules back into the population so as not to loose this genetic material.

4.3.1 Rule Identification and Preservation

We can meet the first requirement (valuable rule identification) by simply counting the number of occurrences of a given gene or rule within the overall population. Genes are then sorted by their frequency. Our basic assumption here is that more useful genes will appear more frequently within the total population. The concept of gene salience has been proposed in [32] but in the domain of GAs with fixed gene locations within a chromosome.

Once frequent genes are identified and sorted, we consider requirement number 2, creating a data structure to hold valuable rules which would be accessible to all chromosomes within the population. We construct an array of *slots*, each capable of holding one valuable rule. Each slot also has an associated variable (*reference_frequency*) to store the number of times it is referred to by chromosomes in the population.

Memory is initialized with m empty slots, where m is an input parameter to our modified GA. The first update to memory is simply a matter of taking the m most frequent genes and loading them into m memory slots along with their associated frequency

Access to rules in memory is controlled at an individual level by supplementing an individual's base chromosome with a “memory” chromosome. The memory chromosome consists of *m* bits set to either 1 or 0. Each bit is directly associated with a given slot (bit one corresponds to slot 1, bit two corresponds to slot 2, etc.) A one in a bit position indicates that the associated slot is referenced by the individual. A “0” in a particular bit position indicates that the contents of that memory slot should be ignored.

The following brief example is provided to better illustrate the ideas put forth above. Assume that a 3-slot memory exists. Slots contain the 4-bit strings indicated by the second column below:

Slot #	Contents
1	1001
2	0110
3	0111

Individual X within the population has the following 3-bit memory chromosome in addition to its base chromosome.

Memory Chromosome	Base Chromosome
101	1101 1011 0001 0101

Prior to fitness evaluation, the contents of slots referenced by the memory chromosome are combined with the rules already in the base chromosome to create a full rule set. This augmented rule set is the one sent to the fitness function (simulator) for evaluation. For our individual X above, the augmented rule set might appear as follows:

Augmented Chromosome for Individual X
0111 *1101 1011 0001* **1001** *0101*

Note that the two rules in bold typeface are obtained from memory slots referenced by the bits in the memory chromosome. The rule “0110” from slot 2 is not included since X’s memory chromosome did not reference this slot. The other four rules (in italics) of X’s augmented rule set are from its own base chromosome.

We experimented with three different methods of combining rules from memory with an individual’s base rule set. The above example illustrates a random insertion of memory rules into the chromosome. The other two methods tried included (1) adding the rules to the very front of the base chromosome and (2) appending the rules on to the back of the base chromosome. The MAV simulator which is our fitness function selects rules from within a set based on a condition-match algorithm. Position within a rule set has minimal impact on the performance of the MAVs. Use of our memory mechanism with location dependent rule applications may require alternative methods for augmentation (add or append) or some new method that tracks rules and locations.

4.3.2 Evolving Selective Memory Access

We now have a memory mechanism that saves valuable rules and allows individuals within a population to access specific slots prior to fitness evaluation. Requirement 3 specifies that each individual must be able to evolve their access to such rules. This was easily met by performing crossover and mutation on memory chromosomes in addition to individuals’ base chromosomes. Memory chromsome crossover and mutation is implemented in our system as follows:

1. Select two parents from the current generation.

2. Determine if crossover is to take place based on crossover rate (e.g., 70% chance of crossover occurring). If crossover to occur:

   - perform 1-point inter-gene crossover between parents using their base chromosomes only.
   - Perform 1-point intra-gene crossover of memory chromosomes based on crossover rate. (e.g., 70% chance of crossover occurring).

   If crossover does not occur, copy both parents (base and memory chromosomes) into next generation.

3. Using rates set in the GA’s parameters, mutate both base and memory chromosomes for all children by flipping bits (a “1” becomes a “0” and a “0” becomes a “1”).

Crossover between two parents includes both their base and memory chromosomes. However, crossover of a memory chromosome may occur less often than that of a base. The effective crossover rate for memory chromosomes is equal to the base crossover rate times the memory crossover rate (e.g., 70% x 70% = 49%.) Mutation of both types of chromosomes always occurs but the rates can be set independent of one another via parameters in the GA. The end result of this is the ability of the GA to evolve both base and memory chromosomes simultaneously.

4.3.3 Updating Memory

The last three requirements for our mechanism necessitate some form of memory update. Our GA contains an update routine that was activated every *g* generations. The value of *g* is set by way of an input parameter. For our experiments it was found that g=10 gave us the best results.

Each updated consists of the following steps:

1. Identify valuable genes within the base chromosomes of the entire population by counting their frequency. Place these genes in a list sorted by frequency of appearance.

2. Compare rules in the valuable gene list with those currently in memory slots. Remove duplicate rules from the valuable gene list (they are already in memory).

3. Determine the frequency with which memory chromosomes within the population reference a slot in memory. Update the slot's *reference_frequency* with this number.

4. If a gene appears more often in the population's base chromosomes than a current slot is referenced, replace the rule in the slot with the newly found valuable gene.

5. For memory rules that are replaced, determine which individuals in the population currently reference the slot. Add the rule to be replaced back into each individual's base chromosome at some randomly selected position. This insures that individuals who reference a given rule will retain a copy of the rule for future use.

6. Reset the bit position associated with updated slots in all memory chromosomes. For our experiments, this is accomplished by randomly setting the reference bit to "1" thirty percent of the time. As a result, 30 percent of the population has access to the new slot content immediately following an update.

It should be noted that a valuable gene added to memory is not removed from any individuals in the population. In our experiments, roughly 30% would have two copies of the gene, one from their base chromosome and one from the reference bit in the reset memory chromosome. The other 70% would still have a copy of the valuable gene in their base chromosome.

4.3.4 Memory Parameters

The specific details of the memory mechanism are set through runtime parameters. During a series of test runs, we found that the following settings appeared to give the best performance:

- Memory Size = 10 Slots
- Update Interval = Every 10 Generations
- Memory Crossover Type = 1-Point
- Memory Crossover Rate = 70%
- Memory Mutation Type = Bit Change
- Memory Mutation Rate = 1%
- Combination Method = Random Insertion
- Return Method = Random Insertion
- Reset Rate for New Slot = 30%

All experiments with active chunking use these same parameter settings.

5 Results of Experiments

5.1 Testing the Functionality of Memory

We perform a series of experiments to test the functionality of our chunking mechanism as well as its ability to evolve rule sets for MAVs. The set of functional experiments focus on how the mechanism operates. Our objective is to answer the following three questions:

- Will a GA intentionally take advantage of memory contents to improve fitness?
- Can a GA differentiate between memory items with positive and negative fitness effects?
- Can a GA evolve the actual contents of memory to its advantage?

We begin our functional experimentation by determining if our chunking GA would take advantage of the contents of a static, pre-set memory. For the first test we use a simple MaxSum fitness function. Each gene within a base chromosome (or memory slot referenced by the memory chromosome) represents a positive binary integer value. The raw fitness for a given individual is the sum of all integers in the base chromosome as well as any integer values contained in referenced memory slots.

A chunking GA with 10 memory slots is used. The contents of each slot are initialized by concatenating a series of eleven randomly generated "1"s and "0"s with roughly a 50/50 distribution. Columns (a) and (b) of Table 1 show an example of the memory structure and contents from one of our experiments. The bit string in each slot is converted during fitness evaluation to the number contained in column (c) "Gene/Slot Value". Memory chromosomes for the 100 individuals within the population are also randomly generated. Table 1, column (f) gives the initial frequencies with which memory slots are referenced by the population at the start of the GA run. On average each memory slot is referenced by 48.9 out of 100 individuals at the start of the run.

Column (g) of Table 1 shows the reference frequency of each memory slot after 50 generations of the chunking GA with MaxSum fitness function. This column indicates the numbers of individuals within a population of 100 that have a “1” in the memory chromosome bit position associated with each slot. For example, Slot 3, the highest valued slot is referenced by 94 out of 100 individuals within the population at the end of 50 generations. On average, 84.4 individuals reference each memory slot. This is substantially higher than the initial starting average of 48.9 referencing memory chromosomes from the start of the run.

The numbers in column (g) clearly show that our chunking GA evolves memory chromosomes which make use of the contents of shared memory. In general, it appears that higher valued slots (which contribute more to fitness) are more likely to be referenced in evolved memory chromosomes. All experiments described in this subsection were repeated for 10 runs using different initial memory content and memory chromosomes. Although we describe specific results from a single run, all runs produced similar behaviors and results.

But we need to ask ourselves, “Is this a statistical fluke?” To verify the validity of our results, we run a baseline GA with no memory, i.e. inactive memory slots. Crossover and mutation continue to occur for memory chromosomes but they reference empty slots which do not add any value to the fitness of an individual. Table 1 columns (d) and (e) show the reference frequency for each memory slot beginning with generation 0 (initial frequencies) and after 50 generations. On average, 48 individuals reference each slot at the start of a run; 43.4 individuals, after 50 generations. With chunking turned off, referenced slots have no impact. As expected, the average number of memory references appear to remain random over a run.

The results from these first two experiments clearly answer the first question posed at the start of this section. Our chunking mechanism allows a GA to intentionally take advantage of memory slots if those slots have an impact on fitness.

But can our chunking GA differentiate between memory slots which impact fitness in positive or negative manners? To answer this question, we develop a new fitness function called the MaxSum with Odd Penalty. The MaxSum with Odd Penalty function is a variation on the MaxSum but with the following difference. Even numbers are added to an individual’s raw fitness, odd numbers are divided by 2 and *subtracted* from raw fitness. In other words, odd numbers have a detrimental impact on an individual’s overall fitness.

We test a chunking GA with 10 memory slots on the MaxSum with Odd Penalty fitness function. The initial reference frequencies for this GA run are the same as those for MaxSum given in column (f). Column (h) of Table 1 shows the frequency of slot references within the population after 50 generations. On average, 90.7 chromosomes reference slots containing even integers and 25.8 chromosomes reference slots with odd integers.

These results indicate that our chunking GA is able to evolve memory chromosomes that take advantage of slots with positive fitness contributions (even integer content) and avoid slots with a negative impact on fitness (odd integer content).

Memory Composition			**Slot Reference Frequencies**				
			Chunking Turned Off		**With Chunking Activated**		
						After 50 Generations	
(a) Slot Num.	**(b) Slot/Gene Bit String**	**(c) Slot/ Gene Value**	**(d) Initial Frequency**	**(e) MaxSum After 50 Generations**	**(f) Initial Frequency**	**(g) MaxSum**	**(h) MaxSum w/ Odd Penalty**
0	10001010101	1109	60	53	50	100	2
1	10110010101	1429	51	7	48	97	57
2	11110001010	1930	36	48	49	94	94
3	11111100110	2022	46	27	49	94	96
4	00011110110	246	42	91	49	82	74
5	01100000001	769	50	29	39	76	7
6	01011001011	715	48	35	47	30	65
7	00110101111	431	49	82	56	77	16
8	11011010010	1746	49	10	46	95	99
9	10110100101	1445	49	52	56	99	8
Avg. References per Slot			48.0	43.4	48.9	84.4	90.7 even 25.8 odd

Table 1 – Contents of Static Memory and Slot Reference Frequencies for Three GA Runs

Our last functional test is designed to determine if a chunking GA will evolve the actual contents of memory in addition to references within the memory chromosomes. Previous experiments use a static, unchanging memory structure randomly initialized at the start of each GA run. For this last test, we construct a GA with updateable memory using the MaxSum with Odd Penalty function. Memory slots and chromosomes are initialized with the same content and frequencies shown in Table 1, columns (b) and (f). The chunking GA runs for 50 generations with memory updates occurring every 10 generations (total of 5 updates in all).

Slot	Gene Bit String	Gene Value	Access Frequency
0	***11101110000***	***1904***	***60***
1	***10110001010***	***1418***	***94***
2	11110001010	1930	99
3	11111100110	2022	99
4	***10110000100***	***1412***	***93***
5	***10100110010***	***1330***	***97***
6	***11100001100***	***1804***	***97***
7	00110101111	431	98
8	11011010010	1746	95
9	***11001000000***	***1600***	***100***
Avg. References per Slot			93.2

Table 2 – Evolved Contents of Dynamic Memory and Associated Slot Reference Frequencies for MaxSum w/ Odd Penalty After 50 Generations

Table 2 shows the contents of memory at the end of a sample run and the reference frequency for each slot. Note that the contents of slots in bolded, italicized font are replaced over the course of the run. Of these six slots, five originally contained odd integer values which detract from raw fitness during evaluation. These odd integers were replaced with high positive integer values. Slot 4 originally contained the value 246 which was replaced with an even integer of higher value (1412). Slot 7 contains an odd value of 431 that is not swapped out during any memory update. This neglect may be the result of its low negative impact (only reduced fitness by 215.5) or the result of hitchhiking [33] since the reference bit associated with slot 7 is surrounded by bits which themselves reference very high even integers.

A comparison of Tables 1 and 2 clearly shows that our chunking GA operates and functions as we had hoped. It allows a GA to beneficially evolve the contents of a shared communal memory with access to that memory on an individual basis through memory chromosomes.

5.2 Developing Rules Sets for MAVs

The preceding functional experiments confirm that our chunking GA operates as required. We can now apply this new mechanism to developing rule sets for MAVs. All experiments described in this section incorporate GA, simulator and memory design parameters previously detailed in Section 4.

MAV rules are fed to the simulator as 14-bit strings. Each rule is a single condition-action pair. We permit a maximum of 110 rules per MAV: 100 rules from an individual's base chromosome and an additional 10 rules from shared memory. Within these rules, bits 1-11 form the condition part of a MAV rule: bits 1-8 are matched against range sensor inputs indicating boundaries or other MAVs and bits 9-11 are matched against the camera sensor input and indicate the direction of the center of interest for all features within the camera's range. Bits 12-14 encode the action by indicating one of eight directions in which the MAV can move.

We have divided our MAV research efforts into two phases. The first phase as reported in this work is concerned primarily with a) testing our chunking GA and b) developing rule sets which keep MAVs from being destroyed (i.e., moving out of the simulation boundary or running into one another.) To simplify this task, we have predefined the action for all MAVs to be "move towards the center of interest". The second phase of our research will use our chunking GA to evolve a) full condition-action rule sets, b) rule sets for a wider range of environments such as low level surveillance or moving points of interest, and c) separate rules sets for decomposable tasks (analogous to subgoals in Soar). This second phase is currently underway.

For phase one, we implement the "move toward the center of interest" behavior by having the GA evolve the range and camera sensor bits associated with each rule's condition part, then set the direction of movement (bits 12-14) equal to the value of the bits defining the camera sensor condition (bits 9-11). We assume that an MAV's goal should always be to move toward the point of highest interest within its sensor range. Again, our objective is to evolve rule sets which move the MAV toward interesting features with minimal self-destruction.

A total of 60 GA runs are conducted for this work – 30 with a standard GA (no chunking) and 30 runs using our chunking GA. The chunking GA runs are divided into two different groups. For the first group consisting of 18 runs, memory contents are emptied between each run. No carry over of valuable rules occurs from one run to the next. The remaining 12 runs employ a cumulative memory. The evolved contents of memory from the current run carry over to the next run. In all experiments base and memory chromosomes are reinitialized at the start of each run to random bit strings regardless of what occurs in memory.

A qualitative review of the best individuals from all runs shows similar behavior. All five MAVs survive and stay within the simulation area. Individually, the MAVs tend to move across the simulation area until a high interest feature is discovered. The MAVs then hover over the interesting feature unless "bumped out" by another MAV. In terms of best fitness, all three groups of runs (standard, re-initialized memory, cumulative memory) produce similar results as shown in Figure 4 almost reaching the upper bound of potential fitness.

The differences between the three groups is best illustrated by plots of average fitness shown in Figure 5. The chunking GA with reinitialized memory performs much better than a standard GA over the course of an entire run. The two GAs have similar performance for the first 10 generations. At that time, the chunking GA updates memory with the first set of salient rules causing the fitness plots to start diverging. Using a cumulative memory enhances performance further due to the fact that valuable rules from one run carry over to the next.

One of the most interesting characteristics of the chunking GA is its apparent ability to minimize the size of rule sets. GA runs in this work utilize variable length chromosomes. As such, each GA is able to evolve both the content and number of rules. Figure 6 illustrates the average number of rules contained in all base chromosomes for the entire population at each generation. Without chunking, the average rule size continues to increase over time. A limit of 100 rules per base chromosome coded into our GA prevented the average non-chunking GA from going beyond approximately 50 rules per chromosome.

The chunking GA appears to reduce the number of rules needed to achieve best fitness without any forced parsimony on the part of the GA program. In these experiments our chunking GA evolve both fit <u>and</u> compact rule sets.

An additional point of interest resulted from an examination of the fitness of different subsets of chunked rule sets. We conducted a series of simulation runs using a single high fitness individual from a test population and using the contents of memory at the time that individual appeared in the GA run. MAVs were manually loaded with rule sets consisting of 1) only rules derived from the base chromosome, 2) only rules derived from memory referenced by the individual and 3) all rules contained in memory regardless of reference.

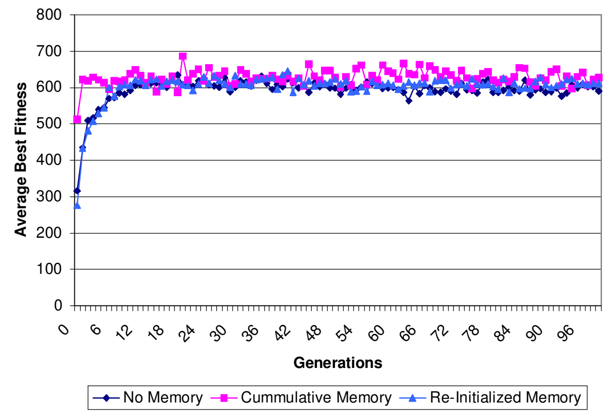

Figure 4 – Average Best Fitness Over 100 Generations

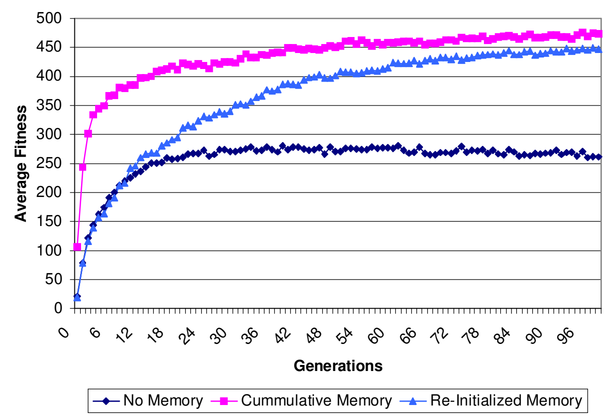

Figure 5 – Average Fitness over 100 Generations

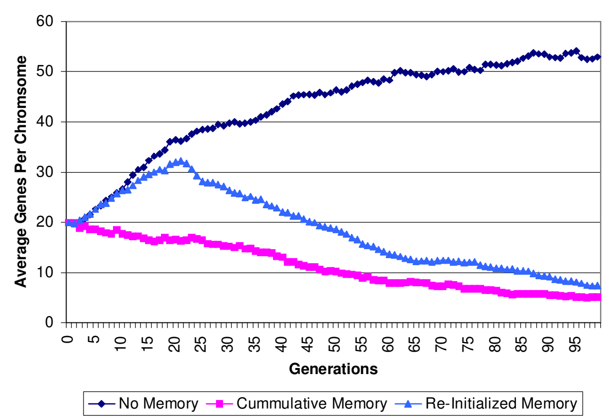

Figure 6 – Average Number of Genes per Base Chromosome over 100 Generations

Results indicate that the base-only rule set performs the worst followed by referenced memory rules. The performance increase obtained by combining the two rule subsets (the augmented rule set) was substantial and exceeded the sum of their individual finesses. The fitness of the rule set consisting of all rules in memory was almost equal to that of the individuals augmented rule set but within a very tight fitness range.

6 Conclusions

We apply a modified GA to the problem of learning control rules for a distributed team of MAVs. Our GA extends the simple GA by incorporating a memory mechanism that is inspired by Soar's chunking mechanism. This memory mechanism strives to achieve some of the benefits that Soar derives from its chunking operator, including the ability to chunk and store useful information, the ability to learn within and across trials, and improvement in performance over time.

The memory mechanism for our chunking GA is designed to recognize and disseminate useful information learned from current and past trials (potential solutions) for use with future trials. This mechanism does not explicitly learn problem specific data, so it should be transferable to other problem domains. Within this paper, we demonstrate its use on three different types of problems.

The overall contribution of chunking appears to be positive. Comparisons of a GA with and without the chunking mechanism shows equivalent performance in terms of best fitness and improved performance in terms of average fitness. Transfer of information across trials results in faster learning. In addition, a GA using the chunking mechanism appears to be able to achieve its performance with smaller rule sets. Fewer rules means better performance for the GA (less work in crossover, mutation, and fitness evaluations) as well as the MAV (more efficient rule selection, smaller storage payload).

Operationally, our chunking GA performs as required in Section 4.3. Detailed examinations of the changes in the memory content over individual GA runs indicate that a GA can selectively evolve references to static memory and that a GA can use memory updating to dynamically change memory contents for better performance.

Future work will focus on evaluation of the phase two runs which examine the evolution of full condition-action rule pairs. These experiments increase the complexity of the evolved MAV rules sets by allowing the GA to evolve the direction of movement (instead of using a default direction toward the area of highest interest). Preliminary experiments are promising and indicate similar results to the phase one results reported here (higher average fitness, shorter rule sets).

Other areas of future work include the development of separate memories for decomposable problems, analogous to subgoaling in Soar, and the investigation of learning specific subtasks in separate runs then combining memory contents for situations that require multiple subtasks.

Acknowledgements

This work was supported by grants from Science Applications International Corporation (SAIC Grant #4400035627) and the University of Central Florida High Tech Corridor Matching Funds Program. The authors would like to thank David Pratt for his input and suggestions on this work.

References

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

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

[3] G.A. Miller: "The magical number seven, plus or minus two: Some limits on our capacity for processing information", *The Psychological Review*, vol. 63, pp. 81-97, 1956.

[4] A. Newell & P.S. Rosenbloom: "Mechanisms of Skill Acquisition and the Law of Practice", *Cognitive Skills and Their Acquisition*, J.R. Anderson, ed., pp.1-55, Erlbaum, Hillsdale, NJ, 1981

[5] P.S. Rosenbloom & A. Newell: "Learning by Chunking: A Production-System Model of Practice", *Production System Models of Learning and Development*, pp. 221-286, Bradford Books/The MIT Press, Cambridge, MA, 1987.

[6] J.E. Laird, P.S. Rosenbloom, & A. Newell: "Towards chunking as a general learning mechanism", *Proc. Nat'l Conf. Artificial Intelligence*, pp. 188-192, 1984

[7] J.E. Laird, A. Newell & P.S. Rosenbloom: "Soar: An architecture for general intelligence", *Artificial Intelligence*, Vol. 33, pp. 1-64, 1987

[8] P.S. Rosenbloom, J.E. Laird & A. Newell eds, *The Soar Papers*, MIT Press, Cambridge, MA 1993.

[9] A.S. Wu, A.C. Schultz, A. Agah: "Evolving control for distributed micro air vehicles", *Proceedings of the 1999 IEEE International Symposium on Computational Intelligence in Robotics and Automation*, pg. 174-179, 1999.

[10] J.J. Grefenstette, C.L. Ramsey, and A.C. Schultz: "Learning sequential decision rules using simulation models and competition", *Machine Learning*, Vol. 5, Number 4, pp. 355-381, 1990.

[11] J.H. Holland & J.S. Reitman: "Cognitive Systems Based on Adaptive Algorithms", *Pattern-Directed Inference Systems*, D.A. Waterman & F. Hayes-Roth eds., Academic Press, NY, 1978.

[12] R. E. Smith and D. E. Goldberg: "Diploidy and dominance in artificial genetic search", *Complex Systems*, Vol. 6, Number 3, pp. 251-285, 1992.

[13] K. Yoshida and N. Adachi: "A diploid genetic algorithm for preserving population diversity", *Parallel Problem Solving from Nature 3*, 1994.

[14] K. P. Ng and K. C. Wong: "A new diploid scheme and dominance change mechanism for non-stationary function optimization", *Proc. 6th Int'l Conference on Genetic Algorithms*, 1995.

[15] J. Lewis: "A comparative study of diploid and haploid binary genetic algorithms", Master's Thesis, University of Edinburgh, 1997.

[16] D.E. Goldberg, G. Korb & K. Deb: "Messy genetic algorithms: Motivation, analysis , and first results", *Complex Systems*, pp. 3:493-530, 1989.

[17] D. Dasgupta and D. R. MacGregor: "Nonstationary function optimization using the structured genetic algorithm", *Parallel Problem Solving from Nature 2*, pp. 145-154, 1992.

[18] A. S. Wu and R. K. Lindsay: "A comparison of the fixed and floating building block representation in the genetic algorithm", *Evolutionary Computation*, Vol. 4, Number 2, pp. 169-193, 1996.

[19] W. Banzhaf: "Genotype-phenotype mapping and neutral variation - A case study in genetic programming", *Parallel Problem Solving from Nature 3*, pp. 322-332, 1994.

[20] H. Kargupta: "The gene expression messy genetic algorithm", "*Proc. IEEE Int'l Conference on Evolutionary Computation*", pp. 814-815, 1996.

[21] G. R. Harik: "Learning gene linkage to efficiently solve problems of bounded difficulty using genetic algorithms", Ph.D. Thesis, University of Michigan, 1997.

[22] M. Shackleton, R. Shipman, and M. Ebner: "An investigation of redundant genotype-phenotype mappings and their role in evolutionary search", *Proc. Congress on Evolutionary Computation*, pp. 493-500, 2000.

[23] T. Blickle & L. Thiele: "Genetic programming and redundancy", *Genetic Algorithms Within the Framework of Evolutionary Computation* (Workshop at KI-94), pp.33-38, 1994.

[24] J. Koza: *Genetic Programming*, MIT Press, Cambridge, MA, 1992.

[25] W.B. Langdon & R. Poli: "Fitness causes bloat", *2nd On-line World Conference on Soft Computing in Engineering Design and Manufacturing*, 1997.

[26] P. Nordin & W. Banzhaf: "Complexity compression evaluation", *Proc. 6th Int'l Conference on Genetic Algorithms*, pp. 310-317, 1995.

[27] J. Rosca: "Generality versus size in genetic programming", *Genetic Programming*, 1996.

[28] T. Soule, J.A. Foster & J. Dickinson: "Code growth in genetic programming", *Genetic Programming*, pp. 215-233, 1996.

[29] I. Harvey: "Species adaptation genetic algorithms: A basis for a continuing SAGA", *Proc. 1st European Conference on Artificial Life*, pp. 346-354, 1992.

[30] D.S. Burke, K.A. De Jong, J.J. Grefenstette, C.L. Ramsey & A.S. Wu: "Putting more genetics in genetic algorithms", *Evolutionary Computation*, pp. 6(4):387-410, 1998.

[31] C.L. Ramsey, K.A. De Jong, J.J. Grefenstette, A.S. Wu & D.S. Burke: "Genome length as an evolutionary self-adaptation", *Parallel Problem Solving from Nature 5*, pp. 345-353, 1998.

[32] D. Thierens, D.E. Goldberg & A.B. Pereira: "Domino convergence, drift and the temporal-salience structure of problems", *Proc. IEEE Int'l Conf. Evolutionary Computation*, IEEE Press, New York, NY, pp. 535-540, 1998.

[33] S. Forrest and M. Mitchell: "Relative building block fitness and the building block hypothesis", *Foundations of Genetic Algorithms 2*, pp. 109-126, 1992.

Author Biographies

HAL STRINGER is currently attending the University of Central Florida (UCF) as a graduate student working towards a Master's Degree in Computer Science. Before attending UCF, he was employed in private industry where he specialized in telecommunications and applied information technology. He received a Bachelor of Science degree in Business Administration with a major in Computer Information Sciences from the University of Florida (UF) in 1979.

ANNIE S. WU Annie S. Wu is an assistant professor in the School of Electrical Engineering and Computer Science at the University of Central Florida (UCF). Before joining UCF, she was a National Research Council Postdoctoral Associate at the Naval Research Laboratory. She received her Ph.D. in Computer Science and Engineering from the University of Michigan.