Winnowing Wheat from Chaff: The Chunking GA

Hal Stringer and Annie S. Wu

School of Computer Science
University of Central Florida, Orlando, FL 32816
{stringer, aswu}@cs.ucf.edu

Abstract. In this work, we investigate the ability of a Chunking GA (ChGA) to reduce the size of variable length chromosomes and control bloat. The ChGA consists of a standard genetic algorithm augmented by a communal building block memory system and associated memory chromosomes and operators. A new *m*x*n* MaxSum fitness function used for this work is also described. Results show that a ChGA equipped with memory capacity equal to or greater than the minimal size of an optimal solution naturally eliminates unexpressed genes.

1 Introduction

Variable length evolutionary algorithms often evolve overly long chromosomes in which useful information is interspersed with unused information. This "bloat" or non-coding DNA can increase the computational resources required to evolve a solution. For problems 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. A variety of methods have been employed to reduce bloat in GA and GP systems, including the addition of parsimony pressure in the fitness function and active testing for and removal of non-functional regions. Parsimony pressure if too high can favoring short but non-optimal solutions. The implementation of both parsimony pressure and active removal methods require additional computational resources during a GA or GP run.

In [1], we investigate the use of a new form of genetic algorithm coined the "Chunking GA" or ChGA to develop rule sets for autonomous micro air vehicles (MAVs). Inspired by the chunking operator in SOAR [2], the ChGA identifies and preserves good individual building blocks in the form of genes (useful rules) from solutions that a GA has already considered. Identified rules are then made available to other individuals in the population for evolving potential future solutions.

The ChGA exhibits a number of unique properties in experiments performed using a MAV simulator. One of these is the ChGA's ability to reduce the size of individual base chromosomes (rule sets) over time. This reduction occurs without explicit parsimony pressure built into either the ChGA or the fitness function.

To use an agricultural analogy, the ChGA appears able to separate good rules from extraneous rules like a farmer who separates wheat from the chaff. We refer to this property as the "winnowing effect" – the ability to reduce variable length chromosomes to a minimal sized, optimal solution. In this work, we investigate the ChGA's winnowing ability and show that the ability to reduce solution size is directly related to the size of a ChGA's memory capacity.

2 Related Research

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 [3][4]. 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 [5][6][7][8].

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 [9], the SAGA system [10], the SAMUEL learning system [11], and the Virtual Virus (VIV) project [12][13]. 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.

In a ChGA, we use a simple variable length representation. In addition, each individual in the population incorporates a short fixed length chromosome to serve as a reference to slots within a shared communal memoryThe size of a complete solution therefore varies in two ways. First, the base chromosome can vary in length as it evolves. Second, prior to fitness evaluation, the base chromosome is augmented with referenced slots in memory, potentially increasing the size of a full solution.

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.

A 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 [14][15][16][17], the messy GA [9], the structured GA [18], and the floating representation [19]. Additional studies on the dynamics of redundant representations include [20][21][22][23].

Research in the area of explicit memory systems for the GA is more rare. In [24] an external memory structure is constructed to select and retain entire chromosomes. This “online” approach is used to construct individuals in the next generation. [25] presents a more “offline” approach. A GA's past results are used as the starting point for solving a new but similar problem – a form of case-based reasoning.

Most of these memory-related works focus on saving entire chromosomes or individuals from the population for later use. The ChGA is a departure from this approach, saving single genes or building blocks that have been extracted from the population at large. The ChGA's focus on building blocks or genes combined with a shared communal memory appears to be new to GA research.

3 Overview of the Chunking GA

The ChGA starts with a genetic algorithm using genes (bit substrings) of fixed equal length within a variable length chromosome. As an example, each gene might represent a single condition/action rule for a MAV. The set of rules (all genes contained in a chromosome) would serve as input to a simulator that returned a fitness value for the individual. Using a variable length chromosome allows the GA to evolve individuals with rule sets of different sizes.

The following three elements are added to this variable length genetic algorithm in order to implement the memory system of our Chunking GA:

1. *Shared Communal Memory*. A shared memory structure with some l number of slots is added. Each slot is capable of holding a single gene, building block or rule. All slots are of equal size (fixed number of bits). The number of available slots (l) is set by parameter prior to each ChGA run and remains constant during that run.
2. *Memory Chromosomes*. Individuals within the population have two chromosomes: a *base* chromosome and a *memory* chromosome. The base chromosome is similar to a variable length haploid chromosome used with a simple GA. The memory chromosome serves as a reference to slots in the shared memory structure. Prior to fitness evaluation, genes located in slots indicated by the memory chromosome are added to the genes in the base chromosome to create an *augmented* chromosome. The augmented chromosome is then sent to the fitness function for evaluation.
3. *Memory operators*. Operators are added to the GA to manage the shared memory structure. These operators identify good genes from the pool of base chromosomes and add them to memory, in some cases replacing good genes with better genes. Other memory operators construct augmented chromosomes by combining an individual's base chromosome with genes from referenced memory slots.

Access to rules in memory is controlled through each individual's memory chromosome. The memory chromosome consists of l bits set to either 1 or 0. Each bit is directly associated with a given slot (bit position one corresponds to slot 1, bit two corresponds to slot 2, etc.) A "1" 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 for illustration purposes. Assume that a 3-slot memory structure exists. Slots contain 4-bit genes as indicated below:

	Slot 1	Slot 2	Slot 3
Contents:	1001	0110	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 genes contained in slots referenced by the memory chromosome are randomly inserted into the base chromosome to create an augmented chromosome representing a complete solution. This augmented chromosome is then sent to the fitness function for evaluation. For our individual X above, the augmented chromosome might appear as follows:

0111 1101 1011 0001 ***1001*** 0101

The two rules in bold italicized typeface were obtained from memory slots referenced by "1" bits in the first and third position of X's memory chromosome. The rule "0110" from slot 2 was not included since X's memory chromosome does not reference this slot (bit position two contains a "0"). The other four rules (non-italics) of X's augmented chromosome are from its own base chromosome.

The execution of a ChGA occurs as shown in the following pseudo code:

```
procedure Chunking_GA

  // Initialize Memory
     for (i=1 to number of slots){
        set slot to random bit string;}

  // Initialize Population
     for (j=1 to population size){
        set memory chromosome to random bit string;
        set base chromosome to random bit string;}

  // Begin GA Run
     while (stopping condition not satisfied){
        for (j=1 to population size){  // Evaluate Fitness
           augmented chromosome = base + referenced slots
           evaluate fitness of augmented chromosome;}
        for (j=1 to population size / 2){ // Reproduction
           select two parents for crossover;
           crossover base chromosomes;
           crossover memory chromosomes;
           update children with resulting chromosomes;}
        for (j=1 to population size){  // Mutation
           perform mutation on child base chromosomes;
           perform mutation on child memory chromosomes;}
        copy children to next generation;
        if (update interval reached)  // Update Memory
           determine frequent genes from population
           update memory
     }
end procedure Chunking_GA
```

During execution, the ChGA must periodically identify good genes and determine if they should be added to memory. We do this by counting the number of occurrences (frequency) of each gene within the base chromosomes of the entire population. Genes are then sorted by their frequency. The basic assumption is that genes which are more useful or valuable will appear more frequently within the population. There may be other more accurate methods for identifying good genes but in this work we use frequency as a measure of "goodness" due to its simplicity.

Once frequent genes are identified and sorted, we compare them with the existing contents of memory. Memory is updated if a gene appears more often in the population's base chromosomes than a current slot is referenced. The contents of the less referenced slot is then replaced with the newly found frequent gene.

4 Experiment Design

Experiments conducted for this work have a twofold purpose. First, we want to investigate the ChGA's ability to eliminate unnecessary genes from variable length chromosomes. Second, we want to illustrate the ChGA using a fitness function somewhat simpler than the MAV simulator used in [1]. A simpler fitness function would consume less computational resources and allow for direct analysis of results due to *a priori* knowledge of optimal solutions to the function.

4.1 A Simpler Fitness Function

It was first noted in [1] that the majority of rule sets evolved by the ChGA became smaller over time with no loss in overall fitness. This anti-bloat property occurred despite the absence of any explicit parsimony pressure in the ChGA or in the MAV simulator fitness function. Why then did this winnowing occur? Were reductions in base chromosome size a fluke? Were they a bi-product of the simulator? Or were they the result of some random occurrence?

In order to answer these questions efficiently, we require a new fitness function that meets a number of important criteria. Chief among these criteria is our ability to know the optimal solution to the function *a priori* and the exact number and composition of genes which represent the optimal solution. Other important criteria include: (1) a fitness function that is computationally simple allowing for numerous, fast evaluations (the MAV simulator require up to six seconds per evaluation); (2) a fitness function that is easily replicable by other researchers; and (3) solutions must be represented with variable length bit strings yet allow for the existence of unexpressed genes that do not contribute to the fitness of an individual.

We have named the fitness function which meets these criteria the "*m* x *n* MaxSum" function. During evaluation, each gene within a chromosome is viewed as a 2-tuple <*m*, *n*> where the first *m*-bits represent a positive integer index and the remaining *n*-bits represent a positive integer value which contributes to an individual's fitness. For this work we have utilized three versions of this function:

- 2x8 MaxSum: Indices range from 0-3, Values range from 0-255
- 3x8 MaxSum: Indices range from 0-8, Values range from 0-255
- 4x8 MaxSum: Indices range from 0-16, Values range from 0-255

The function is the sum of the largest value associated with each of the 2^m indices. In the case of the 2x8 MaxSum, the optimal fitness of a chromosome is 1020 based on the presence of the following 2-tuples appearing somewhere in the chromosome: <0, 255>, <1,255>, <2, 255> and <3, 255>.

Chromosomes can be of any length with this fitness function since only the largest value for each index is added to the sum. As a result, many of the genes in a chromosome can go unexpressed and serve no specific purpose.

Using the 2x8 example, individuals within a population can have a fitness less than the optimal (1020) for one of two reasons: 1) no gene exists for one or more of the permitted indices (underspecification) or 2) the maximum value represented by a gene's *n*-bits represents a number less than 255. No effort is made to make up for underspecification. A chromosome must contain 2^m tuples, one for each index in order to reach the optimal solution.

Our winnowing hypothesis states that the ChGA's ability to reduce chromosome size is tied to the size of the optimal solution. In previous work, a memory size of 10 slots was sufficient to cause anti-bloating to take place. Experiments for this work take advantage of the fact that we know the optimal size of the solution to any *m*x*n* MaxSum function – that size is 2^m. Knowing this, we are able to option a ChGA with memories of different sizes relative to the known size of the optimal solution.

There are a few caveats that should be made about our choice of fitness function. First, we recognize that the *m*x*n* MaxSum is a simple, linear function and does not contain any epistatic relationships between genes. Each gene can be optimized independently. Further, the function can be solved in a number of other ways including a simple fixed length GA. But our goal is not to put the ChGA to the test in terms of problem complexity. Rather, we are investigating only a single property of this GA and chose a fitness function that makes this both possible and efficient.

4.2 Chunking GA Details and Parameters

A series of experiments were performed to test our winnowing hypothesis. Each set of runs used a different size memory. The number of memory slots were chosen to be less than, equal to, and greater than the minimal number of genes in an optimal *m*x*n* MaxSum solution (2^m). Specifically, the following memory sizes are tested:

- For 2x8 MaxSum: No Memory (0 slots), 2, 4, 6, 8, 12, 16, & 24 slots.
- For 3x8 MaxSum: No Memory (0 slots), 4, 6, 8, 10, 12, 16, & 24 slots.
- For 4x8 MaxSum: No Memory (0 slots), 10, 12, 16, 24, 32, 48 & 56 slots.

If our hypothesis holds, we should see reduction in base chromosome sizes for 2x8, 3x8 and 4x8 MaxSum functions when memory contains a number of slots greater than or equal to 4, 8, and 16, respectively.

Features and parameters incorporated into the ChGA for the majority of our experiments are listed below. Unless otherwise indicated, results from all experiments were averaged over 50 runs.

- Generations per Run = 300
- Population Size = 100 Individuals
- Genes per Chromosome = 20 Initial, 100 Maximal
- Number of Bits per Gene = 10, 11 or 12 (varies based on fitness function)
- Selection Method = Rank Proportional
- Crossover Type = 1-Point (both base and memory chromosomes)
- Crossover Rate = 70% (both base and memory chromosomes)
- Mutation Type = Bit Flip (both base and memory chromosomes)
- Mutation Rate = 1% (both base and memory chromosomes)
- Number of Memory Slots = varies with each experiment
- Memory Update Interval = every 10 generations

The majority of these parameters are reasonably self-evident but one in particular should be explained. At the start of each run, an initial population is generated using random bit strings. Each base chromosome starts out with 20 genes. As the ChGA executes, base chromosomes are free to increase in size up to 100 genes. Any base chromosomes larger than 100 genes that result from the crossover of two large individuals is right truncated back to 100.

5 Results from Experiments

The results of our experiments confirm our hypothesis. The winnowing effect of the ChGA is directly related to the minimal number of genes required for an optimal solution. Figures 1 through 3 show the results from experiments with different *m*x*n* MaxSum problems. In all cases, the chunking GA reduces base gene size when the number of memory slots is large enough to hold the minimal size solution (2^m).

When the number of available slots is less than the minimal size, the base chromosome grows and remains at an average length of 80 genes. Variable length base chromosomes are unable to go beyond this average size due to the 100-gene maximum cap specified in the ChGA's parameters.

Figures 1, 2 and 3 also illustrate an apparent correlation between speed of reduction and number of available slots. The greater the ChGA's memory capacity relative to the minimal solution size, the faster the reduction in base chromosome size.

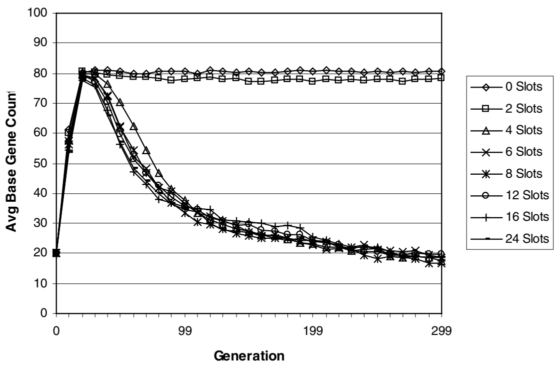

Fig. 1. Average Base Gene Counts for 2x8 MaxSum Problems. Base gene count decreases when memory holds 4 or more slots

But why do base chromosomes reduce in size to begin with? An analysis of memory contents indicates that the ChGA copies good genes into the available slots. Individuals in the population then evolve memory chromosomes to take advantage of these good genes. As an example, Table 1 gives the memory contents at the end of three different randomly selected runs.

In the Table 1, 8-slot example, memory contains the optimal genes for solving a 3x8 MaxSum problem. Individuals in the population that reference all of these genes will have the optimal solution regardless of the contents of their base chromosome.

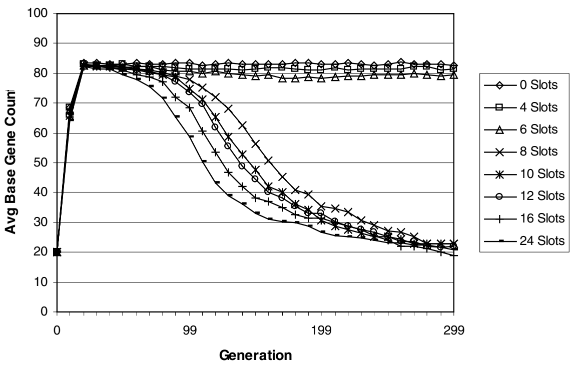

Fig. 2. Average Base Gene Counts for 3x8 MaxSum problems. Base gene count decreases when memory holds 8 or more slots. Shows speed of reduction related to number of slots.

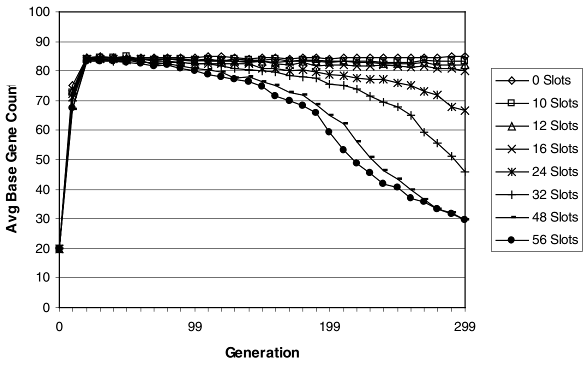

Fig. 3. Average Base Gene Counts for 4x8 MaxSum problems. Base gene count decreases when memory holds more than 16 slots. Shows speed of reduction related to number of slots.

And as seen in Figure 2, the ChGA can then begin reducing the size of the base chromosome. In one of the more surprising results of this work, the base chromosome on many individuals actually reduces to zero length. The entire solution is contained in memory as referenced by the individual's memory chromosome.

Table 1's 4-slot example provides a contrast to the winnowing effect. The example has very good genes for indexes 0, 3, 4 and 5 but not enough room to hold an entire solution. As a result, the population must use its base chromosomes to find the four missing genes. At this point, the ChGA acts as a standard GA with variable length chromosomes and thus is subject to bloat.

The 12-slot example illustrates another interesting point. Here memory contains more slots than that required for the minimal optimal solution. For each index, a near maximal value is included. Redundant slots (those with duplicate indices) allow the individual to select from more than one possibility. Note that the frequency is lower for lower value slots.

Table 1. Samples of memory slot contents and frequency for 3x8 MaxSum runs. For each Content entry *a*/*b*, *a* represents the index while *b* represents the value. Frequency entries show the number of indivuals out of 100 which reference that slot in thier memory chromosome.

	4 Slots		8 Slots		12 Slots	
Slot #	Content	Freq.	Content	Freq	Content	Freq
1	5/255	95	1/255	99	6/255	100
2	3/255	92	7/255	100	5/78	37
3	4/254	98	5/255	99	1/114	25
4	0/255	99	0/255	100	5/255	97
5	n/a	n/a	3/255	99	1/254	93
6	n/a	n/a	6/255	98	3/255	99
7	n/a	n/a	4/255	100	1/255	98
8	n/a	n/a	2/255	99	2/255	100
9	n/a	n/a	n/a	n/a	7/255	99
10	n/a	n/a	n/a	n/a	4/255	99
11	n/a	n/a	n/a	n/a	1/240	62
12	n/a	n/a	n/a	n/a	0/255	97

But exactly when does the reduction in base chromosome size take place? The following figures give us some idea. Figures 4, 5 and 6 show average (online) fitness, average best (offline) fitness and the average base chromosome size (gene count).

We include Figure 4 for completeness. This graph illustrates how the ChGA behaves without any memory (0 slots). The size of the base chromosome increases quickly to its maximum capacity. Only the ChGA's 100-gene truncation parameter keeps the size of the base chromosome from increasing indefinitely.

Figure 5 shows base chromosome size increasing at the start of a run. This appears to continue until a best fit individual arises that contains the optimal solution. Base chromosome size begins to decrease slowly at first until the average fitness of the population is also very high or close to optimal. Winnowing then picks up speed dramatically. A quick inspection of memory shows that memory slots contain most, if not all of the optimal gene set when this reduction speed up occurs.

Figure 6 shows results similar to that of Figure 5 and further illustrates the fact that reduction in base chromosome size occurs earlier given larger memory capacity. We suspect that this happens since the larger memory allows critical genes to be "loaded" into slots earlier – there is more opportunity to be added to memory earlier.

Due to space limitations we are not able to present graphically, the results for all of the different experiments run. We did find similar results for all of the various combinations of memory size and *m*x*n* MaxSum functions listed earlier in this work.

6 Conclusions

In the introduction, we put forth the idea that the Chunking GA is able to reduce solution or chromosome size for variable length GAs. We hypothesize that this winnowing effect is influenced by the size of a ChGA's memory capacity.

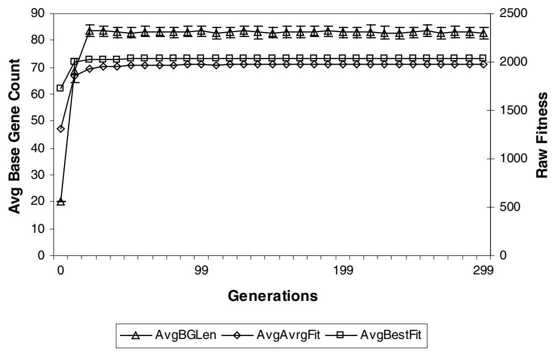

Fig. 4. Fitness vs. Base Gene Count (+/–1 std. deviation) for 3x8 MaxSum without memory

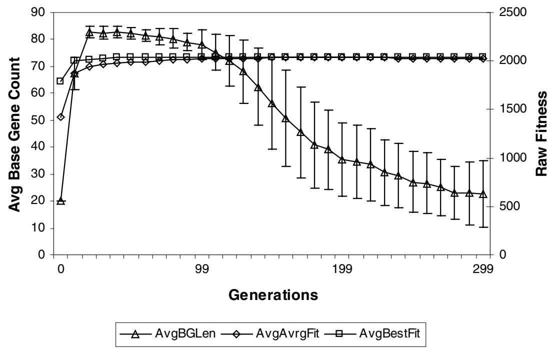

Fig. 5. Fitness vs. Base Gene Count (+/–1 std. deviation) for 3x8 MaxSum w/ 8-slot memory

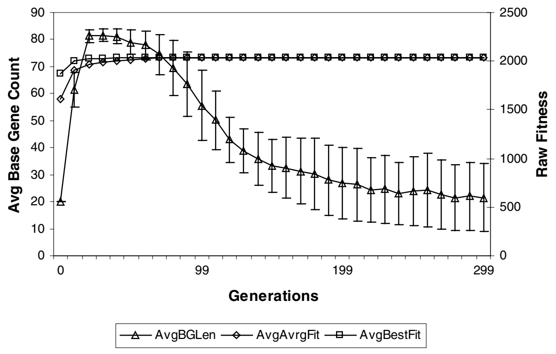

Fig. 6. Fitness vs. Base Gene Count (+/–1 std. deviation) for 3x8 MaxSum w/ 32-slot memory

Experimental results indicate that this is the case. Specifically, winnowing occurs when the memory capacity is sufficient to hold the minimal optimal solution.

Providing even greater memory capacity than that required for the minimal solution further speeds up the reduction in chromosome length. This speedup, however, is not without cost. Memory chromosomes may point to slots containing optimal genes as well as slots containing redundant counterparts. Deciding which slots contain the best genes, even from a reduced set, may be difficult. The best approach is to size memory as small as possible to capture the minimal size solution.

Our studies so far do not indicate that the ChGA necessarily yields "better" solutions unless we take into account shorter size as a criteria. If we do, then the ChGA provides optimal solutions both in fitness and in size. These shorter solutions are possible without incorporating explicit parsimony pressure in the fitness function.

The underlying cause to the ChGA's winnowing effect can be attributed to the natural desire of a variable length GA to evolve shorter chromosomes. In [26] we show that absent any selection pressure, no more than 1/3 of all child chromosomes produced by a variable length GA will be longer than their longest parent. Over time, a variable length GA will instead evolve shorter and shorter chromosomes until little or no genetic material exists in the population.

A ChGA with the proper size memory is able to select and store genes forming the best solution in its memory structure. The population then converges on a solution with a memory chromosome referencing all of the best slots in memory. Once all individuals use the same slots, they will all evaluate to the same fitness. Selection among equally fit individuals becomes random selection. At this point, the ChGA begins producing children with shorter and shorter base chromosomes until all that is left are the slots referenced by memory chromosomes.

Variable length GAs have uses in a variety of problem domains. Their use can be further enhanced by the ability to minimize the size of a solution. Without a reduction in size, the GA practitioner may not be able to determine which genes are the ones which really create the true solution. The Chunking GA illustrates an approach for finding and isolating the best genes, building blocks or rules which make up a complete solution.

References

1. A.S. Wu & H. Stringer: "Learning using chunking in evolutionary algorithms", *Proc. 11th Conf. on Computer-Generated Forces and Behavior Representations*, pp. 243-254, 2002.
2. J.E. Laird, A. Newell & P.S. Rosenbloom: "Soar: An architecture for general intelligence", *Artificial Intelligence*, Vol. 33, pp. 1-64, 1987.
3. T. Blickle & L. Thiele: "Genetic programming and redundancy", *Genetic Algorithms Within the Framework of Evolutionary Computation* (Workshop at KI-94), pp.33-38, 1994.
4. J. Koza: *Genetic Programming*, MIT Press, Cambridge, MA, 1992.
5. W.B. Langdon & R. Poli: "Fitness causes bloat", *2nd On-line World Conference on Soft Computing in Engineering Design and Manufacturing*, 1997.
6. P. Nordin & W. Banzhaf: "Complexity compression evaluation", *Proc. 6th Int'l Conference on Genetic Algorithms*, pp. 310-317, 1995.
7. J. Rosca: "Generality versus size in genetic programming", *Genetic Programming*, 1996.
8. T. Soule, J.A. Foster & J. Dickinson: "Code growth in genetic programming", *Genetic Programming*, pp. 215-233, 1996.

9. D.E. Goldberg, G. Korb & K. Deb: "Messy genetic algorithms: Motivation, analysis , and first results", *Complex Systems*, pp. 3:493-530, 1989.
10. I. Harvey: "Species adaptation genetic algorithms: A basis for a continuing SAGA", *Proc. 1st European Conference on Artificial Life*, pp. 346-354, 1992.
11. J.J. Grefenstette, C.L. Ramsey, and A.C. Schultz: "Learning sequential decision rules using simulation models and competition", Machine Learning, 5:4, pp. 355-381, 1990.
12. 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.
13. C.L. Ramsey, K.A. De Jong, J.J. Grefenstette, A.S. Wu & D.S. Burke: "Genome length as an evolutionary self-adaptation", *PPSN 5*, pp. 345-353, 1998.
14. R. E. Smith and D. E. Goldberg: "Diploidy and dominance in artificial genetic search", *Complex Systems*, Vol. 6, Number 3, pp. 251-285, 1992.
15. K. Yoshida and N. Adachi: "A diploid genetic algorithm for preserving population diversity", *Parallel Problem Solving from Nature 3*, 1994.
16. K. P. Ng and K. C. Wong: "A new diploid scheme and dominance change mechanism for non-stationary function optimization", *Proc. 6th Int'l Conf. on Genetic Algorithms*, 1995.
17. J. Lewis: "A comparative study of diploid and haploid binary genetic algorithms", Master's Thesis, University of Edinburgh, 1997.
18. D. Dasgupta and D. R. MacGregor: "Nonstationary function optimization using the structured genetic algorithm", *Parallel Problem Solving from Nature 2*, pp. 145-154, 1992.
19. A. S. Wu and R. K. Lindsay: "A comparison of the fixed and floating building block representation in the genetic algorithm", *Evolutionary Computation*, 4:2, pp. 169-193, 1996.
20. W. Banzhaf: "Genotype-phenotype mapping and neutral variation - A case study in genetic programming", *Parallel Problem Solving from Nature 3*, pp. 322-332, 1994.
21. H. Kargupta: "The gene expression messy genetic algorithm", "*Proc. IEEE Int'l Conference on Evolutionary Computation*", pp. 814-815, 1996.
22. G. R. Harik: "Learning gene linkage to efficiently solve problems of bounded difficulty using genetic algorithms", Ph.D. Thesis, University of Michigan, 1997.
23. 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.
24. J.S. Byassee & K.E. Mathias: "Expediting Genetic Search with dynamic memory". *Proc of the Genetic and Evolutionary Computation Conference 2002*, 2002.
25. C.L. Ramsey & J.J. Grefenstette: "Case-based initialization of genetic algorithms", *In Proceedings of the 5th International Conference on Genetic Algorithms*, 1993.
26. H. Stringer & A.S. Wu: "Bloat is unnatural: An analysis of changes in variable chromosome length absent selection pressure", Technical Report CS-TR-04-01, University of Central Florida, 2004.