Natural Metaphoric Optimization Algorithms

by

Kent Arthur Spaulding, B.A.

Report

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

December 1998

Natural Metaphoric Optimization Algorithms

Approved by
Supervising Committee:

Dedication

This report is dedicated to my wife, Tracy, and the Peanut, without whose patience and support it would not have been possible.

Acknowledgements

I would like to thank all of the fine software engineers and programmers who have helped me develop an understanding of software and software systems development. I would also like to thank my manager, Tim Magnuson, and Trajecta's CEO, Trey Herschap, for patiently supporting my research efforts.

November 20, 1998

Abstract

Natural Metaphoric Optimization Algorithms

Kent Arthur Spaulding, M.S.E.

The University of Texas at Austin, 1998

Supervisors: Calvin Lin and Craig Chase

In the current business climate the quest for profit in the face of lower margins drives companies to be more efficient and more careful when making decisions. Software systems need to enable users to state their goals and resource constraints and then suggest an optimal plan that achieves that goal. Global optimization techniques can provide these kinds of answers. Many global optimization techniques are difficult to use. This report reviews promising and accessible techniques for global optimization that are based on metaphors from the natural sciences. We apply some of these algorithms to several test problems using genetic programming and dynamic constraint optimization techniques.

Table of Contents

List of Tables xi

List of Figures xii

List of Illustrations xiii

Introduction 1

- Motivation 1
- Objectives 2
- Approach 3
- Overview 4

Literature Review 6

- Introduction 6
- Technical Overview 6
  - A Few Search Strategies 8
  - Continuous Optimization 10
  - Combinatorial Optimization 11
  - Metaheuristic Searching 12
  - Global and Local Searching 13
- Metaphoric Optimization Algorithms 14
- Natural Metaphoric Optimization Algorithms 15
  - Genetic Algorithms 15
    - Strengths and Weaknesses of GA's 17
  - Evolutionary Programming 18
    - Strengths and Weaknesses of EP 20
  - Simulated Annealing 21
    - The Process of Annealing 21
    - Strengths and Weaknesses of SA 23

Genetic Programming....................24
Strengths and Weaknesses of GP....................25
Grammar-Based Genetic Algorithms....................25
Strengths and Weaknesses of GBGA's....................26
Ant Colony Optimization Systems....................27
Ant Colony Behavior....................27
The Traveling Salesman Problem....................30
Evolution of Ant-Based Algorithms....................30
The Ant System Algorithm....................32
The Ant Colony System....................35
MAX-MIN Ant System....................35
Continuous Ant Colony Optimization....................36
Global Search in CACO....................37
Local Search in CACO....................38
Constrained Optimization in CACO....................39
Another Global Optimization Algorithm of Interest....................40
GRG2....................40
Strengths and Weaknesses of GRG2....................42
Summary of Literature Review....................42
An Introduction to MasterMind and Possible Solutions....................43
Introduction....................43
MasterMind and Its Variants....................43
Solving 3-3 MasterMind....................44
The Mini MasterMind Language....................44
An Implicit Strategy for MML....................45
MML in More Detail....................46
Estimating the 3-3 MasterMind Search Space....................49
An Approach to Solving 3-3 MasterMind Using a GBGA....................51
An Approach to Solving 3-3 MasterMind Using ACO....................52

Solving 4-6 MasterMind....55
The Dynamic Constraints Approach....55
Estimating the 4-6 MasterMind Search Space....57
Solving 4-6 MasterMind with a GBGA....57
Solving 4-6 MasterMind with an ACO Algorithm....58

Experimental Results....60
Introduction....60
3-3 MasterMind Results....61
3-3 GBGA Results....61
Objective Function....62
Intermediate Results....63
Non-Elitist, Default Crossover and Mutation....64
Elitist, Default Crossover and Mutation....65
Non-Elitist, 0.9 Crossover and 0.15 Mutation....66
Analysis of the 4.37 Result....67
Reanalyzing the 4.37 Result....69
Analysis of the 3.93 Result....70
Comparison with Published Solutions....72
4-6 MasterMind Results....72
GBGA Results....72
Summary of Experimental Results....73

Conclusion....74
Main Findings....74
Original Contributions....75
Limitations....75
Opportunities for Further Work....75
Solution- and Approach-Neutral Possibilities....76
The MML Approach....76

The Guess-to-Guess Approach 76
Dynamically Specifying Constraints Using a Grammar 77

References 78

Vita 81

List of Tables

Table 1: A Representation of a 3-3 MasterMind Program 47
Table 2: An Example Row with Pruning Operators..48
Table 3: Impossible Responses in an MML Program 50

List of Figures

Figure 1: Historical Progression of Ant-Based Algorithms 31
Figure 2: The Ant System Algorithm..34
Figure 3: Finding 2 Digits with 3 Values Using TSP..54
Figure 4: The Best Early Non-Elitist Performer...65
Figure 5: The Best Early Elitist Performer...66
Figure 6: The 4.37 Result 67
Figure 7: The Reduced 4.37 Result 68
Figure 8: The Optimal First Row for 3-3 69
Figure 9: The Reduced 3.93 Result 70
Figure 10: A Graph of Progress Towards the 3.93 Result.....................................71

List of Illustrations

Illustration 1: Ants and Obstacles..29

Introduction

MOTIVATION

The current business climate, with its raft of mergers and acquisitions, is driving businesses to compete against one another for lower and lower profit margins. This is true across many industries, including Telecommunications, Cable and Broadcast Television, Computers, Consumer Electronics, Banking and Consumer Credit. The quest for profit in the face of lower margins drives companies to be more efficient, and hence more careful when making decisions.

To illustrate, consider the credit card industry where the need for good decisions is profound. In the current business climate, the only way for a credit card company to maintain profitability is to accept increased risk for the same return. Hence, card issuers face a common dilemma. How does one make decisions in an environment where there is a great reward for being correct and severe penalties for being wrong?

Software in the form of data-mining tools used for advanced data analysis addresses this question by evaluating proposed plans of action. Data-mining software finds patterns in vast data warehouses and is then used to predict the probable outcome of a given proposed plan. This all well and good, but begs the question, “If this is a good plan, then how can it be better?”

Some data-mining tools address this issue by using optimization technologies to fine-tune a given plan. Imagine being able to state your goals along with some set of resource constraints and have software suggest an optimal plan that achieves that goal while minimizing consumed resources. Global optimization techniques can provide these kinds of answers; the better the optimization algorithm, the better the plan and the higher the rewards.

What's the downside? Many global optimization techniques are difficult to comprehend, and hence difficult to use.

Given the current business climate and the need for optimization, there is a trend to incorporate more and more intelligence into software (Watson 97). Given this trend, it clearly behooves Software Engineers to understand optimization technologies.

This report investigates a couple of promising and accessible techniques for global optimization.

OBJECTIVES

This report has two primary objectives. First, to build a broad understanding of the field of non-linear programming, which incorporates a wide-ranging variety of 'global optimization' tools and techniques. The second objective is to learn to apply interesting optimization techniques to a meaningful problem. Finally, we keep an eye towards using optimization technologies in future software engineering projects.

Extending the state-of-the-art is not an objective; this research is being undertaken in order to provide the author with an understanding of global-optimization techniques and the beginnings of a process to follow for learning and applying new techniques to novel problems. In other words, this research should result in the ability to answer high-level questions regarding chosen techniques.

For example, what are the relative strengths and weaknesses of a given method? What kinds of problems are best suited for attack via this technique? How difficult (or easy) is it to map a given problem into a given technique? How easy is it to incorporate a given technology into a software system? As optimization becomes more and more useful, these questions will need to be answered on an on-going basis.

Generally, this report should provide the beginnings of a process to follow for learning about, evaluating, and applying new optimization techniques as they are discovered and appear in the literature.

APPROACH

The methodology used to compile this report was quite straightforward. It started with an interest in 'metaphoric optimization techniques,' that is, techniques based on observations of physical phenomena. More specifically, this interest is focused on algorithms grown from studies of biological systems. These techniques are in contrast to purely mathematically based techniques like the popular Generalized Reduced Gradient (GRG2) method or even branch and bound techniques. The descriptions of metaphoric optimization techniques are rooted in familiar terminology, which make them more accessible than many other techniques.

The next step was to review the literature and find specific metaphoric optimization techniques of interest. Grammar-Based Genetic Algorithms (GBGA) and Ant Colony Optimization (ACO) were selected from a plethora of options. These appeared to be the most accessible methods, and have also demonstrated promising results when applied to standard test problems (Antonisse 91)(Colorni 91).

Next, it was time to find a problem to solve. We settled on the game of MasterMind[1]. Why MasterMind? First, it has several variations and hence a variety of search spaces and strategies to choose from. Second, many variants do not have known optimal solutions. Third, there are some published results for certain variants for both a straight genetic algorithm and a simulated annealing solution. Fourth, MasterMind can be tackled with the chosen optimization techniques in interesting ways. Finally, MasterMind should be illustrative of the various strengths and weaknesses of the GBGA and ACO algorithms. MasterMind serves as an excellent test bed, and yes, it is fun.

Why not a business problem? In the end, access to appropriate and publishable data became an issue. All of the stated research goals are met just using MasterMind.

The remainder of the research was simply of matter of finding or developing code to use for the evaluation of the techniques and performing experiments that attempt to solve the test problems.

OVERVIEW

This report is divided into five major sections. This introduction is first. A section covering the current literature on selected optimization techniques follows the introduction. This section focuses on Natural Metaphoric Optimization Algorithms, with an extra emphasis on Ant Colony based techniques and

[1] MasterMind is a trademark of Invicta Plastics, Inc.

Grammar-Based Genetic Algorithms. The third section provides an introduction to the test problems and their anticipated experimental solutions. Section four focuses on the experiments, including lessons learned during the process and the experimental results. The fifth major section provides a conclusion. It also considers additional research and discusses the possibilities for future work based on this research.

Literature Review

INTRODUCTION

This section provides a brief literature review covering the optimization techniques and technologies that were investigated while formulating solutions for MasterMind. This review is by no means intended to be complete; rather, it is intended to serve as a brief introduction to some of the concepts involved in Nonlinear Programming[2] (NLP) and optimization techniques. It should also serve to define terms and techniques used and discussed in the remainder of the report.

This section begins by providing a high-level technical overview of NLP and optimization. This section is largely a consolidation of the overviews by Fourer and Greening (Fourer 97)(Greening 92). After the technical overview, this section discusses several Natural Metaphoric Optimization Algorithms at a high-level. Grammar-Based Genetic Algorithms and Ant Colony Optimization are then discussed in more detail. Finally, we look at other non-metaphoric techniques that might be of interest to Software Engineers.

TECHNICAL OVERVIEW

Optimization and global optimization techniques fall into a field of study called Nonlinear Programming (NLP). NLP is used to find good solutions to problems that can be expressed in terms of an *objective function* and a number of

[2] In this context, the term, "Programming", does not refer to act of developing code, but rather the act of "planning."

constraint functions. An objective function takes a vector of input values as an argument and returns a vector of output values.

The objective function describes the search space in which one wants to find a minimum or maximum set of output values. It is often convenient to think of objective functions as defining an n-dimensional landscape, where n is the number of input values over which one wanders in order to find the high or low point. Constraint functions limit the search of this landscape to specific areas. Simply put, all optimization problems are just a search for the best combination of input values to an objective function that, given some constraints, produce the optimal set of output values.

While it sounds simple in concept, it is actually very difficult to find good general solutions for all sets of possible objective and constraint functions. First, these search spaces are very large. Additionally, many problems have a number of local minima or maxima that are difficult to locate and/or hard to distinguish from one another.

To illustrate the concept of multiple local minima, consider the following minimization problem. Imagine that you are walking on a ranch and have the task of finding the lowest point on the ranch so you can drill a well (objective function). You know the boundaries of the ranch when you see them (constraint functions), but have no map to follow. You decide to traverse the ranch, but how do you know that you are at the lowest place on the ranch? When you are in depressions you cannot see over the hills around you. You can keep notes, so one strategy might be to stand on the highest hill you can find, look around for promising depressions, and then search each one for depth. Assuming you can find the highest hill, you might not be able to see all of the depressions to search and there may a large number of depressions. The task is clearly not easy.

It should be clear from the ranch example that even though optimization sounds simple, it is often quite difficult. However, the basic concept is simple; you search a landscape to obtain an objective.

A Few Search Strategies

There are many strategies for searching, including *exhaustive*, *random*, *greedy* and *heuristic* searches. The next few paragraphs look at these gross strategies at a very high-level.

In an *exhaustive search*, every possible set of inputs is tested to find the one that results in the best, as judged by some measure, set of outputs.

In a *random search* we just randomly generate and test combinations of inputs until we find a set of inputs that is good enough. Random search hopes to quickly find a good answer. Naturally, it is not always quick and is always sub-optimal. Random search amounts to random guessing.

A *greedy search* is related to a random search, but employs some intelligence. Like a random search, a greedy search randomly generates and tests combinations of inputs; however, when it finds a good combination it sets it aside and then begins testing other combinations of inputs in the same vicinity as the current best result. The term "greedy" connotes a strong preference for better solutions, and is applied to many algorithms. It follows that whenever a greedy search finds a better result, it sets it aside, and continues search in that local area.

It stops after a set number of tries or when it has found a solution that is sufficient. It has the same weaknesses as random search, but on average can finds solutions that are good enough more rapidly than straight random guessing.

A *heuristic search* follows some strategy, like a human might. For many searches, one could follow a strategy of always moving in an uphill direction, hoping to find better and better results. If the results get better for some time and then start to get worse, you have reached a summit. This is the *hill climbing* strategy. It is commonly used in NLP systems. The use of a strategy makes hill climbing a heuristic search. Hill climbing is more complicated then it sounds. If you reach a summit, how do you know it is the highest summit? There are, naturally, other strategies available.

An as-of-yet unstated goal of searching is to find a near-optimal solution in a search space in a reasonable period of time. All four types of searches may fail to find a good solution, but can find good approximations. How long do we have to wait for a good approximation? In all cases, one could search forever unless some preset time limit is imposed. It may not be immediately obvious as to why exhaustive searches may take forever. It suffices to say that even for simple sounding problems, one could have an infinite number of possible input conditions. For example, one of the inputs may be a real number. There are an infinite number of real numbers, therefore, a function taking real numbers as input has an infinite number of possible combinations of inputs. One cannot possibly test all input combinations for these functions.

Optimization problems are broken into two general classes, based on the type of inputs to the problem. Problems with an infinite number of input or output values are referred to as *continuous optimization* problems. Problems that have a limited set of possible input values (no matter how big the set) are called *combinatorial optimization* problems, or sometimes *discrete optimization* problems.

Continuous Optimization

Continuous optimization problems are the most difficult form of optimization. Note that this is the case, even in the presence of constraints on the inputs. These problems can be very difficult to solve.

Imagine a simple case where one is searching for the maximum value of y given the Cartesian coordinates x and y (from the set of Real numbers), where $x=y$ and $x < 1$. This function defines a line of slope 1, from the origin to the coordinates (1,1) exclusive. It is bounded by the constraint $x < 1$, but it has no maximum value since given any x, we can always pick another that is closer to 12. Obviously, exhaustive search is not possible. If we try a random search we run into the problem that the odds of randomly generating values of x that are very close to 1 are extremely low. Random search could take an unbounded amount of time to find good answers, even in the presence of constraints. We could use a heuristic search, assuming our heuristic can tell us what value is good enough.

If we can construct an example as simple as $x = y$, then it should be clear that continuous optimization problems are very difficult to solve[3].

[3] Strictly speaking, this is not a fair example. The given function has no true optimal value. Objective functions are generally *unimodal* or *multimodal*, meaning that they have one true

Combinatorial Optimization

Many other optimization problems can be characterized as a search for the best combination of a finite set of discrete values to meet a given set of goals. This does not make them easy, but one might argue that they are easier than continuous optimization problems.

The search space for a combinatorial problem can still be quite large. For example, imagine trying to find the substitution values of 0 and 1 for a given Boolean equation such that the result of the equation is true. It has been shown that as the number of terms in an arbitrary Boolean expression increases, the equation is exponentially more difficult to solve. This is, in fact, a well-known NP-Complete problem. Therefore, searching can still take an enormous number of tries before arriving at a solution.

As another example of a combinatorial optimization problem, consider a pharmaceutical salesperson that has a fixed number of expensive samples for several different products to give away to doctors. If the right combinations and quantities of samples go to the right doctors and in turn their patients, the doctors will write more prescriptions on average and the salesperson’s company will realize more profits. Therefore, this salesperson would like to know how to best allocate combinations of samples for each doctor in order to maximize the overall effectiveness of the samples.

One might also imagine a case where a program is trying to decrypt an encoded message by guessing a key. If the program had a means to evaluate how

optimum or many, respectively. This example is simply meant to be illustrative of the problems with the various strategies.

close its current guess is to the actual key, it could use combinatorial optimization techniques to arrive at the correct sequence of characters in the key.

The solutions to a wide class of problems are simply the appropriate sequence of characters or combination of items. The search for discrete combinations, given a set of goals, is called combinatorial optimization.

Metaheuristic Searching

Clearly, random guessing is not the best strategy for finding solutions. There are several improvements that can be made by following a heuristic that we use to make strategic decisions. In the simplest case, we want to avoid making the same guess more than once. We therefore need to keep a history of guesses in order to avoid this. Next, we could make ’educated’ guesses. Simply adding the abilities to only try guesses once and to make ’educated’ guess improves our situation quite a bit.

We can make educated guesses in a couple of ways. First, we could strive to make better guesses based on the results of previous guesses, or even previous attempts at solving similar problems. This implies a need for a history of those previous attempts. Second, we could improve our guessing by knowing more things about the search space. If we are looking for wild elephants, we should not search Canada.

Optimization techniques that employ these capabilities are said to be using heuristic search. Optimization techniques that can employ a number of different heuristics, based on context, use metaheuristic searching. Heuristic searching allows a given technique to quickly locate regions of the search space that contain good solutions. Metaheuristic searches decide how to make these decisions during the search process.

One can think of a metaheuristics as a top-level general strategist that guide other heuristics to search for feasible solutions. The other heuristics can be thought of as tacticians.

The use of a metaheuristic is not always explicit. Examples of metaheuristic techniques that incorporate the metaheuristic into their algorithms are simulated annealing, genetic algorithms, evolutionary programming, and Tabu Search. None of these use an explicit metaheuristic - most are driven by a metaphor that serves as the metaheuristic.

Global and Local Searching

Many algorithms use one method to search the solution space in the large, and another algorithm to search within local regions of the space that look like they contain good solutions. Searching the space in the large is called *global searching*. Limiting the search to nearby areas is called *local search*. When hunting a particular elephant, you consult a book that tells you to go to Africa. When you get there, you hire a local guide to help you find the elephant you want. This is an example of global versus local search.

In general terms, optimization algorithms have been developed to allow us to find good solutions to problems with very large search spaces, without having to wait forever. According to Levy, researchers from a diverse set of disciplines, from astronomy to economics and population studies to mathematical theory have developed these algorithms (Levy 94).

Naturally, all of the known algorithms have their own strengths and weaknesses. For example, some are easier to use than others and some perform better on one class of problems versus another.

The most interesting and successful searches employ heuristics and/or metaheuristics to rapidly converge on good solutions.

NLP is simple in concept, complex in implementation and use. This section provided a brief overview of the concepts. The web is a good place to get a deeper overview and pointers to a variety of resources. The web provided much of the information in this section (Fourer 97).

The next section discusses some easily accessible NLP methods.

METAPHORIC OPTIMIZATION ALGORITHMS

Most NLP techniques are rooted in theories of Mathematics, and are therefore difficult to comprehend without a strong mathematical background. This section introduces terminology for techniques that are based on metaphors from domains outside of computer science and mathematics.

’Metaphoric Optimization Algorithms’ are based on observations of physical phenomena like autocatalytic processes in chemistry or evolutionary biological systems. These techniques are in contrast to purely mathematically based techniques like GRG2 or many linear and non-linear programming solutions. The descriptions of metaphoric optimization techniques are rooted in familiar terminology, which make them more accessible than many other techniques.

NATURAL METAPHORIC OPTIMIZATION ALGORITHMS

Natural Metaphoric Optimization Algorithms (NMOA) are Metaphoric Optimization Algorithms that are based on metaphors derived from the Natural Sciences. NMOA's are a subclass of MOA's.

The next few subsections discuss several Natural Metaphoric Optimization Algorithms. There are a wide variety of NMOA's available; these subsections give a brief overview of several of these techniques and discuss their general strengths and weaknesses.

Genetic Algorithms

John Holland, from the University of Michigan, first described the Genetic Algorithm (GA) in 1975. This section describes the Genetic Algorithm and then discusses its strengths and weaknesses.

Genetic algorithms use principles from biology to provide an efficient means of searching a large search space for a set of near optimal solutions. The possible solutions in the search space are represented as fixed and uniform length *chromosomes*, as one might expect in a biological system. The chromosomes are built up from genes. The genes typically encode the set of variables that compose a solution. Optimal solutions are found by creating a population of chromosomes and playing *survival of the fittest* from one generation to the next.

The basic algorithm generates a population of chromosomes representing some random portions of the search space. It then ranks all individual chromosomes using a *fitness function*. Next, it discards some portion of the least fit chromosomes and then randomly *mutates* some of the population to introduce some random wandering through the search space. After this the algorithm replaces the discarded individuals by randomly combining – using a crossover function - the best-fit chromosomes with one another. This results in a new generation. The algorithm repeats; continuing until the average level of fitness stops changing or some preset number of generations has passed.

To give a simple example, if we are searching the x,y plane for the maximum value along a curve defined by the function foo(x,y), a chromosome might consist of two genes representing x and y.

Fitness functions assign values within some range to chromosomes, based on the *genotype* or the *phenotype*. The genotype is simply the set of genes, as they stand. It can be evaluated by applying some heuristic that has knowledge about good sequences of genes. Evaluating the phenotype means looking at the functionality of the genes. When evaluating phenotypes, the contents of the individual genes do not matter. What matters is how the individual behaves according to some criteria.

If we are evaluating strings, the genotype can be thought of as the bits in the string of values. The phenotype can be thought of as the characters identified by the bits.

In the *foo(x,y)* example, we evaluate the phenotype. We favor x and y values that result in high positive distances from the x-axis and rank them ahead of those that result in smaller distances. This ranking serves as the fitness function for the algorithm.

The least fit chromosomes are discarded from the gene pool and some of the remaining population is mutated. Mutation randomly changes a few bits in a few chromosomes. Again in our example, the mutation function may toggle a few of the bits in the x and y values of the individuals.

Next, new chromosomes are created to take the place of the discarded individuals. A process like mating creates these new chromosomes by using a *crossover function* to swap some of the genes of two of the more fit chromosomes into a new instance. The genes are typically swapped somewhat randomly - just like in real cells. For our example, we might choose to create children by crossing the x and y values in the parents to create two new chromosomes.

The GA can be seen as a hill-climbing algorithm. The fitness function defines the 'landscape' of hills. The GA keeps track of where it has been in the genes of the population of chromosomes. Mutation and crossover allow for some level of backtracking in the space.

Antonisse and Merelo provide a good overview of Genetic Algorithms in their works (Antonisse 91)(Merelo 96).

Strengths and Weaknesses of GA's

Genetic algorithms can be faster than more traditional search methods, especially when run on parallel hardware. The fitness function takes the most processor time in a normal application of the GA. It is easy to distribute the population of chromosome across many machines, letting each machine evaluate the fitness a sub-population of the global populace. Each machine can also independently apply the mutation operator to its sub-population. Crossover is the only time that the machines need to communicate about the global population. They can do this and then start independently evaluating the next generation of chromosomes. The GA is a natural candidate for distribution across machines or for execution on parallel machines.

These issues complicate the use of GA's.

GA's find a near optimal solution, but not the optimal solution. They are not good at finding the one best solution, but are good at making quick approximations and therefore do not do true global optimization.

Representing the search space as a chromosome is difficult since chromosomes are essentially long fixed length strings. Long strings do not intuitively map to many problem domains. This is especially true if all of the chromosomes need to be of a uniform length.

GA's also suffer from the problem that many illegal chromosomes are created by both mutations and crossover. This effectively increases the search space because portions of it are searched more than once. This is called the *illegal chromosome problem*, and can be addressed by using custom crossover functions and/or robust fitness functions.

Evolutionary Programming

This section introduces the Evolutionary Programming optimization technique and then discusses its relative strengths and weaknesses.

Evolutionary Programming is conceptually related to Genetic Algorithms. It is illustrative to discuss the two techniques in terms of one another, so some the terminology used here is used in both this section and the Genetic Algorithms section. See Fogel for details (Fogel 95)(Fogel 97).

Lawrence J. Fogel invented Evolutionary Programming (EP) in 1960. EP is based on an observation of evolutionary biology, at the species level. If you consider generations of a species and its competitors within an ecological niche as 'solutions' for occupying that niche, you can understand the general idea behind EP.

EP leverages the notion that solutions, serving something like parents, can be mutated to produce new more viable solutions as their 'offspring.' In GA's, 'individuals' are like individual organisms. The 'individuals' in EP are species, not individual animals.

The evolution of species (and sometimes subspecies) into better and better solutions is realized in the EP paradigm by the following process:

1. Randomly generate a set of finite automata machines to serve as a population of solutions. This is the first group of *parents*.
2. Copy each of the solutions into a new population of *offspring* and then apply a *mutation* operator that alters the behavior of the individuals in the new population. The behavior of each offspring is compared to that of its parent in order to achieve the appropriate distribution of changes.
3. Evaluate each known individual (from parents and offspring) for *fitness*. Fitness can be defined by its ability to achieve a certain goal (like recognize a given input sequence). Keep some number of individuals to serve as the next generation of parents.

Steps 2 and 3 continue until you have a set of finite automata that achieve the desired goal.

EP is thus similar, at least conceptually, to Genetic Algorithms. However, EP differs in several key aspects.

First, EP is focused on the behavioral differences between parents and their offspring rather than the representation of the individuals. Any representation of individuals can be used, whereas GA's typically represent individual solutions as strings. In other words, EP operates in the domain of the *phenotype*, rather than the *genotype*. It is not the structure of the genes that matters, but rather their expression and their impact on the behavior of the 'animal', in this case, a finite automaton or a program.

Second, EP handles mutation differently than Genetic Algorithms. The mutation operator still randomly alters the individuals, but it is defined in a way that favors mutations that create small variances in behavior versus those that create large variances. Additionally, as the algorithm progresses and good solutions are being found, the mutation operator favors small variances even more than in early generations.

Third, EP systems typically compare individuals against each other in a tournament setting. GA's typically rank individuals based on their performance when provided as input to the fitness function.

Strengths and Weaknesses of EP

EP is a useful method for optimization when other techniques like GRG2 cannot be applied. EP is most useful for combinatorial optimization, and can be used for continuous optimization, especially when there are many potential solutions as opposed to one global solution. These are the same classes of problems that are well suited to a Genetic Algorithms approach.

EP's main advantage over GA's is the ability to represent solutions to the problem at hand in a more flexible manner.

Simulated Annealing

Simulated Annealing is a Metaphoric Optimization Algorithm that stems from the theories of Thermodynamics. It is widely used in a variety of applications, including VLSI circuit design and distributed scheduling problems. Simulated Annealing has even been used to play MasterMind (Bernier 97).

Aarts and Korst give detailed coverage to Simulated Annealing and a relative, the Boltzmann Machine, in their book (Aarts 89).

This section describes annealing at a very high level in order to give the reader some insight into the metaphor. It then discusses the Simulated Annealing Algorithm and some of its relative strengths and weaknesses.

The Process of Annealing

Annealing is a generic term describing a process of treating a material to improve or enhance certain properties. In the annealing process, one heats a material to a given temperature, holds it there until it reaches quasi-equilibrium, and then allows cooling at a slow rate through phase transitions. This process allows materials to seek a lowest energy state.

Annealing is frequently used to soften metallic materials. It can also simultaneously produce desired changes in other properties. In the case of metals, these changes may be an improvement in workability, facilitation of cold work, and/or improvement of mechanical or electrical properties. The annealing process can influence other properties as well. For example, silicon wafers are sliced from large cylindrical blocks of silicon. Before slicing, the blocks are annealed in order to ensure that the distribution of materials within the blocks is even across all regions.

How does annealing work? When a material is heated, the atoms of the various elements in the material are excited to varying degrees. When the material is cooled at specific rates, the material slowly approaches temperatures near phase transitions (e.g. water to ice). At these transition points, the atoms tend to ’settle’ into configurations that can be determined ahead of time. The specific configuration of material depends on the rate of cooling. Since the process is somewhat deterministic, by altering the rate of cooling one can effectively control the microstructure of the material.

How do optimization algorithms take advantage of this? According to Greening (Greening 95), simulated annealing mimics the physical annealing process in software.

Simulated Annealing is a modification of the "greedy algorithm", which is a well-known heuristic used to find approximate solutions in NP-Hard searches. In the very simplest terms, the "greedy algorithm" simply makes a random guess to determine a trial-state. It checks this state against its current best state and if the cost of the trial-state is better, it keeps it. It then keeps guessing in a nearby area until some stopping criteria is met - typically a criteria like ’no improvement has been seen for several iterations.' Like many optimization algorithms, it is easy for the greedy algorithm to get caught in local minima.

Simulated annealing combats this weakness in the greedy algorithm by adding the notion of a 'temperature' to the algorithm. Whenever the cost of a trial-state is compared and the 'temperature' is high, the algorithm is likely to choose to keep the state with the higher cost. This tends to throw the algorithm out of local minima. As the temperature is reduced, the algorithm is less likely to choose a new state with a higher cost over a state with a lower cost. In other words, as the temperature goes from high to low, the algorithm tends to explore smaller and smaller valleys in the objective function's surface.

The likelihood of choosing higher cost states over lower cost states is determined by the temperature function, which, given a time, returns a temperature. This function is usually monotonically decreasing. Variants of the algorithm use different temperature functions, some slow the rate of temperature change near phase transitions (the tops of peaks in the landscape). The algorithm stops when the trial-state and the best state exhibit a distribution of state transitions that fit the Boltzmann distribution. This distribution is straight out of the study of Thermodynamics and implies that the system has reached quasi-equilibrium for a given state. The Simulated Annealing algorithm is simply a simulation of a thermodynamic system.

Strengths and Weaknesses of SA

Simulated annealing makes good approximations for many applications. It makes better approximations when given more time. Results are also affected by experimentation with the temperature function. Despite that, it still get caught in local minima, so does not do true global optimization.

Simulated Annealing works reasonably well, but it is computationally expensive. In fact, there is not a theoretical basis for the claim that it solves NP-Hard problems in polynomial time. However, under certain conditions it converges on solutions in polynomial time (Greening 95).

In the web-based MasterMind solution, Simulated Annealing does not perform as well as Genetic Algorithms (Bernier 97). In many other areas, it performs better.

Genetic Programming

This section presents Genetic Programming and then briefly illustrates the strengths and weaknesses of the technique.

John Koza, from Stanford, is the researcher most frequently associated with Genetic Programming. Genetic programming is a specialized form of a Genetic Algorithm. In general, GP systems search through a set of programs looking for one that solves a particular problem. GP systems represent programs in a number of ways, but they all basically reduce down to lists of symbols that are interpreted as program instructions (Antonisse 91).

Genetic Programming systems have been applied to problems like planning, minimax gaming strategies, data mining applications (function matching), and emergent behavior problems. They can be quite effective in these areas.

Strengths and Weaknesses of GP

Most GP systems have the same strengths and weaknesses as Genetic Algorithms. However, some GPs allow for variable length solutions, which gives them an edge for some problems.

The crossover techniques are usually very specific to the problem at hand, especially considering the variable length strings. Unfortunately, this makes it harder to do crossover so there is a trade-off for allowing some variation in length.

Genetic Programming systems have been used to solve some very difficult problems. In fact, they have performed well on problems that were thought to be almost intractable. However, they are not generally easy to use. They cannot be generally applied to large classes of problems. In other words, they require a great deal of effort to set up and execute.

Grammar-Based Genetic Algorithms

This section discusses a specialization of the GA, called a Grammar-Based Genetic Algorithm (GBGA). The GBGA is the focus of this report, and is simply introduced here. The sections regarding the experiments go into more detail. This section is intended to serve as an introduction.

The Grammar-Based Genetic Algorithm is a generalized form of Genetic Programming (Antonisse 91). A GBGA represents the set of chromosomes using a formal grammar. The grammar is typically context-free and thus represented in BNF. This grammar defines a search space contoured by an objective function implemented as a parser for the BNF grammar.

The BNF grammar can serve as a set of constraints that define a set of programs, each of which is an expression of the phenotype of legal individuals. These strings are the phenotypes and the parse-tree is considered to be the genotype.

An individual is some legal combination of terms from the BNF. These individuals are ranked for fitness using an objective function that either walks the parse tree (genotype) for an individual or sends the string representation (phenotype) to an interpreter that judges relative fitness.

The GBGA finds optimal solutions in the set of legal grammars based on some fitness criteria.

What about the crossover function? A GBGA system uses a generalized parse-tree crossover function to create new individuals in a population. Basically, sub-trees of two mating individuals can be swapped. This is guaranteed to create two new legal individuals, thus avoiding the illegal chromosome problem found in some GA systems.

Mutation can be accomplished by changing the terminal nodes of the parse tree to some other legal value from the grammar.

Strengths and Weaknesses of GBGA's

GBGA's simplify the representation of the search space by allowing it to be expressed in a BNF.

GBGA's simplify the crossover function by allowing for the crossing of grammar trees. Usually, the developer can simply use the general crossover function.

GBGA’s do not suffer from the illegal chromosome problem, so the search space is effectively smaller.

Like all GA systems, a GBGA finds a good solution, not necessarily the best solution.

ANT COLONY OPTIMIZATION SYSTEMS

Ant Colony Optimization systems are the most interesting and recent NMOAs. This section discusses several of these algorithms in some detail.

The general Ant Colony Optimization algorithm is based in part on the natural behavior of ant colonies, and was first discussed in the work done by Dorigo, Maniezzo and Colorni (Colorni 91). This section discusses the metaphor for the algorithm, the general algorithm itself, and some of the systems based on this work.

Ant Colony Behavior

Given the ubiquity of ants, it is safe to say that we have all spent time watching real ants forage for food. Ants are nearly blind and appear to wander aimlessly, but when an ant finds food and begins to carry some back to the nest other ants are sure to arrive shortly. Before long there is a highway of ants taking food back to the colony’s nest.

When there is highway of ants carrying food, what happens if we now put an obstacle in the trail? The ants quickly navigate around the obstacle, and eventually, almost all of the ants take the shortest path around it.

How do animals like ants manage to find any path to food, let alone the shortest path between food and the nest?

Edward O. Wilson discovered that ants communicate by using chemical signals (Holldobler 90). Ants deposit chemical markers, which are called pheromones, on the ground as they travel. The pheromones are deposited in varying quantities and act like a trail of breadcrumbs that other ants can follow. Ants use pheromones to communicate their trail to other ants.

A single ant moves at random, and upon encountering a pheromone trail, it may decide to follow it based on the amount of pheromone on the trail. If it does follow the trail, it deposits more chemicals, thus reinforcing the trail. The more ants that follow a trail, the more likely it becomes that other ants will follow it. The probability of an ant following a trail is related to the number of ants that previously followed that trail.

It should be clear how trails of pheromone allow ants to find food and cooperate once they have discovered it; however, the exact mechanism for taking the shortest route around obstacles is not quite so obvious.

When ants arrive at a new obstacle, there is no established trail of pheromones so they are just as likely to choose one direction or the other. Having chosen one, they continue on until they find the nest. They drop their food, and return to the food source. As they return, they again encounter the obstacle, but they take the path with the most pheromone on it.

Why does one path have more pheromone than the other path? Imagine that ants lay pheromones down at a constant rate, one unit per inch of travel, and ants move one inch every second. Now consider a food source and a nest that are ten inches apart. There is a four-inch wide obstacle in between the food and the nest. The obstacle is situated so that if the ants go to one side they travel a total of sixteen inches for a one-way trip. If they go the other way, they travel twelve inches one-way. Pheromone will accumulate more quickly on the short side, since a round trip on that path deposits 24 units, which over one minute corresponds to 60/24 units per inch per ant. The longer trail only has 60/32 units per inch per ant of pheromone. Over time, more ants will take the shorter path and even more units of pheromone will accumulate. This explains why more pheromone accumulates on the shorter path. This extra accumulation allows ants to find the shortest paths.

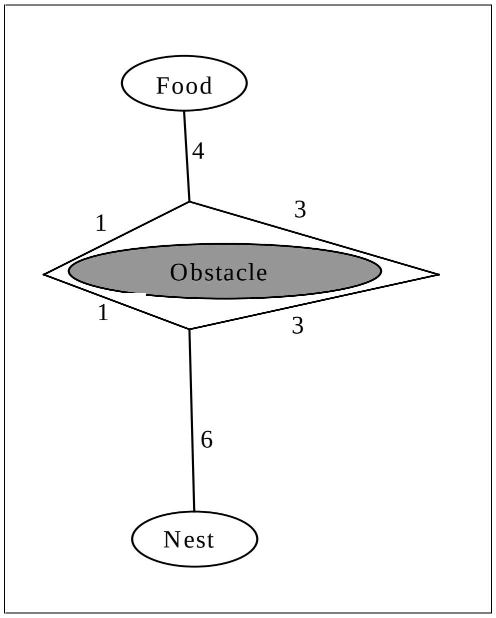

Illustration 1: Ants and Obstacles

Note that in the real world, food sources get consumed and eventually exhausted. Once a source is exhausted, ants stop going there because the pheromones on the trail evaporate. This is an important aspect of the behavior of ants.

The observation that ants find shortest paths led Dorigo, Maniezzo and Colorni to develop an optimization algorithm based on ant behavior. As it turns out, there is a class of optimization problems ideally suited for ant colony techniques (Colorni 91).

The Traveling Salesman Problem

The Traveling Salesman Problem (TSP) is a very difficult NP-Hard problem and many different techniques have been developed to solve it. The TSP is defined as follows: "Given a set of *N* cities and the distance between each pair of cities, find the shortest trip that includes each city exactly once and ends with the city of origin." A number from 1 to *N* can represent each city. A matrix *d* is created such that d_{ij} gives the distance from city *i* to city *j*. A trip can then be shown as a vector of *N* numbers, and the distance traveled is simply the sum of the distances between the cities in the vector.

There are two flavors of the TSP, symmetric and asymmetric. Symmetric TSP uses the same cost (distance) for both directions of travel between any two cities. It is represented using undirected graphs. Asymmetric TSP allows the cost of travelling from Austin to Buda to differ from the cost of travelling from Buda to Austin. It is represented using directed graphs. Asymmetric TSP is a more difficult problem.

Evolution of Ant-Based Algorithms

Since its inception, Ant Colony Optimization has been successfully, and repeatedly, applied to TSP. It has also been applied to other combinatorial optimization problems like Job Shop Scheduling and the Quadratic Assignment Problem.

The behavior of ants first appeared in optimization algorithms in 1991 (Colorni 91). One of the authors of this paper, Marco Dorigo, is most often associated with this concept as it was the main contribution of his PhD thesis

(Dorigo 92). Dorigo's thesis covers the first functional ant based system, Ant Colony Optimization (ACO). ACO has been refined over the years, improving performance and becoming applicable to a wider variety of problems.

- ➔Ant Colony Optimization
  - ➔ Ant System
    - ➔Ant Colony System
    - ➔MAX-MIN Ant System
    - ➔Continuous Ant Colony Optimization

Figure 1: Historical Progression of Ant-Based Algorithms

The next few sections discuss the progression of Ant based algorithms. The defining characteristics of the original algorithm (ACO) are captured in all the other algorithms, so it is not discussed.

The Ant System Algorithm

This section describes the Ant System (AS) and how it applied to the Traveling Salesman Problem. This section paraphrases a number of papers (Dorigo 97b)(Dorigo 96)(Wodrich 97).

The Ant System is an improvement on the original ACO. Some of the features of the algorithm are tailored to the TSP, but the same approach can be applied to other combinatorial problems.

The algorithm is used to solve TSP as follows. Given a set of N towns, the matrix d is created to contain the distance between towns. There are a constant number of m ants dispersed among the cities, so that at time t there are $b_i(t)$ ants in town i. Initially ants may be dispersed randomly among the cities, or all in the same city. Assume that random dispersion is used. Ants move to another city at the start of each time unit. They select which city based on the level of the pheromone on the edges of the graph and the distance between their current city and its neighbors. Initially, the level of pheromone on the trial is set to the same small value for all edges so all ants will select cities with equal probability.

Each ant acts according to these rules:

1. An ant chooses a next step as a function of the distance and the amount of pheromone on the connecting edge.
2. Ants always make legal round-trips by storing a list of the cities that the ant has already visited.

3. After completing a trip of the cities, pheromone is deposited on the routes traveled and then the list is cleared. This is done for each ant.

If $\tau_{ij}(t)$ is the amount of pheromone trail at time *t* on the route linking cities *i* and *j*, then the equation for updating the trail is given by :

$$\tau_{ij}(t+n) = \rho \cdot \tau_{ij}(t) + \Delta\tau_{ij}$$

where $\rho \in [0,1]$ is a constant governing the rate of pheromone evaporation, and

$$\Delta\tau_{ij} = \sum_{k=1}^{m} \Delta\tau_{ij}^{k}$$

where m is the number of ants, and

$$\Delta\tau_{ij}^{k} = \begin{cases} \dfrac{Q}{L_k} & \text{if ant } k \text{ uses the route from city } i \text{ to } j \text{ in its tour} \\ 0 & \text{otherwise} \end{cases}$$

where *Q* is a constant, and L_k is the tour length of the *k*th ant.

The amount of pheromone added to the trail is inversely proportional to the distance traveled. This makes shorter routes more attractive. So, the ants quickly start following similar routes. Like real ants, they all eventually converge towards the shortest route found.

In practice, ants do not actually converge on the shortest route all that quickly. However, it is possible to give the ants a heuristic to accelerate the process. At any given junction, ants should favor towns that are closer. If the ants use the distance between towns in their decisions they find shorter routes sooner.

The probability of an ant selecting a valid town is given by:

$$p_{ij}(t) = \frac{\left[\tau_{ij}(t)\right]^{\alpha}\left[1/d_{ij}\right]^{\beta}}{\sum\limits_{k \notin \text{Tabu List}} \left[\tau_{ik}(t)\right]^{\alpha}\left[1/d_{ik}\right]^{\beta}}$$

The use of this formula allows the ants to use all of the information available to them. This tends to make the ants "greedy." It makes for a more efficient search.

As shown in Figure 2, the AS algorithm is simple even when using the heuristic.

1. Initialize pheromone trail on all routes.
2. Place ants at random locations and clear the lists of cities in each ant.
3. For t = 1 to N, select which city each ant moves to next and add it to the ant's list of cities.
4. Compute the length of ant's tour and save the shortest tour found so far.
5. Evaporate some of the pheromone.
6. For each ant, add pheromone trail to the routes used by the ant.
7. Repeat steps 2 through 6 until a maximum number of iterations have passed or the ants stop finding shorter trips.

Figure 2: The Ant System Algorithm

There are slightly more complex variations that alter the way pheromone trail is added; but the basics are just that, basic. This makes the algorithm very accessible.

The Ant Colony System

The Ant Colony System (ACS) is a refinement on the Ant System (Dorigo 97a). This algorithm has three differences when compared to AS. In ACS, only a single ant updates the pheromone level at the end of the process, rather than all ants. The function used to determine the probabilities that an ant chooses a known good city versus taking a random choice is tunable, and finally, as ants walk they locally update the pheromone level on trials.

These differences allow the user to tune the algorithm and behavior of the ants, which gave it some measurable improvements over AS.

Dorigo and his colleagues showed that ACS outperforms SA and EP for the TSP, in both its symmetric and asymmetric forms. It performs fairly well when compared to specialized TSP algorithms, like opt-3, but it is not the best performer to date.

MAX-MIN Ant System

The MAX-MIN Ant System was developed by Stutzle and Hoos (Stutzle 97) as an improvement to the Ant System. It differs from the Ant System in two key aspects. First, when an ant completes trip, only the best ant deposits pheromone on its path, rather than all of the ants. In other words, only the ant that took the shortest trip for this round deposits pheromone. This is similar to the approach in ACS. Second, the system also allows the definition of a maximum and a minimum amount of pheromone allowed per trail, hence the name. These variances allow the MMAS to provide an extra parameter that helps avoid premature convergence on good trips in large TSP problems. The authors were able to tackle larger problems than the AS could.

The MMAS and the Ant System algorithm solve combinatorial problems. They cannot be directly applied to problems where there are both discrete and continuous variables. However, a more recent algorithm allows searches in continuous spaces.

Continuous Ant Colony Optimization

Mark Wodrich and George Bilchev have developed an algorithm suitable for problems containing continuous input and/or output variables (Wodrich 97). This algorithm is interesting because it combines some notions from GA’s with notions from the Ant System in order to solve continuous optimization problems. It solves them fairly well, and we cover this in some detail.

The Continuous Ant Colony Optimization (CACO) algorithm uses the ant metaphor, but also borrows from GA’s. It divides the set of ants into two classes. One type of ant is used to perform a global search for promising regions of the search space. The global search employs concepts from GA’s and is performed by *global ants*. The other ants are used to perform local search within the most promising regions. These *local ants* follow an algorithm derived from the Ant System. The two sets of ants combine to do true global optimization.

This section gives an overview of the system. For more details, see Wodrich and Bilchev (Wodrich 97).

Global Search in CACO

A subset of the ants (about 80%) handle the Global Search. There are two aspects to global search. First, the ants must be able to search a finite space. Second, they need to avoid local extrema. CACO has an interesting approach to handling both of these issues.

In the Ant System, the ants select destinations from a finite set of possibilities. The probability of selecting a destination is based on pheromone level. To map the continuous space to a finite space, the CACO algorithm divides the search space into a finite set of regions. Each region acts as a destination for the local ants to explore and serves as a trial solution.

Regions' positions are typically represented as a vector of real numbers. Initially, the regions are distributed randomly in the search space. They evolve over time due to the actions of the ants so that they slowly move towards areas of high fitness.

To avoid local extrema the global ants have to be able to search wide areas around the regions. The algorithm uses aspects of Genetic Algorithms to allow regions to exchange information. The end result is that ants can move regions large distances by employing a crossover-like function. This is also like the mechanism used in Population Based Incremental Learning (PBIL) algorithm, which has been shown to give good global search characteristics (Baluja 94).

Note that the global technique has no metaphoric link to ant colony behavior. This link is provided in the local search technique.

Local Search in CACO

The local ants are responsible for local search and comprise about 20% of the population of the ants. These ants provide the metaphoric link to ant colonies.

Global ants recruit local ants to search promising regions by using pheromones. This allows collaboration between the two groups. Pheromones focus attention on promising regions of the search space.

The basic outline of the search process is as follows:

1. An ant selects a region with a probability proportional to the pheromone value of that region. This is as if the ant went from a virtual nest to the region.
2. After arrival the ant moves a short distance and calculates the fitness at this point.
3. The region maintains the direction of the last ant. If the last ant improved the fitness, the new ant goes the same way. If not, it goes a random direction.
4. If the ant finds a higher fitness value, the region is moved and the ant deposits pheromone in proportion to the improvement made in the fitness. The region has a counter that is decremented.
5. If the ant does not find a better value, the counter is incremented. This will cause the next step in that direction to be smaller.

The algorithm provides parameters that control a function that controls the size of the step made by an ant in a local search. This function makes the step size depend on the region's counter. The step size decreases as the counter grows. This enables ants to refine the area of the local search and improves convergence if a higher fitness is repeatedly found in the same direction.

Bilchev and Wodrich point out that one of the weaknesses of this approach is that ants can mistakenly search a region more than once. It would be very expensive in large spaces to maintain a history of regions. In other words, since local search moves regions, regions can move back to a previously exhausted local maximum.

Constrained Optimization in CACO

As described, the CACO algorithm does not handle constrained optimization problems. However, Bilchev and Wodrich show that with a few modifications, it can be adapted to effectively handle constraints.

The primary adaptation is the incorporation of the concept of constraint violation. Given a set of constraints, every trial solution can be tested to determine whether it lies within the feasible region. A *constraint violation* is a measure of how far outside legal a solution lies. The violation is expressed as the sum of the violations of each given constraint.

During local search, ants search for local improvements in fitness. The improvement in fitness acts like a food source that is exploited by the ants. With the constraint handling mechanism, a point is only accepted as a "food source" if its constraint violation is below an acceptable threshold. Like temperature in SA, the acceptable constraint violation is changed over time, decreasing linearly from an initial value to the desired final constraint tolerance. This causes ants to be move back into feasible regions as the search progresses. They effectively pay more and more attention to the constraints over time. The increasing enforcement of constraints allows limited exploration within non-feasible regions at the start of the algorithm. This helps avoid premature convergence.

In the CACO algorithm was shown to perform quite well when compared to other published algorithms over a fairly wide set of test problems (Wodrich 97). It also looks to be reasonably easy implement and to incorporate into an application solution.

ANOTHER GLOBAL OPTIMIZATION ALGORITHM OF INTEREST

From the perspective Software Engineering, there is another optimization technique worth mentioning. This section very briefly introduces GRG2, which is a well-known, oft-used, and relatively fast algorithm that can be embedded into many software systems.

GRG2

GRG2 is an algorithm that solves nonlinear optimization problems using Generalized Reduced Gradient methods (Lasdon 78). Leon Lasdon and others developed it over the last couple of decades. GRG2 can be found in many software systems, including Microsoft Excel and Borland's Quattro Pro. It is very fast, and should be in a Software Engineer's toolbox.

GRG2 can solve problems that seek to minimize or maximize some function, $g_p(X)$, subject to $glb_i \leq g_i(X) \leq gub_i$ where $i=1,...,m$, $i \neq p$ and $xlb_j \leq x_j \leq xub_j$ where for $j=1,...,n$.

X is a vector on n variables, $x_1 ,...,x_n$, and the functions $g_1 ,...,g_m$ all depend on X. Any of these functions may be nonlinear. Any of the bounds may be infinite and any of the constraints may be absent. If there are no constraints, the problem is solved as an unconstrained optimization problem. GRG2 uses first partial derivatives of each function g_i with respect to each variable x_j. These are generally computed by finite difference approximation.

GRG2 works with a set of input functions. These functions are used in some calculation that finds some output value. It can optionally be given a list of upper and lower bounds on the values of the input functions. Given this set of inputs and the optional list of bounds, GRG2 can find maximum or minimum output values. It finds the set of values for all of the input functions that produces a minimum or maximum final value. This set of values will never contain any values outside the bounds given in the optional list of bounds.

The algorithm operates in two phases. Phase I finds legal solutions and Phase II finds optimal solutions.

Phase one is structured as an optimization problem in its own right. It is only run if the initial values of the variables do not satisfy all of the g_i constraints. The Phase I objective function is the sum of the constraint violations. It can also include a fraction of the true objective. The Phase I optimization terminates either with a message that the problem is infeasible, or returns with a legal and feasible solution.

Phase II begins with a legal solution and attempts to optimize the true objective function. At the conclusion of Phase II, an optimal value has been found.

Strengths and Weaknesses of GRG2

GRG2 can solve problems involving up to 200 variables, which makes it applicable to many problem domains. However, it can get trapped in local extrema, requiring a restart with slightly different inputs. The algorithm cannot tell that it is stuck. Most implementations of GRG2 monitor the algorithm for a timeout period and then restart the algorithm with slightly different inputs when the timeout expires.

The GRG2 algorithm is relatively fast and is fairly simple to integrate into software products. It is therefore widely used.

SUMMARY OF LITERATURE REVIEW

We looked at a number of optimization techniques and introduced some new terminology. Almost all of the techniques discussed are all classified as Metaphoric Optimization Algorithms, with most being members of a subclass called Natural Metaphoric Optimization Algorithms. We also looked at one very commonly used algorithm called GRG2. All of these techniques are interesting because they are easily accessible and understandable.

It is important for Software Engineers to be aware of these techniques. They are fairly easy to incorporate into software systems with a need for optimization.

An Introduction to MasterMind and Possible Solutions

INTRODUCTION

This section introduces the game of MasterMind, defines two variants of the game, and finally frames the approaches that can be used to solve each of the variants. These variants of MasterMind serve as test problems. Their solutions involve distinctly different attacks.

We formulate solutions for these variants using GBGA approaches and ACO approaches. In order to try some hands-on experimentation, we implement one of the solutions using a GBGA. The next major section covers the experimental results.

MASTERMIND AND ITS VARIANTS

MasterMind is a relatively straightforward two-player guessing game. Assume we have players A and B. Player A selects a fixed length sequence of colors from a set of available colors. Player A's initial sequence is called the *target* sequence. It is saved and hidden from Player B who then repeatedly tries to guess the target sequence. Player A responds to each guess with a set of chips indicating how close the guess is to the target. When Player B has finally guessed the target, B receives a score based on how many guesses were taken. Obviously, low scores are better than high scores.

Let's look at the response to guesses in a little more detail. After each guess, Player A responds with a sequence of Black and White chips. Black chips state how many places in the guess are exactly correct. White chips indicate how many colors are correct but not in the right position.

The response defines constraints on the next guess, so Player B uses this information to make an informed decision. Based on the response from a guess and the history of guesses made previously, Player B knows that certain guesses are not reasonable. Furthermore, Player B knows that some reasonable guesses are better than others; they should lead to more information in the next response. The ultimate goal is to find to next best guess that meets the constraints imposed by the response. The problem boils down to a constraint-based search for the next best guess.

MasterMind is typically played with a sequence of four items and six possible colors. This is the 4-6 version, with the first cardinal number representing the number of items in a guess and the second number representing the number of possible colors. We selected two variants, 3-3 and 4-6, as our test problems and provide two different approaches for solving these problems.

SOLVING 3-3 MASTERMIND

We use a genetic programming approach to solve a 3-3 version of MasterMind. This section introduces the programs that we use to play 3-3 MasterMind. These programs are captured in a static table that can play all possible games of MasterMind.

The Mini MasterMind Language

As discussed, the response to a guess specifies constraints on the next guess, so the game boils down a constraint-based search for a next guess. One way to implement a constraint based search is to prune all of the solutions in a search space that do not meet the given constraints and then choose from the remaining solutions. This is the “guess and prune” approach.

In terms of MasterMind, a player simply makes a guess, prunes the set of possible remaining guesses according to the response from the previous guess, and then makes another guess.

To use genetic programming, we have to first define the notion of a program. The Mini-MasterMind Language (MML) captures this notion of “guess and prune” in a string representing a program. MML can thus be used to define programs that play the game.

Since this game is about making a guess and eliminating any unreasonable next guesses, the MML language simply defines a table of guesses, in a fixed order, and pruning operators for each possible response to a guess. The MML interpreter maintains a list of unused guesses, and after each response, removes any guesses from the unused list that do not make sense based on the response. It then moves to the next guess in the list and offers it as the new current guess.

An Implicit Strategy for MML

All of the MML programs follow a strategy. They always try to maximize the number of black pegs in a response in order to pin down the colors in the target. This strategy to “maximize the black pegs” is followed by minimizing the number of different colors in a guess.

To do this, the first row of a MML table always represents a guess of RRR. This has the effect of identifying the number of reds in the target. By guessing RRR a player is guaranteed to get a response containing only black chips, one per red item in the target. If we know how many reds are in the target, then we should certainly never make follow-on guesses that do not have exactly that number of reds. This allows us to eliminate a large percentage of the guesses after we get our first response.

How can we guarantee that we only guess the correct number of reds from this point on? We use pruning operators to remove guesses from a list of unused guesses. Applying the correct pruning operators for a given response will remove all of the guesses with the wrong number of reds from our set of unused guesses. Now we just make any guess from the remaining set.

It should be noted that this strategy works for variants of the game with a small difference in the number of positions and the number of colors. When there are significantly more colors than positions, one should probably try to maximize the number of different colors in a given guess. This could be called the "maximize white pegs" strategy.

Since MML programs only play 3-3 MasterMind, the "maximize black pegs" strategy serves as a common heuristic for all MML programs.

MML in More Detail

Each possible MML program represents a 'maximize black pegs solution' to 3-3 MasterMind. These programs are very similar to finite state machines (FSM).

Consider a specific MML program that represents a specific FSM. In this FSM, the most recent guess represents the *state* and the response to that guess represents an *event*. Events cause actions that prune the search space to eliminate invalid next guesses. The state machine is, in essence, dynamically updated so that only ‘valid’ next guesses are achievable. After pruning, the current state is updated to the next guess.

For the 3-3 variant, MasterMind programs are represented as a table that has one row per possible guess and one column per possible response. These tables therefore have 27 rows and eight columns.

Each cell in the table contains a list of zero to three pruning operations. Pruning operations are used to remove particular types of guesses from the list of unused guesses. For example, the operator “Red 1” (R1) would remove all guesses containing exactly one red chip. Since we have three positions, we want operators that can prune guess with 0, 1, or 2 chips of a given color. In the case of red, we have operators R0, R1, and R2. This implies that we have 9 operators in total, 3 per color. This is not sufficient. It may be the case that we do not want to prune anything for a given response to a guess. What should we prune if we guess RYG and get one black chip as response? For these cases, we need a no-op. This leaves us with 10 pruning operations.

Table 1: A Representation of a 3-3 MasterMind Program

	BWW	BB	BW	WW	B	W	WWW	None
RRR	{op}	{op}	{op}	{op}	{op}	{op}	{op}	{op}
RRG	{op}	{op}	{op}	{op}	{op}	{op}	{op}	{op}
RRY	{op}	{op}	{op}	{op}	{op}	{op}	{op}	{op}
RGR	{op}	{op}	{op}	{op}	{op}	{op}	{op}	{op}
RGG	{op}	{op}	{op}	{op}	{op}	{op}	{op}	{op}
RGY	{op}	{op}	{op}	{op}	{op}	{op}	{op}	{op}
RYR	{op}	{op}	{op}	{op}	{op}	{op}	{op}	{op}
RYG	{op}	{op}	{op}	{op}	{op}	{op}	{op}	{op}

RYY	{op}	{op}	{op}	{op}	{op}	{op}	{op}	{op}
GRR	{op}	{op}	{op}	{op}	{op}	{op}	{op}	{op}
GRG	{op}	{op}	{op}	{op}	{op}	{op}	{op}	{op}
GGR	And so on until YYY							

The first column represents the set of possible guesses (not all guesses are shown.) The row headers represent the set of responses. The cell values, represented by {op}, are any non-repeating combination of the pruning operators, including no-ops. Applying a pruning operator removes some set of matching guesses from the set of available guesses. Note that 'BBB' is not shown since this response indicates a win.

The next figure shows an example of a very small portion of an MML program. Table 2 shows the first row of table, complete with a few pruning operators.

Table 2: An Example Row with Pruning Operators

	BWW	**BB**	**BW**	**WW**	**B**	**W**	**WWW**	**None**
RRR		R1 R0			R2 R0			R0

The row in Table 2 specifies the actions to take for a given response to a guess of RRR. If the response is BB then we know that there are two reds in the target and we can apply pruning operators that throw away all guesses that have either exactly 1 red and all guesses that have exactly 0 reds. It should be clear that by providing pruning operators for all 27 possible guesses and their responses, we build programs that play MasterMind.

A near-optimal solution for 3-3 MasterMind should exist as one of these programs and the problem can now be restated: *Among the set of all MML programs, find the program that is the best, on average, at playing the game.*

Estimating the 3-3 MasterMind Search Space

This search space is huge. There are 27 times 8 cells in the table (216). There are up to 3 of the 10 operators in each cell so each cell can be thought of as holding a 3-digit, base 10, number. The entire table is therefore a string of 216*3 digits (648). It therefore appears that there are 10^{648} possible solutions.

The space is actually quite a bit smaller than that. Note that the cells of the table contain *sets* of operators. Therefore, no operator should be repeated and the order of operators within each cell does not matter. These factors significantly reduce the number of possible combinations per cell. This in turn reduces the search space.

Since we want to disallow combinations of digits within each group of 3, we could assume that we can use (10 choose 3) as the number of possible combinations per cell. However, we want to allow the no-op to be repeated. Therefore, for our analysis, we define a total of 3 no-ops to go along with the 9 pruning operators. This effectively allows the no-ops to repeat. We have a total of 12 operators and use (12 choose 3) as the number of items per cell. There are 440 possible combinations per cell.

This leaves 216 cells with 440 possible combinations per cell. This is a total of 440^{216} combinations, which is a search space of roughly 10^{218} combinations. This is still a huge search space, but certainly not as bad as before.

We can gain even more ground by noting that there is noise in this representation. Some of the responses for a given guess are not possible. For example, the guess ‘RRR’ will never result in a response containing white chips. Table 3 identifies the impossible responses.

Table 3: Impossible Responses in an MML Program

	BWW	BB	BW	WW	B	W	WWW	None
RRR	■		■	■		■	■	
RRG							■	
RRY							■	
RGR							■	
RGG							■	
RGY								
RYR							■	
RYG								
RYY							■	
GRR							■	
GRG							■	
GRY								
GGR							■	
GGG	■		■	■		■	■	
GGY	And so on until YYY							

Table 3 shows that the WWW response can only occur when all three chips in the guess are a different color. There are 3! (6) possible guesses in which all of the color are different. Subtract 6 from 27 total rows and there are 21 cases that can be removed from consideration. Furthermore, responses with white chips cannot occur when all the chips are the same color. This adds 12 more cases to the total of impossible responses. There are a total of 33 responses that cannot occur and hence should not count as part of the search. This further reduces our search space to *just* 440^{183}, or 10^{185}. This is still a huge space, and it does not appear to be reducible[4]. Given the apparent size of the space, it cannot be searched exhaustively. Therefore, we did not attempt to compute the optimal solution. *When using MML, the lower bound on guesses per game of the 3-3 variant is unknown.*

An Approach to Solving 3-3 MasterMind Using a GBGA

Since GBGA's use a formal grammar, in Backus-Naur-Form (BNF), to define a search space we defined the Mini MasterMind Language (MML) as a BNF grammar and built an interpreter to serve as the objective function.

MML programs are executed via this interpreter that was built using flex and yacc. As expected, the interpreter actually maintains a list of possible guesses, and after each response, removes any guesses from consideration that were specified by the pruning operations for that response.

Note that partial solutions can be expressed in this grammar. For example, we know that certain cells are never 'hit' so we can write the grammar to provide empty operator lists in those cells.

The ability to provide partial solutions allows the user to run preliminary experiments, learn something unexpected about the problem, and hopefully provide a partial result that helps reduce the search space.

Rather than implement an entire GBGA system, we used the Genetic Programming Kernel (GPK) which is freely available on the Internet. Helmut Hoerner developed the GPK, which is a Grammar-Based-Genetic Algorithm

[4] It is, but we will let the story unfold as the report progresses.

system that applies the general Genetic Algorithm to populations of individuals as defined by a BNF grammar representing the search space.

The code is free for educational and research use. It is implemented in C++ and is delivered as source. The code is available at the web site for The Austrian Institute of Economics, Vienna, Austria. The free code was last updated in 1996. A more recent version of this system is available as a commercial ActiveX control. Hoerner based the code on research by Andreas Geyer-Schulz (Geyer 97)(Bohm 97).

The GPK system is fairly easy to use, and seems to work well enough to satisfy our goal of learning to apply these algorithms to actual problems. There were naturally some issues. These are discussed in the Experimental Results section.

An Approach to Solving 3-3 MasterMind Using ACO

The table used by the GBGA can also be seen as a very long *number* of 183 digits in base 440. Therefore, we could use one of the ACO algorithms to search the range of possible numbers from 0 to 440^{183} for good values that play MasterMind well.

This approach allows us to reuse the MML interpreter as the objective function for the algorithm. Our objective function simply takes a number, converts it to MML, and then runs it through the interpreter; just like we do in the GBGA approach.

One option might be to use the CACO algorithm. The CACO algorithm searches promising regions of the search space for optimal solutions. It might be applied to search our space, although it will require is to define 185 variables for use.

Additionally, it is not clear that our problem space can be searched by hill-climbing. The values our variables take (from 0 to 439) do not relate to each other. More specifically, they do not define a total or even a partial order amongst themselves. This implies that the ‘direction’ of travel for a given variable is not particularly helpful. In other words, given the way the variables in our table interact, the notion of ‘direction’ is meaningless. This implies that hills cannot be defined so that we cannot possibly climb them in any reasoned fashion. Therefore, we chose to propose a solution using the MAX-MIN Ant System.

The MAX-MIN Ant System was designed to solve Traveling Salesman Problems (TSP) and other graph-based problems. It is fairly easy to restate our problem as a graph that can be traversed with a goal of finding the best path through the graph.

Naturally, there is more than one way to graph this problem. In any method the graph will be huge.

As a small example of a graph that can be used, consider a similar problem in which we find 2 digits, each with 3 possible values. This is significantly easier to represent than 183 digits with 440 values each, but it is essentially the same problem. Figure 3 shows the problem of finding 2 digits with 3 possible values as a TSP problem.

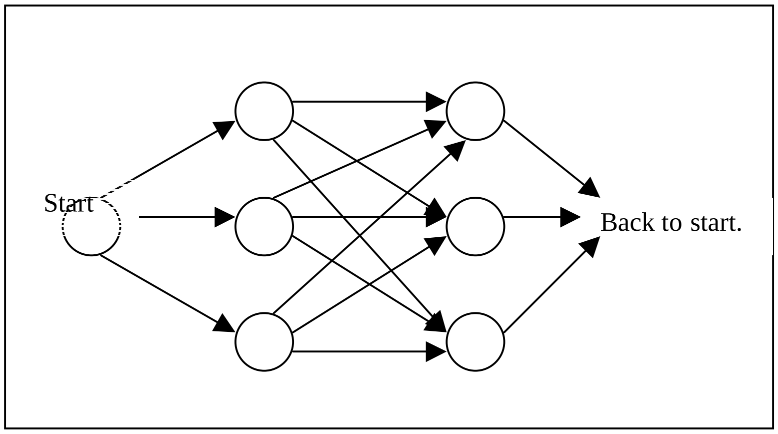

Figure 3: Finding 2 Digits with 3 Values Using TSP

Each column of nodes represents a digit; each node in the column represents a specific value of that digit. All distances in the graph can be set to the same value. The objective is to find the shortest route from the start to the start, by traversing the graph. This will result in an ordering of nodes, and hence a number that can be used to generate an MML table.

This representation would be very large for the full MML case, but would allow the MMAS algorithm to yield results. Note that this representation does not take advantage of the distance heuristic used in MMAS, so it may converge quite slowly. However, this representation is quite flexible. This makes it possible to use partial solutions as a starting point.

In fact, partial solutions can be presented in two manners. First, if certain subsequences are known to be optimal, this knowledge can be represented in the graph by removing invalid digits within the column representing those sequences. In other words, edges that are known to connect sub-optimal sequences can be removed. Second, one could add and then alter distances for the edges to give the ants a preference for certain routes.

The ability to use partial solutions as a starting point should allow a more rapid discovery of good programs.

SOLVING 4-6 MASTERMIND

The MML-based solutions try to solve all possible games of 3-3 MasterMind with one search. This approach will not work for 4-6 MasterMind because the search space is very large. Each increase in the number of colors or the number of chips increases the search space in three dimensions. The search space correlates directly to the table size. Each increase in colors or chips results in an increased in number of rows, number of columns, and number of operators per cell in the table. Therefore, the "guess and prune" search space for 4-6 MasterMind is completely intractable.

We can dramatically reduce our search space if we run our search algorithm once for each guess in each game. This new approach solves one guess at a time, and illustrates some other properties of the search algorithm, e.g. its performance under constraints and the ease of incorporating a decision heuristic into the game.

The Dynamic Constraints Approach

The 4-6 solution plays "guess by guess." The basic algorithm is simple. At every guess search for a best next guess based on *constraints* defined by all of the responses to previous guesses. This is in line with the experiments by Merelo in genMM (Merelo 96). This approach dramatically reduces the size of the search space for each execution of the optimization algorithm.

How can we define the constraints based on a response to a guess? Black chips identify the exact number of chips in the guess that are correct. If b represents the number of black chips in the response, then we must assure that all ‘next-guesses’ have at least b chips in the exact same position. Furthermore, if w represents the number of white chips in the response then we must ensure that the next guess has w colors that are the same. These constraints on the next guess are composed, with black chips taking precedence. Additionally, we should never make the same guess twice. So, a *matching rule* is defined as a guess and its response. We need to maintain a list of these as we progress. We also maintain a list of prior guesses. By using the information in these two lists, we can search for a next best guess amongst the set of possible guesses.

During each search for a next guess, we want to consider all matching rules. Our fitness function will calculate how close a potential guess is to each rule, and return the summation of these calculations.

Our solution differs from Merelo’s in one key aspect. We try to follow a similar strategy to that used in our MML solution. First, we define an order in which we will try to identify colors. For example, if the colors in 4-6 MasterMind are Red, Orange, Yellow, Green, Indigo, and Purple, then we can try to pin them down in that order. We start with RRRR guess and then when selecting a next guess from the set of guesses that match as many of our rules as possible, we favor the guess with the most of the next color in our order, Orange. This drives us towards maximizing the number of black chips in the responses. In theory, the addition of this strategy will garner better results.

Estimating the 4-6 MasterMind Search Space

The search space is small. There are 4 chips of 6 colors, or a base 6 number with 4 places. This gives 6^4 combinations. There are 1296 possible combinations, so the space has a magnitude of only 10^3.

Genetic Algorithms find a near-optimal solution, so we do not expect this algorithm to obtain the theoretical optimum. For the 4-6 game, this has been posited to be either an average of 4.34 guess per game if 6 guesses are allowed (Koyama 94) or to have a slightly higher average taking a maximum of 5 guesses (Knuth 76-77). Koyama's solution uses an exhaustive search.

Why don't we use an exhaustive search? This is a solved problem so we do not expect to learn anything new from an exhaustive search. Since our goal is to learn about optimization algorithms and different approaches to similar problems we use the techniques under investigation. We then compare any experimental results to the known optimum for the 4-6 variant in order to judge the effectiveness of our solution.

Solving 4-6 MasterMind with a GBGA

The implementation using a GBGA is fairly straightforward.

First, we need a grammar that defines a current guess. This is trivial since the population consists of 4 digit strings of base 6.

Merelo's solution relies heavily on a crossover technique called *transposition.* Transposition boils down to the ability to swap digits in the string.

Therefore, the generic crossover function for the GBGA appears to be ideal for this application.

Next, we need a fitness function that provides a range of fitness given the prior guesses and the responses to the prior guesses. For a given matching rule, we count the number of identical matches to the prior guess and the number of colors that are same and then, favoring blacks, return a value that is 10 * the identicals + the same color values. We do this for each matching rule and overall fitness is simply the sum of these values.

The difficult item is to incorporate the color-ordering heuristic. Basically, we add one additional point to fitness for each chip in a matching rule that is the same color as our current focus. Focus progresses as we try guesses of all one color, so it is initially R, then follows a progression. We move to the next color when we cannot find any black pegs in a response to guess containing the current focus color.

We also throw out any next guess that is in the list of all prior guesses.

This composition of evaluation techniques should perform reasonably. We do not expect it to be optimal, since the GBGA does not generally find the optimal answer, but rather a close approximation.

Solving 4-6 MasterMind with an ACO Algorithm

It is not clear how this can be tackled with an ACO algorithm. Minimally, an approach would require a graph representing the possible guesses. This would be like the graph seen in the MML-Based ACO approach.

It is conceivable that the distances on the edges in this graph could be manipulated from guess to guess to incorporate the various matching rules that we have discovered so far. We could then essentially add pheromones from one ant based on results of the same fitness function as the GBGA solution.

It is not clear exactly how to do this, which may outline a weakness in the idea of using ACO for this problem.

Experimental Results

INTRODUCTION

This section presents the results of the experiments with the two variants of MasterMind. We discuss the 3-3 results and then the 4-6 results. For each variant and each optimization technique, we generally follow a progression of topics including experimental setup, intermediate results, tuning the implementation, and finally an analysis of the final results. When available we make a comparison against other research's published results.

By 'tuning the implementation' we mean that these discussions also report on any issues encountered during experimentation and any changes from the original framework of the solution that were required. These changes were made in order to get the best solution possible within the project's limited timeframe. In other words, this section will read somewhat like a laboratory notebook capturing the results and observations made during the experiments.

When discussing performance, we will normally talk about the number of executions of the objective function that were required, rather than timing data. However, when appropriate, we will discuss run lengths for the various experiments. For reference, the experiments were performed on a Dell Dimension 266MHz Pentium II machine with 128MB of RAM and a 512K L2 cache.

3-3 MasterMind Results

This section covers the results of the experiments with a GBGA and the AS algorithm for the 3-3 variant of MasterMind. We begin, as usual, with the GBGA results and follow with the AS results.

3-3 GBGA Results

As mentioned earlier, this solution was implemented using the Genetic Programming Kernel. The GPK system is a bit unwieldy to use since it requires the recompilation of the entire code set into one executable in order to run the GA. This required many cycles of build, run, rebuild, run, and so forth. Every change to the GA parameters (like elitist vs. non-elitist strategies or population size) required rebuilding the executable.

The GPK system was designed to use an input grammar to define a search space and to naturally use an interpreter as the objective function. Despite this, we found that integrating the MML interpreter was quite difficult. We used common tools (flex and yacc) for development and expected a cleaner integration between the GPK and the interpreter. The objective function, a parser, had to run many times per execution. This required us to enable the interpreter to restart many times within a single execution. Flex and yacc are not normally used in environments like this, so knowing this in advance would have been helpful. Thankfully, a ready-made solution was available (Levin 92).

Integrating the interpreter was also complicated by the fact that GA and interpreter live in the same executable. We needed to repeatedly rebuild the entire GA executable for each bug fix in the interpreter.

The GPK does not use the yacc grammar as input for the BNF. Initially, this seemed like a flaw since we had to specify the grammar twice. However, since the grammar is specified in a separate file from the GA, it can be manipulated to only search portions of the search space without having to change the interpreter. For example, the GPK input grammar can be written in such a way that the cells in the table that cannot exist are left empty. It can also be written to fix the values at a certain row or column in our table. Specifying the grammar twice turned out to have great advantages.

The GPK system works, but it could make it easier to set up problems by making it clear how to use common tools to develop interpreters.

Objective Function

We measured fitness by passing the phenotype for each individual to the MML interpreter for every possible target sequence. In other words, the interpreter was used to play all possible games. The sum of all of the resulting number of guesses was returned as a measure of fitness. Lower totals of guesses were rated as more fit than larger totals.

In order to give a measure of fitness, the total number of guesses for all games was subtracted from the upper bound of 729. It would normally be 756 (27 games times 28 for a very bad program). Given our strategy, we know that at least one guess (RRR) always results in a score of 1. This leaves 756 – 27 as the upper bound.

The interpreter is used for our objective function, so it needed to execute as quickly as possible. It maintains a list of guesses and an array specifying if a given operator prunes a particular guess. This makes pruning very easy and very fast since it is simply a lookup on the table[5].

Intermediate Results

We ran the GA many times, varying the input parameters based on observations of each run. It took a lot of searching to find a ‘good’ solution. The parameters fed into the GA can make a big difference on the results.

While there are no solid heuristics for determining what parameters to use, early runs appeared to get stuck in local minima. The solutions converged early in the run and stopped making much progress. We therefore slowly increased the amount of mutation present in the system in order to introduce more randomness and climb out of these valleys.

Recall that the empty solution (no pruning at all) requires an average of 14 guesses.

Almost all runs started with 70 individuals, ran for 1200 generations, and had the following parameters the same:

- They used a Selection Strategy based on Linear Ranking. This just takes the top performers.
- They use Stochastic Sampling for the initial population generation.
- They use Random Permutation for the Mating Strategy.
- The mating pairs are selected using the (Selection Heuristic) that selects the mates from adjacent quartiles of the population (pow2).
- The Crossover ratio was varied between .85 and .9.

[5] Despite this, there was a bug in the first implementation, discussed later.

- The Mutation rate was dramatically varied, from 0.005 to .25.

The parameters are largely the default for the system, with the exception of generations, population size, mutation, and crossover. The default crossover ratios and mutations rates have been shown to be good starting points for many problems using a GA, but did not function well in this case. The space may just be too large.

There are two strategies that can be used when removing the least fit individuals from a population. These are called the *elitist* strategy and the *non-elitist* strategy. In the elitist strategy, all individuals less fit than a limit are discarded. The elitist strategy is very strict with regards to the fitness of the population. On the other hand, the non-elitist strategy allows some sub-par individuals to survive, usually in the hopes that their genotype may be hiding some useful genes for future populations. It allows a bit more randomness into the system.

The next two subsections discuss the best results for the non-elitist strategy and the elitist strategy. These experiments were performed before altering the mutation and crossover rates.

Non-Elitist, Default Crossover and Mutation

The non-elitist solution used all of the above parameters and followed a strategy that allows some poorly fit individuals to survive onto the next generation.

This setup arrived at a solution of 9.3 guesses per game. This is better than the brainless version, but not as good as might be expected.

The top performer was:

```
R0 Y2,Y2,Y0,Y2,Y0,G1 Y0 G1,G1 Y0 R0,R1
Y2,G1, , ,G1 G0, , ,
R0 Y2,Y2,Y0,Y2,Y0,G1 Y0 G1,G1 Y0 R0,R1
 ,Y0, ,R2 Y2,Y0 G1 R1, ,R2,R0 G0 Y1
Y1 G1 G2,G2 G1 G2,Y2 R1 Y0,R1 Y2,G1 G1,G1 Y0,Y2 Y0,Y1 Y1 G2
 ,G0 R2 Y0,Y1 G2 Y2, , ,R0 G0 R2,Y2 R0,G1 G1 G1
 ,G0 R2 Y0,Y1 G2 Y2, , ,R0 G0 R2,Y2 R0,G1 G1 G1
 ,Y0, ,R2 Y2,Y0 G1 R1, ,R2,R0 G0 Y1
G1 Y1, , ,R0 R2,G0,G0,R1 Y2,
R0 Y0, ,G0 Y2,G1 R1 R2,R1,R2,R2 Y1,G1 G0 R1
G0,Y2,R0 Y0, ,R1 Y0, ,R1,Y2
G2 Y2 Y2, ,R1,Y2 R2, ,G1, ,G1
G2,R1,R1,R2 G1,G1 R0 G0, ,Y0,R0 G1
R0,Y0, ,G0 R2, ,G0 Y1,R1,G2 G1 R2
Y0 G2,Y1 R1 Y2,Y0 Y0 Y0,Y1,Y1 R1,R0, ,Y0
 ,G2 Y0,Y0 R0,Y0,R2 R0 G2,Y2,R0,Y2 G1 Y0
R0 R0,G0,Y0,R0, ,G0 Y0 Y0,G0,Y1
 ,G2 R1,Y0 Y2,Y2 G1 R1,R1 R1 Y1,Y2 G1,R1 G1 Y0,R2 R2 R0
R1,G0 Y2, ,Y2 G0,R1,Y0 Y2 G1,R0,R0 R2 R2
 ,Y0, ,R2 Y2,Y0 G1 R1, ,R2,R0 G0 Y1
Y1 R2,R1,Y1 Y0 G2, ,Y2,R0 R0 R0,R2 R0,G1
Y1 Y1 Y0,G1 G1,Y1 Y1 R1, ,R2 R0 Y2,R1 R2, ,G0 Y1 R2
Y2,G0 Y1 R1, , ,Y1 R2,R0 G2 G0,R1,
 ,Y0, ,R2 Y2,Y0 G1 R1, ,R2,R0 G0 Y1
G1 Y1, , ,R0 R2,G0,G0,R1 Y2,
 ,G2 Y0,Y0 R0,Y0,R2 R0 G2,Y2,R0,Y2 G1 Y0
Y0 G2,Y1 R1 Y2,Y0 Y0 Y0,Y1,Y1 R1,R0, ,Y0
```

Figure 4: The Best Early Non-Elitist Performer

Elitist, Default Crossover and Mutation

The elitist strategy has the same criteria, but keeps the best individuals. It always discards unfit individuals.

The results were 10.6 guesses per game. This was not particularly great either, but still better than average.

The top performer was:

```
R1 G0 R0, ,Y1 G2,G1 G2 R1, ,R1 G1 Y0,G2 R2 G0,
R1 G1,R0,Y2,G0 Y2,G2 Y1,G1,G0 G1,Y2
Y1,G2 Y0,G1 G0,G1 G1,R2,G0 Y0,R1,Y1 G1
R2 Y2 Y1,R1, ,G0 Y0,R0,R0 G2 R1,Y0 R0,G0 R2 G1
G2 Y0,Y0 G0 R0,Y0,R1 R0,G2 Y0 G1,Y2 Y2,G2 R0 G1,G1 G0 Y0
Y2,R0 Y2,Y0 Y2 G1,Y1 Y0,G2 Y2 R2,Y2 G0,R0 G0,G1 R2 R1
R2 Y2 Y1,R1, ,G0 Y0,R0,R0 G2 R1,Y0 R0,G0 R2 G1
Y0 R0,G2 G0,G1,G2 R1,Y0,G1 G0,Y2 Y2,
R2 Y2, ,R0 G0 R2,Y2 R0 R2,R0 Y0,Y2 R0 G0,R2,Y0
```

```
R1 G0 R0, ,Y1 G2,G1 G2 R1, ,R1 G1 Y0,G2 R2 G0,
G2 Y0,Y0 G0 R0,Y0,R1 R0,G2 Y0 G1,Y2 Y2,G2 R0 G1,G1 G0 Y0
G2 Y0,Y0 G0 R0,Y0,R1 R0,G2 Y0 G1,Y2 Y2,G2 R0 G1,G1 G0 Y0
 , , , ,Y1,Y2 R0 G2,R1 R1 G1,R2 R0
R2 Y2, ,R0 G0 R2,Y2 R0 R2,R0 Y0,Y2 R0 G0,R2,Y0
R1 R1,R1, , , ,G2 R0 Y1,Y2 Y0,Y2 R2 Y1
Y1 Y1 Y0,G1 G1,Y1 Y1 R1, ,R2 R0 Y2,R1 R2, ,G0 Y1 R2
Y0,G1,G0,R0 Y0,R2 R1,G2,Y0 R1 G0,
Y0,Y1 Y0,G0,G1,G1 G2 R1, ,Y2 G1,Y0
Y1,G2 Y0,G1 G0,G1 G1,R2,G0 Y0,R1,Y1 G1
Y1,G1 G0 R2,Y2 R0,G0 Y2, ,Y2 G2 G0,Y1 R2 Y1,
R2 Y2, ,R0 G0 R2,Y2 R0 R2,R0 Y0,Y2 R0 G0,R2,Y0
R0,G0 R1,R2 R2 G0,G0 Y1 Y2,Y1 R0, ,R0 R2 G0,Y0
 ,Y1 R0,G1,Y0 G2 R1, , ,R0,G0
R2,Y0 Y2,R0,Y0,G1,R1 Y2,Y1,Y2 Y0 R2
G1 Y2 Y2,R0 G2 G1,G2 G0,R0, ,Y1 G0 G0,G2 Y0,Y0 Y0
G1 R2,R0 Y2 G1, ,G2,Y2 Y1,R2 R1,G0 R1,Y0
Y1 R1,R0,G1, ,R2 G0 Y1, , ,G0
```

Figure 5: The Best Early Elitist Performer

These early results were encouraging, but it appeared that the system was getting stuck in local minimums.

Given these results, from this point forward we use the non-elitist strategy exclusively[6].

Non-Elitist, 0.9 Crossover and 0.15 Mutation

After significantly altering the mutation rate, which introduced a great deal of randomness into the system, the best program took an average of 4.37 guesses per game.

The top performer was:

```
R2,G1 R1 R1,Y2 Y0 G2, ,R2,R2 Y1 Y0,G2 R0 Y0,G1 R0 R1
G1 G1 G0,G2 Y2 Y2, ,G2,R2,G0 R2,G2 G0,Y1
G2 R0 Y0,Y1 Y0,G0,R0 R0 G2,Y0 R1,R2 R1 R2,R2 Y1 G2,Y0
 ,R1 Y1 G2,G1,G1 R1 Y1,G2 R1 R0, ,G0,R2 G1
Y2 Y2 G1,R0 G1 R2,R0, ,G0 Y1 Y1,R1 R0 Y1, ,
R1 G0,Y0 Y1 Y0,G1,R1 G2 G0,G2,G2,Y0 Y0 Y0,R1 G0 Y1
R1 R0, ,G0 Y0 G2,R1 R2,G0,G0,R1 G2 Y0,Y2
R1 G0,Y0 Y1 Y0,G1, ,G2,G2,Y0 Y0 Y0,R1 G0 Y1
```

[6] We actually did periodically run the experiments with the elitist strategy, but it never faired as well so the results are not reported.

```
R2 R0,Y1 Y0,G0,R0 R0 G2,Y0 R1,R2 R1 R2,R2 Y1 G2,Y0
G2 R0 Y0,Y1 Y0,G0,R0 R0 G2,Y0 R1,R2 R1 R2,R2 Y1 G2,Y0
R1 G0,Y0 Y1 Y0,G1, ,G2,G2,Y0 Y0 Y0,R2 R2 Y0
R2 R0,Y1 Y0,G0,R0 R0 G2,Y0 R1,R2 R1 R2,R2 Y1 G2,Y0
G2 G2,Y0 R0, ,R1 Y2 Y0,G1 R2 G1, ,R1,
R1 G0,Y0 Y1 Y0,G1, ,G2,G2,Y0 Y0 Y0,R2 R2 Y0
R1 G0,Y0 Y1 Y0,G1, ,G2,R0 R0 G2,Y0 Y0 Y0,R1 G0 Y1
Y2,R0,Y2 R2, ,Y0 Y0 Y2,G1 R0 G2, ,G1 G2 R1
Y2 Y2 G1,R0 G1 R2,R0, ,Y1 Y0,R1 R0 Y1, ,
 ,R1 Y1 G2,G1,G1 R1 Y1,G2 R1 R0, ,G0,R2 G1
R1 G0 Y1,R0 G1 R2,R0, ,Y1 Y0,R1 R0 Y1, ,
 , ,G1,G0 Y0 G2,G2 Y0 G2, ,G0 G0 Y1,
 ,R1 Y1 G2,G1,G1 R1 Y1,G2 R1 R0, ,G0,R2 G1
R2 R0,Y1 Y0,G0,R0 R0 G2,Y0 R1,R2 R1 R2,R2 Y1 G2,Y0
R1 G0,Y0 Y1 Y0,G1,R1 G2 G0,G2,G2,Y0 Y0 Y0,R1 G0 Y1
 , ,G1, ,R2 G1, ,R2,G0 Y2
G1,R2 Y1,G2 G1, ,R2, ,R0,G1 G2
Y2,R0,Y2 R2, ,Y0 Y0 Y2,G1 R0 G2, ,G1 G2 R1
Y0 Y1,G1,R0 R0 Y1,Y2 G0,G2 R0,G1,Y2 G2,Y1 Y2
```

Figure 6: The 4.37 Result

Note that this solution used a Crossover ratio of 0.9 and an extremely high mutation rate of 0.15. We also ran for 2000 generations. Relying on mutation to this extent is atypical, but sometimes necessary with large search spaces.

It is quite interesting given the monstrous search space that the GBGA allowed the algorithm to find a decent result in finite time. We only evaluated 140,000 individuals over the course of the run. This is miniscule portion of the space.

Analysis of the 4.37 Result

This seems to be a promising result. Although, there is no known minimum, by extrapolating from Merelo's results we suspect the optimal value for 3-3 MasterMind is somewhere between 3 and 4.

There are few caveats and complicating factors.

After evaluating the outputs one more time, .it was clear that he 4.37 result was obtained reasonably early in the process, and yet nothing better was found.

This could mean that 4.37 is very close to optimal, or it could mean that the system might still be getting stuck on local minima.

We are searching a larger than necessary space. First, our solution does not use sets for the list of pruning operators because this would require look-ahead from the parser. The GPK only accepts BNF grammars, so look-ahead is out of the question. Second, we are not accounting for impossible responses. These two factors introduce noise into our search space, effectively making it larger than necessary. We could try to specify the grammar a little more precisely, by at least removing the illegal responses from the grammar.

After analyzing the solution, it became apparent that less pruning is required than originally anticipated. In fact, the 3-3 solution can be reduced to the following program.

```
, ,R1, , ,R2, , ,R1
 ,G0, , , ,R2,R2, ,Y1
 ,Y0, , , , , , ,
 , , , , , , , ,G1
 ,G1, ,R0, ,Y1,Y1, ,
 ,G0, , , , ,Y0,
 ,R0, , , , , , ,
 ,G0, , , , ,Y0,
 ,R0, , , , ,R2, ,
 , , , , , , , ,
 , , , , ,G2,G2, ,
 ,R0, , , , , , ,
 , , , , , , , ,
 , , , , ,G2, , ,
 , , , , , , , ,
 , , , , , , , ,
 ,Y2, , , , , , ,
       with the remaining lines empty
```

Figure 7: The Reduced 4.37 Result

This seems to indicate that we should be able to use one operator per cell, from a set of 10. The search space is therefore *only* 10^{183}.

Reanalyzing the 4.37 Result

When considering the use of one operator per cell and how the table worked, it was clear that at least two operators per cell should allow better tables, at least in very specific cases. For example, if the first guess is RRR and the response is BB then the target had 2 reds in it. In this case, the program should prune all guesses with exactly 1 red and all guesses with 0 reds. However, trying this did not improve our result!

As it turns out, the interpreter had an error. Sometimes additional operators in a cell would put pruned guesses back into the list! In other words, the interpreter was not evaluating the fitness properly.

The process of discovering this error led to the discovery of the optimal first row in the table. The optimal first row is given in Figure 8.

```
, ,R1 R0, , ,R2 R0, , ,R2 R1
```

Figure 8: The Optimal First Row for 3-3

All further experiments take advantage of this discovery and use the input grammar for the GPK system to specify partial solutions using an optimal first row.

After fixing the interpreter and running all of the experiments again (using an optimal first row) we found the results to be consistent with the first run through the experiments. Each solution was marginally improved.

Rerunning the experiments resulted in a program with an average of 3.93 guesses per game.

The winner (reduced) was:

```
, ,R1 R0, , ,R2 R0, , ,R2 R1
 ,Y1, , , , , , ,
 ,G1, , , , , , ,
 , , , , , , , ,
 ,Y2 G1, , , ,Y1,Y1 Y0, ,
 ,G2, , , , , ,G0,
 , , , , , , , ,
 ,G2, , , , , ,G0,
 , , , , , , , ,
 , , , , , , , ,
 , , , , , , , ,
 , , , , , , , ,
 , , , , , , , ,
 , , , , ,G2, , ,
 ,Y2, , , ,G1 Y1, , ,
       with the remaining 12 lines empty
```

Figure 9: The Reduced 3.93 Result

Analysis of the 3.93 Result

This result was found with the following parameters.

- The Crossover ratio was .85.
- The Mutation rate was 0.09.
- The algorithm ran for 2000 generations.

Figure 10 shows the progress made during the execution of the algorithm.

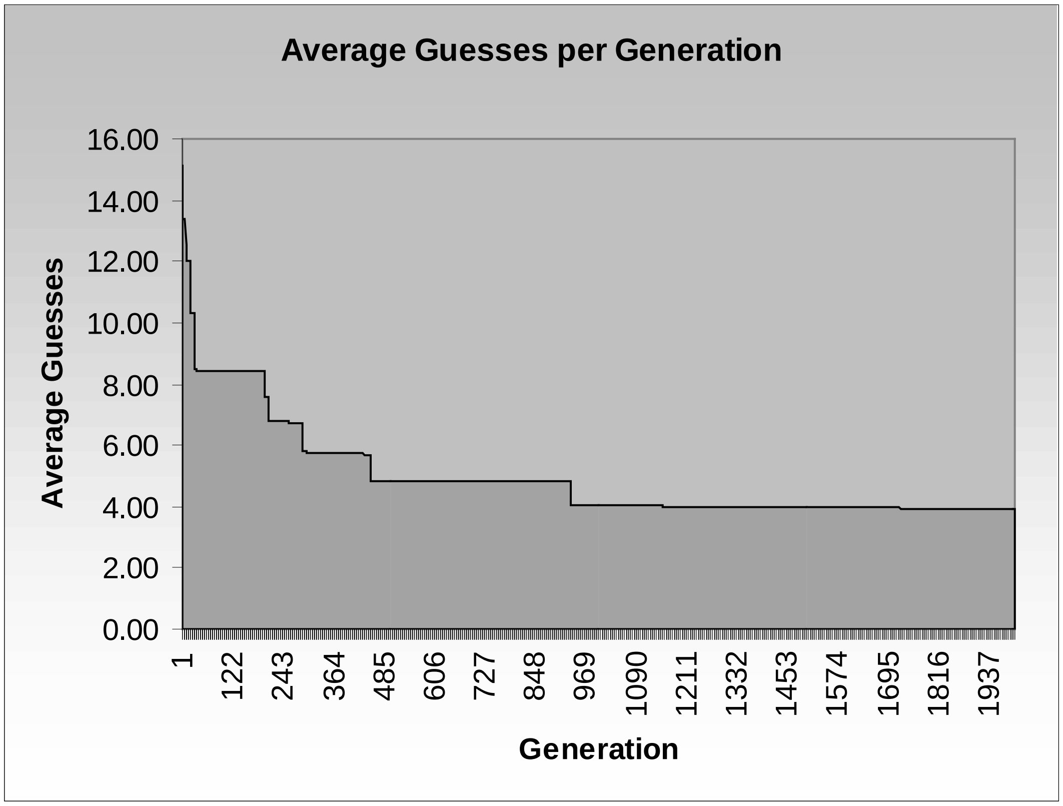

Figure 10: A Graph of Progress Towards the 3.93 Result

The final result of 3.93 guesses per game was reached after 1726 generations. It represented an improvement of .04 guesses per game. This means that is used 1 less move to play all possible games. The prior 3.96 result came after 1151 generations. Clearly, progress has bottomed out.

This reduced solution is very sparse. It has had a significant number of operators removed because they had no impact. The items they pruned were most likely already pruned, so those operators were redundant.

From these results, it seems clear that it should be worthwhile to try a using a few searches with the last 10 lines empty and only 2 operators per cell. A better table may exist in this space.

Comparison with Published Solutions

None of the published solutions we looked at did 3-3 MasterMind, but the 6 peg, 6 colors version of Merelo's used between 5.4 and 5.8 guesses per game. The Simulated Annealing version (by Bernier) averaged between 5.6 and 6.1 guesses. These results are according to the web page at http://kal-el.ugr.es/mastermind/.

Intuition tells us to expect results between 3 and 4 guesses for the optimal 3-color version. Our GBGA search got close to this result, at 3.93. Of course, intuition is not always accurate. The optimal solution is still unknown. The GBGA and the MML approach clearly work. *The question is still "How well do they work?"*

4-6 MASTERMIND RESULTS

This section captures our results using the GBGA and the 4-6 "guess by guess" solution.

GBGA Results

This is still under construction. So far, we can play, but poorly.

SUMMARY OF EXPERIMENTAL RESULTS

All of the GBGA results illustrate the idea that the power of the Genetic Algorithm comes as much from making the user think about the problem in novel and precise ways as much from the algorithm itself.

Conclusion

This section concludes the report. It discusses the main findings and limitations of this research. It also discusses opportunities and suggestions for follow-on work.

MAIN FINDINGS

Natural Metaphoric Optimization Algorithms are accessible and relatively easy to apply to combinatorial optimization problems. From a Software Engineering perspective, GBGAs and ACO algorithms are the most promising recently developed algorithms.

Given that NMOAs are accessible and perform well, it is important to monitor the research in the natural sciences in the hopes of finding other applicable metaphors. The process of evolution has had a long time to find solid optimization strategies. There are undoubtedly many more metaphors in Nature that can be leveraged by software engineers and computer scientists.

Having said that, it could be argued that the power of Genetic Algorithm based systems, which include Genetic Programming systems, comes from the thought processes they require of the user, not the algorithm itself. They allow the user to solve hard problems because they force the user to know the problem domain extremely well in order to get good results from the system. They in essence force the user to think about the problem in a different light and the search for a solution simply becomes frosting.

Naturally, this observation has both good and bad aspects.

Original Contributions

This report makes some original contributions. The MML-based approach, with its notion of dynamically updated finite state machines, is a novel solution for MasterMind. The MML experiments produced promising early results, while only searching a miniscule portion of the overall space. Additionally, this general notion and approach appear to be applicable to other Genetic Programming tasks.

The guess-to-guess approach, while simply a variation of previous research, also gave interesting results, albeit very preliminary. It represents an interesting general framework for solving problems with potentially huge search spaces. This work on the guess-to-guess approach also led to an apparently unknown application of GBGA's to constraint-based optimization problems. This application is discussed in the Opportunities for Further Work section that follows.

LIMITATIONS

This report serves more as a survey than as experimental research. Many more experiments, on a wide variety of problems need to be performed to properly assess the value of Grammar-Based Genetic Algorithms.

OPPORTUNITIES FOR FURTHER WORK

The results for both 3-3 and 4-6 MasterMind using a GBGA are quite encouraging, but leave several unanswered questions and possibilities for further research. This section looks at these opportunities, covering those that are not solution-specific or approach-specific and then those that apply to each variant of MasterMind used for this report.

Solution- and Approach-Neutral Possibilities

First and foremost, it would be interesting to implement and test an ACO algorithm for several variants of MasterMind and then compare the results with those found here.

With regards to the GBGA-based experiments, several other avenues of research should be followed.

The GBGA we use leverages the transposition operator in crossover to an extreme. It may be illustrative to try additional operators in both variants of the game.

The MML Approach

In the MML solution, one could try to optimize each row individually and then test the results of a program that combines all of the rows.

Another MML approach would use 2 phases of GA's. The first phase would approximate optimal rows and then use slight variants of these to populate the starting population of a second phase. This could be run in parallel for the first phase, and may allow the exploration of more of the search space. This 2-phase approach might also provide interesting results by allowing the combination of various optimization techniques for each phase.

The Guess-to-Guess Approach

This approach will scale to larger variants of the games, e.g. 7-7 with a space of magnitude 10^6. We may investigate these variants in follow-on research.

We might also want to compare the approach taken here against a directed search that explicitly incorporates our heuristic. It would be interesting to compare this approach for both the 3-3 and 4-6 variants to the results found here.

Dynamically Specifying Constraints Using a Grammar

Two observations about GBGA's and the approaches taken here lead to a very intriguing insight and opportunity for follow-on research. First, all Grammar-Based Genetic Algorithms share one major strength - the ability to specify and restrict the search space using a grammar that allows the generation of legal chromosomes. Second, Genetic Algorithms are not particularly good at constraint-based optimization since the constraints can be difficult to express in the fitness function. These observations lead to the following insight. The Guess-to-Guess approach to MasterMind might be significantly improved if we *dynamically capture the constraints in a new grammar for each guess.*

For example, if we guess "RRRR" and get no blacks in response, we know there are no Reds in the solution. We could generate a new grammar that reflects this and use it to drive the search for the next guess. This reduces the search space and simplifies the fitness function, which can now focus simply on the number of blacks and whites and not the colors involved. Generating new grammars based on a heuristic and observation of constraint is non-trivial, but deserves attention in follow-on work.

References

Antonisse, H.J. (1991). A Grammar-Based Genetic Algorithm. In G.J.E. Rawlins (ed.), *Foundations of Genetic Algorithms* (pp. 193-204). San Mateo, CA: Morgan Kaufmann Publishers.

Aarts, J., and Korst, E. (1989). *Simulated Annealing and Boltzmann Machines.* John Wiley & Sons.

Baluja, S. (1994). *CMU-CS-94-163 Population-Based Incremental Learning: A Method for Integrating Genetic Search Based Function; Optimization and Competitive Learning.* Pittsburgh: Carnegie Mellon University.

Bernier, J.L., et al. (1996). Solving MasterMind Using GAs and Simulated Annealing: A Case of Dynamic Constraint Optimization, *Proceedings of PPSN, Parallel Problem Solving from Nature IV*, (pp. 554-563). LNCS 1141, Springer-Verlag.

Böhm, W., and Geyer-Schulz, A. (1997). Exact Uniform Initialization for Genetic Programming. In R.K. Belew and M. Vose (eds.), *Foundations of Genetic Algorithms 4* (pp. 379-407). San Francisco: Morgan Kaufmann.

Colorni A., M. Dorigo & V. Maniezzo (1991). Distributed Optimization by Ant Colonies. In F.Varela and P.Bourgine (Eds.), *Proceedings of ECAL91 - European Conference on Artificial Life*, (,pp.134-142). Paris, France, Elsevier Publishing.

Dorigo, M., and Gambardella, L.M. (1997). Ant Colony System: A Cooperative Learning Approach to the Traveling Salesman Problem. *IEEE Transactions on Evolutionary Computation,* 1, 1, 53-66.

Dorigo, M., and Gambardella, L.M. (1997). Ant Colonies for the Travelling Salesman Problem. *BioSystems,* 43, 73-81.

Dorigo, M., Maniezzo, V., and Colorni, A. (1996). The Ant System: Optimisation by a Colony of Cooperating Agents. *IEEE Transactions on Systems, Man, and Cybernetics,* Part B, 26, 1-13.

Fogel, D. (1995). A Comparison of Evolutionary Programming and Genetic Algorithms on Selected Constrained Optimization Problems. *Simulation,* 64, 397-404.

Fogel, L. J. (1997). "What’s Evolutionary Programming (EP)?". WorldWide Web, Available at ftp://ftp.Germany.EU.net/pub/research/softcomp/EC/FAQ/supplements/what-is-ep.

Fourer, R. and Gregory, J. W. (1997). "Linear Programming FAQ". WorldWide Web, Available at http://www.mcs.anl.gov/home/otc/faq/nonlinear-programming-faq.html.

Geyer-Schulz, A. (1997). The Next 700 Programming Languages for Genetic Programming. In J.R. Koza, W. Banzhaf, et al. (eds.), *Genetic Programming* (pp. 128-136). San Francisco: Morgan Kaufmann.

Greening, R. (1995). *Simulated Annealing with Errors.* PhD Thesis. Los Angeles: University of California, Department of Computer Science.

Holldobler, B., and Wilson, E.O. (1990). *The Ants.* Berlin: Springer-Verlag.

Koyama, K., and Lai, T.W. (1994). An Optimal Mastermind Strategy. *Journal of Recreational Mathematics*.

Knuth, D. (1976-77). The computer as Master Mind. *Journal of Recreational Mathematics*, 9, 1-6.

Lasdon, L. (1978). Design and Testing of a Generalized Reduced Gradient Code for Nonlinear Programming. *ACM Transactions on Mathematical Software*, March 1978, 34-50.

Levin, J.R., Mason, T., and Brown, D. (1992). *lex & yacc.* Sebastopol, CA: O'Reilly and Associates.

Levy, S. (1992). *Artificial Life - A Report from the Frontier Where Computers Meet Biology*. New York, NY: Random House, Inc.

Merelo, J.J. (1996). *GeNeura Technical Report G-96-1 Genetic Mastermind, a Case of Dynamic Constraint Optimization.* Granada: University of Granada, Spain.

Stützle T., and Hoos, H. (1997). Improvements on the Ant System: Introducing the MAX-MIN Ant System. *ICANNGA97- Third International Conference on Artificial Neural Networks and Genetic Algorithms*, University of East Anglia, Norwich, UK, Wien: Springer Verlag.

Wodrich, M., and Bilchev, G. (1997). Cooperative Distributed Search: The Ants' Way. *Control and Cybernetics*, 26, 413-446.

Watson, M. (1997). *Intelligent Java Applications for the Internet and Intranets.* San Francisco, CA: Morgan Kaufmann Publishers.

Vita

Kent Arthur Spaulding was born in Denver, Colorado, on September 28, 1965. His parents are Arthur D. Spaulding (deceased) and Jennifer D. Spaulding. In 1990, Kent completed a Bachelor of Arts degree majoring in Computer Science Applications in Russian at The University of Colorado. During much of the time in undergraduate school, Kent worked as a student co-op for IBM Boulder where he developed an interest in object-oriented and distributed systems. He moved to Austin, Texas in 1990 and began working for The Continuum Company where he was part of team that developed a three-tiered object-request-broker development and runtime system for integrating legacy applications and business process objects. In 1993, Kent became Principal Engineer at Zephyr Technologies, and ported the SOMobjects Toolkit to Win32 as support for the OpenDoc project, as well as developing a communications framework for handheld computers. In 1996, Kent worked at Texas Instruments on a novel JavaBeans composition tool. Kent currently works at Trajecta, Inc, a small data-mining and predictive optimization startup.

Permanent address: 4510 Avenue B
Austin, TX 78751

This report was typed by the author.