University of Tennessee, Knoxville

TRACE: Tennessee Research and Creative Exchange

Masters Theses

Graduate School

12-1999

A real-valued genetic algorithm for optimizing pumped storage scheduling

Ryan Thomas

Follow this and additional works at: https://trace.tennessee.edu/utk_gradthes

Recommended Citation

Thomas, Ryan, "A real-valued genetic algorithm for optimizing pumped storage scheduling. " Master's Thesis, University of Tennessee, 1999.
https://trace.tennessee.edu/utk_gradthes/10037

This Thesis is brought to you for free and open access by the Graduate School at TRACE: Tennessee Research and Creative Exchange. It has been accepted for inclusion in Masters Theses by an authorized administrator of TRACE: Tennessee Research and Creative Exchange. For more information, please contact trace@utk.edu.

To the Graduate Council:

I am submitting herewith a thesis written by Ryan Thomas entitled "A real-valued genetic algorithm for optimizing pumped storage scheduling." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Engineering Science.

Robert E. Uhrig, Major Professor

We have read this thesis and recommend its acceptance:

Michael Vose, J. A. M. Boulet

Accepted for the Council:
Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

To the Graduate Council:

I am submitting herewith a thesis written by Ryan Thomas entitled "A Real-Valued Genetic Algorithm for Optimizing Pumped Storage Scheduling." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Engineering Science.

Robert E. Uhrig

Robert E. Uhrig, Major Professor

We have read this thesis and

recommend its acceptance:

[illegible]

[illegible]

Accepted for the Council:

[illegible]

Associate Vice Chancellor and

Dean of the Graduate School

A REAL-VALUED GENETIC ALGORITHM FOR OPTIMIZING PUMPED STORAGE SCHEDULING

A Thesis

Presented for the

Master of Science Degree

The University of Tennessee, Knoxville

Ryan Thomas

December 1999

Copyright © Ryan Thomas, 1999

All rights reserved

DEDICATION

This thesis is dedicated to my parents

Richard and Sue Thomas

ACKNOWLEDGEMENTS

The author would like to thank the many people who have made writing this thesis possible. My academic advisor, Dr. Robert E. Uhrig, has provided invaluable insight and guidance throughout my time in Graduate School.

The other members of the thesis committee, Dr. Michael Vose and Dr. J.A.M. Boulet are sincerely acknowledged for their technical advice and constructive criticism.

Support from the Nuclear Engineering Department at the University of Tennessee in the form of a Graduate Research Assistantship has funded this research and my degree. The Department of Mechanical, Aerospace, and Engineering Science has also offered a course of study that has been ideal for me. Dr. Wesley Hines has been a wonderful teacher and his MATLAB programming tips have been numerous.

Further thanks go to the Tennessee Valley Authority, and specifically Robert Taylor, for providing data and answering questions.

Special recognition goes to my parents, Richard and Sue Thomas, for their love and support.

ABSTRACT

This thesis discusses the construction and use of a real-valued Genetic Algorithm to optimize weekly scheduling for TVA's Raccoon Mountain Pumped Storage facility. Pumped storage systems are primarily employed to meet peak load conditions and to replace power plants during scheduled maintenance outages. Scheduling for this facility involves determining when to pump water into an elevated reservoir and when to release the water for generation. Using a historical weekly load, a Genetic Algorithm was constructed and used to search for schedules that maximize monetary return. The algorithm proves to be capable of finding schedules that optimize usage and simultaneously reduce peak system loads.

PREFACE

This work was done during my time as a Research Assistant for the Artificial Intelligence Group in the Nuclear Engineering Department at the University of Tennessee. To retain compatibility with other work being done by the group, the program described herein was written in MATLAB. Though interpreted languages are not ideal for Genetic Algorithms, MATLAB provides a rich set of tools and is widely used for scientific and engineering programs. Fully commented source code appears in the Appendix. Language specific implementation details will only be discussed in the text when deemed truly insightful. The remainder of this section provides organization notes for the thesis.

Chapter One serves as an introduction to the physical characteristics of pumped storage schemes and the Raccoon Mountain facility. A brief definition of Genetic Algorithms is given to prepare the reader for a more in-depth treatment later. A concise statement of the problem to be solved concludes the chapter.

Chapter Two details the theory of Genetic Algorithms. Different selection schemes are reviewed and compared. Operators for performing crossover and mutation are then examined. The chapter concludes with a review of some of the real-world problems to which Genetic Algorithms have been successfully applied.

Chapter Three deals with the design and implementation of a MATLAB toolbox for Genetic and Evolutionary Algorithms. This is intended to be a general purpose GA, which will be applicable to many future problems. The application of the toolbox to the Raccoon Mountain facility is then detailed.

Chapter Four presents and discusses the results of this application. The profit of operation, the weekly water levels and the system load impact are used to judge performance. A comparison of GA runs with varied parameter settings is also performed in order to find the settings that provide the best results in the least time.

Chapter Five provides a summary of the work, and contains concluding remarks as well as recommendations for future research.

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

Pumped Storage Schemes and Raccoon Mountain

Pumped storage systems are primarily employed to meet peak load conditions and to replace power plants during scheduled maintenance outages. The use of pumped storage can also reduce peak region-wide loads, helping to minimize the ill effects associated with frequently starting and stopping electrical generation equipment.

The main components of a pumped storage system are: two connected water reservoirs at different elevations, pumps for moving water up, and turbines for generating electricity when the water moves down. The lower reservoir, or tail pond, is generally a river or a lake, though artificial reservoirs and even the sea have been used. The upper reservoir, or head pond, is typically an artificial reservoir or a lake. Combination pump-turbines are normally used to provide both pumping and generating. Figure 1 below shows a conceptual drawing of such a system.

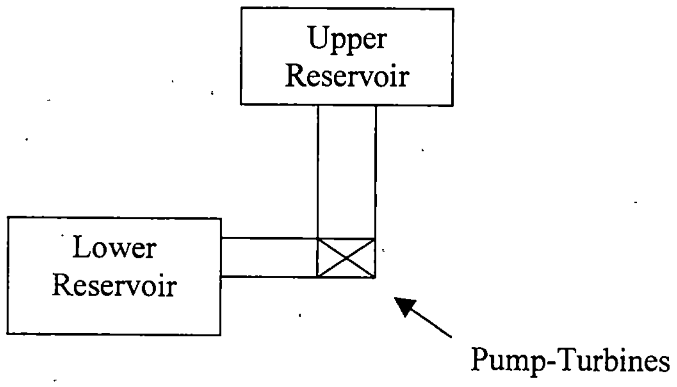


Figure 1- Pumped Storage Conceptual Model

The Second Law of Thermodynamics dictates that more energy is needed to pump a given amount of water to a higher elevation than can be regained from that amount returned during generation. What makes these schemes attractive however, is that the price of electricity fluctuates based upon the total system-wide demand. When the demand is low it can be met most economically, when "sunk" capital costs are not considered, by using low cost nuclear, coal and hydroelectric plants. As demand increases though, more expensive and less replaceable oil and gas fuels must be used, or the electricity must be purchased from other electricity generators. Thus, for a pumped storage system by itself to operate without a financial loss, pumping must be done when the cost of electricity is lower than the cost when generating. Figure 2 shows a hypothetical weekly region-wide demand, starting from 8:00 Monday morning.

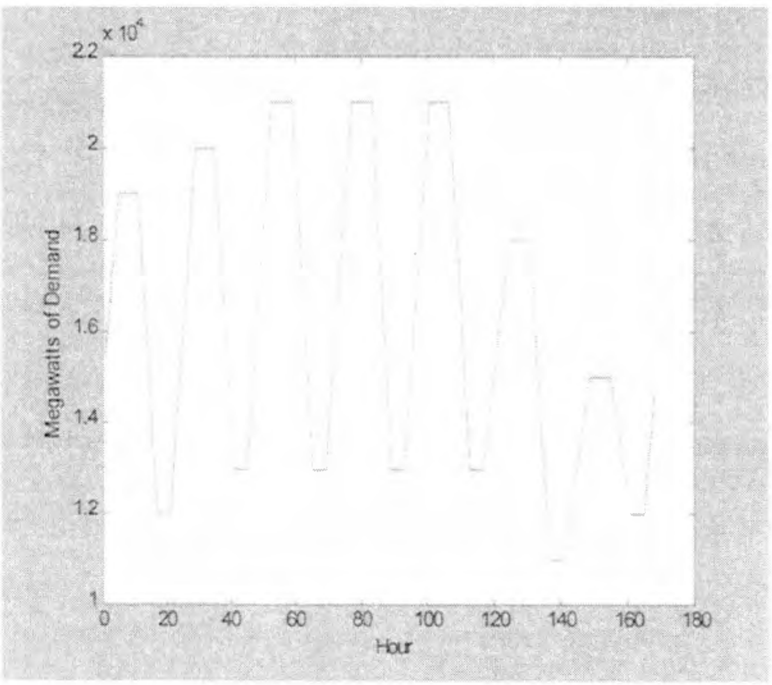


Figure 2 – Demand Curve

Pumped storage schemes are generally part of a larger power generation system. This allows electricity needed for pumping to be taken from the power grid and the electricity generated to be returned to the grid. By scheduling the periods of generation to coincide with periods of high demand, higher cost plants can be kept off line and the use of expensive purchased electricity can be avoided.

The Tennessee Valley Authority (TVA) operates the Raccoon Mountain pumped storage facility near Chattanooga Tennessee and is the nations largest producer of public power. This facility is located on the Tennessee River at mile 444.6, between the Chickamauga Dam upstream and the Nickajack Dam downstream. This river serves as the lower reservoir, and an artificial upper reservoir has been created in the mountain above. The two reservoirs are connected by a series of tunnels that contain four 382.5 MW pump-turbines, for a total capacity of 1530MW. Figure 3 depicts the physical layout of the Raccoon Mountain facility.

The upper reservoir can be filled by 27 hours of pumping, and emptied in 20 hours of generating. However, the system operates under the restriction that the tail pond, the Nickajack Reservoir, must remain between 632 and 634 feet to maintain navigability. Water discharged from the Chickamauga Dam upstream takes two hours to reach the Raccoon Mountain site. Since the Nickajack Dam and the Raccoon Mountain site are both remotely controlled from the Chickamauga Dam, the combined oversight necessary to ensure proper water depth is available. For the purposes of this study, the navigability requirements will not be considered since adjusting the flow through the Chickamauga Dam can control the lower reservoir water level.

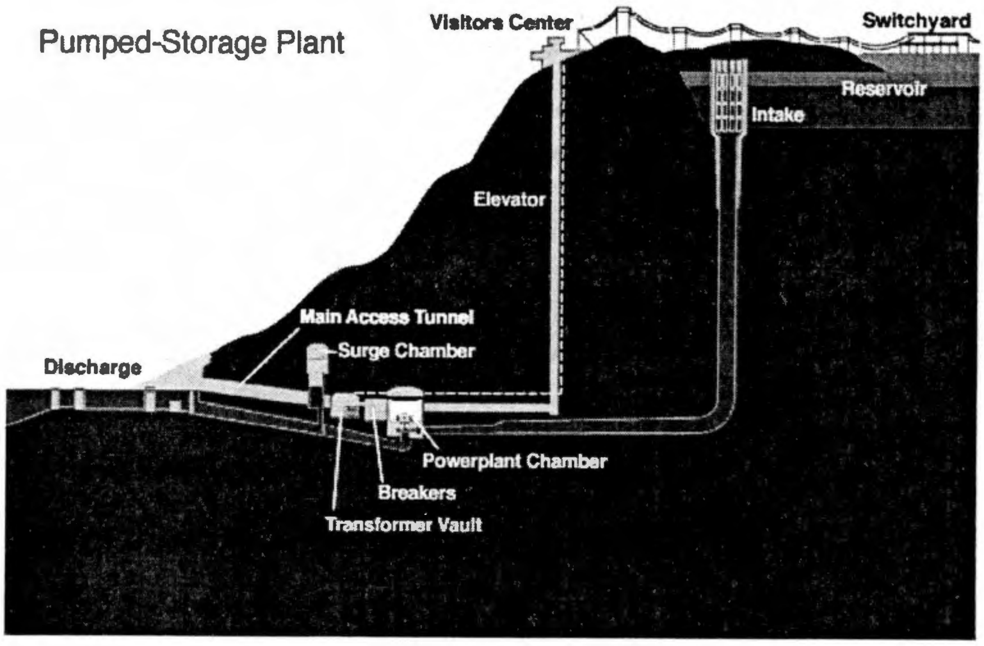


Figure 3 – Raccoon Mountain Facility Diagram
Source: TVA Web Site, www.tva.gov/power/pumpstorart.htm

Electricity demand follows a weekly cycle in the TVA region, with the highest peak demands occurring in the summer and winter months. By examining historical data and trends and taking weather predictions into account, it is possible to make some prediction of the demand or load on the system for an upcoming week. Given this prediction, some technique must be used to determine the weekly schedule for pumping and generating that results in the most profitable operation. This is further complicated due to the amount of electrical input and output of the pump-turbines being a function of the water level of the head pond.

The head pond water level can vary from 1530 to 1672 feet above sea level, and the pump-turbines are located 1167 feet below the full pool level. The input power required when pumping and the output power produced when generating are shown plotted against the head pond water level in Figure 4 below. The breaks in the curves are due to manipulation of intake valves to maintain efficiency, and the differing discharge rates.

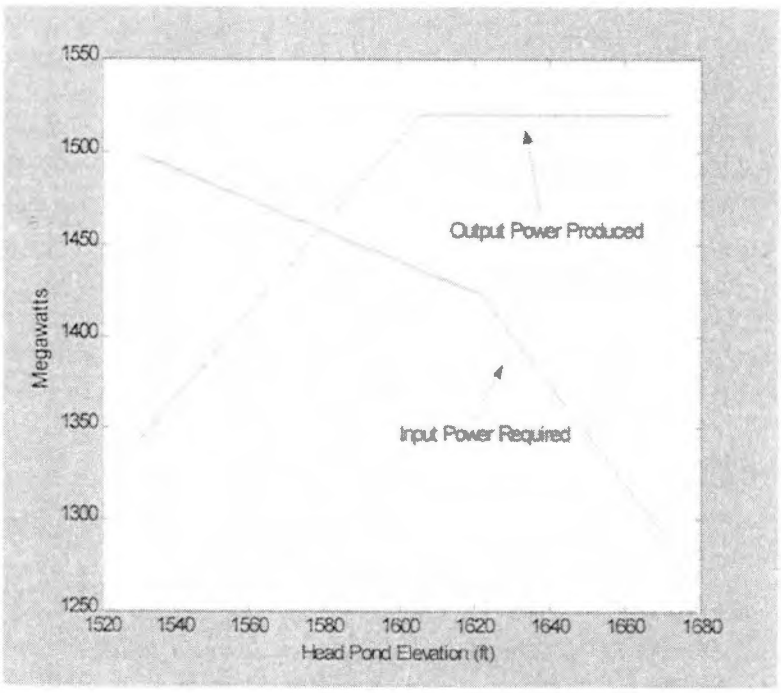


Figure 4 – Pump-Turbine Curves

Since solving directly for the optimal schedule would be prohibitively time consuming due to the non-linearity of the system, a faster method is needed. The most profitable schedule of operation based on a given weekly demand prediction can be determined by searching the space of all possible weekly schedules for the one that offers the highest return. One hour is taken as the minimum unit of time in the schedule. If we have a predicted demand for electricity, it is possible to completely describe a weekly schedule by specifying for each hour how many, if any, of the pump-turbines are running and in which direction. Thus, a weekly schedule could be represented as a string of 168 whole numbers on [-4, 4], where negative indicates pumping and positive indicates generation. This leads to 9^{168}, or about 2.05 X 10^{160}, possible schedules. For comparison, the universe is estimated to be 10^{18} seconds old and to contain around 10^{80} particles. This implies that an exhaustive search of the space would not be feasible. This thesis describes the implementation and use of an alternate search strategy, the Genetic Algorithm.

Introduction to Genetic Algorithms

John Holland originally introduced Genetic Algorithms in 1975 as a method for modeling and explaining adaptation in natural systems [9]. Genetic algorithms, or GA's, typically involve operations such as crossover, mutation and selection, which are based on their chromosomal counterparts in genetics. Since their introduction, Genetic Algorithms have been reexamined as a method for the optimization of mathematical functions. By searching the space of all possible inputs to a given function, Genetic Algorithms can be used to find the inputs that maximize or minimize that function. GAs can also be used to optimize physical systems by creating a function that models the operation of the system.

Genetic Algorithms represent a single input to the function as an array of digits, commonly called a chromosome. A small random collection of such chromosomes constitutes an initial population. Each member of the population is then used as an input to the function, and a numerical output value is calculated. The function that guides the optimization process is referred to as the fitness function, and the output for a specific input is the fitness score. By selecting pairs of chromosomes based upon fitness scores, a new population is generated from the current one by the operations of crossover and mutation. Crossover involves creating child chromosomes by using different parts of the chromosome of each parent. Mutation generally involves a random change in a single digit of the child chromosome. Often, the best chromosome from each generation is intentionally passed through to the next unchanged. This ensures that the algorithm does not abandon a good solution until a better one is found, and is referred to as elitism.

By repeating this process through a number of iterations, or generations, it is possible to optimize a wide variety of classes of problems. This amounts to a "greedy" parallel hill-climbing approach. Many researchers have worked to develop different types of selection schemes, crossover operators and mutation operators, which will be further explored in the next chapter.

Problem Statement

A relatively quick and automatic method is desired that will provide pumping and generating schedules for the Raccoon Mountain pumped storage facility. The method should be able to accept a prediction of a weekly region-wide load, and return the schedule that provides the maximum profit. A Genetic Algorithm has been chosen as the method to search for the best schedule.

2. THEORY AND LITERATURE REVIEW

Genetic Algorithms Basics

Though Genetic Algorithms were originally created to simulate adaptation in natural systems, current research focuses on ways GAs can use evolution to perform function optimization. The Genetic Algorithm conducts a search of the space of inputs to the function being optimized by relying on the principals of Darwinian natural selection.

Genetic Algorithms have been successfully applied to many different types of problems, though several factors limit the success of a GA on a specific function. Problems that require a good, but not optimal, solution are ideal for GAs. The manner in which points on the search space are represented is an important consideration. An acceptable performance measure or fitness value must be available. It must also be feasible to test many potential solutions.

Points on the search space are typically encoded as binary strings, though real-valued representations are also widely used. Individual strings are referred to as chromosomes, and a set of chromosomes is termed the population. A set of randomly chosen points on the search space provides an initial population. Each iteration of the algorithm is called a generation, and involves creating a new population by reproduction between chromosomes of the current population.

Each generation consists of four steps: evaluation, selection, crossover, and mutation. During evaluation each member of the population is used as an input to the function being optimized, and a numerical fitness score is determined. Chromosomes are then selected, based on fitness, to be parents for the next generation. The selected parents are paired together, and produce pairs of offspring. The crossover operation determines how the genetic material of the parents will be passed on to the children. Mutation involves changes to individual positions on the chromosomes, and allows new information to enter the population.

Many different selection schemes exist, as well as several types of crossover and mutation operators. Choosing the proper selection scheme and reproduction operators for a given problem can influence the success of the search. The next sections will further explore how selection, crossover, and mutation work.

Selection Schemes

Selection of parents is based on fitness scores, though several different schemes exist. Four specific schemes, Roulette Wheel, Universal Stochastic Sampling, Ranking, and Tournament will be individually explored.

Roulette Wheel selection chooses parents for the next generation proportional to their fitness scores. The fitness of each individual in the population is summed, and a wheel is simulated with this as the circumference. Each chromosome is then assigned to a portion of the wheel equal to its percent of the fitness of the total population. Random numbers indicating points on the circumference are generated to simulate spins of the wheel. The chromosome corresponding to the section of the wheel indicated by the random number is then selected as a parent for the next generation. This process continues until the desired number of parents has been selected.

Roulette Wheel selection generally offers fast convergence properties, but premature convergence to local maxima or minima is sometimes encountered. If "super" chromosomes exist in the population, with fitness scores many times those of the other chromosomes, problems can arise leading to loss of genetic information from the population and premature convergence. Roulette Wheel selection is also highly sensitive to the fitness function. Care must be taken to insure that the function provides a suitable range of output values.

The Stochastic Universal Sampling method is quite similar to Roulette Wheel, but only one random spin of the wheel is made. The location of the spin denotes the first parent, and from there the wheel is marked at even intervals around the entire circumference to identify successive parents. When the wheel circumference is normalized, the interval between two indicated selections is the inverse of the population size.

This selection method is more efficiently implemented than standard Roulette Wheel, and also helps solve the problem of premature convergence. Since spins are equally distributed around the wheel, a single individual is less likely to be multiply selected. The sensitivity to the fitness function has not been removed though, and can still result in problems.

Ranking selection provides a way to reduce sensitivity to the fitness function, and also helps to control premature convergence. Instead of choosing parents based proportionally on fitness, only the relative position within the population is considered. A ranking function is constructed, which maps a chromosome's relative position to a numerical score. This function can be linear or non-linear, depending on the selection pressure desired. A linear version of the ranking function might assign the chromosome with the worst fitness score a value of one, the second worst a value of two, and so on through the best, which would have a value equal to the population size. Using these new values as fitness scores, proportional selection is then performed.

By scaling fitness according to rank, premature convergence due to "super" individuals becomes less of a problem. There is also less sensitivity to the output range of the fitness function. These improvements, however, generally come at the price of somewhat slower convergence.

Tournament selection also provides an alternative to proportional selection. In this scheme, a randomly selected subset of chromosomes from the population is chosen, and the one with the highest fitness is selected as a parent. The number of chromosomes selected to compete in each tournament is referred to as the tournament size. The tournament process is repeated until all parent positions have been filled. Sometimes referred to as local competition, this method can also be used on distributed systems. The population subset might then consist of neighboring chromosomes.

Tournament selection normally shows slow, steady convergence. This scheme also has a low sensitivity to the fitness function, and is unaffected by "super" individuals. The choice of tournament size directly influences the selection pressure. With a small tournament size, the odds of a chromosome with a low fitness score being selected as a parent increase. Larger tournaments are more likely to contain chromosomes having high fitness, thus reducing the chances of the information in a low scoring chromosome being passed on to the next generation.

Each selection scheme has advantages and disadvantages, and some trial and error testing is necessary to determine the scheme that works best for a specific problem. Construction of a good ranking function and the determination of a good tournament size are crucial to successful application. The fitness function must also be carefully constructed when Roulette Wheel or Universal Stochastic Sampling is used.

Crossover Operators

After completion of the selection process, the chromosomes chosen to be parents for the next generation are recombined to form children. This recombination can take many forms, and the specific crossover operators used can greatly effect the results of the search. Two of the most widely used crossover operators are Discrete and N-point. Both operators work on pairs of parent chromosomes, though extension to three or more parents is possible.

Discrete crossover, sometimes referred to as Uniform crossover, can be used in both binary and real-valued chromosome representations. Two chromosomes that have been chosen to be parents are used to create two child chromosomes. For each location on the child chromosomes an equal probability random selection is made to determine which parent contributes the genetic material.

The following example demonstrates the Discrete crossover operator in action. Two parent chromosomes are shown below, each having a length of four.

Parent 1: 1 1 1 1   Parent 2: 0 0 0 0

Two random vectors are generated, and indicate the parent to be used at each position.

Vector 1: 2 1 1 2   Vector 2: 1 2 1 1

The offspring created contain information from both parents, and are shown below.

Child 1: 0 1 1 0   Child 2: 1 0 1 1

Information in the population can be lost using Discrete crossover, since both children may contain the genetic material of the same parent at a given location. This is seen in the example above at position three in the chromosome, where the information from parent two has been lost. If this occurs too often, it is possible that the GA will not find the global maximum due to insufficient information in the population.

N-point crossover is normally used when chromosomes are represented using a binary encoding, though it has also been employed with real-valued chromosomes. This operator mimics the crossover process that is found in animal DNA. A total of N locations on the parent chromosomes are selected, and the genetic material of the parents is exchanged about these points.

An example of N-point crossover for N equal to three is given below. Two chromosomes selected to be parents are shown, each having six positions.

Parent 1: 1 1 1 1 1 1 Parent 2: 0 0 0 0 0 0

Three crossover points are then chosen uniformly at random, and are located after the position indicated by the random number.

Crossover Locations: 1 3 5

The two children created are shown below, with the vertical bar indicating crossover location.

Child 1: 1 | 0 0 | 1 1 | 0 Child 2: 0 | 1 1 | 0 0 | 1

The N-point crossover operator passes on all information contained in the parents to the next generation. Since corresponding locations on the child chromosomes never come from the same parent, no information is lost. The one restriction on this operator is that the number of crossover points must be at least one less than the length of the chromosome.

Both the Discrete and N-point crossover operators typically contain a parameter, the crossover rate, which determines how often the operator is applied. The crossover rate ranges between zero and one, with zero indicating the operator is never applied, and one indicating that the operator is always applied. For each pair of parents a random number is generated on [0, 1]. If this random number is greater than the crossover rate, the operator is not applied to that pair of parents. If the operator is not applied, the children produced are identical to the parents.

Mutation Operators

Crossover operators can only recombine genetic information that already exists in the population. The optimal solution to the given problem, however, may not be reachable with only the information that was supplied by the random initial population. Mutation forms the second half of the reproduction process, and serves to introduce new information into the population. Many types of mutation operators exist, and problem specific operators can often be created to incorporate domain knowledge into the Genetic Algorithm. Three basic mutation operators, Bit Flip, Random Replacement, and Small Shift will be discussed further.

The Bit Flip operator is only used with a binary chromosome representation. This operator causes the value of a particular point on the chromosome to change to its opposite. An example of a Bit Flip mutation is shown below.

Before Mutation: 1 1 1 1 After Mutation: 1 1 0 1

Here the third location in the chromosome has been selected to undergo the mutation. The process by which the locations for mutation are chosen will be discussed later.

The Random Replacement operator can be used with both binary and real-valued chromosome representations. In this operator the value of a position on the chromosome is changed to a value selected uniformly at random from all allowable values for that position. In a binary representation this reduces to a Bit Flip operator that only operates half as often, since roughly half of the time the bit chosen will be replaced by the same value. In real-valued representations, the chance of replacing the initial value with the same number varies with the size of the set of allowable values.

The Small Shift operator is only defined for non-binary representations. Here, a small random number is added to the current value at the position being mutated. The random number may be chosen from a uniform or normal distribution, and may be negative. Normally distributed random numbers tend to keep the mutated value close to the original, while uniformly distributed random numbers do not. The new value must be checked to insure that it is still within the allowable range for that position. If not, some correction must be made to produce an allowable value. Some correction methods include rounding, random replacement, and setting the position to the nearest legal value.

Mutation operators also contain a parameter that determines how often they are applied. This is known as the mutation rate, and lies on [0, 1]. While the crossover rate is only checked once for each pair of parents, the mutation rate must be checked for each position on each chromosome. If a randomly generated number is greater than the mutation rate, a mutation occurs at that position. This can cause slow program execution due to the large number of comparisons that must be made each generation. The use of pre-created mutation masks can alleviate this concern in binary representations, though are generally not as effective with real-valued representations.

Genetic Algorithm Applications

Genetic Algorithms are increasingly being used to optimize and control physical systems. A fitness function that models the system can often be created, and a GA can then be used to find the best inputs to the function. Genetic Algorithms also form the basis for Learning Classifier Systems, which can be used for system control.

Genetic Algorithms have been successfully applied to a large number of real-world problems. One of the first such applications was a gas pipeline control system created by David Goldberg [5], a student of John Holland. Greffenstette [7] mentions several GA applications including message routing, scheduling, and automated design. Entire conferences have been devoted to applications of Genetic Algorithms and evolutionary techniques to specific disciplines, such as Image Analysis, Signal Processing and Telecommunications [13]. GAs have also found application in such varied domains as physics [14], condition monitoring [1], robot control [3,4], game playing [16], and ecology [20].

Genetic Algorithms can also be used in conjunction with other techniques. By adding a hill climbing post-processor, for example, it may be possible to reach a global maximum from a nearly optimal solution obtained by using a GA. Hybrid artificial intelligence techniques involving Genetic Algorithms, Neural Networks, and Fuzzy Logic are becoming increasingly common. Lee and Takagi [10] describe a fuzzy control system, operating on GA parameters such as mutation rate and population size, which attempts to combat premature convergence. The use of GAs to evolve fuzzy rule bases has also been explored. GAs have been used with Neural Networks to optimize neuron connections and weights [12], as well as for automated architecture selection [18], and optimal selection of inputs [8].

3. METHODS

Design of Genetic Algorithm Toolbox

The MATLAB programming environment provides a wide array of tools and features designed to simplify numerical programming. The MATLAB language is functionally a C interpreter with no pointers and only one data type, the matrix. Though interpreted languages are not ideal for Genetic Algorithms due to speed considerations, the built-in math functions and graphics tools yield a very versatile programming environment.

Since Genetic Algorithms can be used on a wide variety of problems, the Toolbox was designed to be a general purpose GA framework. This framework provides access to the selection schemes, crossover operators, and mutation operators, which are common to all GA implementations. Ideally, only one Genetic Algorithm should need to be constructed in a given programming language. Isolating all of the parts of the program that change from problem to problem allows the GA to be tailored to a specific problem with minimal reprogramming.

The Toolbox consists of seven main function files: **GASearch, Select, Crossover, Mutate, Evaluate, GAgui**, and the **Objective Function**. In addition, a driver function named **GA** is included for setting program parameters, and another function, **RMplot**, is used to control graphical output. The **Objective Function** and the driver function contain all of the problem specific information, and are the only files changed when applying the GA to another problem. Figure 5 shows the relations between these files.

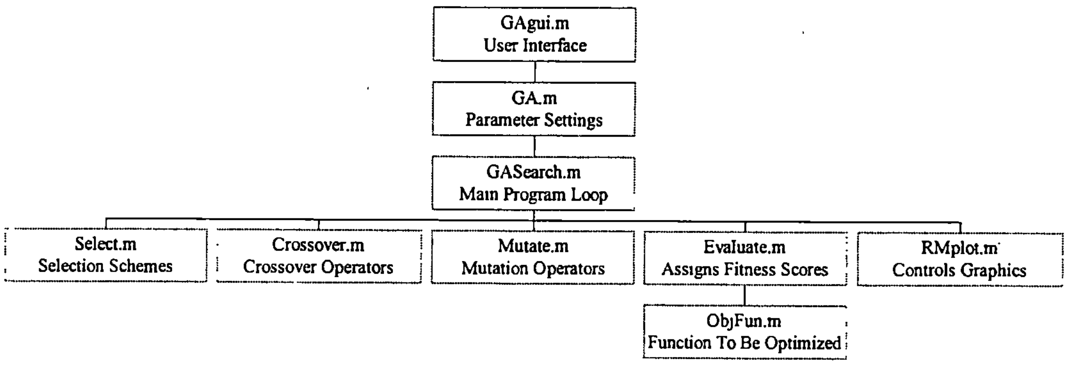


Figure 5 – Toolbox Organization

The Toolbox is to be used to conduct a search for the set of numerical parameters to the **Objective Function** that cause it to return the highest value. By writing an **Objective Function** that models a physical system to be optimized, many combinations of system parameter values can be evaluated in an automatic manner.

The main body of the algorithm is the **GASearch** function. The function first allocates space for an initial population and initializes them to uniformly distributed random numbers. The function then begins a loop that iterates the algorithm through successive generations. Every generation, the **Select** function is called to choose members of the population based on the value they return as parameters to the **Objective Function**. The individuals chosen are then recombined to form a new set of potential solutions using the **Crossover** function. This new set is then subjected to the **Mutation** function, which allows new information to enter the population. Every member of the new population is then used as a parameter to the **Objective Function** and assigned a fitness score by the **Evaluate** function. If the fitness goal has not been met and the maximum number of generation has not elapsed then the **GASearch** function begins another iteration of the loop. Each function will be discussed in detail later in this chapter.

The **GASearch** program requires eight matrices as inputs that are used to define the GA parameters. The input arguments are: maximum number of generations, the fitness goal, population size, crossover rate, mutation rate, maximum and minimum value ranges, and a flag for the **Evaluate** function. Though each of these parameters is a matrix, all have only one row.

The maximum number of generations and the fitness goal have one column, and together provide bounds to the loop outlined above. The population size matrix has three columns; the first is the true population size, the second specifies the type of selection scheme to be used, and the third gives a tournament size for tournament selection. Since a two-parent – two-child reproduction scheme is being used, the true population size must be even, and will be made so if entered incorrectly. The crossover size and mutation rate matrices each have two columns. Both list the true appropriate rate as a number on [0.0, 1.0] in the first column, and specify a particular operator to be used in the second. The population maximum and population minimum matrices fix the chromosome length implicitly, and together form a range over which the search may proceed. The flag for the **Evaluate** function determines whether or not the program should compile a list of all solutions that have been evaluated. If such a list is used, it is possible to save time by not re-evaluating previously seen solutions. However, this is only practical when ample memory is available and the evaluation of the **Objective Function** takes substantially more time than searching the list.

The **Select, Crossover** and **Mutate** functions all share a similar style of implementation in terms of how a scheme or operator is chosen. Each use the second column of the appropriate **GASearch** input matrix to specify between the available methods. Each of these functions can also be expanded should a need for new methods be experienced. This allows domain knowledge to be used to create problem specific operators that can better solve the given problem.

The **Select** function offers four basic selection schemes: Roulette Wheel, Stochastic Universal Sampling, Linear Ranking, and Tournament selection. Each method is used to create a list of parents for the next generation. Roulette Wheel assigns each solution a portion of a one-unit circumference wheel proportional to its fitness value. Random numbers are then generated on [0, 1] to simulate spins of the wheel. The solution assigned to the portion of the wheel that contains the random number is selected to be a parent. Stochastic Universal Sampling is similar, except that only one random number is generated on [0, 1/population size]. The first parent is chosen by where the random number falls, and the rest are found by successively adding one over the population size to the initial random number. Linear Ranking assigns the worst solution a new fitness of 1 and the best solution a new fitness equal to the population size, with all others linearly in between. The next generation parents are then determined by roulette wheel selection with the new fitness scores. Tournament selection uses the third column of the population size **GASearch** parameter to specify a number of individuals to be chosen from the population. The individual with the highest fitness of those chosen is then put on the list of parents for the next generation. This is repeated until the list is complete. The completed list is returned to the **GASearch** function, which in turn passes it to the **Crossover** function.

The **Selection** function is also used to insure that the algorithm passes a copy of the best solution yet found to the next generation. This is done so that the search will maintain a good position on the search space until a better on is located. This is known as elitism, and in this implementation a pristine copy and a copy to be mutated are perpetuated from one generation to the next. A side effect of this is that the algorithm will always provide a solution equal to or better than the one it begins with.

The **Crossover** function uses the second column of the crossover size **GASearch** input parameter to determine which crossover operator should be used. If the value of the parameter is zero, Discrete crossover is performed. This operator selects the first two individuals on the parent list, and creates two children by selecting for each position on each child chromosome which parent contributes the genetic material. This is then repeated for each pair of parents on the list to create the next generation. If the value of the operator type parameter is not zero, then an N-point operator is used, where N is the value of the parameter. This operator randomly selects a crossover position on the parent chromosomes and creates two temporary children. One of these temporary children has the genetic material of the first parent up to the crossover point and that of the second parent beyond that point. The second temporary child has the material of parent two first and then parent one. This process repeats N times on the temporary children for each set of parents until the new population is obtained. In both the Discrete and N-point operators the first column of the crossover size matrix is used as the crossover rate. For each pair of parents, a random number is generated on [0, 1]. If the number is above the crossover rate no crossover occurs, and the children are exact copies of the parents.

The first two parents on the list are both the same member of the population due to the elitism in the **Selection** function. Since neither the Discrete nor the N-point crossover operators perform any useful function when the two parents are the same, two pristine copies of the best individual in the last generation will carry through to the new population. The new population is returned to the **GASearch** function, which transfers it to the **Mutation** function.

The **Mutation** function offers the options of a small normally distributed Small Shift operator, a viral Neighbor Invasion operator and a uniformly distributed Random Replacement operator. The second column of the mutation rate matrix specifies which of these operators should be used. The normal Small Shift operator generates a uniform random number on [0, 1] and compares this to the first column of the mutation rate matrix, the true mutation rate. If the value of this random number is less than the mutation rate a small normally distributed random number is generated, and added to the first position on the chromosome. This process is repeated at every location on the chromosome. The Neighbor Invasion operator generates a uniform random number for comparison with the mutation rate to determine if the operator is to be applied. If the operator is to be used, another uniform random number is generated. Forty five percent of the time the value of the position on the chromosome is changed to match that of its left neighbor and forty five percent of the time the value is changed to match the right neighbor. The remaining ten percent corresponds to a random replacement from the allowable value range at that position. This operator is also applied at every position on the chromosome. The uniformly distributed Random Replacement operator mimics the random replacement operation previously mentioned without the neighbor invasion.

The specified mutation operator is applied to all members of the new population except the first. Since the first and second are both the most fit individual from the last generation, this completes the elitism implementation by providing one pristine copy of the best and one mutated copy. The new population has now been obtained from the previous, and is passed back to the **GASearch** function.

GASearch then calls the **Evaluate** function with the new population and the table flag as arguments. For each member of the population, this function calls the **Objective Function** with the chromosome as its input parameters. If the table flag is set then each chromosome is checked against the table before being evaluated. The output of the **Objective Function** for each member of the population is recorded on a list, which will be passed back to **GASearch** and used by the **Select** function in the next iteration.

Once the search has terminated, the **GASearch** function returns the best set of parameters and a list of all the fitness values achieved. A sorted list of fitness values is also provided, along with an index matrix to transform between the two. The **GA** and **GAgui** functions are used to handle the construction of the **GASearch** input matrices, and to construct meaningful interpretations of the progress and results.

The **GAgui** file defines a graphical user interface to the program. This interface can be used to set all of the parameter values to the GA, and also launches the search and displays the progress. The interface was constructed using the MATLAB GUI design tools, and was a fairly automatic process. This interface allows the program to be used with minimal direct manipulation of the underlying source code

The **Objective Function** must be written to model the problem to be solved. Indeed, the Objective Function is the only connection between the GA and the system being optimized. The use of a separate function such as this allows great latitude in the types of models that can be made. Individual positions on the chromosome can correspond to different parameters within the modeled system, each having separate allowable ranges. These parameters could represent such things as the coefficients of a function, loop bounds and array values, strategies for playing tic-tac-toe, even variables to be selected for training an Autoassociative Neural Network. The use of MATLAB makes applications like these easily realizable. The next section details the application of the Toolbox to the Raccoon Mountain Pumped Storage scheduling problem.

Application To Raccoon Mountain

Several factors must be accounted for to accurately model the Raccoon Mountain Facility. First, the value of energy as a function of demand needs to be represented. Second, since the input and output power of the pump-turbines depends upon the water level this relationship must be characterized. Finally, the schedules should not cause the water level in the upper reservoir to drop below 1530 feet or to rise above 1672 feet, and the pool must be returned to the full level by the last hour of the week.

The value of energy based on demand was obtained from a curve fit for three points: 0.008 $/kWh at 8000 MW, 0.012 $/kWh at 14000 MW, and 0.024 $/kWh at 20000 MW. This fit results in an equation for the cost of energy as a function of the region-wide demand. This equation is

$$y = 1.511 - 0.1777\,x + 0.01111\,x^2 ,$$

where y is the cost in cents per kilowatt hour and x is the demand in gigawatts. Figure 6 shows this relation.

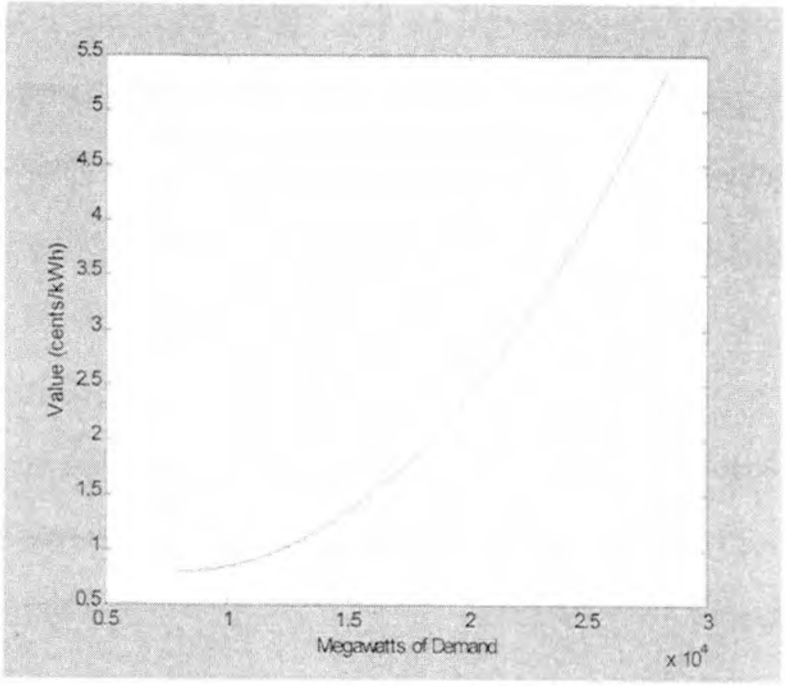


Figure 6 – Incremental Cost of Energy

The input power required for pumping and the output power acquired when generating were modeled from the pump curves provided by TVA. These curves are shown in Figure 7. The breaks in the two curves are due to intake vane manipulation done to retain efficiency, and the differing discharge rates.

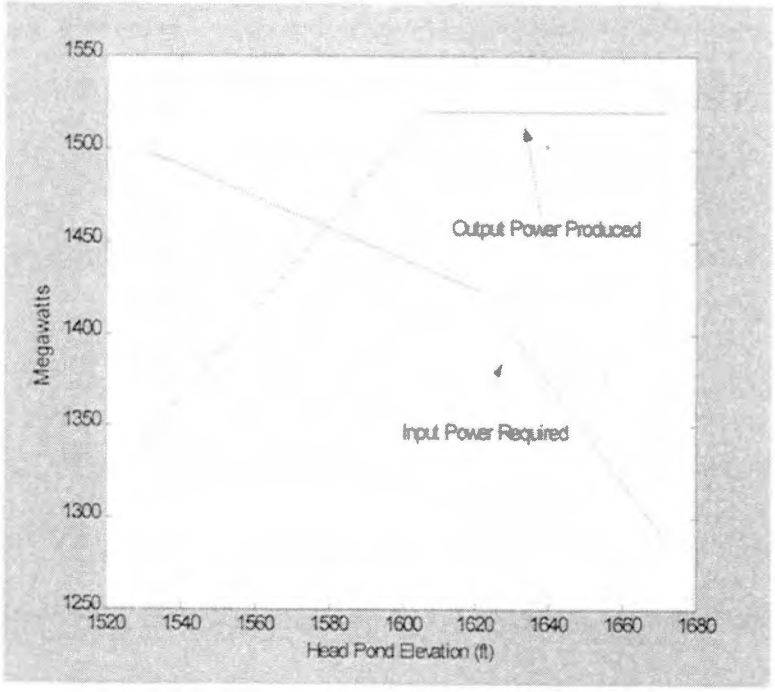


Figure 7 – Pump-Turbine Power Requirements

By declaring two single row matrices and initializing them based upon the equations, simulations of the power value and input-output curves were created. A range of demands from 5000 MW to 28282 MW was used, and a range of head pond levels from 1530 ft. to 1672 ft. It is then easy to determine the costs and effects associated with operating the facility, when provided with the upper reservoir pool level and the region-wide demand, by indexing into these matrices. There is a simplifying assumption being made here; namely that the discharge rates from the pump-turbines remain constant. Though this is not accurate, it allows for much less complicated calculations. Dividing the 142 feet of head variation by the 27 hours of pumping that is needed to fill the pool yields an average pumping rate of 1.32407 feet per hour per pump. Likewise, the turbines were each found to empty 1.7875 per hour, based on 20 hours to empty the entire reservoir. This eliminates the need to deal directly with the efficiencies of the pump-turbines.

To insure that the schedules conform to the operating limitations of the facility, matrices for the hourly allowable minimum and maximum water levels were created. The maximum level is 1672 feet. The minimum pool level is 1530 feet from hours 1 to 140 and then climbs linearly to 1672 by hour 168, since a maximum of 27 hours of pumping are needed to fill the pool. The correction of schedules that do not meet these criteria will be discussed later.

The weekly schedule was fixed to start on Monday morning at 8AM with a full upper reservoir. By interpreting the value of a location on the chromosome as the number of pumps or turbines in action during a given hour, it is possible to simulate the operation of the Raccoon Mountain facility and to calculate the financial profit or loss of a proposed schedule. This profit or loss figure is then used as the fitness score for the schedule being evaluated.

Beginning at the first position on the chromosome, the **Objective Function** calculates the amount of power needed or produced by the action specified at the present head level. This value is then added or subtracted from the predicted demand level for that point in the week. This new value for demand represents the additional load on the region-wide system when running the pumps, or the reduced amount of load that needs to be otherwise met when running the turbines. The cost of this action is then determined by indexing into the value of energy matrix at the new demand level and converting to dollars per Megawatt hour.

The change in water level associated with the action specified is next determined. The new water level at the end of the hour's operation is then compared to the minimum and maximum allowable pool levels. If a particular hourly action would cause the water level to exceed the maximum or drop below the minimum level for that hour, the value of the chromosome is replaced by the nearest legal value. For example, if the use of four pumps would cause the water level at the next hour to be 1673 feet, the schedule will be corrected so that only three pumps are used and the constraint is met. This correction method allows any schedule to be transformed into a viable one during evaluation.

Once the cost of the action is computed and its effect on the water level is known the procedure is repeated using the action specified in the next position on the chromosome. This continues until the entire weekly schedule has been evaluated. The cost of each hourly action is summed, and this represents the weekly operating return for the proposed schedule. This is then used as the fitness score, and has units of thousands of dollars.

The energy value curve and the input-output curves, as well as the predicted demand and the head level constraints, are declared and constructed as global variables in the **GA** driver function associated with the application. The parameter matrices used as inputs to **GASearch** are also constructed in this driver. When the search terminates, either by reaching the fitness goal or the maximum number of generations, the **GA** file saves the contents of the workspace and displays the best solution produced by the search.

The **GASearch** function was also modified to call a function that provides a real-time progress display. The function, **RMplot,** is used to create three plots. The first plot shows the improvements in fitness by monitoring the return of the best schedule of each generation. The second plot displays the upper reservoir's hourly water level using the current best schedule. The final plot uses the same schedule to depict the impact of the Raccoon Mountain facility on the region-wide demand. A sample of the display is shown in Figure 8.

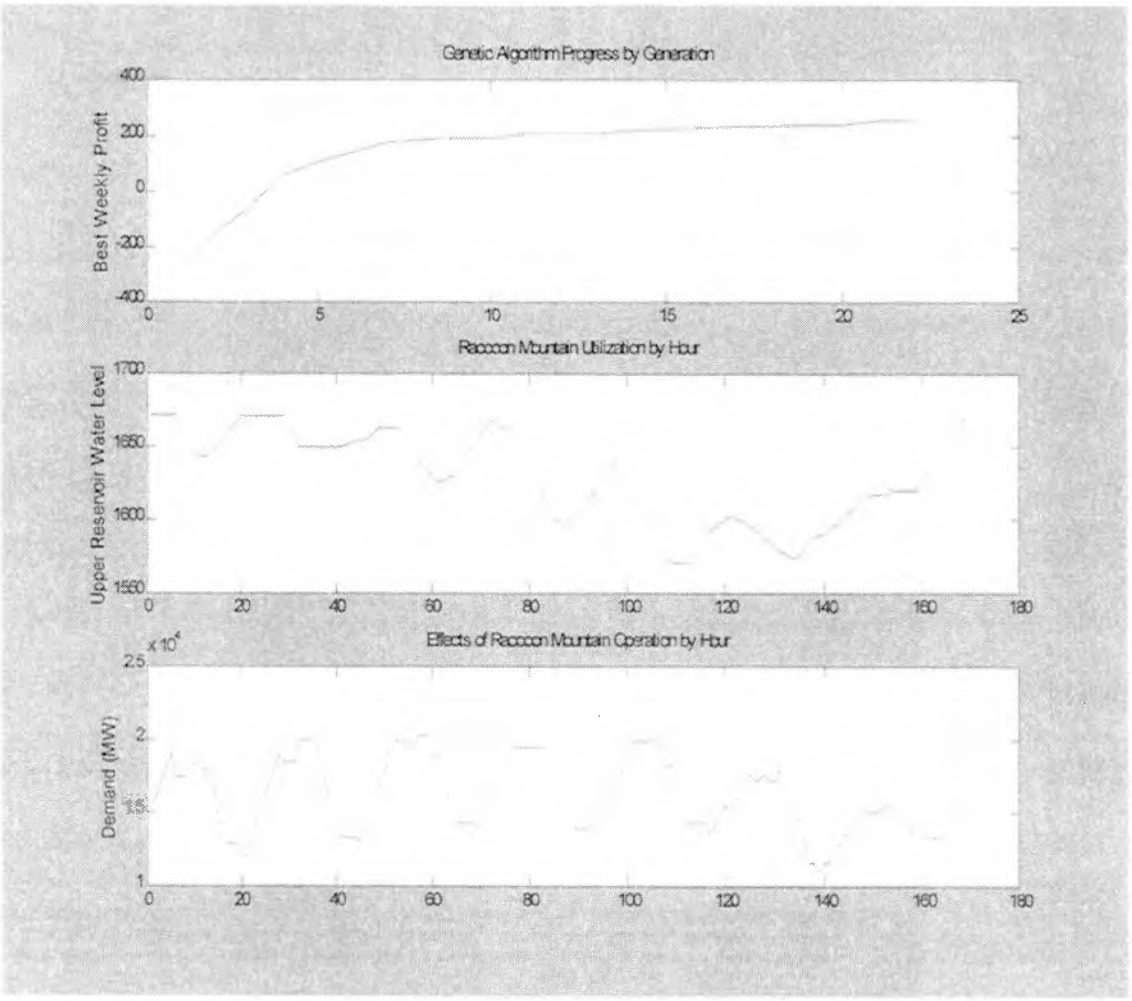


Figure 8 – Sample Results Display

4. RESULTS

Parameter Settings

A large amount of trial and error testing was done in order to decide what combinations of selection scheme, crossover operator, and mutation operator produced the best results in the least time. By examining the Progress by Generation indicator on the output display it is possible to see the effects of different combinations. Almost all of the combinations showed a rapid evolution from the random initial population, followed by a leveling off when the population converged. Progress up to the convergence point appears to be guided primarily by the crossover operation, and beyond that point all further gains are generally the result of mutation. The slope of the progress curve up to the convergence point shows wide variation when different combinations are employed, and is used as the basis for comparison. All combinations were tested using the sample weekly load curve shown in Figure 2.

A typical fitness for the best random schedule in the initial population is on the order of -$300,000 per week, which illustrates the potential losses that could be incurred without proper scheduling. All of the combinations explored were eventually able to evolve schedules that produced a profit. However, the number of generations a particular combination of settings needed to reach this point varied greatly.

Tournament selection appears to offer the quickest and most steady progress, regardless of the crossover and mutation operators used. Some improvement in the fitness is made every generation until the convergence point. Roulette Wheel, Universal

Stochastic Sampling, and Linear Ranking all experienced generations where no improvement was made. This often happened as early as two or three generations into the run. All three of these schemes showed slower progress than Tournament selection, and generally had final profit results that were significantly lower. The tournament size producing the best results was typically a little over half of the population size.

The N-point and Discrete crossover operators displayed very little difference in their effects on the progress of the search. N-point was chosen as the default operator since Discrete crossover can cause information in the population to be lost. A range of values for N was tested to determine the impact of multiple crossover points. When less than about six crossover points are used, progress tends to be slow. However, when the value of N is between six and about thirty, progress is much more rapid. Increasing the number of crossover points above thirty does not significantly improve the results, but does increase processing time.

The use of the Neighbor Invasion mutation operator appears to have a significant impact on the progress of the search. Schedules produced using this operator also conform more closely to what human schedulers would produce. Specifically there are few periods where pumps or turbines are turned on and off rapidly. An example of this would be a gradual progression from one pump to four pumps occurring over four hours, instead of going directly from one pump in hour one, to four pumps in hours two and three, and then back to one pump in hour four. The Random Replacement operator produced the opposite results. The smoothing effects that the Neighbor Invasion operator has on the schedule are lost. This not only causes problems with the pump-turbines at

Raccoon Mountain, but would also have an adverse effect on the other power generation facilities in the TVA system. The Random Replacement operator was also not able to reach as high of a fitness level as the Neighbor Invasion operator for the same number of generations.

The population size and the crossover and mutation rates also directly affected the speed at which the search progressed. If the population size was very small, convergence typically occurred before a highly profitable schedule could be found. Trial and error indicated that a population size of about one hundred was sufficient to prevent premature convergence due to lack of information in the initial population. The GA relies on crossover to exploit existing information and mutation to incorporate new information into the population. Since the main work of the search is done by the crossover operation, a high crossover rate is generally preferred. A fairly low mutation rate enables the search to gain new information without abandoning information that already exists. The default parameter settings for crossover and mutation were chosen as 0.8 and 0.4 respectively.

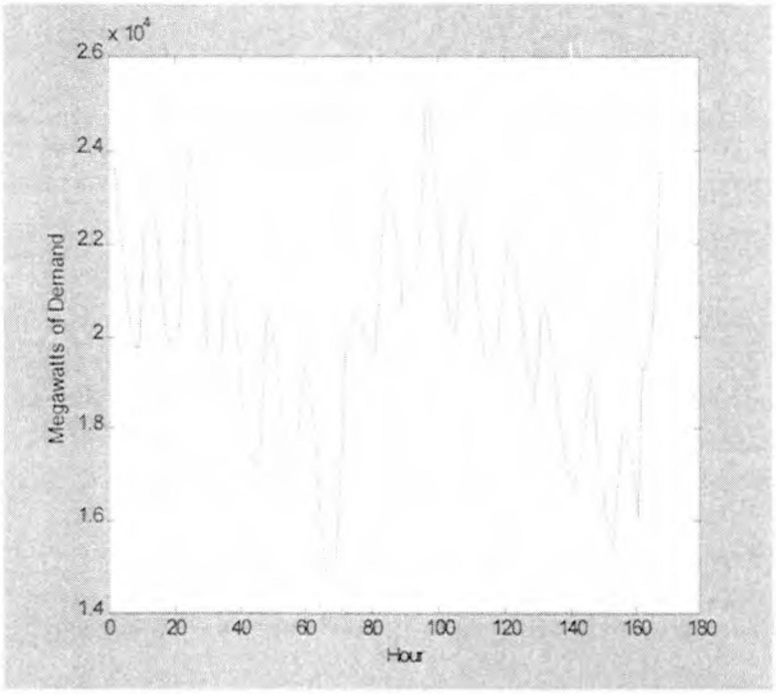
The best settings identified for selection, crossover, mutation, and the population size were incorporated into the **GAgui** file as the default selections, and are shown in Table 1 below. Historical data about typical weekly demand curves for each season was provided by TVA, and these default parameters were used to conduct searches for the best schedule for each demand curve. The results of the searches will be examined in the next section.

Table 1- Default Parameter Settings

Parameter Name	Parameter Setting
Population Size	100
Selection Scheme	Tournament
Tournament Size	55
Crossover Operator	N-point
Crossover Points	6
Crossover Rate	0.8
Mutation Operator	Neighbor Invasion
Mutation Rate	0.4

Seasonal Loads

The highest demand for electricity in the TVA region is in the summer and winter months. Typical hourly demand in the winter ranges from 15,000 to 25,000 MW during this time, and peaks twice daily. Since spring weather in the region is fairly mild, less electricity is used for heating than in the winter months. This results in hourly demands ranging from 10,000 to 18,000 MW, and also shows a double daily peak. Summers in the region tend to be quite warm, and the hourly demand for electricity ranges from 14,000 to 25,000 MW. The demand curve for the summer months, however, peaks only once per day. Fall weather is similar to the spring, and results in hourly demands from 11,000 to 17,000 MW. Figures 9 through 12 show representative demand loads for each season.



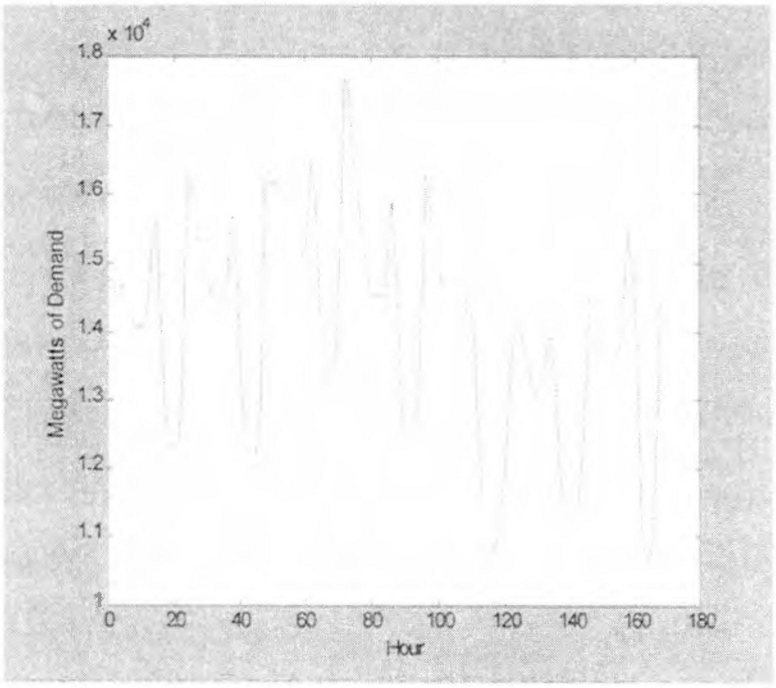


Figure 10 – Typical Spring Demand

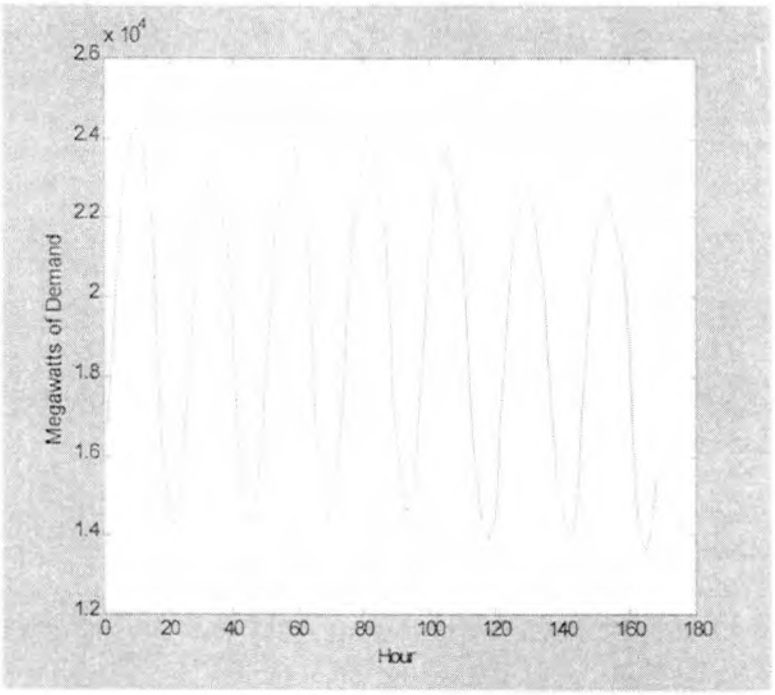


Figure 11 – Typical Summer Demand

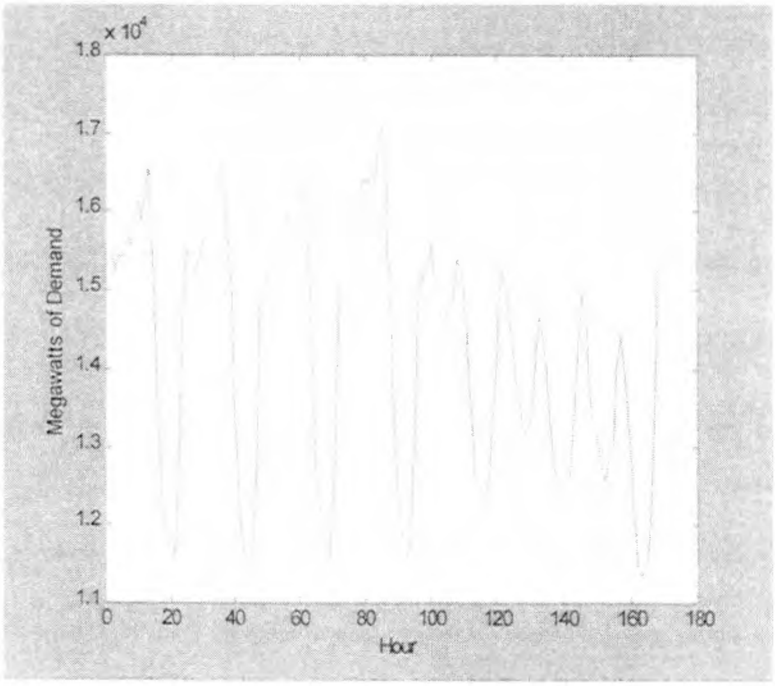


Figure 12 – Typical Fall Demand

Ten thousand random schedules were generated for each seasonal curve to provide a comparison between the Genetic Algorithm and Random Search. The results of the Random Search are shown in Table 2. Four runs of the Genetic Algorithm were then conducted for each of the seasonal demand curves. All runs used the default settings listed in the last section, and each run lasted one hundred generations. The results of each run are tabulated in Table 3, where the fitness score corresponds to weekly profit in thousands of dollars. For an equal number of schedule evaluations the Genetic Algorithm outperforms the simple Random Search in every case.

The GA performed extremely well on the summer demand curve, most likely due to the large variation between minimum and maximum demand. The demand is also high enough in the summer to push the TVA system into the steep portion of the incremental cost curve. While the winter demand curve also has a large range, the daily variations do not appear to be great enough to cause the schedules to assume a more or less daily pattern as is observed with the summer curve.

Table 2 – Random Search Results

Season	**Best Fitness (1000's of Dollars)**
Winter	-261.8886
Spring	-172.1430
Fall	-166.9189
Summer	-114.4241

Table 3 – Genetic Algorithm Results

Season	**Run Number**	**Best Fitness (1000's of Dollars)**
Winter	1	244.3848
Winter	2	214.9610
Winter	3	271.7147
Winter	4	250.2554
Spring	1	72.6647
Spring	2	64.6047
Spring	3	65.9362
Spring	4	66.6481
Summer	1	801.6419
Summer	2	756.2335
Summer	3	753.3519
Summer	4	736.1586
Fall	1	60.2173
Fall	2	55.3109
Fall	3	51.2546
Fall	4	53.0878

The spring and fall demand curves both show a fair amount of daily variation, but both are on the less steeply sloping part of the incremental cost curve than the summer and winter demands. However, the GA finds weekly cycles in both cases that result in a modest profit. Since the profitability of the pumped storage system relies heavily on fluctuations in the price of energy, less return will be generated by the system when there is little variation in daily and weekly demand.

For all four seasonal demand curves the Genetic Algorithm also succeeds in shaving peak region-wide loads. This is an emergent property of the GA, and is consistent with the current operation of Raccoon Mountain. Figures 13 through 16 show the graphical output generated by the **Rmplot** function for the fourth run for each season. The rate of progress of the GA for each run was fairly consistent, and in each case the population converged before the end of the hundred generations. This implies that by the end of the search the GA was mainly relying on mutation. Thus it appears that increasing the population size or extending the number of generations might produce more profitable schedules.

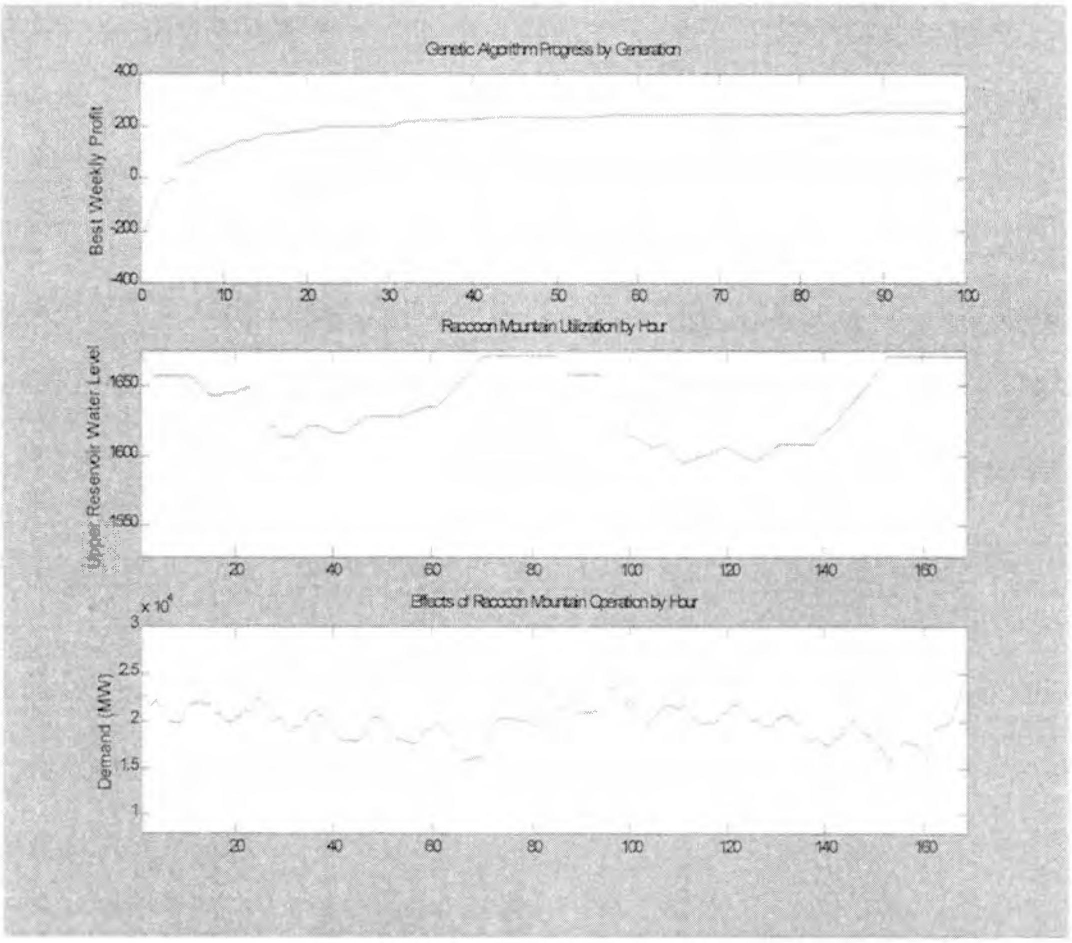


Figure 13 – Sample Winter Results

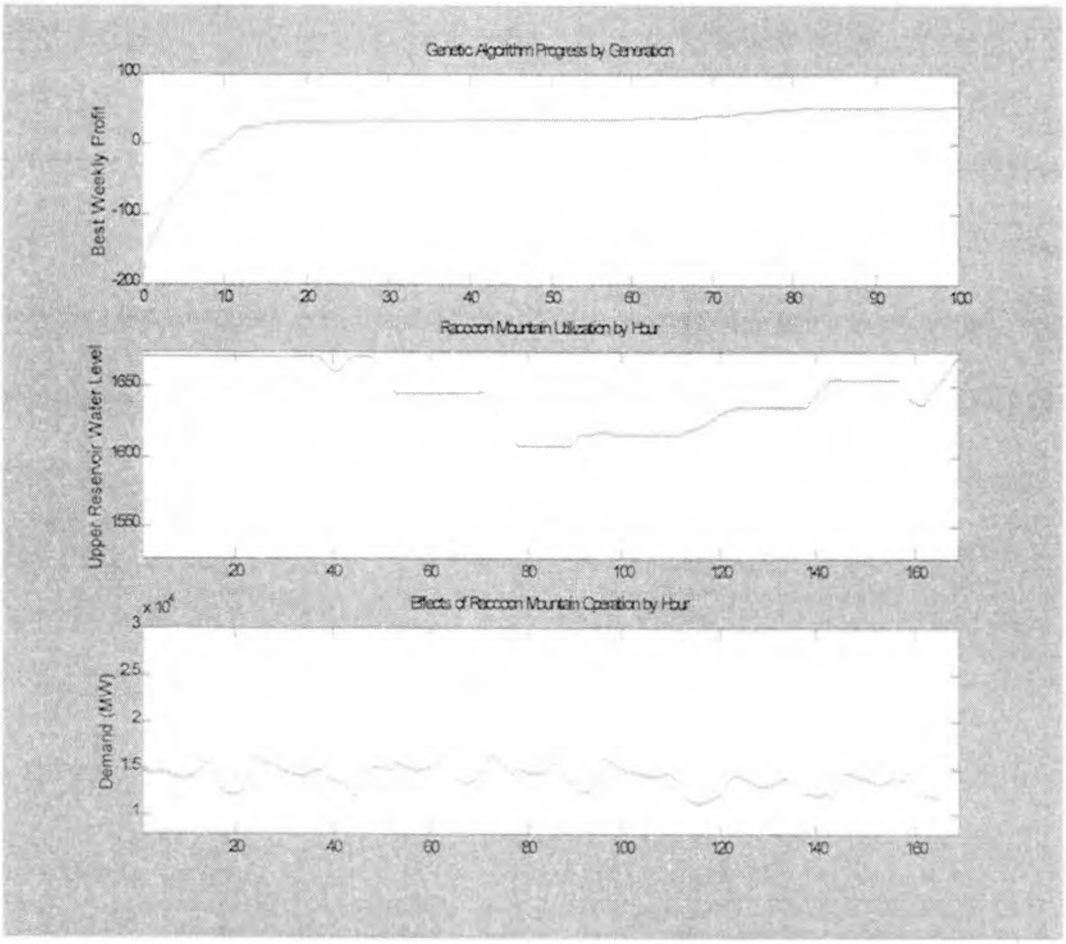


Figure 14 – Sample Spring Results

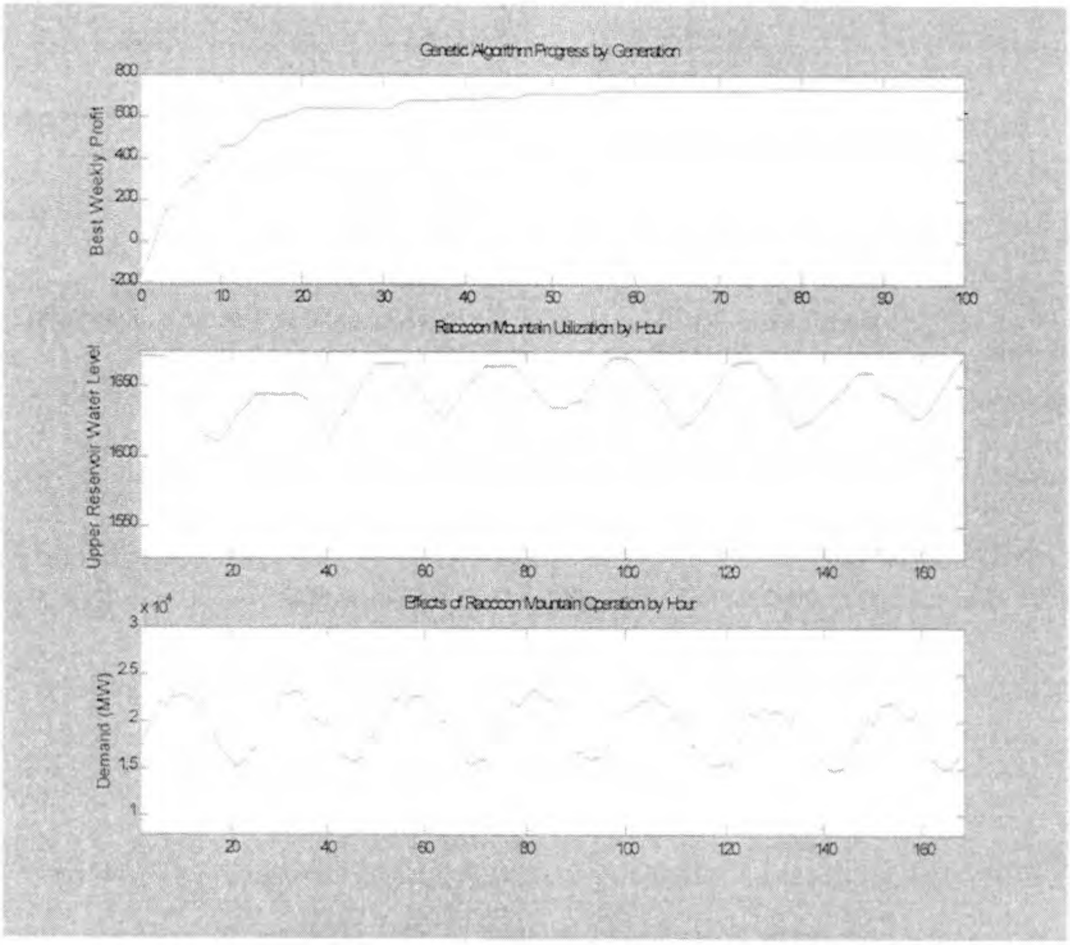


Figure 15 – Sample Summer Results

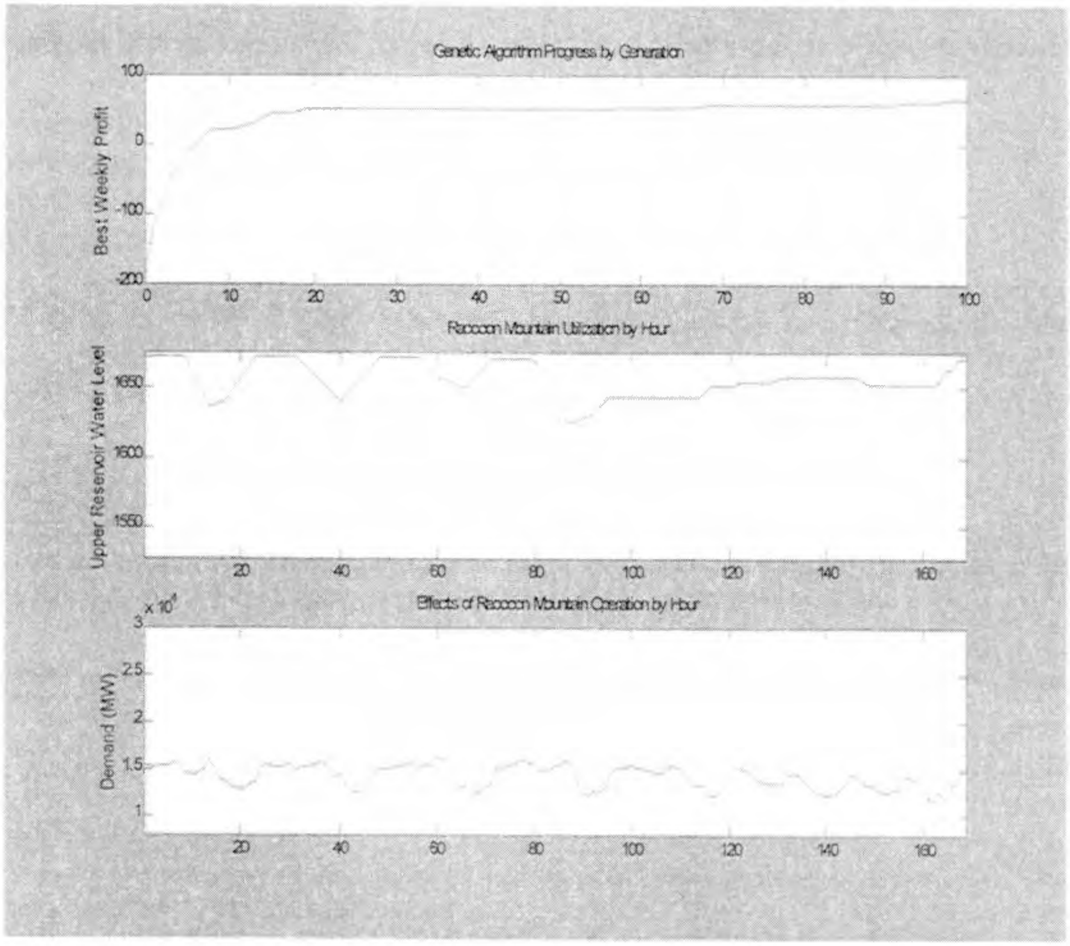


Figure 16 – Sample Fall Results

5. CONCLUDING REMARKS AND RECOMMENDATIONS FOR FUTURE RESEARCH

The Genetic Algorithm approach to schedule optimization has generated highly profitable schedules in a relatively short time. A typical run with a population size of one hundred evolving through one hundred generations involves only ten thousand schedule evaluations, which is less than 5^{-155} percent of the total search space. A Random Search of the space with the same amount of schedule evaluations can not generally produce a schedule that shows a profit, and an Exhaustive Search of the space is not possible.

The Genetic Algorithm also provides two improvements over the current scheduling algorithm used by TVA. The current method utilizes Dynamic Programming, and requires a substantial amount of time to run. Ten thousand schedule evaluations can be conducted by the GA in less than eight minutes on a Pentium III 500Mhz personal computer. Converting the GA software from MATLAB to C or C++ would reduce computing time by two or three orders of magnitude.

The scheduling program in use now also requires that the program be completely rerun every time new demand prediction information becomes available. The GA, however, can use the results of a previous search as part of the initial population for a new search when updated demand predictions are received. This can greatly reduce computation time necessary to produce a profitable schedule.

The current TVA scheduling program includes some operating considerations that the Genetic Algorithm does not. Inter-region buying and selling of electricity, scheduled maintenance outages, and proper pump-turbine discharge rates are all incorporated into the current Dynamic Programming algorithm. Inclusion of these factors into the Genetic Algorithm could improve performance and provide a more accurate model of the system.

Two of the most frequently encountered problems with Genetic Algorithms are the selection of proper parameter settings and premature convergence. Both of these problems could be addressed by the inclusion of a Fuzzy Logic control system as described by Lee and Takagi [10]. This would create a Dynamic Parametric Genetic Algorithm, which uses fuzzy rules to adjust parameters during a run based on the progress of the search. This eliminates the need for trial and error testing to determine good operating parameters, and makes the GA more robust to premature convergence.

REFERENCES

REFERENCES

1. Benzing, D.,"Space Shuttle Main Engine Condition Monitoring Using Genetic Algorithms and Radial Basis Function Neural Networks", *Industrial Applications of Genetic Algorithms*, CRC Press 1999.

2. Boston, W., "Conceptual Design for the Integration of Raccoon Mountain Pumped Storage into the TVA Generation Control System", TVA Internal Document, 1977.

3. Dain, R., "Development of Mobile Robot Wall-Following Algorithms Using Genetic Algorithms", *Industrial Applications of Genetic Algorithms*, CRC Press, 1999.

4. Fukuda, T., Kubota, N., Arakawa, T.,"GA Algorithms in Intelligent Robots", *Fuzzy Evolutionary Computation*, Kluwer Academic Publishers, 1997.

5. Goldberg, D., "Computer-Aided Pipeline Operation Using Genetic Algorithms and Rule Learning. Part I: Genetic Algorithms in Pipeline Optimization", *Engineering with Computers 3*, 1987.

6. Goldberg, D., *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley, 1990.

7. Greffenstette, J., *Genetic Algorithms Made Easy*, 1991.

8. Guo, Z., *Nuclear Power Plant Fault Diagnostics and Thermal Performance Studies Using Neural Networks and Genetic Algorithms*, Ph.D. Dissertation, 1992.

9. Holland, J., *Adaptation in Natural and Artificial Systems*, University of Michigan Press, 1975.

10. Lee, M. and Takagi, H., "A Framework for Studying the Effects of Dynamic Crossover, Mutation and Population Sizing in Genetic Algorithms," *Advances in Fuzzy Logic, Neural Networks and Genetic Algorithms*, Springer-Verlag, 1995.

11. Levy, S., *Artificial Life*, Vintage Books, 1992.

12. Nishita, K., "Tuning BAMA Optimized Neural Networks Using Genetic Algorithms", *Industrial Applications of Genetic Algorithms*, CRC Press, 1999.

13. Poli, R., et. al. (Eds.), *Evolutionary Image Analysis, Signal Processing and Telecommunications*, Springer, 1999.

14. Reichert, A., "Using A Genetic Algorithm to Determine the Optimum Two-Impulse Transfer between Coplanar, Elliptical Orbits", *Industrial Applications of Genetic Algorithms*, CRC Press, 1999.

15. Robinson, R. A., *A Study of a Proposed Pumped-Storage Project for Use as Peaking Capacity on the TVA System*, Masters Thesis, 1965.

16. Shi, J., "Genetic Algorithms for Game Playing", *Industrial Applications of Genetic Algorithms*, CRC Press, 1999.

17. TVA, *Final of Environmental Statement for Raccoon Mountain Pumped-Storage Plant*, Tennessee Valley Authority, 1976.

18. Uhrig, R. E. and Tsoukalas, L. H., *Fuzzy and Neural Approaches in Engineering*, John Wiley, 1997.

19. U. N. Economic Commission for Europe, *Future Role of Pumped-Storage Schemes for Peak-Load Hydro-Electric Supply*, United Nations Publication, 1968.

20. Wand, S., "Simulation of an Artificial Eco-System Using Genetic Algorithms", *Industrial Applications of Genetic Algorithms*, CRC Press, 1999.

APPENDIX:

SOURCE CODE

FOR

MATLAB GA TOOLBOX

Source Code for GAgui.m

```
%GAgui
% Graphical User Interface to Genetic Algorithm Toolbox
% Constructed using MATLAB Guide
%
% Ryan Thomas
% Copyright 1999

load def_params
load GAgui

h0 = figure('Color',[0.8 0.8 0.8], ...
    'Colormap',mat0, ...
    'FileName','C:\MATLABR11\bin\testgui.fig', ...
    'HandleVisibility','callback', ...
    'PaperPosition',[18 180 576 432], ...
    'PaperUnits','points', ...
    'Position',[163 358 554 199], ...
    'Tag','Fig1', ...
    'ToolBar','none');
h1 = uicontrol('Parent',h0, ...
    'Units','points', ...
    'BackgroundColor',[0.752941176470588 0.752941176470588 0.752941176470588],
    'Callback','GA(popsize,maxgens,xrate,murate);', ...
    'ListboxTop',0, ...
    'Position',[250.5 82.5 80.25 50.25], ...
    'String','RUN', ...
    'Tag','Parameter List');
h1 = uicontrol('Parent',h0, ...
    'Units','points', ...
    'BackgroundColor',[1 1 1], ...
    'Callback',mat1, ...
    'Position',[32.25 69 84 63], ...
    'String',mat2, ...
    'Style','listbox', ...
    'Tag','Parameter List', ...
    'Value',1);
h1 = uicontrol('Parent',h0, ...
    'Units','points', ...
    'BackgroundColor',[1 1 1], ...
    'ListboxTop',0, ...
    'Position',[32.25 36 84 15], ...
    'String','100', ...
```

```
        'Style','edit', ...
        'Tag','Value Box', ...
        'Value',2);
h1 = uicontrol('Parent',h0, ...
        'Units','points', ...
        'BackgroundColor',[0.752941176470588 0.752941176470588 0.752941176470588],
        'Callback',mat3, ...
        'ListboxTop',0, ...
        'Position',[150 117 65.25 15], ...
        'String',mat4, ...
        'Style','popupmenu', ...
        'Tag','PopupMenu1', ...
        'Value',4);
h1 = uicontrol('Parent',h0, ...
        'Units','points', ...
        'BackgroundColor',[0.752941176470588 0.752941176470588 0.752941176470588],
        'Callback',mat5, ...
        'ListboxTop',0, ...
        'Position',[150 36 45 15], ...
        'String','Set Value', ...
        'Tag','Pushbutton1');
h1 = uicontrol('Parent',h0, ...
        'Units','points', ...
        'BackgroundColor',[0.752941176470588 0.752941176470588 0.752941176470588],
        'Callback',mat6, ...
        'ListboxTop',0, ...
        'Position',[150 90 65.25 15], ...
        'String',mat7, ...
        'Style','popupmenu', ...
        'Tag','PopupMenu2', ...
        'Value',2);
h1 = uicontrol('Parent',h0, ...
        'Units','points', ...
        'Callback',mat8, ...
        'ListboxTop',0, ...
        'Position',[150 63 65.25 15], ...
        'String',mat9, ...
        'Style','popupmenu', ...
        'Tag','PopupMenu3', ...
        'Value',2);
h1 = uicontrol('Parent',h0, ...
        'Units','points', ...
        'BackgroundColor',[0.752941176470588 0.752941176470588 0.752941176470588],
        'ListboxTop',0, ...
```

```
'Position',[265.5 46.5 55.5 15.75], ...
'Style','text', ...
'Tag','Progress Display');
```

Source Code for GA.m

```
function GA(popsize,maxgens,xrate,murate)
% Controls Creation of GA Parameters and Raccoon Mountain curves
%
% Ryan Thomas
% Copyright 1999

home
global allpumps   %whether to allow pumps to be used individually
allpumps=4;            %1=[-1,1]  4=[-4,4]

%default parameters, loaded in GAgui
%maxgens=500;popsize=[100,4,55];
%xrate=[.8,30];murate=[.4,2];

%choose operating parameters
pumpcurve=1;loads=5;costcurve=2;
usetable=0;goal=500;

if allpumps==4
  popmax=4*ones(1,168);
  popmin=-4*ones(1,168);
elseif allpumps==1
  popmax=1*ones(1,168);
  popmin=-1*ones(1,168);
end

global t
t=0:.0416:7;   %daily sine time
global lcurve
lcurve=zeros(1,size(t,2));
if loads==0
  load myload
elseif loads==1
  lcurve=14500+3500*sin(2*pi*(t-.0416));  %even daily sine wave
elseif loads==2 | loads ==4
    lcurve=[15500,16375,17250,18125,19000,19000,19000,19000,19000,19000,19000,
17833,16667,15501,14335,13169,12000,12000,12000,12000,12000,13000,
14000,15000,16000,17000,18000,19000,20000,20000,20000,20000,20000,20000,20000,
%6pm tuesday
18834,17668,16502,15336,14170,13000,13000,13000,13000,13000,14000,15000,
16000,17000,18000,19000,20000,21000,21000,21000,21000,21000,21000,21000,
%6pm wed
19667,18334,17000,15668,14335,13000,13000,13000,13000,13000,14000,15000,16000,
17000,18000,19000,20000,21000,21000,21000,21000,21000,21000,21000,
%6pm thurs
19667,18334,17000,15668,14335,13000,13000,13000,13000,13000,14000,15000,16000,
17000,18000,19000,20000,21000,21000,21000,21000,21000,21000,21000,
%6pm fri
19667,18334,17000,15668,14335,13000,13000,13000,13000,13000,13625,14250,14875,
15500,16125,16750,17375,18000,18000,18000,18000,18000,18000,18000,
%6pm sat
16834,15668,14502,13336,12170,11000,11000,11000,11000,11000,11500,12000,12500,
13000,13500,14000,14500,15000,15000,15000,15000,15000,15000,15000,
%6pm sun
14500,14000,13500,13000,12500,12000,12000,12000,12000,12000,12875,13750,14625]
;
elseif loads==3
 lcurve=[14000.00,15000.00,16000.00,17000.00,18000.00,18000.00,18000.00,18000.00,
18000.00,18000.00,
18000.00,17000.00,16000.00,15000.00,14000.00,13000.00,12000.00,12000.00,12000.00,
12000.00,12000.00,12500.00,13000.00,13500.00,14000.00,14500.00,15000.00,15500.00,
16000.00,16000.00,16000.00,16000.00,16000.00,16000.00,16000.00,15000.00,14000.00,
13000.00,12000.00,11000.00,10000.00,10000.00,10000.00,10000.00,10000.00,11000.00,
12000.00,13000.00,14000.00,15000.00,16000.00,17000.00,18000.00,18000.00,18000.00,
18000.00,18000.00,18000.00,18000.00,16666.50,15333.00,14000.00,12667.00,11333.50,
10000.00,10000.00,10000.00,10000.00,10000.00,11125.00,12250.00,13375.00,14500.00,
15625.00,16750.00,17875.00,19000.00,19000.00,19000.00,19000.00,19000.00,19000.00,
19000.00,17833.50,16667.00,15500.00,14333.00,13166.50,12000.00,12000.00,12000.00,
12000.00,12000.00,12875.00,13750.00,14625.00,15500.00,16375.00,17250.00,18125.00,
19000.00,19000.00,19000.00,19000.00,19000.00,19000.00,19000.00,17500.00,16000.00,
14500.00,13000.00,11500.00,10000.00,10000.00,10000.00,10000.00,10000.00,10500.00,
11000.00,11500.00,12000.00,12500.00,13000.00,13500.00,14000.00,14000.00,14000.00,
14000.00,14000.00,14000.00,14000.00,12833.50,11667.00,10500.00,9333.00,8416.50,
7500.00,7500.00,7500.00,7500.00,7500.00,8000.00,8500.00,9250.00,10000.00,10750.00,
11500.00,12250.00,13000.00,13000.00,13000.00,13000.00,13000.00,13000.00,13000.00,
12500.00,12000.00,11500.00,11000.00,10500.00,10000.00,10000.00,10000.00,10000.00,
10000.00,11000.00,12000.00,13000.00 ];
elseif loads==5
  hand=figure(3);
  plot([0])
  for i=1:168
    xlim([.9 i+(i-.9)])
    ylim([8000 25500])
    grid on
    hold on
    plot([1:i-1],lcurve(1,1:i-1))
    [dummy,lcurve(1,i)]=ginput(1);
    hold off
  end
  clf reset
  plot(lcurve)
  xlim([1 167])
  ylim([8000 30000])
  title('New Load - Press any key')
  pause
  close(hand)
end
if loads==4                    % with 5% random "noise"
  lcurve=lcurve+(lcurve*.05.*randn(1,168));
end

global capacity
capacity=5000:1:28282;        %MW
global cost
if costcurve==1
  cost=.132 + 42e-6*capacity + 4.667e-9*(capacity.^2);
elseif costcurve==2
  cost=1.511-.1777*(capacity./1000)+.01111*((capacity./1000).^2);
end

global minhead
minhead=zeros(1,169);
for i=1:142, minhead(i)=1530;end
for i=169:-1:143, minhead(i)=1672-(169-I)*5.296;end
global output
global input
output=zeros(1,143);
input=zeros(1,143);

if pumpcurve==1        %from conceptual study
  for i=1:76,
    output(i)=1340+(180/76)*i;
  end
  for i=77:143,
    output(i)=1520;
  end
  for i=1:91,
    input(i)=1500+(-75/90)*i;
  end
  for i=92:143,
    input(i)=1425+(-135/50)*(i-92);
  end
end % if pumpcurve==1

if pumpcurve==2       %from engineering judgement
  for i=1:76,
  output(i)=1000+(300/76)*i;
end
for i=77:143,
  output(i)=1300+(200/65)*(i-76);
end
for i=1:76,
  input(i)=1500+(200/76)*i;
end
for i=77:143,
  input(i)=1700+(300/65)*(i-76);
end
end%if pumpcurve==2;

if pumpcurve==3       %from energy in storage chart
  head=1530:1:1672;
  depth=head-1529;
  eg=0.084507*depth+.00019837*(depth.^2)+.000005588*(depth.^3);
  ep=41-.077465*depth-.00029756*(depth.^2)-.0000083820*(depth.^3);
  egp=.084507+.00039674*depth+.000016764*(depth.^2);
  epp=-.077465-.00059512*depth-.000025146*(depth.^2);
  input=-epp*1000*5.259;
  output=egp*1000*7.1;
end    %if pumpcurve==3;

[table,sorted_table,SI,best] =
GAsearch(maxgens,popsize,xrate,murate,popmax,popmin,usetable,goal);
%display results and settings

best
popsize=popsize
xrate=xrate
murate=murate
pumpcurve=pumpcurve
loads=loads
costcurve=costcurve

%save all active variables to a file
%to recall, use "load optimal"
```

save optimal

Source Code for GASearch.m

```
function [table,sorted_table,SI,best] =
GASearch(maxgens,popsize,xrate,murate,popmax,popmin,usetable,goal);
% [table,sorted_table,SI] =
GASearch(maxgens,popsize,xrate,murate,popmax,popmin,usetable,goal)
% GASearch attempts to maximize ObjFun.m with respect to its parameters.
%
% Input Parameters:
%   maxgens(1x1) = maximum number of generations to compute
%   popsize(1x2) = [population size ,selection type]
%   xrate  (1x2) = [crossover rate (<1.0) ,crossover type]
%   murate (1x2) = [mutation rate (<<1.0) ,mutation type]
%   popmax (1xr) = the maximum acceptable parameter string
%   popmin (1xr) = the minimum acceptable parameter string
%     where: r is the number of parameters to ObjFun.m
%   usetable(1x1)= 1 to check against repeat table, 0 to evaluate all
%   goal   (1x1) = fitness goal
% Output:
%   table(Kxr+1) = evaluation ordered table of all evaluated parameter combinations
%     where: the column r+1 contains the fitness
%   sorted_table = the same table sorted by increasing fitness
%   SI         = reordering matrix such that sorted_table=table(SI,:)
%
% Ryan Thomas
% Copyright 1999

global lcurve;
global input;
global output;
global allpumps;
global tableindex;
tableindex=1;

if rem(popsize(1,1),2)~=0
  popsize(1,1)=popsize(1,1)+1;
end

%normal random starting pop
pop=rand(popsize(1,1),size(popmin,2));
a=(popmax);
for i=1:popsize(1,1)
  pop(i,:)=a.*pop(i,:);
end
```

```
pop=ceil(pop);
for i=1:popsize(1,1)
  for j=1:size(pop,2);
    k=rand;
    if k<.5
      pop(i,j)=-pop(i,j);
    end
  end
end

tablesize=popsize(1,1)*maxgens;
if usetable==1
table=zeros(tablesize,size(popmax,2)+1);
else
  table=zeros(tablesize,1);
end
h1=gcf;
h2=figure(2);
[fitness,table,pop]=evaluate(pop,table,usetable);
i=1;
progress=zeros(1,maxgens+1);
best_initial_fitness=max(fitness)
progress(1,1)=best_initial_fitness;
while max(fitness)<goal
  gen=i                    %only used to diplay progress to console
    nextgenind=select(fitness,popsize);
    newpop=crossover(pop,nextgenind,xrate);
    pop=mutate(newpop,murate,popmax,popmin);
  [fitness,table,pop]=evaluate(pop,table,usetable);
  best_fitness=max(fitness)
  progress(1,i+1)=best_fitness;
  ProgressHandle=findobj(h1,'Tag','Progress Display');
  set(ProgressHandle,'String',best_fitness);
  if i>5
    if progress(1,i+1)==progress(1,i-1)
      disp('convergence correction');
      tableindex=tableindex-popsize(1,1);
      w_ind=find(fitness==min(fitness));
      temp=floor((1+popmax).*rand(1,168));
      for newi=1:168
        newk=rand;
        if newk<.5
          temp(1,newi)=-temp(1,newi);
        end
```

```
         end
         pop(w_ind(1,1),:)=temp;
         fitness(w_ind)=max(fitness)-.01*abs(max(fitness));
       end
     end
     rmplot(progress,fitness,pop,i,maxgens)
     i=i+1;
     if i>maxgens
       break
     end
   end
   if usetable==1
     best=sorted_table(size(sorted_table,1),[1:size(table,2)-1]);
   else
     top=max(fitness);
     bestind=find(fitness==top);
     best=pop(bestind(1),:);                   %only one of best
   end
   [sorted_table,SI]=sortrows(table,size(table,2));
```

Source Code for Select.m

```
function nextgenind=select(fitness,popsize);
% nextgenind=select(fitness,popsize)
% SELECT chooses population members for reproduction based on fitness
%
% Inputs:
%   fitness (N x 1) = fitness of each population member
%   popsize(1x2) = [population size ,selection type]
% Outputs:
%   nextgenind (N/2 x 2) = indices of parents
%
%  selection types: 1 == roulette wheel selection
%                   2 == stochastic universal sampling
%                   3 == linear ranking
%                   4 == tournament selection
%
%
% Ryan Thomas
% Copyright 1999

N=popsize(1,1);

if popsize(1,2)==1     %roulette wheel
  best=max(fitness);
  best=find(fitness==best);
  nextgenind=[best(1,1),best(1,1)];        %elistist
  f2=fitness/sum(fitness);
  wheel=zeros(popsize(1,1),1);
  wheel(1)=f2(1);
  for i=2:size(fitness,1)
    wheel(i)=wheel(i-1)+f2(i);
  end
  spins=rand(popsize(1,1)-2,1);
  nextgenind=[best(1,1),best(1,1)];
  for i=1:2:size(spins,1)
    ind1=find(wheel>=spins(i));
    ind2=find(wheel>=spins(i+1));
    nextgenind=[nextgenind;ind1(1,1),ind2(1,1)];
  end
end

if popsize(1,2)==2     %stochastic universal sampling
  best=max(fitness);
```

```
    best=find(fitness==best);
    nextgenind=[best(1,1),best(1,1)];          %elistist
    f2=fitness/sum(fitness);
    wheel=zeros(popsize(1,1),1);
    wheel(1)=f2(1);
    for i=2:size(fitness,1)
      wheel(i)=wheel(i-1)+f2(i);
    end
    spins=zeros(popsize(1,1)-2,1);
    dist=1/(popsize(1,1)-2);
    spins(1)=dist*rand(1,1);
    for i=2:(popsize(1,1)-2)
      spins(i)=spins(i-1)+dist;
    end
    nextgenind=[best(1,1),best(1,1)];
    for i=1:2:size(spins,1)
      ind1=find(wheel>=spins(i));
      ind2=find(wheel>=spins(i+1));
      nextgenind=[nextgenind;ind1(1,1),ind2(1,1)];
    end
  end

  if popsize(1,2)==3                   % linear ranking
    best=max(fitness);
    best=find(fitness==best);
    nextgenind=[best(1,1),best(1,1)];          %elistist
    oldfitness=fitness;
    [f2,ind]=sortrows(fitness);
    for i=1:(popsize(1,1))
      fitness(ind(i))=popsize(1,1)+1-i;
    end
    f2=fitness/sum(fitness);
    wheel=zeros(popsize(1,1),1);
    wheel(1)=f2(1);
    for i=2:size(fitness,1)
      wheel(i)=wheel(i-1)+f2(i);
    end
    spins=rand(popsize(1,1)-2,1);
    nextgenind=[best(1,1),best(1,1)];
    for i=1:2:size(spins,1)
      ind1=find(wheel>=spins(i));
      ind2=find(wheel>=spins(i+1));
      nextgenind=[nextgenind;ind1(1,1),ind2(1,1)];
    end
```

```
   fitness=oldfitness;
end

if popsize(1,2)==4      %tournament
   best=max(fitness);
   best=find(fitness==best);
   nextgenind=[best(1,1),best(1,1)];        %elistist
   a=popsize(1,3);                    %number of combatants
   for i=1:2:(N-2)
      %select combatants
      tnext=[];
      for k=1:2
         temp=-10e10*ones(popsize(1,1),1);
         for j=1:a
            b=ceil(popsize(1,1)*rand(1,1));
            temp(b)=fitness(b);
         end
         %select best
         t2=max(temp);
         ind=find(temp==t2);
         tnext=[tnext,ind(1,1)];
      end
      nextgenind=[nextgenind;tnext];
   end
end
```

Source Code for Crossover.m

```
function newpop=crossover(pop,nextgenind,xrate)
% newpop=crossover(pop,nextgenind,xrate)
% CROSSOVER begins the reproduction process
% Inputs:
%   pop (Nxr) = current population
%   nextgenind (N/2 x 2) = parents selected for reproduction
%    xrate  (1x2) = [crossover rate (<1.0) ,crossover type]
% Outputs:
%   newpop (Nxr) = intermediate population
%
%  Crossover types: 0 == discrete
%                          n == n point
%
% Ryan Thomas
% Copyright 1999

[N,n]=size(pop);


if xrate(1,2)==0               %discrete
  a=0;
  for i=1:2:(N-1)
    a=a+1;
    d=nextgenind(a,1);
    e=nextgenind(a,2);
    g=rand;
    if g<xrate(1,1);
      k=round(rand(1,168));
      m=round(rand(1,168));
      k1=find(k==1);
      k0=find(k==0);
      m1=find(m==1);
      m0=find(m==0);
      newpop(i,k1)=pop(d,k1);
      newpop(i,k0)=pop(e,k0);
      newpop(i+1,m1)=pop(d,m1);
      newpop(i+1,m0)=pop(e,m0);
    else
      newpop(i,:)=pop(d,:);
      newpop(i+1,:)=pop(e,:);
    end
  end
```

```
end

if xrate(1,2)>=1                % n point
  a=0;
  for i=1:2:(N-1)        %thru pop
    tpop1=zeros(1,168);
    tpop2=zeros(1,168);
    flag=0;
    a=a+1;
    g=rand;
    d=nextgenind(a,1);
    e=nextgenind(a,2);
    k=ceil((n-1)*rand);
    if g<=xrate(1,1)
      tpop1=[pop(d,1:k),pop(e,k+1:n)];
      tpop2=[pop(e,1:k),pop(d,k+1:n)];
    else
      newpop(i,:)=pop(d,:);
      newpop(i+1,:)=pop(e,:);
      flag=1;
    end
    if flag==0
      for points=2:xrate(1,2)          %number of xover points
        k=ceil((n-1)*rand);
        tpop3=[tpop1(1,1:k),tpop2(1,k+1:n)];
        tpop4=[tpop2(1,1:k),tpop1(1,k+1:n)];
        tpop1=tpop3;
        tpop2=tpop4;
      end
      newpop(i,:)=tpop1(1,:);
        newpop(i+1,:)=tpop2(1,:);
    end
  end
end
```

Source Code for Mutate.m

```
function pop=mutate(newpop,murate,popmax,popmin)
% pop=mutate(newpop,murate,popmax,popmin)
% MUTATE completes reproduction
%
% Inputs:
%   newpop (Nxr) = intermediate population
%   murate (1x2) = [mutation rate (<<1.0) ,mutation type]
%   popmax (1xr) = the maximum acceptable parameter string
%   popmin (1xr) = the minimum acceptable parameter string
% Outputs:
%   pop    (Nxr) = next generation population
%
% mutation types: 1 == small randn
%             2 == invade neighbor w/10% random replace
%                  <<allpumps must be defined>>
%             3 == random replace <<must manually change range>>
%             4 == randperm(8) only one spot  <<for 8 queens>>
%             5 == randperm(8) all <<for 8 queens>>
%
%
% Ryan Thomas
% Copyright 1999

global allpumps

[N,n]=size(newpop);
if murate(1,2)==1      %small normal(+/- < ~5) integer movement
  for i=1:N
    for j=1:n
      k=rand;
      if ((k < murate(1,1)) & (i>1))%skips 1 copy of best
        a=newpop(i,j)+(floor(3*randn));
        if a>popmax(1,j)
          a=popmax(1,j);
        end
        if a<popmin(1,j)
          a=popmin(1,j);
        end
        pop(i,j)=a;
      else
        pop(i,j)=newpop(i,j);
      end
      end
    end
  end

  if murate(1,2)==2       %neighbor replacement w/10% random replace
    for i=1:N
      for j=1:n
        k=rand;
        if ((k < murate(1,1)) & (i>1))%skips 1 copy of best
          if ((j~=1) & (j~=n))
            k=rand;
            if k<.45
              pop(i,j)=newpop(i,j-1);
            else if k>.55
                pop(i,j)=newpop(i,j+1);
              else
                if allpumps==4
                  pop(i,j)=floor(9*rand);
                  if pop(i,j)>=5;
                    pop(i,j)=4-pop(i,j);
                  end
                  if pop(i,j)==-5
                    pop(i,j)=-4;
                  end
                elseif allpumps==1
                  pop(i,j)=floor(3*rand);
                  if (pop(i,j)==2 | pop(i,j)==3)
                    pop(i,j)=-1;
                  end
                end
              end
            end
          else
            if j==1
              pop(i,j)=newpop(i,2);
            else
              pop(i,j)=newpop(i,n-1);
            end
          end
        else
          pop(i,j)=newpop(i,j);
        end
      end
    end
  end

  if murate(1,2)==3       %random selection from [-4,4]
    for i=1:N
      for j=1:n
        k=rand;

        if ((k < murate(1,1)) & (i>1))%skips 1 copy of best
          pop(i,j)=floor(9*rand);
          if pop(i,j)>=5;
            pop(i,j)=4-pop(i,j);
          end
        else
          pop(i,j)=newpop(i,j);
        end
      end
    end
  end

  if murate(1,2)==4       %generate randperm(8) and replace cor. j
    for i=1:N
      for j=1:n
        k=rand;

        if ((k < murate(1,1)) & (i>1))%skips 1 copy of best
          t=randperm(8);
          pop(i,j)=t(j);
        else
          pop(i,j)=newpop(i,j);
        end
      end
    end
  end

  if murate(1,2)==5       %generate randperm(8)
    for i=1:N
      k=rand;
      if ((k < murate(1,1)) & (i>1))%skips 1 copy of best
        pop(i,:)=randperm(8);
      else
        pop(i,:)=newpop(i,:);
      end
    end
  end
```

Source Code for Evaluate.m

```
function [fitness,table,pop]=evaluate(pop,table,usetable)
% [fitness,table]=evaluate(pop,table)
% EVALUATE calls ObjFun.m for each member of the pop that
%  has not previously been evaluated.
%
% Inputs:
%   pop (Nxr) = population
%   table(Kxr+1) = list of all results for repeat checking
% Outputs:
%   fitness(Nx1) = fitnesses of the population members
%   table        = the updated table
%
%
% Ryan Thomas
% Copyright 1999

global tableindex;
[N,n]=size(pop);
fitness=zeros(N,1);

for i=1:N
  [tr,tc]=size(table);
  if usetable ~=0
    for tind=1:tr
      a=table(tind,[1:(tc-1)]);
      b=pop(i,:);
      if a==b
        fitness(i,1)=table(tind,tc);
        found=1;
        break;
      else
        found=0;
      end
    end
    if found == 0
      [fitness(i,1),pop(i,:)]=ObjFun(pop(i,:));
      table(tableindex,:)=[pop(i,:),fitness(i,1)];
      tableindex=tableindex+1;
    end
  else
    [fitness(i,1),pop(i,:)]=ObjFun(pop(i,:));
```

```
      table(tableindex,:)=fitness(i,1);
      tableindex=tableindex+1;
    end
  end
```

Source Code for ObjFun.m

```
function [score,solution]=ObjFun(solution)
% Raccoon Mountain Pumped Storage System Model
% Accepts solution = weekly generating schedule, 168 integers on [-1,1] or [-4,4]
% Returns score = weekly operating income
%
% Ryan Thomas
% copyright 1999

global lcurve;
global capacity;
global cost;
global minhead;
global output;
global input;
global allpumps;

hp=1672;
bank=0;
d=zeros(1,168);
for n=1:size(solution,2),
  if solution(n)<0
    g=1.32407*solution(n)*(5-allpumps);
  else
    g=1.7875*solution(n)*(5-allpumps);
  end
  a=lcurve(n);
  if solution(n)<0
    c=solution(n)*input(round(hp)-1529)/allpumps;
  else
    c=solution(n)*output(round(hp)-1529)/allpumps;
  end
  d(n)=a-c;
  nhp=hp-g;

  while ((hp-g)<minhead(n+1) )
    solution(n)=solution(n)-1;
    if solution(n)<-4
      disp('error')
    end

    if solution(n)<0
      g=1.32407*solution(n)*(5-allpumps);
      else
        g=1.7875*solution(n)*(5-allpumps);
      end
      if solution(n)<0
        c=solution(n)*input(round(hp)-1529)/allpumps;
      else
        c=solution(n)*output(round(hp)-1529)/allpumps;
      end
      d(n)=a-c;
      nhp=hp-g;
      p=p+1;
    end
    p=1;
    while ((hp-g)>1672 )
      solution(n)=solution(n)+1;
      if solution(n)<0
        g=1.32407*solution(n)*(5-allpumps);
      else
        g=1.7875*solution(n)*(5-allpumps);
      end
      if solution(n)<0
        c=solution(n)*input(round(hp)-1529)/allpumps;
      else
        c=solution(n)*output(round(hp)-1529)/allpumps;
      end
      d(n)=a-c;
      nhp=hp-g;
      p=p+1;
    end
    if(round(d(n)-capacity(1))<=0)
      disp('error')
      a,c,solution(1,n),d(n),nhp
    end

    e=cost(round(d(n)-capacity(1)));        % to GW
    f=c*1000*e;                          %from cents/kwhr to cents/MWhr
    hp=hp-g;
    bank=bank+f;
  end

  total_bank=bank/100;          % to dollars

  score=total_bank/1000;        % to thousands of dollars
```

Source Code for Rmplot.m

```
function RMplot(progress,fitness,pop,i,maxgens);
% Controls Progress Display for Raccoon Mountain Fitness Function
%
% Ryan Thomas
% Copyright 1999

global allpumps;
global input;
global output;
global lcurve;
subplot(3,1,1), plot([0:i],progress(1,[1:i+1]));
    title('Genetic Algorithm Progress by Generation')
  ylabel('Best Weekly Profit')
  xlim([0 maxgens]);
  best_ind=find(fitness==max(fitness));
  best=pop(best_ind,:);
  tt=size(best,2)+1;
        headlevel=zeros(1,tt);
        headlevel(1)=1672;
   for ii=2:tt,
   if best(ii-1)<0
    g=best(ii-1)*1.32407*(5-allpumps);
   else
     g=best(ii-1)*1.7875*(5-allpumps);
   end
    headlevel(ii)=headlevel(ii-1)-g;
   end

subplot(3,1,2),plot(headlevel);
  xlim([1 169]);
  ylim([1527 1675]);
  title('Raccoon Mountain Utilization by Hour')
    ylabel('Upper Reservoir Water Level')
    d=zeros(1,168);
    hp=1672;
    for n=1:size(best,2),
    a=lcurve(n);
      if best(n)<0
      c=best(n)*input(round(hp)-1529)/allpumps;
    else
        c=best(n)*output(round(hp)-1529)/allpumps;
    end
    d(n)=a-c;
  end

subplot(3,1,3),plot(d)
  hold on
  plot(lcurve,'-.')
  xlim([1 169]);
  ylim([8000 30000]);
  title('Effects of Raccoon Mountain Operation by Hour')
    ylabel('Demand (MW)')
    %xlabel('Hour of the Week')
  hold off

drawnow
```

VITA

Ryan Thomas was born in Kingsport, Tennessee on January 14, 1975. He attended Dobyns-Bennett High School, and graduated in 1993. This same year he enrolled in the University of Tennessee, where he obtained a Bachelor of Science degree in Mechanical Engineering in December 1998. Having always been interested in Robotics and Artificial Intelligence, he continued on at the University of Tennessee to pursue a Master of Science degree in Engineering Science with a concentration in Applications of Artificial Intelligence. In January 1999 he also began work as a graduate research assistant in the Nuclear Engineering Department, working with Dr. Robert E. Uhrig and Dr. J. Wesley Hines.