2006

Faculty scheduling using genetic algorithms

Kevin Soule

Faculty Scheduling Using Genetic Algorithms
Rochester Institute of Technology
Department of Computer Science
Master's Project

Kevin Soule
ksoule@monroecc.edu

April 2, 2006

Abstract

The problem of developing a class schedule for a department of faculty has been proven to be NP-complete. Therefore when the schedule is large enough, finding just one feasible solution can be impossible for any direct search algorithm within a reasonable time. This project is geared toward investigating the possibility of using genetic-based algorithms to solve faculty scheduling problems of 100 courses or larger quickly. Multiple versions of genetic algorithms and heuristics are tested. Many parameter levels for these algorithms are optimized for fastest convergence.

Project Committee

Chairman: *Dr. Peter G. Anderson*
Reader: *Dr. Stanisław P. Radziszowski*
Observer: *Dr. Roxanne Canosa*

Contents

1 Introduction

1.1 Goals of Project

This project takes a look at a specific scheduling problem involving faculty assignments for a medium to large size department within a university or college. An examination is done on how difficult it is to actually come up with a working schedule within a reasonable time. The amount of input data and constraints used towards making a viable schedule are investigated. The use of a genetic algorithm is attempted to generate schedules, and many parameters of that algorithm are manipulated experimentally. Additional heuristics within the algorithm are attempted and compared to one another.

Five separate genetic algorithms are generated and tested against each other for best features to solving the scheduling problem. Conclusions and thoughts are given on why some did in fact work better then others. Much discussion is also done on developing genetic algorithms. This helps in determining their usefulness to replacing the guaranteed solution but time consuming direct algorithm.

All programs were created using the C++ programming language and the algorithms are described in detail. All tests and comparisons were preformed on identical personal computers with matching processing speed and hardware. This was done to allow us to get reasonable approximations of the actual time required to successfully come up with solutions given today's range of consumer machines.

1.2 General Scheduling Problems

The encyclopedia defines scheduling as the process of assigning tasks to a set of resources. It is an important concept in many areas such as computing and manufacturing processes.

In everyday life we know scheduling slips into our daily routine. We have to pick the kids up at 3pm, we have an appointment from 1-2pm and when can we schedule a free hour to pick up groceries? We perform these calculations every day without even thinking about it.

In mathematical terms, a scheduling problem is often solved as an optimization problem, with the objective of maximizing a measure of schedule quality. For example, in order to minimize cost, a company might wish to minimize the amount of workers needed at any one time and still produce a desired amount of goods.

Scheduling is a key concept in multitasking and multiprocessing operating system design and in real-time operating system design. It refers to the way processes are assigned priorities in a priority queue. This assignment is carried out by software known as a scheduler.

Workshops can be another great example of scheduling problems. A company has limited machines to work on, a set of employees with different work skills, and a large back-order of parts to create. How to schedule the use of those

machines when and by who, while not having any down time on a machine or any worker sitting and waiting their turn to use one. These type of problems can get complicated quickly.

1.3 Faculty Scheduling Problem

All of the scheduling examples mentioned above have similar problems. But different sets of inputs and restrictions give each a unique set of hurdles to overcome. Scheduling courses for a school or university is discussed below. As the list of required inputs and restrictions is discussed you will see this problem is no exception.

1.3.1 Scheduling Issues

Normally a school starts with a set amount of classes offered for students (usually based on the previous year's enrollment). This set becomes dynamic as students sign up and additional sections are added and some classes with little interest are cancelled. These changes can happen right up to the last couple of weeks before a given semester begins. The cancellation or addition of classes can also have a domino effect on the current schedule, forcing certain professors to change to other classes, which then bumps others with less seniority. Additional lab rooms may also be needed, which could then force a class that really did not need to be in a particular room to be moved.

Most schools most likely start with last year's schedule and then spend a lot of time tweaking the schedule every time a conflict or change arises. After many years of these small changes, this new schedule could be compounded with new problems that formed from trying to fix others. Fixing all of these may not be possible by simply updating the current version. Yet administrations do not have the time to start over from scratch with such a dynamically changing problem.

Particular schools' resources actually change relatively slowly from year to year. This leads to the idea that an overall data base of current rooms available, lists of professors available, and set of courses offered could be created and kept up to date. This data base, if current, would be the first step in helping to develop other scheduling choices. If a program could be developed to quickly rewrite an entire schedule that uses this data base, the administration would be able to get rid of the old schedule that has gone through many revisions and replace with a fresh new one that may incorporate many new criteria.

Although from a student perspective, a schedule must be mostly set months in advance in order to be able to plan for the courses that they can sign up for and there can't be a major overhaul in the entire schedule in the weeks right before it starts. So it appears the best promise for a schedule maker would be to create a new one from scratch each year based on all relevant changes since the previous year. The schedule maker would still be forced to deal with those

last minute updates. At least in this case, they would be starting with a good schedule with the best features that the program could find.

1.3.2 Input Required

A school has a limited amount of classrooms, labs, and lecture halls to hold courses in. Many full-time professors have only specialized areas of expertise within which they can teach, and have limits to how many classes they can take on. Courses tend to have differing sizes, room requirements, lengths, credits, and even some restrictions on which other classes cannot be taught at the same time. All of these features must be present in any successful schedule and many times a school will only have a few days to finalize the finished version.

Minimum inputs required to generate one schedule:

General Items:

- Time blocks/slots for school (Schedule a class anytime between 8am and 10pm starting at 30min intervals)
- Courses to schedule
- Professors available
- Rooms/labs available

For each course to schedule:

- Length each meeting or credits
- Number of days a week (one-five)
- Room restrictions (lab or lecture)
- Number of sections of that course

For each room on campus:

- Type of room (lab or lecture)
- Size of room
- OR (list of which courses can be held there)

For each professor:

- Teaching abilities (which courses does he/she know)
- Number of credits he/she can teach
- Preferences of early/late classes
- Preferences of which courses he/she likes to teach

1.3.3 Finding the Perfect Schedule

It does not take much calculation to see that with all these many choices, even a small schedule, with perhaps only a few courses to place, has potentially billions upon billions of different permutations of schedules. With such a small example many of these would probably be "good" schedules. But with larger and larger problems, there are exponentially more restrictions placed on each course, until the point is reached that there are no possible schedules to meet all of the initial conditions. Since schools have limited time and a computer running through the almost infinite number of variations of schedules would take potentially many years, some sort of estimation or close approximation of a good schedule is needed quickly. A genetic algorithm has shown a lot of promise in severely reducing processing time. The trade off is that the best result is not guaranteed.

1.4 Basic Genetic Algorithm Description

An algorithm is defined as a step-by-step problem-solving procedure, used in order to solve a problem in a finite number of steps. However, many problems still exist which no algorithms can solve in a reasonable number of steps. Many interesting problems known are classified as NP-complete. A few examples include the travelling salesman problem, subset sum problem, SAT problems, and various scheduling problems [4]. These problems all can be solved, but not with any known algorithm which runs in polynomial time. The only solutions found so far are exponential in nature, which means any of these problems with a large amount of inputs could take computers an unrealistic amount of time to solve. With no feasible method of solution available, other options must be explored to solve these types of problems. This document is going to explore one such method called the genetic algorithm.

One of the features that make NP-complete problems so difficult is that the search space for potential solutions is sometimes so large that not every possible solution can be checked in our lifetime. A genetic algorithm narrows down this potentially large search space by using iteration and evolution techniques to jump over large patches of unlikely candidate solutions.

The basic idea of a genetic algorithm is to begin with an initial set of possible solutions, which can be generated either randomly or by some sort of educated guesses. Each one of these solutions can be referred to as a chromosome (or genome) and the initial set of solutions form the population for the algorithm. Every member of the population is given a rating as to how good a solution it is (the fitness). Most of these initial solutions are bad and seemingly give us no help as to how to find the best solution. However even a bad solution may have one small key ingredient that is needed in finding the best one. This is where genetic algorithm gets its name. At this point a couple of solutions are chosen from among the relatively better ones in the population and a form of gene-splicing occurs. In other words a random part of one solution is grabbed and placed within another. With every iteration, the algorithm creates new solutions from parts of existing ones. These new solutions are then placed into

the population replacing some of the worst solutions that currently reside there, thereby slowly giving favor to better and better fitness values.

It would seem that a lot depends on whether or not the initial set of solutions contains the necessary genetic material needed to find the optimum solution. Most likely with a large search space this would not be the case, therefore additional genetic material must be added occasionally. This can be done in the form of mutation. Every so often, at random intervals a gene from one of the solutions is altered in the hopes that we mutate in a positive direction. Most times we will worsen, but every once in a while we may hit the jackpot and find a gene addition that makes our chromosomes much closer to our best solution. This gives us almost the same model as the theory of evolution in which lower orders (bad fitness) can be evolved into higher beings (good fitness)[6].

1.4.1 Representation (The genome)

Some restrictions became apparent on how our solution (or genome) needed to be represented to work well in our program.

- Can be programmed easily in a computer
- Easily check the fitness level for that solution
- Able to create random solutions
- Cannot represent more than one solution
- Every solution can be represented
- Able to perform mutations
- Easily use crossover techniques (gene splicing)

Many different representations for the individual genomes in the genetic algorithm can be used. Many researchers work mainly with strings of bits as they are the easiest to manipulate, but you can use arrays, trees, lists, or other objects. Whatever representation you use, you must program all the genetic operations to fit this model.

1.4.2 Population

The population is the set of chromosomes that you always have stored on hand. The larger number of these solutions that you start with increases your genetic material to work with, but also increases the time needed to process though all of them. A balance must be found to get enough diversity, but still processes in a reasonable amount of time. This value is determined by experimentation to attempt to optimize solution time of a given problem. Each particular problem may have its own value of best population size depending upon how the program is written, or population size may not have any large effect on solutions. This is one parameter we measure experimentally for our particular genetic algorithm.

1.4.3 Crossover: Combination of Genetic Material

How one could perform the gene-splicing of the chromosomes has almost limitless choices, but is limited to how the original solution is represented. A bit string representation is easy, any section of the bits on one solution could be taken and replaced with the corresponding values on another solution. With an array or object representation one would probably be limited to taking a whole cell or cells in the transfer. There also are many choices for which "genes" should be taken. Since there is no knowledge at the genetic level for what each gene is responsible for, it seems appropriate to have some sort of random choice as to how this is done. But how many changes or what percentage of changes that would give the best diversity is a question for experimentation. This form of gene-splicing is called crossover. The comparison of different crossover types when looking for the quickest or best solution is also an item that is tested for in our experiments.

1.4.4 Mutations

Mutation is another random process in evolution necessary for improvements to take place. The amount of mutation is a serious question for a genetic algorithm. Too little mutation and improvements will come at a very slow rate and the algorithm will not converge fast enough to good solutions. Too much mutation could cause potential improvements to get altered before they can get incorporated into the population. This question of mutation rate is best characterized by breaking it up into two separate components. First, the rate at which individual chromosomes are picked to be mutated must be established. And second, after a chromosome is picked, how much of that chromosome is to be altered? These two ideas may not be mutually exclusive; by increasing one, we may need to decrease the other for similar results. A third question on mutation rate may also include, how many members of the population are to be mutated at one time, giving a more generational approach to mutation. All of these ideas must be explored in order to find any optimum mutation rates toward generating the best solutions.

1.4.5 Fitness Calculation

Being able to calculate how "good" a potential solution is is essential to genetic algorithms. If it is not possible to quickly test this solution for its level of correctness, or at least rank it against others, getting a population to converge in the right direction would not be possible. Being able to represent a solution and then quickly test the fitness of that representation are the first concepts that should be explored before attempting any such genetic algorithm. In order to be able to check the level of this fitness, criteria must be established that represents the ideal solution. Each potential solution can then be compared against a list of criteria, and either a percentage of the criteria met can be reported or a list of failed criteria could be shown. These criteria can also be weighted as to importance for the ideal solution and larger rewards can be given

to solutions that meet the most important points. Alternatively large demerits can be given to those solutions that fail those most important points. Either way a fitness level (or value) can be established allowing for ranking among the population. Once a population is ranked, it is easy to remove bad solutions and keep the better ones.

1.4.6 Evolving to a "Best" Solution

It is one thing to know the criterion that make up an ideal solution to a problem, but knowing that a solution exists that meets these criteria is quite another problem. Using a genetic algorithm forces one to understand that ideal solutions may not exist and settling for a good but suboptimal solution may be the best option. On the other hand, particular sets of input and criteria may result in many different but ideal solutions, all equally good, and we may only be able to find some of these using our algorithm. With this knowledge, we can not just turn on our program and wait for the ideal solution to pop out. Some sort of stopping criteria must be present. Physical time, percentage of best fitness or count of iterations (evolutionary steps) are all possible ways to stop from being forced to wait years for a perfect solution, when 99 percent of all criteria might have been met in the first five minutes or a point has been reached where improvements no longer happen.

2 Faculty Scheduling with Genetic Algorithms

2.1 Schedule Representation

When giving thought on how to represent a single schedule in memory, the first and most likely choice was, as mentioned before, using a bit string. A string of ones and zeros that, when decoded, became a unique schedule that cannot be formed from any other different string. This led to some interesting problems that had to be dealt with. First and foremost, since the number of professors/rooms/courses to schedule is variable, our string should also be of variable length. Limiting the size to write a standard program would force the program to be only able to handle a maximum number for each parameter. We would not want to decide up front that we would allow 8 bits of storage per class to assign a professor. That would put limitations on our program to solving problems that have less then 256 professors.

In addition, any parameter that does not happen to have a number of choices equal to that of a power of two, would add extra choices that would have to be dealt with. For example, a particular schedule might have nine professors to assign. A minimum of 4 bits to store would be needed for each class assignment to pick a professor. Four bits can represent up to 16 different values, which would mean with mutation, almost half of the time a professor is picked that doesn't exist for that problem. Although this would have the effect of raising the fitness value to make this mutation a poor solution, this would also enlarge the search space that we are trying so hard to reduce. Since we never know

ahead of time the values of our resources, we must either keep this enlarged space or add the extra code needed to alter the value back to one in the range.

A bit string also adds to the complexity of the program by forcing us to encode and decode schedules at every iteration so that we can compare them and generate fitness values for each, thereby adding additional processing time.

Even with these problems using a bit string was still possible, but for the best efficiency an array was chosen to represent a schedule. Inside each cell of the array an individual course can be easily represented as a set of values. Each course has many features to be stored such as room location, time and professor. A standard list of these features was devised and an integer value was given to each member of the list. These integer values can then be decoded by using a look up table and checking which room or which professor it represents.

2.1.1 Information stored for each schedule

Every course that is offered has a set of parameters required to be assigned in order for it to be placed inside a schedule. As a course is decided to be offered, we know some standard features already that are unique to that course. But there are also subsets of unknown values that have to be determined based on a given attempted schedule.

The known values include:

type the name of the class

section the section number of the class (many classes have multiple sections)

credit number of credit hours for that particular class (the number of hours of class time per week)

days offered the number of days of the week is it offered (1-5) (a lab meets one day a week, a lecture could meet two or three)

The variable values for the class include:

professor the name of the professor to teach this class

room the location where the class is held

days which actual day(s) of the week the class is held

start starting time of the class (end time is determined by the credits and days held)

These eight different parameters need to be given values for every course in the schedule to have enough information to be able to test the schedule for conflicts. All of the values stored for these parameters are integers, and many represent an item found in a look-up table elsewhere. For the ease of programming later, the course names, professor names, and room assignments are left as simple integers. For a more advanced version of the program, the final schedule that was found could easily convert these values into readable names. Section numbers, course credits, and number of days per week were also integers that exactly matched what was looked for so no changes need be done. The days of the week variable and the start times had to be converted with the relationships below.

Code for days of the week:
$0 - MTWRF$
$1 - MTWR$
$2 - MWF$
$3 - MW$
$4 - WF$
$5 - TR$
$6 - M$
$7 - T$
$8 - W$
$9 - R$
$10 - F$

Code for start times:

$0 - 8:00am$	$9 - 12:30pm$	$18 - 5:00pm$
$1 - 8:30am$	$10 - 1:00pm$	$19 - 5:30pm$
$2 - 9:00am$	$11 - 1:30pm$	$20 - 6:00pm$
$3 - 9:30am$	$12 - 2:00pm$	$21 - 6:30pm$
$4 - 10:00am$	$13 - 2:30pm$	$22 - 7:00pm$
$5 - 10:30am$	$14 - 3:00pm$	$23 - 7:30pm$
$6 - 11:00am$	$15 - 3:30pm$	$24 - 8:00pm$
$7 - 11:30am$	$16 - 4:00pm$	
$8 - 12:00pm$	$17 - 4:30pm$	

Some obvious assumptions were made with these two parameters. Not all possible days of the week combinations are allowed, and start times are only allowed between 8am and 8pm and only on the half hour. These restrictions were placed to restrict final schedules to be somewhat normalized and matched up with what most colleges and universities currently do.

With all eight parameters set for each course, the goal is now to place them altogether in a schedule. As mentioned before, an array accomplishes this feat with no problem. Using an array allows for easy parsing through the entire course list one by one to check for conflicts.

2.1.2 Testing for Fitness

A perfect schedule would contain no room conflicts or time conflicts and even would not violate any professors preferences. the fitness of a schedule is a numerical value representing how close to a perfect working schedule we happen to be. Some of the things that make up a poor schedule are impossible conflicts of resources, such as a schedule that forces a professor to be teaching multiple classes at the same time, or a room that is scheduled with overlapping classes. These types are "hard" constraints that have to be repaired in order to get to a working schedule. But there are other constraints, "soft" constraints, such as forcing a professor to teach one extra class or to give a professor an 8am class on the same day as an 8pm class. These constraints along with other professor preferences are not as important, but still can make many people unhappy with the schedule.

When a list of these constraints is determined along with the weight of importance for each, a method for comparing schedules can be developed. Although other methods can be used, the addition of demerits of a schedule can be very effective in determining the fitness value. Using this method, a schedule would start with a fitness value of zero, and each instance found of a constraint violated would add the demerit value associated with that constraint. After all constraints were tested the total value is connected to that schedule as its fitness. Then each schedule generated could be easily compared with other schedules simply by looking at which one is closer to zero. If a fitness value of zero is ever found, we can say that we have found a perfect schedule (at least according to the constraints checked).

2.2 Constraints

There are limitless numbers of constraints that can be placed on a schedule in computing in this form and would most probably be different for each college's wants and needs. For experimentation, thirteen of the most likely constraints were picked to use as the basis for a perfect schedule. There is no claim that these are the only ones, nor is there any claim that all of these would be important to any scenario. Observation of a department schedule at a local community college added to the importance of some of these constraints. These constraints are broken up into two separate types, the impossible conflict of resources mentioned above, which are called the hard constraints. The preferences and undesired features fall into the category of soft constraints.

2.2.1 Hard Constraints

The five hard constraints tested for in our experimentation are as follows:

- A professor can only teach one class at a time.
- A room can only hold one class at a time.
- A professor can only teach classes they were trained for. (i.e. an English professor can't teach a chemistry course)
- A professor can only teach a certain number of classes in a given semester/quarter. (My guess- only up to 20 credit hours)
- A room can only hold a class in it that has the correct equipment. (i.e. Biology lab can not be scheduled in a lecture hall.)

2.2.2 Soft Constraints

Eight additional constraints that were thought to be the most important and needed for a good schedule:

- A professor prefers not to teach evening classes.
- A professor prefers not to teach morning classes.
- A professor prefers not to be scheduled for both a morning and an evening class on the same day.
- A professor prefers not to teach certain classes.
- Different sections of the same class should not be taught at the same time. (Less scheduling options for students)
- Different sections of the same class should be taught by same professor if possible. (Makes for consistency of material and less preparation time for professors)
- A professor prefers not to teach more than a certain number of credit hours.
- A professor prefers not to teach less than a certain number of credit hours.

2.3 Fitness Calculations

Calculating the fitness of each schedule is a matter of adding up all of the constraint violations that have occurred. But not all constraints are of equal importance. The next section will highlight how we can distinguish between the various levels of violation.

2.3.1 Demerit Levels

As the algorithm searches for better schedules it only looks at the fitness values to determine which is best. A schedule with only one or two violations of criteria still may be a bad schedule if those violations happen to be hard constraints. A schedule with many soft constraint violations may yet be a good schedule as long as the hard constraints are all satisfied. During the iterations, as each constraint is found to be violated, a weighted amount of "demerits" are added to the fitness of that schedule. Since the algorithm searches only for overall fitness, it will deem a few hard constraints equal to that of many soft constraints. Thereby, intelligent weighting of these demerits, will give preferences on which constraints to solve first. The higher demerit value that one constraint is allowed to be, should result in that constraint being solved faster.

Hard constraint demerit values:

Hard Constraints	Demerit values
Each instance of a professor teaching more then one class at a time	250 points
Each instance of a room holding more then one class at a time	250 points
Each instance of a professor teaching a class they were not trained for	250 points
Each instance of a professor teaching over his maximum load	250 points
Each instance of a room holding a class it can't accommodate	250 points

Soft constraint demerit values:

Soft Constraints	Demerit values
Each instance of a professor teaching an evening class when they prefer not to	10 points
Each instance of a professor teaching an morning class when they prefer not to	10 points
Each instance of a professor teaching both a morning and an evening class on the same day	10 points
Each instance of a professor teaching a class that they do not want to teach	20 points
Each instance of different sections of the same class being taught at the same time	5 points
Each instance of different sections of the same class being taught by different professors	5 points
A professor teaching x more hours then they wish	2^x points
A professor teaching x less hours then they wish	2^x points

All hard constraints were given an arbitrary demerit value of 250 points. Each time one was found a large jump in fitness was calculated. This value was picked high enough to leave room for other less important but still valuable restrictions. The soft constraints were given differing values, based solely on opinion of importance related to the hard value of 250 points. The last two constraints involving the number of credit hours each professor needs to or can teach is a variable demerit value. The idea was that the farther away from the value of credits needed should have an exponential higher demerit value, so a calculation is done to get the value by taking two to the power of the difference of the number of credits wanted.

2.4 Programming with a Genetic Algorithm

Now that the basic ideas on how to solve the problem have been formulated, they need to be incorporated into a computer simulation. The first decision made was to pick the computer language C++ for all programming. This language was selected for a variety of reasons. The object oriented design of the language was to come in handy when storing individual class information, plus the language is well known and relatively good at solving many different types of problems. This language (with different environments) was also available at school, work and home, making testing of intermediate points convenient.

During a genetic algorithms course taken a few years back, a professor introduced a program written in C that solved problems using bit strings. This program was neatly set up keeping all parameters and parameter values in separate files for easy updating between experiments. The program had separate functions written for mutation, fitness calculation, crossover techniques, and tournaments between chromosomes. Many of the features added into the base program for this project were evolved from this original version.

2.4.1 C++ Object Simulates a Gene

It was decided that an array of objects would represent one schedule, but not what would represent one class. Each object has eight separate parameters that all need values before it can be placed into a schedule. An array could have been chosen again to represent these eight values, but there was additional functionality that was wanted to act on each class. Among many construction issues, mutation and comparison functions were to be used on the classes each iteration. Creating a representation that could store this functionality was desired and a C++ class was created for these reasons. The class type, called Eaclass (each class) for the programs, was created with eight private variables, constructors, accessors, and mutators. Operators and other functions found to be needed could then be added. A copy of this class was added with other code at the end of this document.

In genetic terms, each instance of this class in the program is one single gene in our string of genetic material. All of these genes stranded together (in an

array) form the chromosome to represent one schedule.

2.4.2 Array of Objects Simulates a Chromosome

The standard C++ template library contains an additional class that approximates an array with efficient insert and removal functions built in. This class called Vector was used to simulate an individual schedule, inserting course objects as they are placed. Each completed vector contained an array of Eaclass objects filled with all needed items for each course. This vector represents our chromosome and is compared against other chromosomes in the population, to find the fittest one.

2.4.3 Building of Initial Population

The basic idea of using a genetic algorithm to solve for a solution is to take a portion of genetic material (initial set of chromosomes), then perform some gene-splicing and mutation techniques to attempt to find better solutions. As you find better solutions the old initial ones are discarded in favor of the better ones. Eventually a solution that is good enough to use is found. But nothing has been mentioned of where the initial set of solutions comes from. Here we have a few options.

Since the algorithm is set up to be able to create new genetic material through mutation and recombination of old materials, it should not matter what the initial values are. We should, if lucky enough, always tend to slide towards optimum solutions. For this reason, the initial tests of the program were written with nothing more then a random number generator that filled up the initial chromosomes with random values. A random spread of values was thought to give quite a bit of genetic diversity at the beginning, hopefully saving the best gene combinations for use in the increasingly better schedules.

This idea of using random values was a valid one, but missed a really important idea. We want the initial set of chromosomes to be saturated with as much good genetic material as we can start with. So developing some routines to fill the initial schedules with more educated guesses could greatly reduce our search time. As an analogy, think of a scavenger hunt where you are given five items to find. If you are told the location of two or three of them before you start, you could possibly cut your search time in half.

Random number generating became our control case, with which we could compare all potential initial generating ideas against. But other ideas were devised, tested and compared and are described later in the report.

2.4.4 Picking Good Genetic Material with Tournaments

The goal of this program is to steadily move towards better solutions. Randomly generating schedules will start the program with a large supply of mostly bad values. With the hope that intermixed throughout are a few good pieces of genetic material. The algorithm's job is to find those few good pieces and

combine them into better and larger good sections of schedule until eventually a workable schedule is received.

Each iteration of the program grabs two chromosomes (schedules) and combines them, taking genetic material from each in the hopes of creating at least one offspring with better fitness. The choice of which two chromosomes are picked is not left totally to chance. If completely random choices were made, many previously bad schedules would be reused an would infect the population making forward progress difficult. To limit the reuse of bad material in generating new schedules a technique is used called tournaments.

This technique randomly picks a small sample of schedules from the current population and keeps track of which two have the best fitness and which two have the worst. The best two are used as parents in crossover operations, while the worst two are simply replaced when new children are developed. This happens in every iteration and guarantees that bad chromosomes (that must have bad genes) are thrown away every iteration. Although we are not guaranteed that the new members will be better, as long as every once in a while a good one is found, forward progress will be made.

2.4.5 Creation of New Genetic Member with Crossover

There are many different crossover techniques that can be used and picking which one to use is a well studied issue. When taking two different schedules and combining them, it is wished that a new schedule is created using parts from both originals. What is not wanted is a new schedule that contains the same class placed more then once. If order is constant in every schedule generated then the first gene found in any chromosome in the population will represent the same class. And all genes thereafter will be in the same order. This means for a crossover to successfully generate a new schedule, it simply must conserve the order of the genes.

A section or multiple sections of one chromosome is picked and swapped with the corresponding section(s) on the other winning chromosome. This generates two new chromosomes from the two originals. These two new chromosomes are then placed back in the population to replace the two losers of the last set of tournaments. For example: There are six classes to schedule. Some random process is run to say that class numbers two and three are to be swapped. Then those two classes are taken from each of the two chromosomes and swapped, leaving the other four untouched for both.

With two new chromosomes, each are now tested for fitness against the criteria and those schedules and fitness values are dumped back into the population awaiting there turn in another tournament. Most are undoubtedly worse than either of the winners and will simply be weeded out at the next tournament, but every once in a while a better solution is found and saved.

Three main crossover types were tried for the base program are explained below.

2.4.6 Crossover types

Uniform Uniform crossover gives an equal chance for any gene (course) to be swapped between the two chromosomes. It simply runs through the entire course list, and generates a random number (0 or 1) for each course. If the number generated was equal to one a swap is preformed on that gene, if it was a zero no swap is done.

See example below - Each vertical column represents a class needing to be scheduled. Each letter represent a class scheduled with different parameter values

```
Parent A -   a b c d e f
Parent B -   g h i j k l

random #'s   0 0 1 0 1 1

Child C -    a b i d k l
Child D -    g h c j e f
```

One-Point One-Point crossover generates only one random number up to the number of total classes stored in the schedule. All courses before that number will be swapped, while all courses after are left alone.

```
Parent A -   a b c d e f
Parent B -   g h i j k l

random number = 3

Child C -    g h i | d e f
Child D -    a b c | j k l
```

Two-Point Two Point crossover generates two random numbers in the range of the total number of classes. Then every course between those two values will be swapped and all the rest stay the same.

```
Parent A -   a b c d e f
Parent B -   g h i j k l

random number's = 2,5

Child C -    a b | i j k | f
Child D -    g h | c d e | l
```

2.4.7 Mutation of Genes

Crossover alone is not enough to generate new genetic material, all we are doing is moving around the same subsets of classes. What is needed at some point is an influx of new class placements. The mutation function is a simple function in the Eaclass class, that when called will randomize some of the variables stored for that class. A new start time is picked, a new professor name is given and a new room location is assigned. No thought of how well this course will fit into the schedule is done at this time. It is mainly needed to add new genetic material into the population.

There are difficult questions to answer about how much mutation is needed. A large amount scatters the population too much making a trend towards good solutions harder to reach, yet small mutations are quickly washed away if they are not immediate improvements. Many tests are needed to find out how often mutations should be done, and when done how much of the population should be affected.

2.4.8 Parameters to Optimize

The base program described so far left many unknowns still to find. What is the ideal population size to have enough genetic diversity but still be able to process in a reasonable amount of time? What size tournament explores the search space, giving us good solutions most of the time, but still picking some weaker ones from time to time. Which crossover type works most efficiently to get to our solution? How often should a mutation of a chromosome be preformed? How many genes in that chromosome should be mutated? When building the initial population how can we give the program a good head start towards a solution saving us much processing time.

All of the parameters and techniques discussed above were tested to find good values for the entire list of genetic algorithm methods tried. Plus some additional parameters that seemed to come with some of our new attempts. The next few pages will summarize the five major methods that were tried and how each of them worked.

3 Genetic Algorithm Methods Attempted

Through the many experiments and ideas attempted to find reasonable solutions to faculty scheduling, five major methods were tried and compared. The following section explains them and gives an idea on how each worked.

3.1 Base Program with Random Values

This base program has already been thoroughly explained in the last section, but a couple of extra features and functions that were attempted will be highlighted here.

3.1.1 Build Population

As the program begins, large amounts of input data are needed. A list of courses to schedule is required, a list of professors with their preferences and abilities, and a list of rooms all have to be accessed at the beginning to fill up look-up tables. This program uses three text files filled with integers that are immediately opened and read. If the text files can not be found, no scheduling can be done.

The initial population of genetic material was generated by a random number generator with no intelligent placement whatsoever. As the initialization routine begins, a vector of classes gets developed class by class. Each class that is added is constructed by placing known values from input data, and then randomly picking a room, professor, day, and start time values. When the vector is filled it is placed into the population array and the process repeats until the population size is the amount specified in the parameter file. Fitness values are then computed for the entire population and the evolution is ready to begin.

3.1.2 Evolve to Solution

The algorithm now began to evolve slowly with the tournaments, crossovers and mutations all searching for better schedules. These processes all worked as explained in the previous section. A feature was also placed in the program that printed out any progress of the best fitness found. And the best fitness value at thirty second intervals was stored in an array to keep for comparison and graphing the results later on.

During some of the runs it was noticed that after a while the population seemed to not be very diverse. Printouts of the entire population at certain points proved this to be true. Most of the population contained similar or the same schedules and mutation seemed to be the only thing running the program forward. Because of this, a function was added that whenever the fitness of every member of the population became the same, a mass mutation was enforced. This function was called the mutate_all function. This had a very good effect at first, however later on it was found that increasing the overall mutation rate had the same results.

As the algorithm progresses, stopping criteria are checked. A fitness value of zero will automatically break the program and print out the final schedule. Since finding a perfect schedule is unlikely in larger cases, a limiting factor of number of iterations is also set as a parameter. One additional feature was added later in the process, which consisted of actual real time limits. This was added so that it was possible to let various different programs run for the exact same amount of time and do comparisons.

3.2 Base Program with Greedy Procedure

This was the first attempt to improve upon the good results of the base program. It was a variation that tried to take advantage of starting the population with lots of good genetic material.

3.2.1 Build Population

This method starts the program just as the other, reading in inputs and filling up look up tables. But this time does not blindly place classes one by one. It uses a somewhat greedier technique to place each class.

As a class comes up for placement and the given parameters are filled, a random number generator once again places a room, professor, and time assignment for it. But this time it checks if this random placement will hurt the partial schedule currently being developed. A partial fitness function was added to test each "potential" new class placement before it is placed. If the class can be placed without harming the current schedule it is placed immediately and then on to the next class. But if the new class violates some constraints it is tossed aside and a new random location is tried.

This is done many times until either a good spot is found or we have run into our limiting parameter, called "greedy", that was added into the parameter file. This parameters value represents how many times each class is attempted to be placed before actually doing it. In the event that it has tried the maximum number of times and still had no luck, it is placed at the location of the best one found so far. So that even in failure the damage is limited. A flow chart simulating this algorithm is included in section 10.

A definite flaw has to be pointed out with this method. If a schedule is always generated class-by-class, does the order of when to place each class matter? The answer to this seems to be yes. What if all classes at the end of the list are three hour long labs? All of the large blocks will be gone by then. To fix this oversight, a permutation was used to randomly order which class is scheduled when so that it has equal chance of finding a good order.

Obviously, the larger value that this "greedy" parameter is, the better chance we have to set a good schedule. But because of the increased processor demands, at some point there has to be the point of diminishing returns. This was the focus of many experiments for the method.

3.2.2 Evolve to Solution

This method takes the greedy base population and evolves exactly as before, hoping that the better supply of genetic material forces faster solutions.

3.3 Ordered Greed Program

Ordered Greed is a well publicized idea [2] that tends to force genetic algorithms to good solutions fast. So it is not a major surprise that it was one of the first attempts to radically improve our results. The idea of Ordered Greed is to methodically place each class in the schedule into the first spot that contains no new conflicts. Hoping that by the end, all classes were successfully placed. This method would not need to search through the entire search space, it only is forced to try new directions each time a conflict is found. Ordered greed follows the idea that if there is a solution to the problem, then it will be found if a

method is developed to try all possible combinations for placing each class and also to determine the correct order of when to place each class. To use ordered greed all you would need is a methodical method of placement and an order of when each class is to be placed. Since the method of placement would not change from attempt to attempt the only way of generating new schedules is by changing the order of when classes are placed into the schedule. Therefore it is not necessary to store a population of schedules, instead a population of permutations is used. Each permutation decoded would represent a single unique schedule when it is processed through the algorithm.

This population of permutations then represents the population of individual schedules as it should be possible to create the schedule simply by sending the permutation through the algorithm. If two identical permutations are sent, we should get the same schedule so long as our algorithm does not contain any random choices along the way. It is not guaranteed that this method will be able to reach every possible schedule in the search space. In fact one can calculate the number of possible schedules found by calculating the number of different permutations. Although this method creates the possibility of losing potential solutions, it is intuitive that most of those solutions lost were heading down the wrong track.

From the description so far it would seem that this must be the answer, all other methods will fail in the light of a brute force attempt at finding good solutions. One major concern is left however - time. How long is it going to take to attempt to place each class into a larger and larger schedule. If the program is forced to search through millions and millions of possible locations for each new placement, the runtime of the program may end up being weeks instead of minutes. This would then prove the failure of the idea.

3.3.1 New Crossover Types for Permutation

Each permutation contains the same set of numbers with no repeats in order to place every class on the list. If an attempt to use one-point, two-point, or uniform crossover was used, the genetic children could contain repeat numbers or missing values, which would create invalid schedules. Since the population in using an ordered greed type program is stored in permutations, new crossover types had to be developed. Three main types were attempted and are described below.

Partially Matched Crossover Partially Matched Crossover or PMX [6] is a method that allows for conservation of all values in the permutation. To generate two children from two parents the following procedure is used.

First align both parents vertically as shown below:

```
Parent A   9 8 4 5 6 7 1 3 2
Parent B   8 7 1 2 3 9 5 4 6
```

Second two crossing sites are picked at random, these create the matching section.

```
Parent A   9 8 4 | 5 6 7 | 1 3 2
Parent B   8 7 1 | 2 3 9 | 5 4 6
```

The area between these two crossing sites now determines our position by position exchanges. Meaning in each original parent the 5 and 2 exchange places and the 6 and 3 swap and the 7 and 9 also swap, leaving us this.

```
Child C   7 8 4 | 2 3 9 | 1 6 5
Child D   8 9 1 | 5 6 7 | 2 4 3
```

Each new child still contains some ordered information it received from both of it's parents, but is also uniquely different from either of them.

Ordered Crossover Ordered Crossover or OX [6] is a similar method to PMX. It starts almost the same as PMX.

Begin by aligning both parents vertically as before and then create two crossing sites that are picked at random, these create the matching section as with the PMX method.

```
Parent A   9 8 4 | 5 6 7 | 1 3 2
Parent B   8 7 1 | 2 3 9 | 5 4 6
```

The area between these two crossing sites still determines our position by position exchanges. However a sliding motion is used to fill the holes left from mapping values to the other. Look at Child D below , values 5, 6, and 7 were picked to be mapped from parent A. Holes (marked with H) are created to show where those locations were on parent B.

```
Child D   8 H 1 | 2 3 9 | H 4 H
```

These holes are then shifted around to be in the location of the matched section with any values currently in there being pushed to the front of the new child.

```
Child D   2 3 9 | H H H | 4 8 1
```

Those values removed are now mapped back into the holes to finish forming the new child. Similar process is done to create the other new offspring.

```
Child C   5 6 7 | 2 3 9 | 1 8 4
Child D   2 3 9 | 5 6 7 | 4 8 1
```

This type of crossover has the effect of holding relative positions between some values and allowing them to be transferred onto children

Merging Crossover Merging Crossover or MOX [1] is a slightly different idea. Taking both parents we randomly merge together both permutations in a sort of card shuffle. Preserving order in both parents.

```
Parent A   9 8 4 5 6 7 1 3 2
Parent B   8 7 1 2 3 9 5 4 6
```

Shuffled deck = **9 8** 8 **4 5** 7 1 2 **6 7 1** 3 **3** 9 5 **2** 4 6

The bold values indicate they were received from parent A. To create the children a parsing of the list is done and the first instance of each value is written down in order to create child C. The second instance to create Child D.

```
Child C     9 8 4 5 7 1 2 6 3
Child D     8 7 1 3 9 5 2 4 6
```

This form of crossover preserves the order of values from both parents and transfers to the child. If a particular value is early in both parents it is forced to be early in both children.

3.3.2 Build Population

Creating the population for Ordered Greed is very simple. It is simply a matter of randomly creating permutations of the length of the number of classes to schedule. The difficult part about creating these permutations is checking each fitness value. This is when the actual schedule must be produced to test all constraints against.

Creating the actual schedule from the permutation uses an algorithm that attempts to place each class one at a time with an appropriate,available professor, a room, and at a good time. The first step was to limit some of the choices so that the entire search space is not needed. Sublists of professors and rooms were created to only include the members that would not violate any hard constraints. The professor sublist was also ordered to promote professors to the beginning that were in the most need of being scheduled.

The algorithm started by picking a professor and a room from the two sublists and then picked the first possible starting time for the class. This set of parameters was then checked to see if it fit well into the overall schedule and

if so it was added. If this turned out to be a bad location, starting time was incremented and tested again. This continued until all possible starting times were checked. If no good time was found a different room was tried. After all possible times for that room was tried the algorithm would continue on until all possible rooms were tried. Finally it would all start over again trying the next available professor on the list. This algorithm would stop at any point that a good set of values was found. If the entire list of possibilities were used up and nothing was found to work, a random placement was done. This algorithm is shown in flow chart form at the back of this report.

As written, this algorithm only checked for violations involving the hard constraints of the problem. Soft constraints such as professor preferences were ignored due to much larger processing time. One such run was attempted early on and resulted in about 50 to 100 times longer run time and only mild improvement over current model. This idea was abandoned due to practical considerations for what this program would be used for. As it is, most other trials took about 30 minutes to solve a 100 class schedule, and no one wants to wait three or four days for a slightly better solution.

3.3.3 Evolve to Solution

The actual genetic evolution of this trial was simple and very similar to the base program. Once the population was built, tournaments were preformed finding the best permutations currently in the population. These were combined with the new crossover methods in the hopes of creating better genetic children. Each new child was sent through the algorithm to test its fitness and placed into the population. The current best fitness value was reported every 30 seconds for graphing purposes, and the point at which a solution with no hard constraints violations was recorded. Trial was stopped after either a perfect solution was found or a maximum time of 30 minutes was reached

3.4 Hybrid of Ordered Greed and the Base Program

Once three separate programs were created and all were working fairly well, the goal became trying to find even better ways for creating the best solutions. It was noticed that the base program had a much better fitness value then the ordered greed program, however the ordered greed program was much more likely to have all hard constraints solved. This made sense since the ordered greed program only looked at the hard constraints. Having hard constraints solved but missing many of the soft ones made for a weak schedule, but having most of the soft constraints solved while still failing with one or two hard ones was probability even worse. It was also noticed that if the hard constraints were solved by ordered greed, it was almost always done very early in the run usually in the building of the population or in the first few minutes after.

This led to an interesting idea. Why not combine the build from the ordered greed program (giving us a population of schedules that may already have the

hard constraints solved) and then allow the program to evolve the rest of the way using the random techniques. This was the hybrid program.

A few variations were attempted such as adding in functions that would periodically refresh the population if the genetic material of all members became stagnant. These functions seemed to only take up processing time with very little advantage. The best modification that was kept in the final version of this program was allowing the Ordered greed program to work beyond the build and perform some evolutions before handing the control to the base program. So a command was placed into the program that would only swap control if either the hard constraints were solved or if a certain amount of processing attempts were made trying ordered greed. The amount of time spent on the ordered greed evolution was solely dependent on the number of classes needed to be scheduled. This worked very well at maximizing the percent of hard constraints solved and still allowing ample time to work on fixing some of the softer ones.

3.5 Generational Approach - Mutation only Program

Through the entire process of developing solutions to these problems it was seen that in many cases all genetic material in the population became very similar to each another. In fact many members of the population were exactly the same. Without new genetic material the algorithm was relying only on mutation to effect change. Early processes were fixed simply by increasing the mutation parameter to a very high level. This brought forth a new idea of attempting a program that uses only mutation as its source of change. First attempted only as a whim, the results were so much better than expected it became one of the main few to experiment with.

3.5.1 Build Population

Starting with good solutions in the population was important, so for this method it was decided to use the base program with greedy build procedure. By pushing the greedy parameter up to a high level, a little start up time was sacrificed in order to be fairly close to a solution by evolution time.

3.5.2 Evolve to Solution

The beauty of this program is in its simplicity, once the initial population was built the best one is saved. This hero is then rewritten over all other spots in the population. Then one by one, with the exception of the hero, a simple mutation is done on every schedule. All of these mutations are checked and the best one from that generation is saved and rewritten over all the rest again. This continues generation by generation saving only the best and using it to develop the next batch.

There are no tournaments, no crossover methods, only a simple mutation to develop and that can be done by picking one class at random and changing its values. Many times this mutation will send you in the wrong direction, but

just enough times it will find something better. A couple different mutation types were tried including mutating more then one class at a time, but the most effective was changing only one at a time.

4 Five Possible Solutions

These five different programs were not planned to be developed from the start, they evolved as it was seen what was working and what was not.

It all started as a idea for a programming project started in a genetic algorithms class. The only goal at this point was to show that using a genetic algorithm was possible to help solve faculty schedules. The idea worked surprisingly well and opened up the possibility for much more research in the field. The next logical step being how can a program be written to actually work for real schools and scheduling situations. The base program was created with this in mind.

After one working program was developed, the immediate thought was on how to improve upon this idea in order to make the program more feasible for larger schedules. Many heuristics were brainstormed and the best idea coming from this contained the greedy procedure to improve the initial population.

Ordered Greed was an exciting well known improvement that had great potential and was a route that had to be tried. In other applications ordered greed solved things in a fraction of the time other methods had. Many trial and error algorithms were created to attempt to get a fully ordered greed program working. These all worked to some extent, but always at the cost of processing time. It was found fairly early on that the only viable part of the program was a way to eliminate hard constraints fast and easy, adding the soft constraints to the puzzle took up to much time and had to be abandoned.

The Hybrid approach came out of the limited success from the ordered greed and the limited success of the random approach, both on separate fronts. Building a population with ordered greed and evolving it with the random Genetic Algorithm was one of the bright spots that came out of the partial failures.

The final program was a whim tried because of the apparent necessity of mutation in every program. It was not really expected and was a very pleasant surprise to see it work. Its apparent advantage over the random method did seem to disappear the longer you let both trials go. But the simplicity of the method keeps it a very strong method for additional testing.

5 Generating Input for Testing

With these five different methods in place, a way was needed in order to quickly compare and contrast which ones were better than others. The problem that started the entire project was to develop a schedule for a mid-size department for a college or university. Effort could have been done to input all the actual data from an area college. The only way to know for sure that one algorithm

is better than the rest is to test them all with many different possible sets of input data. Using only one may inadvertently give favor to one method.

A program was written that develops a sample set of input data that can be tested for all the programs. Although random values were used to determine most of the data, care was taken to ensure that important ratios and probabilities were maintained. For example, a college is not going to have 100 rooms available and 50 professors available if only five classes need to be taught. By examining a current schedule from Monroe Community College it was noticed that there is approximately one-fifth as many professors as there are classes to schedule and there is about one-tenth as many rooms available as classes to schedule.

It was also noticed that for every single distinct course offered, there was on average about one additional section offered. For example some courses may have 3 sections offered, while others may have only one. Additionally not every room can hold all courses, the estimate of about a fifty percent chance that a certain room could hold a certain class was used.

Professor preferences were more of a estimate, using the assumption that about one-third of all professors have preferences to not teach morning classes or evening classes.

The rest of the values needed were simply developed using random numbers. This gave a very quick input set of data that could be experimented on. Most data trails were preformed with 100 courses, 20 professors and 10 open rooms. To type in input data for 100 classes for every single trial would take too long to accurately do any testing.

5.1 How Input is Organized

The input data was organized in three data files. Each file contains a matrix of integer values that stores all sorts of coded information.

The first file was the course file. This contained a column for each distinct course and three rows containing credit hour information, number of days per week, and number of sections offered that semester.

Sample course file:

	Each Column is a distinct course								
Credit hours (1-5)	x	x	x	x	x	x	x	x	...
Days per week (1-5)	x	x	x	x	x	x	x	x	...
Sections of course (0-5)	x	x	x	x	x	x	x	x	...

The second file was the professor preferences file. Each row in this file represents a individual professor. The first column stores a value representing that professors preferences of what times of day they wish to teach. The second column stores a value representing how many credit hours that they can teach that semester. The rest of the column represent the individual courses that are currently being scheduled and store values that indicate whether or not that

professor has the ability or want to teach that course.

Sample professor file:

	Professor preferences	Hours Available to teach	Ability to Teach Certain Courses Each column is a distinct course								
Each row is a different professor	x	y	z	z	z	z	z	z	z	z	...
	x	y	z	z	z	z	z	z	z	z	...
	x	y	z	z	z	z	z	z	z	z	...
	x	y	z	z	z	z	z	z	z	z	...
	.	.	.	.	.	.	.	.	.	.	...
	.	.	.	.	.	.	.	.	.	.	...
	.	.	.	.	.	.	.	.	.	.	...

Code for professors preferences (x values):

- 0 - no preferences
- 1 - prefers no early classes (before 10am)
- 2 - prefers no late classes (after 5pm)
- 3 - prefers no early or late classes

Code for Hours Available to teach (y values):

- 0 - unavailable to teach or 0 hours
- 1 - between 0 and 5 hours
- 2 - between 5 and 10 hours
- 3 - between 10 and 15 hours
- 4 - available for 15 or more hours

Code for Professors ability to teach (z values):

- 0 - Can not teach this course
- 1 - Can teach but doesn't like to
- 2 - Can teach and wants to

The third file is the room file. Each row is a room available and each column is a class being offered. If the value on a particular spot is a one that means that that course can be taught in that room. If it was a zero, then the course can not. A sample file of this is not shown as it is a simple matrix full of ones and zeros.

5.2 Using Random Input Data vs. One Common Set in Testing

Much of the early runs started by generating a new set of input data and running a trial for 30 minutes and then throwing the data away and starting again with a new set. Hundreds of trials were tried of each program in order to optimize the many parameters that needed to be set. All trials were averaged in the hopes of getting a good representation of that programs worth.

It was assumed early in the testing phrase that even if the randomly generated input gave out easy to solve sets of data from time to time, this would even out over large enough testing samples. This assumption proved to be mostly true, but much variability did occur from trial to trial even in the same program trials that final results seemed somewhat untrustworthy.

Finally before testing the final optimized programs against each other a new script file was written to force all five programs to run the same sets of data. Hundreds of different sets of data were forced to be tested with all five programs and a final table emerged. Even if some data was too difficult or too easy it was fair for all programs.

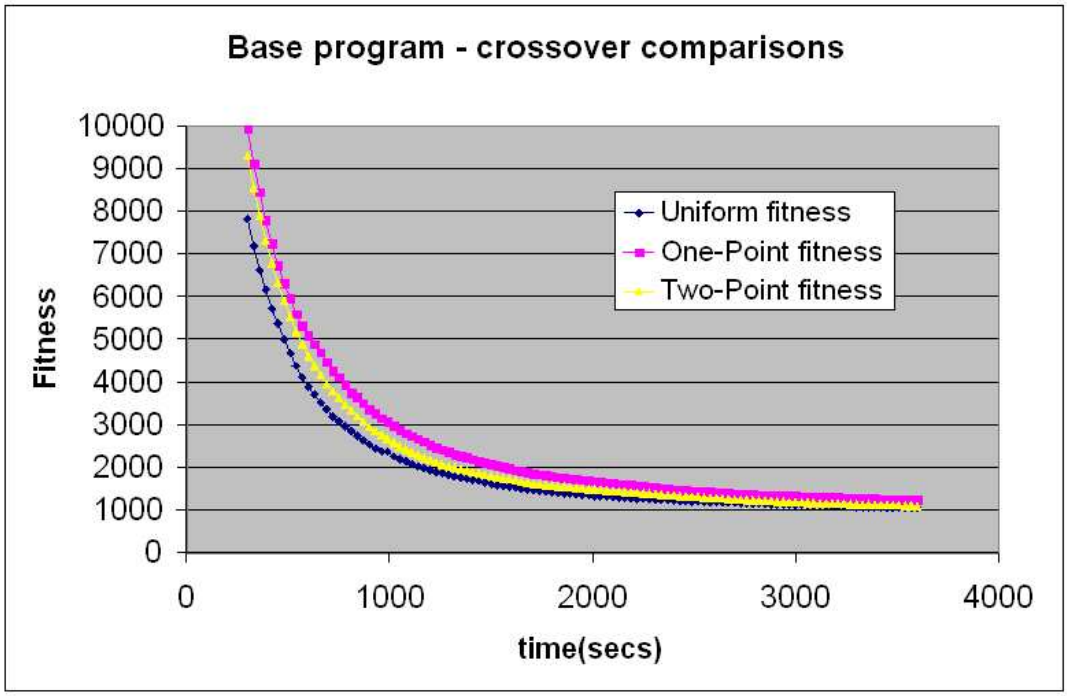
6 Testing Phase

With five different programs all attempting the same goal, it becomes really important that each of the five is optimized to give its best solution when comparing itself to the others. Each of the programs has sets of parameters, such as mutation rate or type of crossover, that needs to be tested individually to find the best set of values. Once all of the parameters are optimized for each version, trial runs can be done comparing them to each other.

6.1 Optimizing Parameters for Base Program

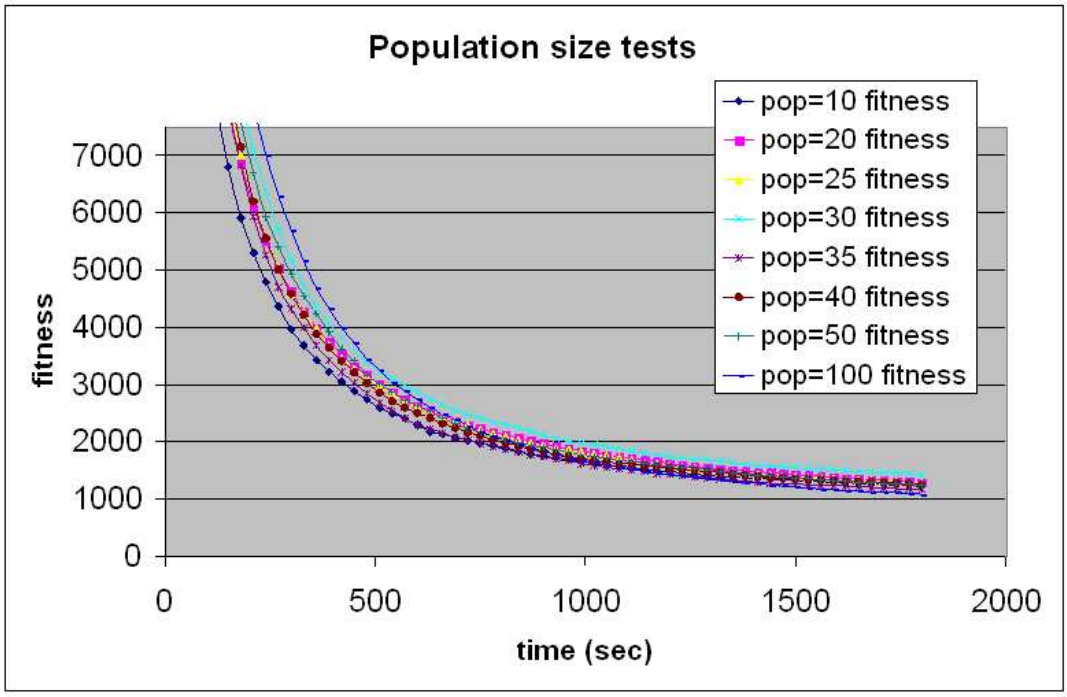
6.1.1 Crossover

The crossover types used for the base program are one-point, two-point and uniform crossover. Hundreds of trials were done comparing each of these three to one another. Trials of both 30 and 60 minute lengths were attempted. All other parameters were held constant at middle range values. In the end very little advantage was seen in any of the three. One sample comparison graph is shown below. This graph was created by running each crossover type 100 times and averaging the current fitness every 30 seconds of the run. It clearly shows all three crossover types converging after 60 minutes.



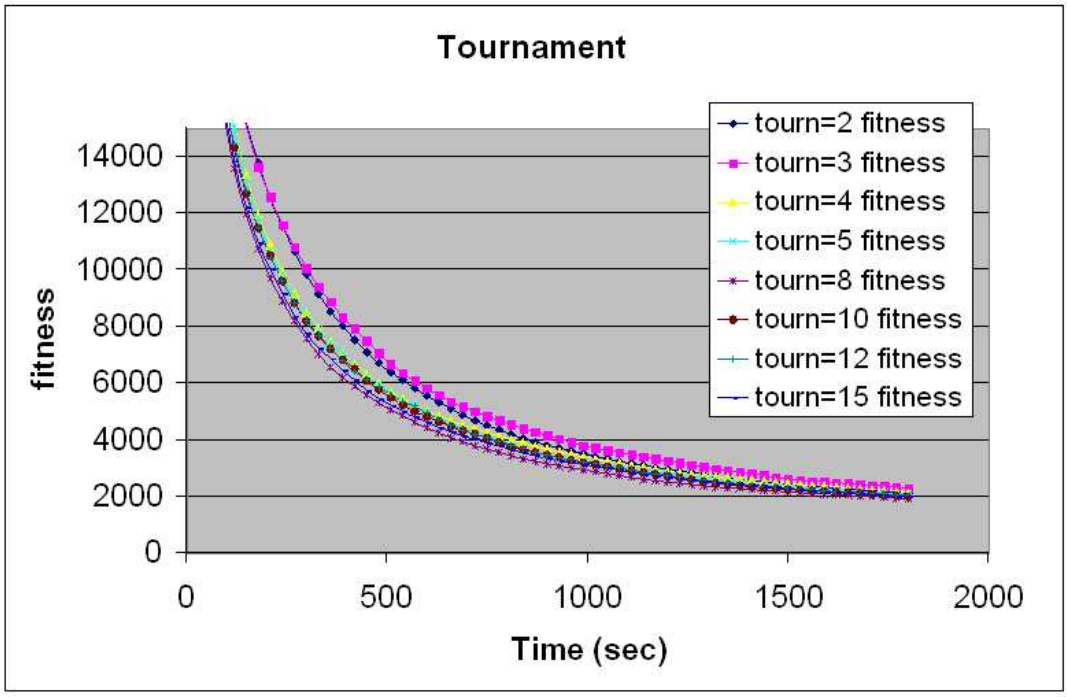
6.1.2 Population Size

The number of schedules stored in the population at one time was another possible source of inefficiency. Small populations could mean less genetic diversity while larger populations could bog down a computer due to extra memory storage. Various sizes of populations were tried from 10 to 100 to see if any differences were observed. Once again, no apparent advantage could be seen by any of the sizes. As you can see below all lines seem to converge together at the bottom after 30 minutes. Each line represents at least 100 trials averaged together. For the reasons of memory requirements most trials from this point on were limited to a population of 50 members.



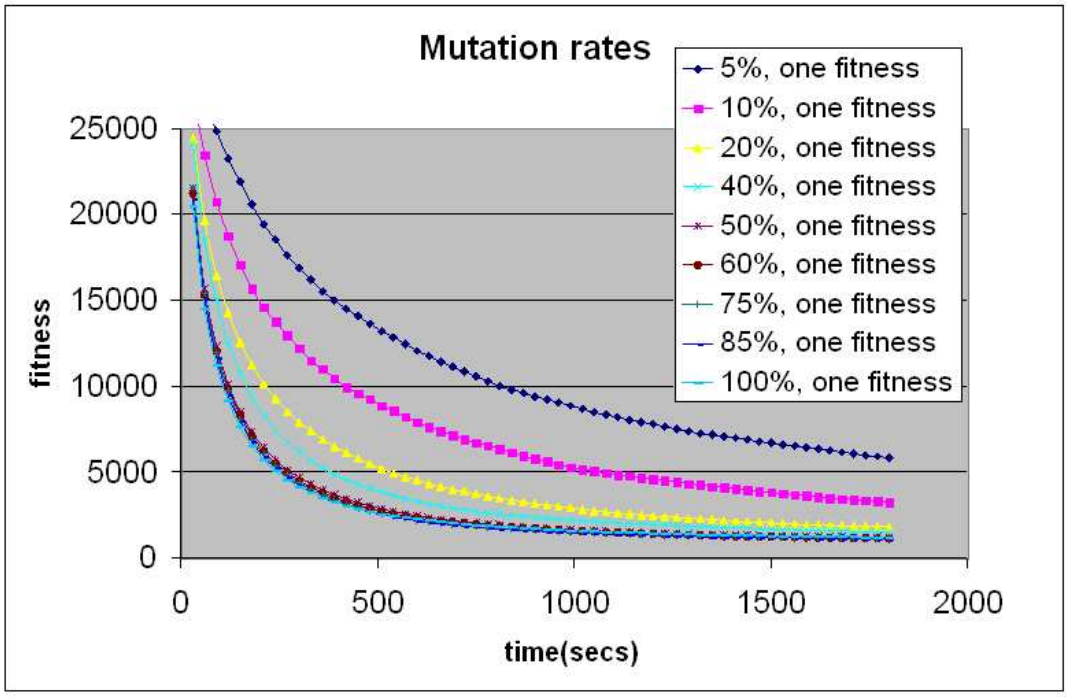
6.1.3 Tournament Size

Increasing or reducing the size of tournaments used to pick members for crossover was another potential parameter to optimize. Larger tournaments would seem to guarantee the likelihood of picking the hero of the population more often, while small tournaments could make it more difficult to find and replace bad schedules. Various tournament sizes from 2 to 15 were tried. Looking at the graph results below, tournament size had no significant affect on fitness. In the initial stages, small tournaments sizes (two and three) were slightly behind the larger values, but after a few minutes all values seemed to converge. For consistency, a tournament size of five was kept and used for all further studies.



6.1.4 Mutation Rates

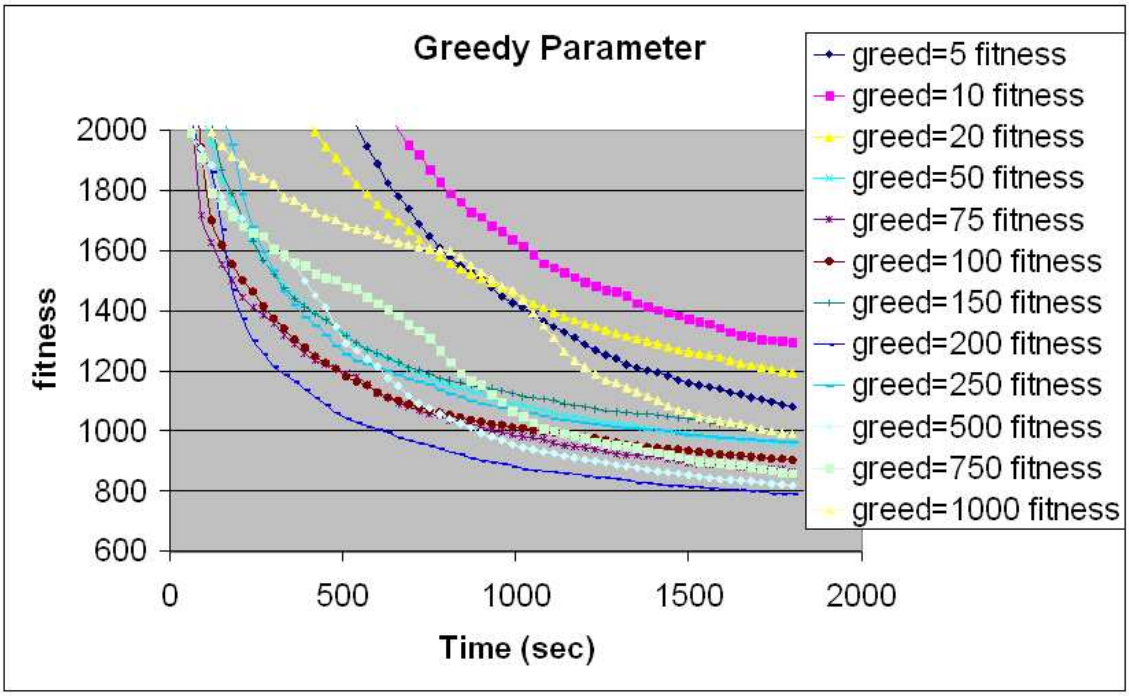
The rate at which a member of the population is mutated and the amount of mutation that is performed on that member were tested. It is seen that a very high mutation rate is needed to continue to keep new genetic material available to use. The best mutation rate turned out to be at about 50 percent of the new children created, but only mutating one class from each schedule. So even though a very high percentage of new children are mutated, only a very small mutation of about one percent (based on 100 classes) was done to each. Mutations of more than one class in a schedule were also attempted but were found to not converge as rapidly.



6.1.5 Greedy Parameter

This value controls how many iterations are tried as each class is attempted to be placed into a new schedule of the population. It seems obvious that the larger a value this is the better a fit it becomes. This trial was done to attempt to find if there is a point at which no more advantage is found versus that amount of time it will continue to take.

The parameter value continues to increase fitness at around 100 and slightly more so at about 200. After this value it seems better fitness values are not found. For most other trials the greedy parameter was set at 100 attempts to place each new class.



6.1.6 Summary of Optimized Parameters for Base Program

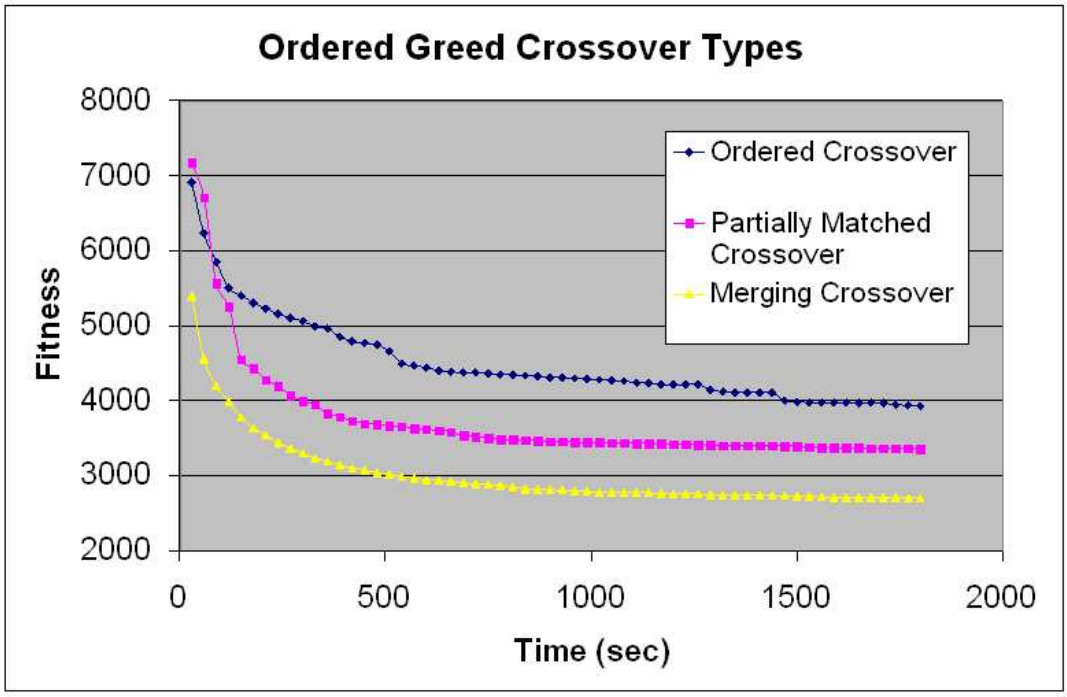
The following table summarizes the optimized parameter values that were chosen to use with our base program and base with greedy procedure program for all other tests.

Parameter	Optimized or chosen value
Crossover	One-Point
Population Size	50 schedules
Tournament Size	5 members
Mutation Rates	50% of new children, but only one class in each
Maximum iterations to place class	 100 attempts

6.2 Optimizing Parameters for Ordered Greed Program

6.2.1 Crossover

Since crossover types were different for permutation based programs, the effectiveness of these new types had to be compared. The three types tested were partially matched crossover (PMX), ordered crossover (OX), and merging crossover (MOX). All types were tested at a 25 percent mutation rate. The results seemed obvious that the merging crossover technique worked better than either of these other two types.

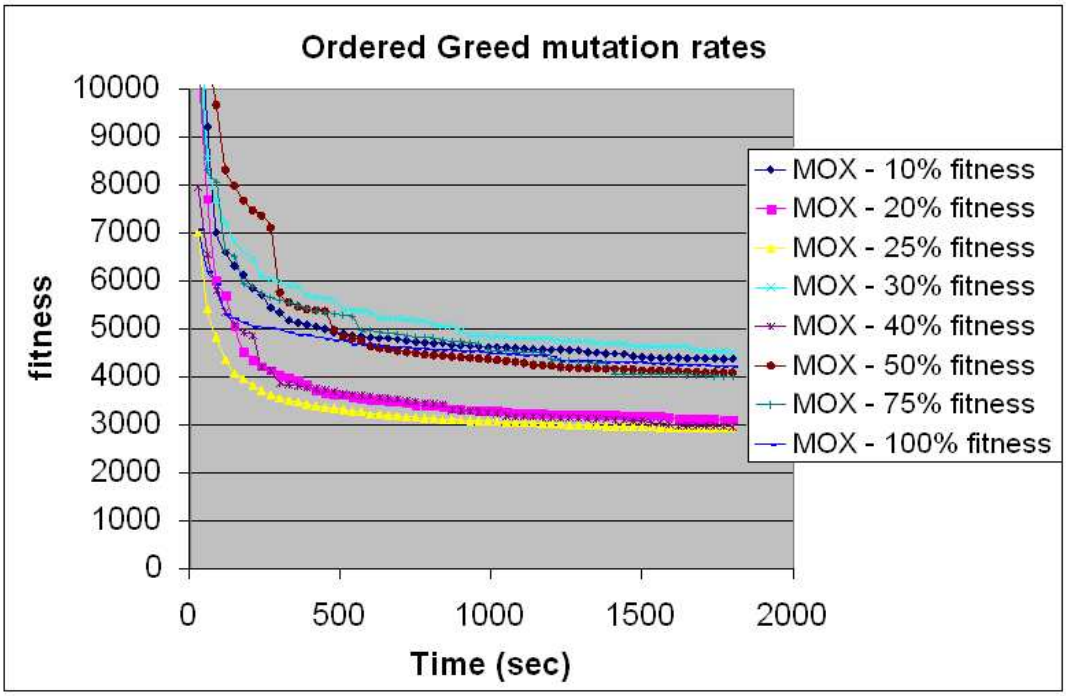


6.2.2 Mutation Rate

With the ordered greed crossover type picked, the next choice was at what mutation rate will the program be the most efficient. Mutation for ordered greed has nothing to do with randomizing classes in the schedule. This mutation was a small swap in the permutation that was used to develop each schedule.

A wide range of values from 10 percent up to 100 percent mutation were picked. It seemed obvious that 100 percent mutation should get us nowhere, but advances through tournaments and crossover even kept that fitness going in the correct direction.

The best mutation levels seemed to circle around 25% with some variability in the readings.



6.2.3 Summary of Ordered Greed Parameters

The merging crossover with a 25 percent mutation rate were the optimum value for those parameters. Population size and tournament size were not retested for this version of the program, and the greedy parameter was not applicable.

6.3 Hybrid Program Tests

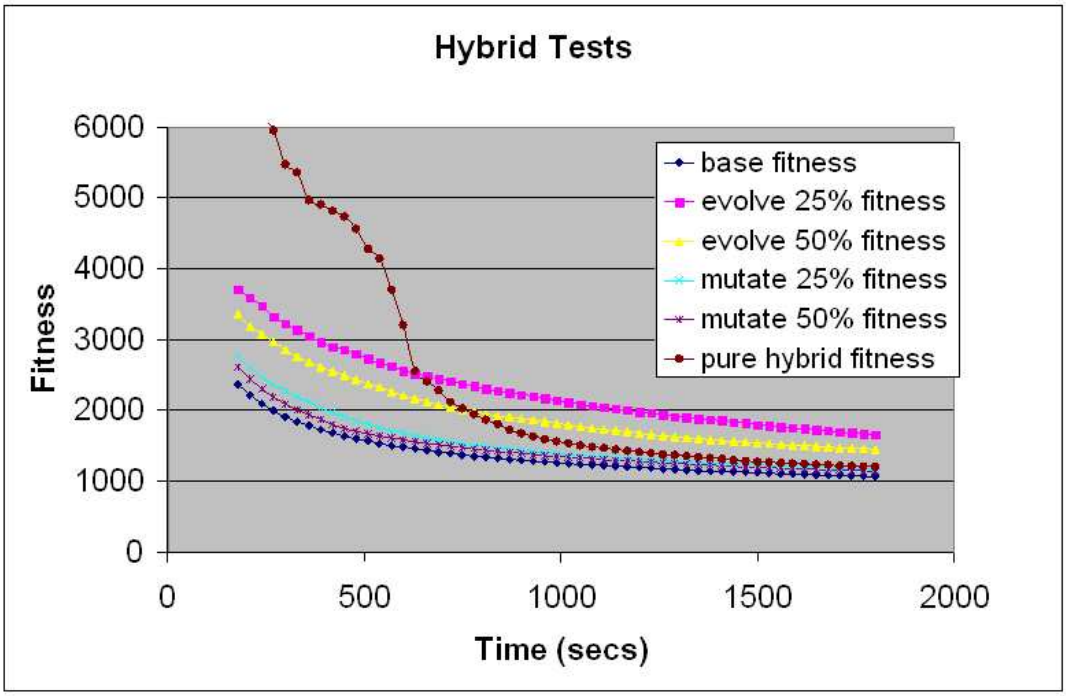
6.3.1 Adding Functions to Fight Stagnant Populations

As stated earlier in this article, the hybrid approach of grabbing the best features of two working programs had much hope for success. Since this program was simply parts of each of the programs written so far there were not any new parameters to optimize. However the lack of diversity in the genetic population did arise at this time. A couple of functions were developed to attempt to combat this issue.

One attempt was a mutate all function. This function would automatically run whenever it was noticed that all current members of the population had identical fitness values. It would simply take each member of the current population and preform a small mutation on it in the attempt to bring in new and better genetic material.

The second attempt was similar, in that it automatically ran when the population was stagnant, but it erased the entire population except the current best. It replaced the population with a newly evolved set of schedules developed by the ordered greed building procedure.

Both of these functions (shown on the next page) turned out to be worse than the simply base hybrid program. This program simply built the population with an ordered greed on the hard constraints procedure, and then evolved using the techniques of the original program.



6.3.2 Allowing Ordered Greed to evolve past Build

The base hybrid program was working well but it was noticed this program still did not approach the success the ordered greed program had with solving hard constraints. This was partially because no evolution was allowed to take place with the ordered greed method after the initial build.

An additional feature was added into the hybrid program, which I called out on the graph as pure hybrid. This allows the program to evolve for a certain amount of time using both methods. It starts by building a population from permutations using an ordered greed method, then continues to evolve using tournaments and merging crossover until it either solves all hard constraints or has tried for a limited amount of attempts. Then the program converts all permutations into the schedules and continues to attempt to evolve using the one-point crossover and the original base algorithm.

The base program still beats this new hybrid variation in overall fitness, however the pure hybrid variety turned out to be about twice as good (60%

to 30%) at solving the hard constraints. This more than made up for a small difference in the overall fitness. The graphs themselves only show the overall fitness values (hard + soft) and do not distinguish, however each program reports at the end of its run if it had any hard constraint failures left.

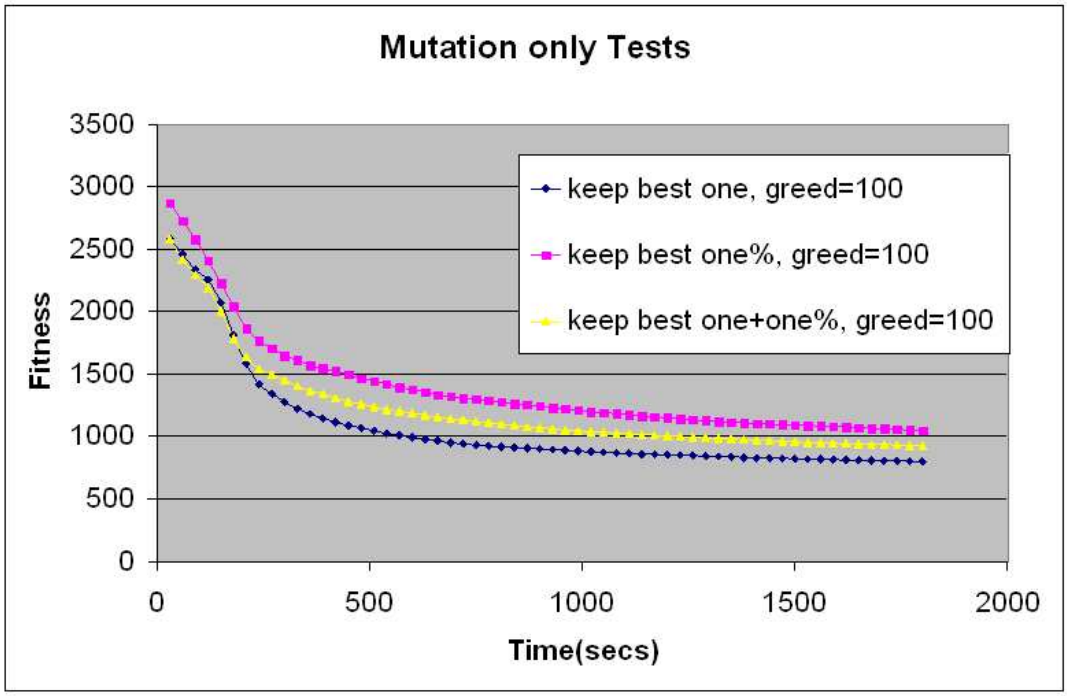
6.4 Generational Tests

6.4.1 Testing different Mutation Rates

With the generational mutation program, there are no tournaments or crossovers, the only parameter to test is how often to mutate. Early tests showed that since every single member of the population was being mutated, the smaller the mutation the better. Mutating only one class for each mutation turned out to work well. There was a question raised about getting stuck into a local maximum. What if there was no way to improve the schedule by only changing one class, maybe two classes needed to be switched. With this in mind three different mutation rates were tried.

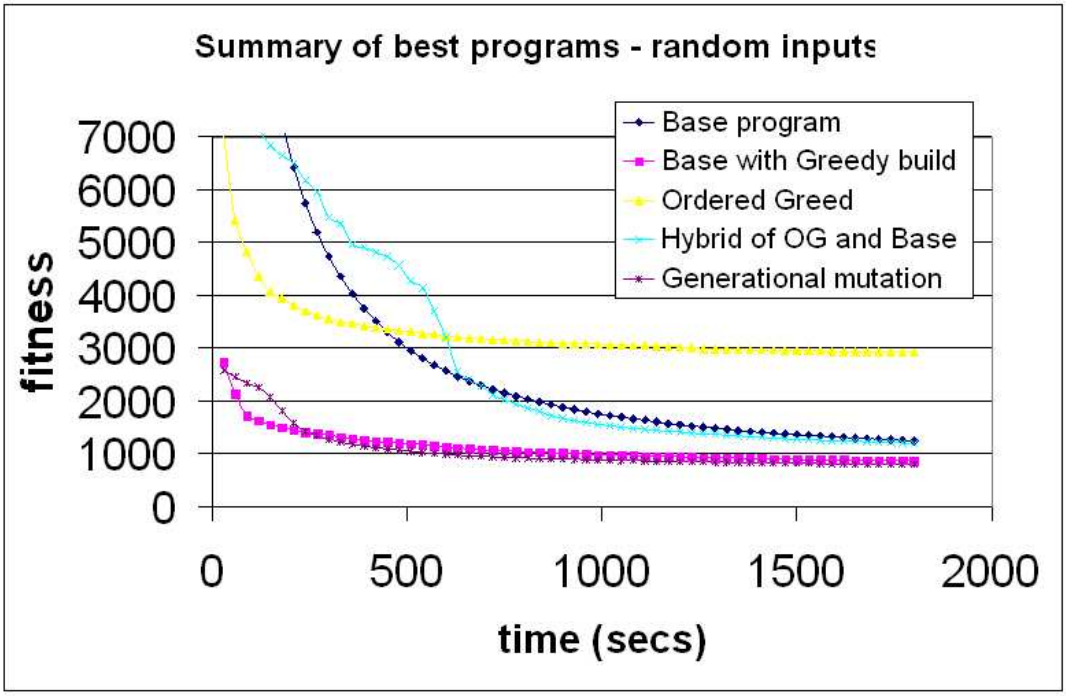
The first attempt was mutating one class at a time. The second was mutating one percent of all classes. That would mean sometimes one class would be mutated, but every once in while either no classes or two classes from the schedule would be picked. The third attempt tried to eliminate the times that no classes would be mutated by forcing at least one class to be mutated and then having one percent of the rest done on top of it.

The results came back as expected, most of the time it seems mutating one class at a time is best. The graph of these results is shown on the next page.


Mutation only Tests
3500
3000
2500
2000
1500
1000
500
0
Fitness
0
500
1000
1500
2000
Time(secs)
keep best one, greed=100
keep best one%, greed=100
keep best one+one%, greed=100

7 Final Results

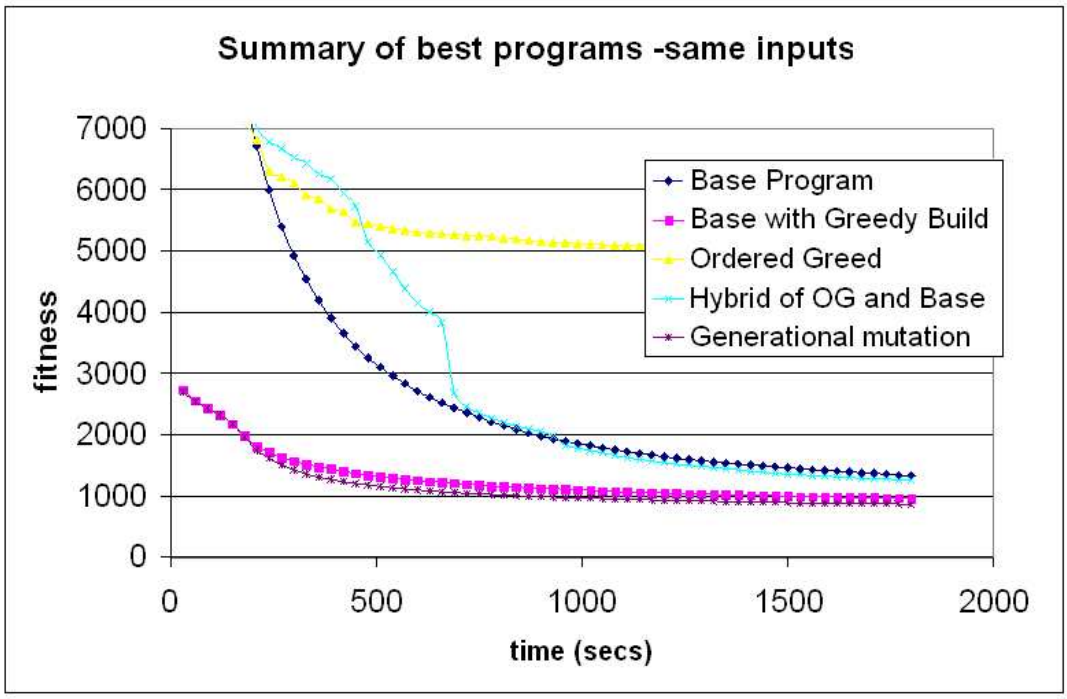
With all of the program parameters optimized it was now time to compare them and determine a winner if any. All five programs were run for at least 200 trials each on random inputs at their most optimized way. This graph summarizes the results.



This graph alone can not show distinction for hard and soft constraints, so the table following was added to give a little more detail.

	Average Lowest Fitness Value	Percent of Schedules with all Hard Constraints Solved
Base Program	1258	5.18
Base with Greedy Build	948	41.49
Ordered Greed	3619	67.5
Hybrid of OG and Base	1200	60.88
Generational Mutations	823	40.87

The first set of data had every trial of all five programs starting with a unique data set and averaging out the final results. In order to verify the results one last time, the trials were done a second time, this time with each of the five programs running on the exact same set of input data. This would eliminate any advantage that one program may have had with getting lucky sets of easy input data. This graph is shown on the next page.



The values although slightly worse than the first set, showed almost exactly the same ratios seen before. This would hint that the data sets created for this

run were slightly harder to solve. The ratios staying the same strongly indicate that this table of results is a accurate summary of the five programs. In this final trial all programs were run 450 times each for 30 minutes at a time.

	Average Lowest Fitness Value	Percent of Schedules with all Hard Constraints Solved
Base Program	1330	6.00
Base with Greedy Build	960	42.13
Ordered Greed	4958	61.33
Hybrid of OG and Base	1263	57.11
Generational Mutations	866	39.33

8 Conclusions

The initial goal of this project was to determine if using genetic algorithms to help in solving a faculty scheduling problem was feasible. This goal was accomplished early on with the first base program. Solutions were developed in a matter of hours for problems with over 100 classes to schedule. Sample data was chosen to be on the difficult side of likeliness with just enough professors to teach all classes (maybe not enough) and a very limited amount of rooms. Since the data was randomly generated, many data sets may actually have had no possible solutions. Even with these choices solutions were developed and although perfect schedules were almost impossible to come by, many schedules were found that would at least satisfy all the hard constraints of the problem. The fact that a problem with such a huge search space to wander though could be accomplished without spending weeks of processor time proves the usefulness of these techniques.

Once the initial goal of having a genetic algorithm that could work to solve this program was completed, the fun part of experimenting to make it work better began. An idea as simple as an intelligent build of the population by adding a greedy procedure as each class was placed had dramatic results, almost unbeaten by any others. The base program with a greedy build had approximately a 25% improvement over the total fitness value and was six to seven times more likely to have all hard constraints solved in only 30 minutes. This was based on a total class schedule of about 100 courses. All total about 40 to 42 percent of every run was able to create a workable schedule versus only five to six percent of the original.

The ordered greed program, which was expected to have strong results in finding good solutions had very mixed reviews. Programs attempted early on that would solve solutions looking for schedules that solved both hard and soft constraints were so time consuming that they proved to be unfeasible in the time frame of solutions that were looked for. But by giving up on the soft constraints and looking for solutions that solved only hard ones, results could be achieved. Of course, as soon as the hard constraints were solved, the evolution to finding better solutions did not go very far. This is why the ordered greed

trials had a much higher final fitness rating than the others, usually anywhere from four to six times worse. However when looking solely at the solution to the hard constraint problem, which is what this program was designed to fix, great results were found. About sixty to seventy percent of all trials ended in solutions that solved all hard constraints, much higher then the forty percent maximum that the base program could produce. Still the high overall fitness value meant that most of these solutions were stacked full of soft constraint violations and much of the professor preference information was ignored.

The hybrid of the ordered greed program and the base seemed a perfect idea. Use ordered greed to do what it does best and solve all of the hard constraints, and once that the program is finished with this, continue evolving just like the base program and try to eliminate the remaining soft constraints. The results showed this worked very well, but a little was sacrificed in both areas to have the time to process both evolution schemes. The final fitness values were not as low as the base program with the greedy build, but very comparable to the original base program. Also the percent of solutions with all hard constraints solved did not reach the same heights as the ordered greed of the middle sixties, but was still very high at between 57 to 61 percent. And for our purposes, solving as many hard constraints as possible should take precedence. Without those solved, a schedule is impossible to make. This program seemed to have the best of both worlds and seems to be the clear cut winner.

The generational mutation program is more of a heuristic set up to test how important mutation is on a system, but really is not a genetic algorithm itself as no gene splicing or combination of parents take place. It did however impress on how fast it was able to find solutions. Most likely saving much processing time by not having many of the tournament and crossover code to pass though, it jumped right into finding good solutions. This program gave us the best final fitness value of all five programs, and was very comparable to the hard constraint value of the base program with greedy procedure. Although very good, and better then expected, it still did not match the universal results of the hybrid program.

After all programming was done it was thought that maybe including the generational mutation code at the end of each of the other programs for a couple minutes may have had the effect of finding even better solutions. Especially the hybrid program, but this may be a topic for another project.

Creating a schedule from scratch is possible by methodically going though every single combination of professor, time, days, and room choices for every single class and comparing them to a large list of constraints. Then picking the choice with the smaller number of constraint violations. However for even a small number of classes (100), and professors (20), and rooms (10), the combinations to choose from quickly rise into the millions. And each of those millions of schedules need to be checked for all violations. As additional classes are added, the search space increases exponentially. So no matter how fast a computer you own, there will always be an amount of classes that is too difficult for your computer to solve by brute force. This project shows that with some careful genetic programming and a few creative ideas, you can almost do the impossible.

9 Possible Improvements

It is always easier to look back on your work and see how tests and experiments could have yielded better data. Many aspects of this project would have been done slightly differently with hindsight. The next couple of paragraphs highlights what could have been done better or what a future experimenter could try to examine.

Using random data for every individual test when optimizing parameters used an assumption that really should not have been made. The thought was that if enough trials were run, any inconsistencies in the data would simply average out. This was probably true, however most of the times only one to two hundred trials were performed per parameter, which turned out to have a still quite a bit of variance in the data. If some of the parameters had a small advantage over another, it may have been missed do to the randomness of the input. Thousands of runs were probably needed to finally show which was better. But given the time constraints on using the lab machines only hundreds were possible. It was only later in the project when a script file was written to allow each program to attempt to solve the same set of random values and a more accurate comparison could be made.

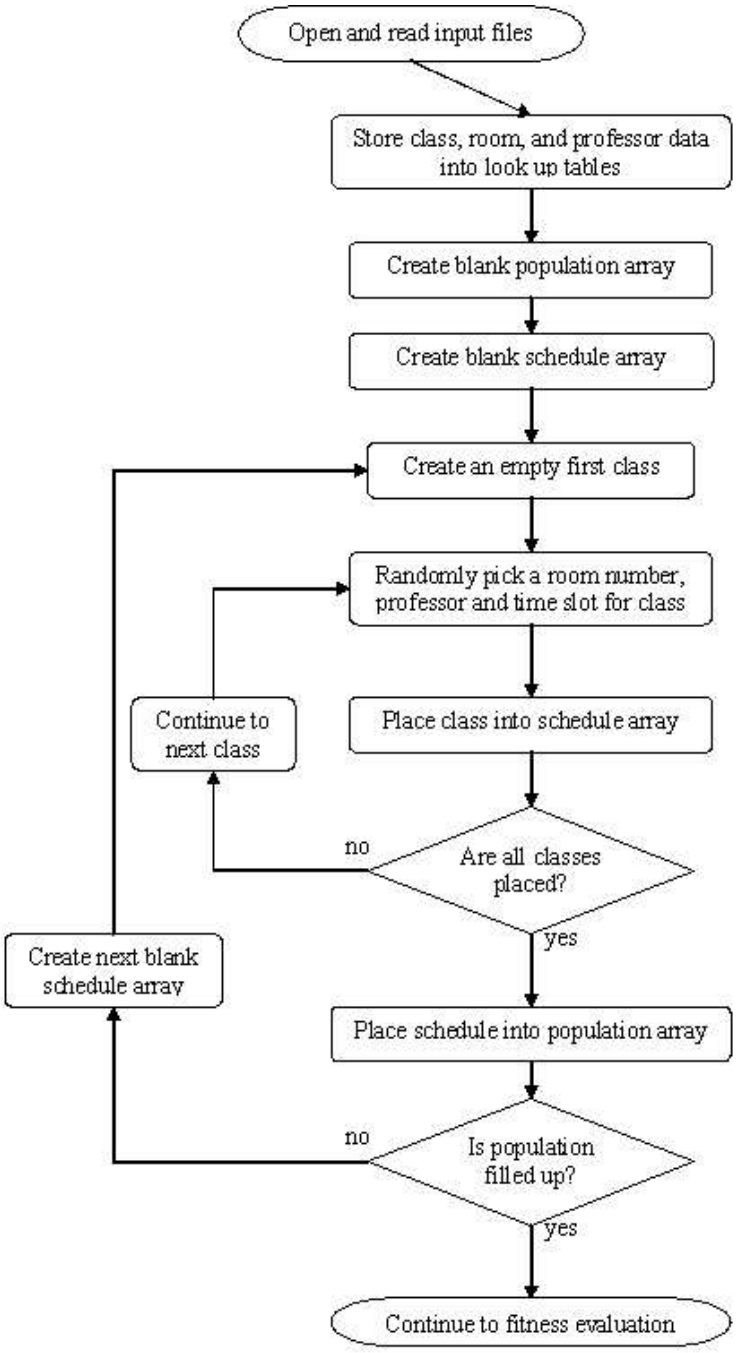
Using the best program to attempt to solve a real world schedule would have made for a interesting final analysis. This was in part planned to be done for this project until some problems arose. These programs were written to solve a scheduling problem for a small to medium sized department in a college or university. One major oversight was not including a process by which resources were shares by other departments. In other words, the Physics department doesn't own the rights to all lecture halls on campus. This problem would go away if we would simply attempt to schedule the entire college (all departments) at one time. However this generally turned out to be thousands of class sections with virtually hundreds of professors. This amount was simply to much for our program to handle at once. Not to mention that amount of input would take weeks to type in. Fixing this oversight, by allowing for some resources to be shared, or allotting certain amount of rooms to each department is a potential fix for a future project.

A smaller problem involved the fact that many times you have specialty classes that just can not be held at the same time as another class. For example, A senior advanced lab should not be offered at the same time as another class that many students need to graduate. Many instances of this were found where many courses need to be taken by the same groups of students during the same semester. So again for a future project a bit more of the needs of the students schedules would have to be taken into account.

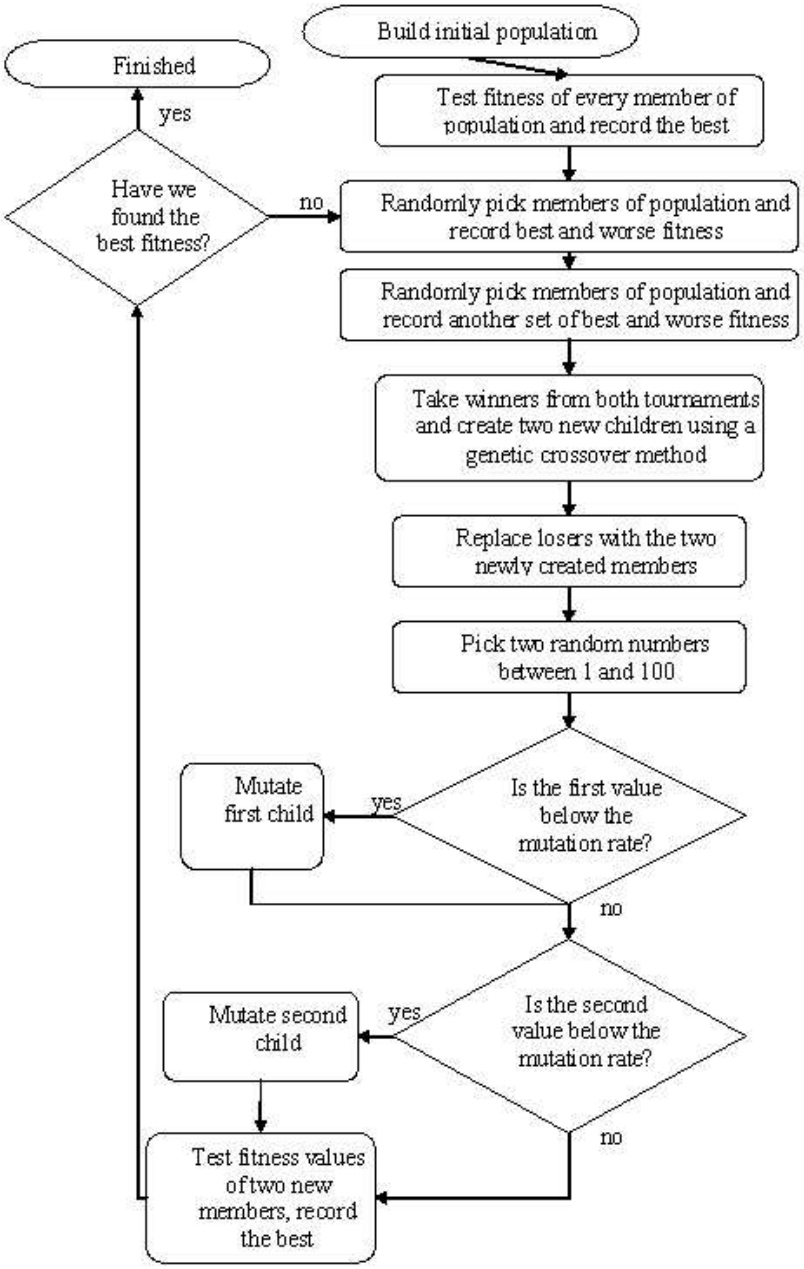
These were by no means the only improvements that could be made. There were probably countless ways to improve the efficiency of the code. Many other crossover ideas, could be tried for both ordered greed and random crossover. And many other program ideas could be tried to check for other advantages. A lot of work is left for many different projects or thesis work.

10 Flow Charts

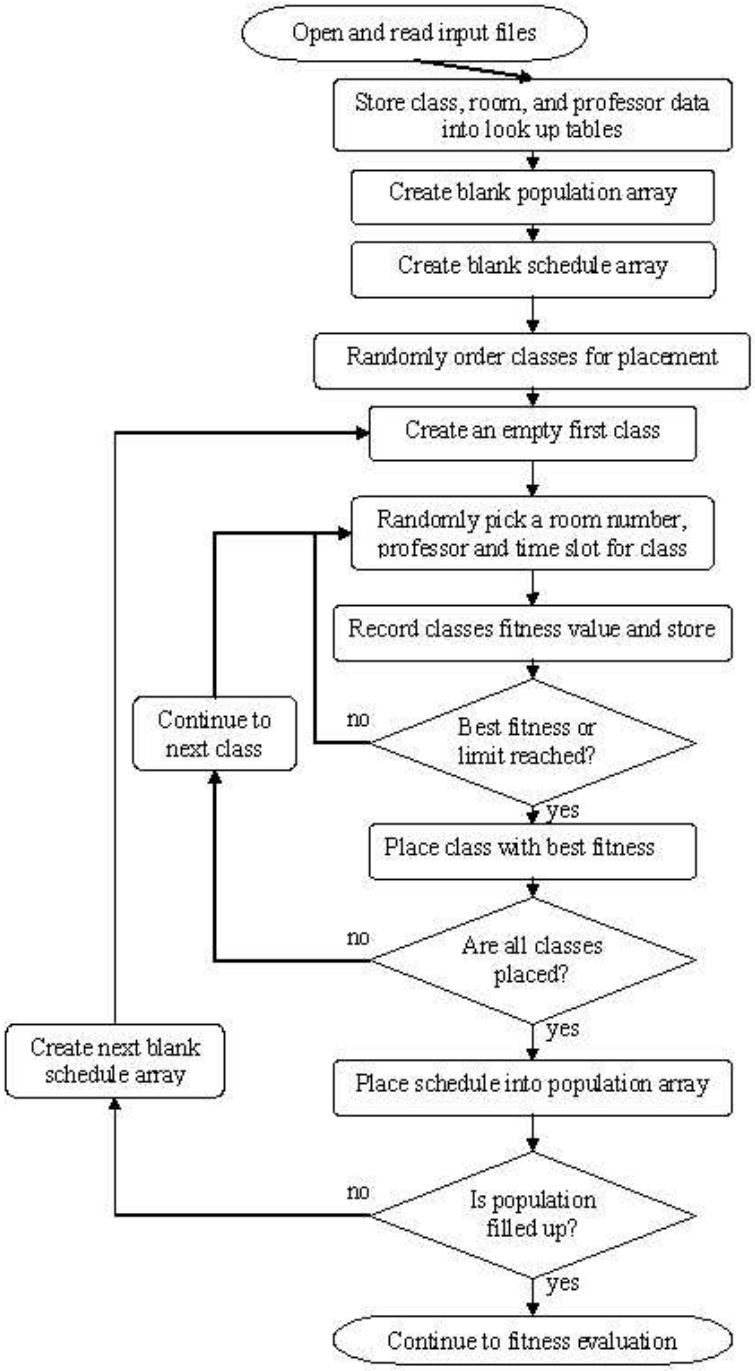
10.1 Building Random Base Population



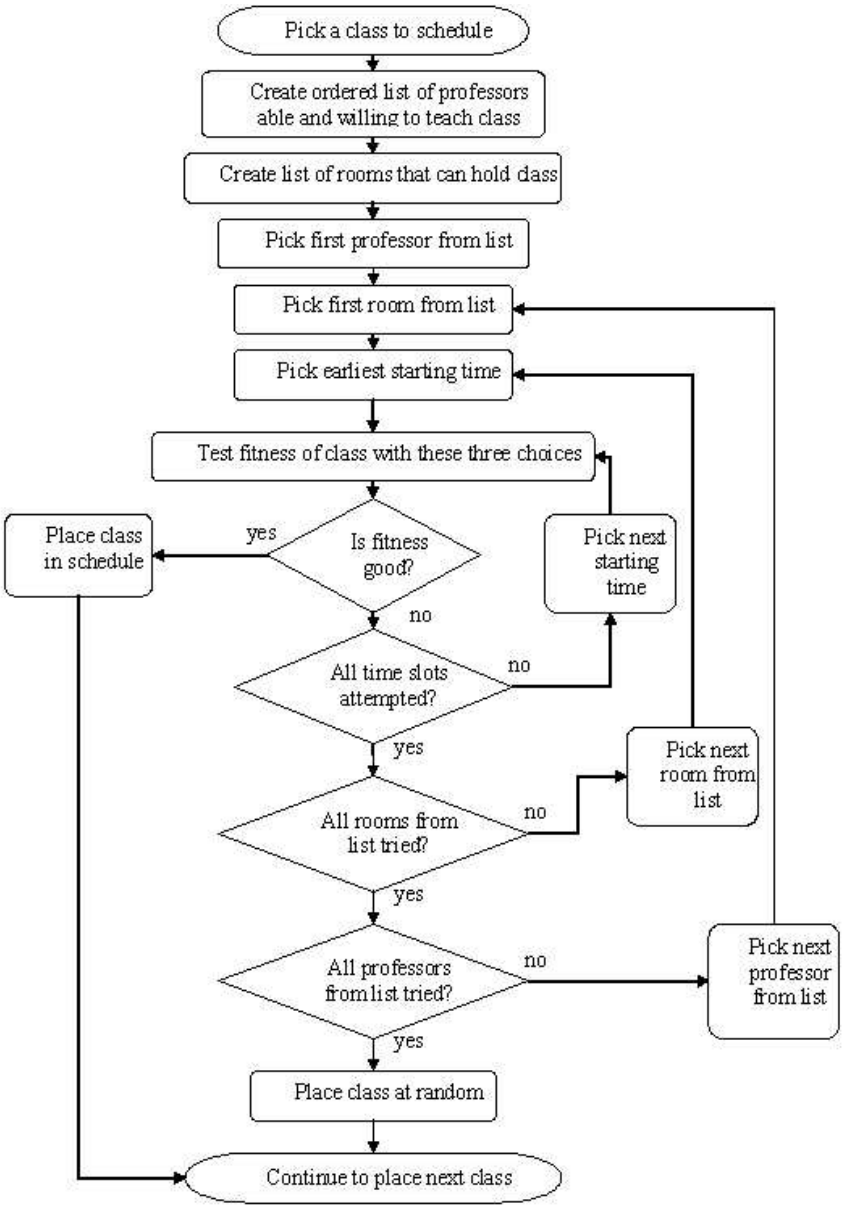
10.2 Main Base Program



10.3 Building Greedy Population



10.4 Creation of Schedule from Permutation using Ordered Greed



References

[1] Peter G. Anderson and Daniel Ashlock. Advances in ordered greed. 2004.

[2] Peter G. Anderson and William T. Gustafson. Ordered greed. In *Proceedings of the ICSC Conference on Soft Computing, Genoa, Italy (SOCO'99)*. International Computer Science Conventions, 1999.

[3] William I. Bereza. Resource scheduling with distributed genetic objects. 2001.

[4] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, second edition, 2001.

[5] H. M. Deitel and P. J. Deitel. *C++ How to Program*. Prentice Hall, New Jersey, thrid edition, 2001.

[6] David Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, 1989.