Faculty Scheduling using Genetic Algorithms
A Master's Project Proposal

Kevin Soule
ksoule@monroecc.edu

January 16, 2006

Abstract

This proposal is geared toward investigating the possibility of using genetic-based algorithms to solve scheduling problems. More specifically, the problem of developing a class schedule for a medium to large size department of faculty at local area colleges. Development of schedules of any form has been shown to be NP-complete. Therefore, finding just one possible solution can be a fairly difficult operation. I would like to investigate the possibilities of retrieving acceptable schedules in relatively quick times using genetic algorithms and techniques.

Project Committee

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

Contents

1 Problem Defined

Organizing a course schedule for a small college or a department in a larger college is a difficult endeavor. Time conflicts, room conflicts, and professor preferences all have to be organized and solved. In many cases an actual solution may not exist. Even a small amount of courses to schedule gives you so many different variable times, rooms and professors that it is infeasible to search through all of them. In fact, scheduling problems in general can be shown to be *NP-complete*. Most likely, even if we are able to find a few good solutions, we would still not be able to be sure that there was not a better one available. Rapidly coming up with at least some useable, if not relatively good, solutions then becomes our top priority. We are definitely willing to make the trade-off and sacrifice the definite ideal solution that may take us an unimaginable amount of time for a technique that will allow us to generate many good solutions in a relatively small amount of time. The technique that I am attempting to use is the Genetic Algorithm.

2 Overview of Genetic Algorithms

The basic idea of using a genetic algorithm to solve any problem is to start with a finite population of potential solutions. This population could be generated either randomly or by using a scheme that develops solutions that are much more likely to solve the problem. Each attempted solution in the population has to have some value associated with it that represents how much of the problem is being solved. This value is called the *fitness* of that solution. The better the fitness value, the closer we are to having the overall problem solved. Once we have the initial population in place, we want to use an evolutionary technique that will replace `"bad"` solutions (i.e. low fitness values) with better ones. In order to understand how this can be done, we can think of each of these individual solutions as an organism with differing genetic materials. This genetic material is the code that generates our schedule solutions. We then want to take these organisms and evolve them by breeding them with each other. By taking parts of the genetic materials from two relatively good parents we can form new individuals that are placed into the population. These new individuals usually will replace one of the organisms with a lower fitness value, which over time will improve the overall population fitness. Most new individuals that are generated do not in fact have better fitness values than most in the population, but by continually removing lower values we will eventually see an increased number of `"good"` solutions. We can repeat this as many times as we wish until we are either satisfied with the current best solution or are confident that additional iterations are unlikely to generate better solutions at a fast enough rate.

The main idea of genetic algorithms is simple enough, but to get them to work well entails a lot of experimentation. First of all there are many different ways to represent in code our `organisms` in the populations, and some have

definite advantages. There are also many different ways to build up the initial population and it is easy to see that starting from a very good initial population will shorten our time required to find our best schedules. Generating children from parents also has many different forms, each with advantages and disadvantages. In addition there are ways to introduce mutation into the population in order to increase genetic diversity. All of these factors can be altered and tweaked to extremely affect the final results. A lot of the effort for this project will come from optimizing these parameters to speed performance to the maximum.

3 Problems with Schedule Making

Just what makes developing a schedule so difficult and time consuming? In analyzing all of the details that go into creating a schedule for faculty teaching at a university, I have come up with 13 distinct constraining factors that have to be kept in mind as we are placing courses in the schedule. I'm sure there are many others that were either missed or ones that I did not deem as important, but these seemed to be the most bothersome and were all of the features of a schedule that I tried to keep solved. These are organized into two sections, the hard constraints and the soft constraints. The hard constraints are what I would signify as impossible problems that have to be solved or we can not make a schedule. The soft constraints were more of annoyance problems but never the less are important to the individuals either teaching or taking these classes. Even within the two sections individual constraints were weighted by overall importance to the solution, so that the worst ones would be taken care of first.

3.1 Hard Constraints

These items must be kept out of our schedule in order for things to work!

- 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. English professor teaching 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 scheduled in a lecture hall.)

3.2 Soft Constraints

These are additional problems that I attempted to keep out of the solutions. Many of these are simple professor preferences.

- 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. (Makes for a long day.)
- 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.
- 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)

4 Goals for the Project

4.1 Overall Goal

My main goal for this project is to show that genetic algorithms can be used to solve the problem of faculty scheduling. I would also like to attempt to experiment and optimize most or all of the differing parameters that are to be used. I will finish with what I can see as a relatively quick working schedule maker program that can not guarantee the best solution, but will return very good ones for even moderately sized problems. The final program will be able to find these solutions in a matter of minutes or hours as opposed to years of searching that may have been needed to run through the entire search space.

4.2 Additional Goals

- Creating two differing versions of the program that use both a random generated schedule and an ordered greed version. Determining the best working version for our needs.
- Determining the best children generating scheme for each version. (i.e. *crossover* methods)
- Optimizing other parameters such as population size and mutation rates, for the final program.
- Experimenting with other approaches, such as a generational approach, and differing ways to mutate schedules.
- Making the program as user friendly as possible

5 Steps to Achieve Goals

I propose to work through the following series of steps in generating my final schedule making program. This should safely end up with a program that optimizes parameters and tests many interesting features and additions of genetic algorithms.

5.1 Initial Program

The first step is to create an initial program that generates totally random schedules and evolves them to attempt to find solutions. A way to represent individual schedules in the computer is to be developed for all future tests. No parameters are optimized as educated guesses are used and the most basic of crossover methods are preformed. Our goal at this point is to find out if any schedules at all can be generated using a genetic algorithm and to prove that reasonable size schedules can be solved in a reasonable amount of time. Experiments of about 100 courses are expected to be mostly solved (all hard constraints at least) within a few hours at most. Initial test input will have to be at first random, but based on actual observations seen in working College schedules. Ratios such as how many rooms and professors are usually available for a specific number of classes are recorded and used. Ideas such as professors preferences are randomly set at a likely probability and then generated with a random number generator. An actual solution to each specific problem is not known to even exist, but trying multiple inputs many different times will eventually show whether solutions can be found. Again our goal is not to solve every case, but show that on average we have a good chance.

5.2 Optimize Initial Program

The goal here is to improve the results in step one by running many experiments with different crossover methods, mutation rates, and population sizes. Many different sets of inputs are also experimented with to hopefully show that the parameters settings are optimized no matter the input. Impossible sets of input as well as relatively easy sets to solve are to be tried. Attempts to increase the fitness of the original population by altering how the initial population of schedules is generated are also to be tried. A somewhat greedier routine instead of completely random seems to have promise. Once the best values for each of these parameters is found they will be kept stable for the rest of the experiments.

5.3 Creating and Comparing Other GA Schemes

The original design must be altered once again in attempt to find improvement with different ideas of genetic algorithms. At least two such ideas will be tried, ordered greed and an generational approach and many variations or combinations of these may result.

Ordered Greed Instead of a completely random approach when it comes to building schedules, a more structured one is to be tried. With ordered greed we force each schedule to be generated by placing classes in a different order and only placing them into locations that are currently free of conflict. This has the potential to rapidly find working schedules in only a few evolutions at the cost of the time required for each revolution. Determining if this benefit outweighs the cost is the goal here.

Generational An idea of allowing a population to be generated and then evolving as usual for a short amount of time. Then at specific times destroy most of the population saving only a small portion of good solutions and rebuilding the rest. This is hoped to widen the diversity and find many additional good schedules.

Generational-by mutation only The idea of creating an initial set of schedules with either Ordered Greed or randomly and then taking only the best and mutating it in different ways to recreate the rest of the population. Then taking only the best from that bunch and repeating. This idea may quickly converge onto a good solution or it may simply get stuck in a local maxima and do nothing.

5.4 Combining to Find Best Solution

All the work of the previous steps will be placed together in the most optimized and best working design in order to most efficiently develop new schedules.

5.5 Make User Friendly

The addition of a *front end* will be necessary in order to make this program usable. Many inputs up front are needed and a way to nicely enter them is important.

6 Design Specification

At the time of this proposal the initial genetic algorithm program had been written and is briefly explained here.

6.1 Inputs Needed

In order to calculate the fitness of any schedule a multitude of inputs are needed. There is in fact so much information that it to difficult to input real values for all of the tests that I wish to perform so I have created a short program that generates random values. There are inputs regarding the courses running, the professor abilities and preferences and also information pertaining to room functions. These inputs are stored in my program as a set of integer values in three distinct matrix look-up tables.

6.1.1 Course Inputs - for each course

- Number of credit hours. (How many hours a week does the course meet.)
- Number of days per week. (Is it a lab that meets on only one day or a class that meets multiple times a week.)
- Number of sections of each course that needs to be scheduled.

6.1.2 Professor Inputs - for each professor

- Ability and preference of which courses to teach.
- Preferences of when to teach classes. (Not wanting to teach early or late ones.)
- Total number of credit hours wanted for the semester.

6.1.3 Room Inputs - for each room

- Basically for each course that can be scheduled we need to know which rooms can hold them.

6.2 How Schedules are Stored

Each class that is to be placed in the schedule has to have at least eight distinct features pertaining to it. The first four are given to us by the inputs and can't be changed. They represent the name of the class, the section number of the class, the number of credit hours for the course, and number of class meetings per week. The other four are variables that we are changing around to make the best schedule. These include the professor, the room, the starting time, and the days of the week that are chosen. Some assumptions were made in deciding how flexible I was to allow the courses to be scheduled.

6.2.1 Restrictions on how classes can be scheduled

- Only certain combinations of days of the week were allowed.
  - 2 days a week - only MW, WF, and TR combinations were allowed.
  - 3 days a week - only MWF was allowed.
  - 4 days a week - only MTWR was allowed.
- Class must start at the same time each day it is offered.
- Classes must start and end on the half hour.
- No classes may start before 8am.
- No classes may start after 8pm.

Since I had so many variables that I wished to store for each class, I decided to represent each class as an object in an object-oriented language. I chose the language C++ to write all of the code. The entire schedule of classes was then represented as an array of these class objects and the entire population of schedules was an array of these arrays. Set up as these arrays it was relatively efficient to parse through when doing calculations.

6.3 How the Program Runs

The program starts by randomly generating a population of schedules given a certain set of input data. Fitnesses of each schedule are calculated and the best is recorded. Random groups of schedules are chosen to compete in tournaments and the highest fitness values are picked to be parents. New schedules are created from parts from both of these parents. These new children have their fitness value calculated and are placed back in the population replacing the losers of the tournament. Every time a new fitness value is recorded it is checked to see if there became a new leader. The program can be run for either a set length of time, a certain number of fitness calculations or when a certain fitness value has been achieved. A schedule representing the best found is then printed to the screen and can be analyzed.

6.4 Fitness Calculations

Many evaluations of fitness have to be calculated before the program is finished. Each time a new schedule is built it has to be tested to see how good it is. The fitness of each new schedule is calculated by checking each of the 13 constraining factors throughout. Each instance of a violation of a constraint is recorded and a demerit value is given. The sum of all of these demerits for the entire schedule is the fitness value. Since some constraints are worse then others, we give them a very high demerit value while others are minor and a small demerit is added. If a schedule has a fitness value of zero then we know it violates none of our constraints and is a perfect schedule.

6.5 Crossover Methods and Mutation

Generating new children from multiple parents can be done in many ways. The idea is that most schedules have at least a few areas where a small amount of courses fit very well. If we can take a good area from one parent and a different good area from another and combine them we can get a really good child. There are many ways to achieve this and they are called crossover methods.

Uniform crossover Randomly taking courses from either parent.

One-Point crossover Taking courses from one parent up to a certain point and then finishing the schedule with courses from another.

Two-Point crossover Same as one-point but adding another swapping location.

Using Ordered Greed to create schedules forces us to store our population differently. The permutation of the order of classes to place in the schedule is what differs each. Crossover methods used in an Ordered Greed program must also be different.

Partially Matched Crossover The two parents are lined up and two random crossing locations are picked. The values between these locations are one-by-one position swapped to the locations in the opposite parent. The result should be two unique permutations.

Ordered Crossover This crossover is similar to the one above instead of position by position swapping, individual values are slid to fill holes left by values being moved.

Merging Crossover Merge the two parents together, then when you have a list twice as long as one parent with doubles of every value, you take the first set of original values and that becomes one child and then take what is left for the other.

Sometimes your population of schedules does not start with enough diversity or ends up with to many of the same type of schedule and normal crossover methods will not be able to get us into a larger search space. Here the idea of mutation is introduced. Every so often a schedule is altered randomly to give us additional options. The rate that this happens at can be controlled and optimized for fastest results.

7 Deliverables

- Final paper with conclusions.
- Experimental data on optimizing parameters and GA type.
- Program code for final version of schedule maker.
- Explanation of use of program.
- Conclusions on best working program version.

8 Defenses Attended

- The Brachistochrone problem

9 Project TimeLine

- Initial Program - Already Completed
- Optimize Initial Program - By January 1, 2005 (Already Completed)

- Creating and Comparing Other GA Schemes - By January 31, 2005 (Already Completed)
- Combining to Find Best Solution - By February 15, 2005 (Already Completed)
- Create a user friendly way to input real data- By July 20, 2005 (Already Completed)
- Write Final Paper and Produce Graphical Conclusions - By January 1, 2006 (Rough Draft Completed)
- Defend - Late February 2006

10 References

- Goldberg, David E. "Genetic Algorithms; in search Optimization and Machine Learning" Addison Wesley, 1989
- Anderson, Peter G. and Gustafson William T. "Ordered Greed" Online Available *http://www.cs.rit.edu/~pga/abstracts.html*
- Bereza, William I. "Resource Scheduling with Distributed Genetic Objects"
  Online Available *http://www.concentric.net/~Bereza/research*
- Anderson, Peter G. and Ashlock, Daniel "Advances in Ordered Greed" Online
  Available *http://www.cs.rit.edu/~pga/abstracts.html*
- Deitel, H. M. and Deitel, P. J. "C++ How to Program" Prentice Hall, 1998