Math Department Schedule Optimization

Jack Meyers, Daniel Morrison

Faculty Sponsors: Dr. Song Chen, Dr. Chad Vidden, Mathematics

ABSTRACT

The purpose of this paper is to look at two approaches to creating an optimal instructor-class schedule for the UW-L Math Department and to compare their results. There were 118 sections of classes to assign to 38 instructors for the semester. These assignments were based on a series of constraints on time and teaching preferences from each instructor. This scheduling problem is an optimization problem and so there is no way to find the optimal solution efficiently, therefore we chose the two most common methods to solve optimization problems. The first method involved a genetic algorithm model while the second approach used a linear integer programming model. The result was two separate schedules that have been optimized and organized into a convenient format for the math department.

INTRODUCTION

Problem Introduction

The UW-L Math Department presented us with the problem of creating a schedule for one semester that involved assigning every instructor a course load as well as times and rooms for every class. We were instructed to use certain constraints for the schedule in order to: place instructors in classes they had the ability to and preferred to teach, take into account the instructors' time preferences and finally to minimize the number of rooms needed for the schedule. The math department expected us to create a program that would be able to take in the data containing the instructor's class and time preferences and create a master schedule.

Constraints

The math department gave us a list of constraints that we then divided into two categories, hard constraints and soft constraints, based on how they affected the final schedule. Hard constraints are used to create a valid schedule and violating them will mean the schedule is unusable, such as an instructor being scheduled to teach a class that the instructor isn't qualified to teach, or that an instructor is teaching in two rooms at the same time. A soft constraint on the other hand is one that helps create a better schedule, such as giving the instructors their preferred classes and times and, if violated, they simply result in a sub-optimal schedule. The constraints are based on the two groups of instructors, professors and lecturers who both have different requirements and expectations. The two groups of constraints are listed below.

Hard Constraints:

1. An instructor cannot teach more than two classes in a row.
2. No instructor can teach two classes at the same time.
3. No two instructors can teach in the same room at the same time.
4. A professor can teach at most one class from the upper level courses and may teach any amount of the lower level courses.
5. A professor can teach up to three courses.
6. A lecturer is only allowed to teach lower level courses.
7. A lecturer can teach up to four courses.

Soft Constraints:

1. An instructor should teach the same class back to back whenever possible.
2. Use the fewest number of classrooms in the final schedule.
3. Minimize the amount of times that an instructor has to change rooms.
4. Each instructor should teach as many of their preferred classes as possible.
5. Each instructor should teach during as many of their preferred times as possible.

After we created an optimal schedule, the math department needed to manually adjust it in order to account for instructors' personal requests that weren't general enough to require a constraint such as a request to leave early to pick up kids from school.

Sample Data

This section contains a set of sample data shown in Table 1 that is used to demonstrate how to interpret the data and will be referenced in later sections. The data contains matrices filled with the elements $\{0,1,2\}$ with 0 corresponding to an impossible event, 1 to a possible event and 2 to a preferred event. Below is an example of the teaching preference matrix.

Table 1. Sample Teaching Preferences Data

Instructor	Math 151	Math 207	Math 208
Instructor 1	0	2	1
Instructor 2	2	1	1

From Table 1 we can read that Instructor 1 can teach Math 207 and 208, and has a preference to teach Math 207. We can also read that Instructor 2 can teach Math 151, 207 and 208, and has a preference to teach Math 151.

Next is an example of the matrix that contained data about the instructors' preferences for when to teach, which can be interpreted in a similar manner.

Table 2. Sample Time Preferences Data

Instructor	7:45 am	8:50 am	9:55 am	11:00 am	12:05 pm	1:10 pm	2:15 pm	3:20 pm	4:25 pm
Instructor 1	0	2	2	2	2	1	1	0	0
Instructor 2	2	2	1	1	1	2	2	1	0

In the real data, there were 38 instructors, of those 28 were professors and 10 lecturers. There were also 116 sections of 25 different classes to be taught and 9 time slots available throughout the day for teaching.

Problem Formation

When researching potential models for this problem we took the description of the problem and determined that the key feature was that we were subjecting this data to a set of constraints. Upon more research we determined that this problem could be expressed as an optimization problem, for which the goal is to maximize a function over a set of constraints. In our problem, the function to maximize is the schedule, which is optimized over the instructors' time and teaching preferences.

OPTIMIZATION

Optimization is a method used to find the maximum value of a function subject to certain constraints, which can be translated to many real life problems. One common example of optimization is the knapsack problem: assume there is a series of items, each with a unique volume and value, and there is a knapsack with a specific volume that is less than the sum of the volumes of this list of items, the goal is to maximize the value of the items in the knapsack without going over the capacity. Another example would be a factory that is trying to maximize its profits subject to machine and labor costs. An optimization problem is defined as some target function to maximize, call it $f(\vec{x})$, and the constraints, $h(\vec{x})$, that it will be maximized over.

An example target function that solves for the teaching assignments can be constructed by considering the sample teaching preferences from Table 1, shown again below, as well as the solution table for that data, seen in Table 3. The solution table contains variables that are to be solved for, which correspond to how many sections of that class the instructor is teaching. Then the target function can be formed by multiplying the corresponding values in Tables 1 and 3, which would result in the equation:

Table 1. Sample Teaching Preferences Data

Instructor	Math 151	Math 207	Math 208
Instructor 1	0	2	1
Instructor 2	2	1	1

Table 3. Sample Teaching Preferences Solutions

Instructor	Math 151	Math 207	Math 208
Instructor 1	x_1	x_3	x_5
Instructor 2	x_2	x_4	x_6

$$f(\vec{x}) = (0)(x_1) + (2)(x_2) + (2)(x_3) + (1)(x_4) + (1)(x_5) + (1)(x_6)$$

The values for each x_i are $\{0,1,2,3\}$ for a professor and $\{0,1,2,3,4\}$ for a lecturer. These values represent the possible number of times an instructor is teaching some course. Then $f(\vec{x})$ can be optimized over $\vec{x}$ if we set constraints on $f(\vec{x})$ which can be formed by creating inequalities using each of the x_i values. Maximizing $f(\vec{x})$ results in the best schedule for all of the instructors.

To illustrate a constraint, we will look at an example constraint that limits the number of classes each instructor can teach to at most two. To construct this constraint we use the sample teaching preferences solution show again below, then we set inequalities for each instructor on their corresponding x_i values, as shown in $h(\vec{x})$.

$$h(\vec{x}) = \begin{cases} x_1 + x_3 + x_5 \leq 2\text{, constrains Instructor 1 to teaching at most two sections} \\ x_2 + x_4 + x_6 \leq 2\text{, constrains Instructor 2 to teaching at most two sections} \end{cases}$$

After maximizing the target function over the list of constraints, the solutions to the x_is were reformatted into a matrix that corresponded to the teaching assignments. The same process was done with the room assignments. We then added class names and times to the schedule so that it was clearly understandable for the math department.

Optimization problems can be solved by two common methods, integer programming and the genetic algorithm. We decided to model our problem with both of these methods using packages from the R programming language [1] and then compare the resulting schedules, since the goal of the project was to explore methods to solve this optimization problem.

A big obstacle to solving optimization problems is that they belong to a class of problems called NP, which means finding the best solution is very hard computationally. As the size of a problem in NP increases, the computation time needed to solve that problem increases at a rate which makes solving for the exact solution impossible due to time constraints. When we modeled our problem we found that by assigning instructors to classes, times and classrooms, the resulting target function would yield: (38 instructors) (116 sections) (9 time slots) (15 rooms) = 595080 variables, assuming we have 15 classrooms. This many variables would result in unreasonable computation time so we broke up the problem into two smaller optimization problems, which when done separately, would solve the main problem more efficiently. First we assigned instructors to their preferred classes in order to have a list of instructors and classes being taught. Second we used the number of classes each instructor was teaching in order to assign them an equal number of times to teach throughout the day. The assignment of times implied room assignment because the actual room assigned was arbitrary so we only needed to worry about the number of rooms needed at each time throughout the day.

GENETIC ALGORITHM

Introduction to Genetic Algorithm

A genetic algorithm solves an optimization problem by modeling the way living organisms evolve to naturally optimize themselves to their environments. The algorithm has two components that have to be designed for each specific problem, a chromosome and a heuristic function, which are used to find the optimal solution. The concept of modeling computation after biological processes has been around since the very beginning of computation,

brought to light by Alan Turing, John von Neumann, and others. This work on evolution based programming for optimization and machine learning began in the 1950s and 1960s. In the beginning each evolution based program was designed specifically for the problem at hand, and generally involved adding randomization into finite-state machines. Typical uses of evolutionary algorithms include calculating trajectories and biological problems like modeling the brain and evolution. John Holland popularized genetic algorithms in the 1960's with the goal of adapting the real process of natural section into a computer algorithm. He developed the idea for implementing a population of chromosomes with different set of bits representing the problem, including the processes for breeding and mutating chromosomes so the population improves. The creation of the population and how to breed chromosomes within it was a major advance for genetic algorithms, as previously only a few chromosomes were used and were randomly mutated rather than being bred together. Holland's work laid the foundation for further advances, and today the concept of biological algorithms has blossomed into the fields of neural networks, machine learning, and evolutionary computation [2, 3]. We used a package in R called *genalg*, for genetic algorithm, which allowed us to model the problem and perform the evolution process to generate an optimal schedule [4].

Chromosome

In essence, the chromosome represents a solution to the given problem. It consists of a fixed length string of usually binary values. The values in different locations of the string correspond to different features of the solution. An example chromosome for the instructor-class assignment portion of the example problem of Table 3, shown in Figure 1, which holds all the information necessary to define a schedule. Notice the length is the product of the number of instructors and number of sections. Each value in the chromosome represents an instructor-class pair, where a zero value means the instructor does not teach that class and a value of one means the instructor does teach the class. So this chromosome means Instructor 1 teaches Section 3, and Instructor 2 teaches Sections 1 and 2. This chromosome is valid, as all sections are taught by exactly one instructor, and all instructors are qualified for the classes they teach. However, this chromosome isn't optimal, like 011100, as Section 2 is taught by Instructor 2 even though Instructor 1 prefers to teach it. A chromosome like 101010 isn't valid as Instructor 1 teaches Section 1, which they are not qualified for. We will describe chromosomes as a vector of bits and use the mathematical notation $\vec{c}$.

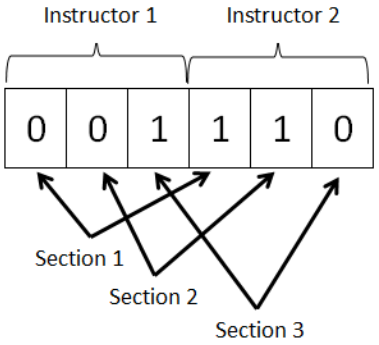

Figure 1. Example chromosome for the simplified problem

When designing a chromosome for a genetic algorithm, the only decision to be made is how long the chromosome needs to be. This depends entirely on how much information a solution to the problem holds; the more complicated the solutions are the longer chromosome is needed to completely describe those complexities. It is useful to design a function that converts a chromosome into a more readable form at the same time as the chromosome design. The genetic algorithm doesn't need this function to operate, but generally the chromosome can't intuitively be read by a person without conversion to another form.

Heuristic Function

The second design element in a genetic algorithm is the heuristic function. This function quantifies how good a chromosome is so the algorithm knows what is optimal, taking into account the hard and soft constraints. Violating a hard constraint means the solution isn't valid, for example two instructors teaching the same same section of a class. In this case the heuristic function will return a minimal value, usually zero, if a hard constraint is violated. Soft constraints cannot be violated, but are met to some degree which determines which chromosomes are better

than others. Soft constraints add to the heuristic score based on how well the soft constraint is met, and they are generally weighted to emphasize which constraints are more important. An abstracted definition of the heuristic is:

$$h(\vec{x}) = \begin{cases} 0 & \text{when } \vec{c} \text{ fails a hard constraint} \\ \sum c_i & \text{when } \vec{c} \text{ meets all hard constraints} \end{cases}$$

where c is a chromosome and c_i is some heuristic score related to how well the input chromosome optimizes the i^{th} constraint. What c_i is, what it means, and how it is calculated depends entirely on the specific problem. Once the the chromosome and constraints are defined for this problem we can create a specific heuristic function that is much better defined.

Algorithm

Given a chromosome size and heuristic function for a specific problem, the algorithm is then able to start finding an optimal chromosome by repeatedly applying a simple process that directly mirrors the biological processes of evolution and natural selection. The algorithm begins by randomly creating an initial population of chromosomes, then uses the heuristic function to rank the population by their suitability as a possible solution. The best chromosomes are then bred and mutated into new chromosomes that make up a new population that consists of marginally better solutions. The size of the population and the size of the population that is bred can be modified as inputs the algorithm. This continues until the heuristic function reaches a maximal value and stops changing. The chromosome with highest heuristic value is then converted into the corresponding solution and the algorithm outputs its result. This process is shown as a flow chart in Figure 2 below.

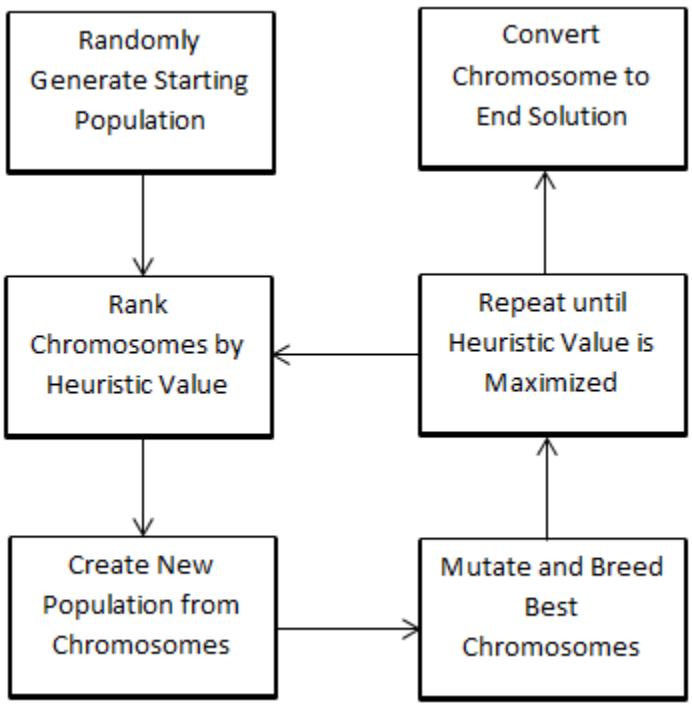

Figure 2. Flow chart for the genetic algorithm

This is a very simple algorithm that knows almost nothing about the problem, only asking how good of a solution it has so far. The only complicated part is evolving existing chromosomes into new ones. There are two ways to evolve chromosomes, called breeding and mutating. Mutating removes portions of a chromosome and randomly reassigns the bits. Which portions of the chromosome are removed is determined randomly, but what percentage of the chromosome is altered is an input to the algorithm. This is done to add new combinations of data

to the population that might not be present. Figure 3 shows this process, and boxes mark the data that was chosen from each parent chromosome.

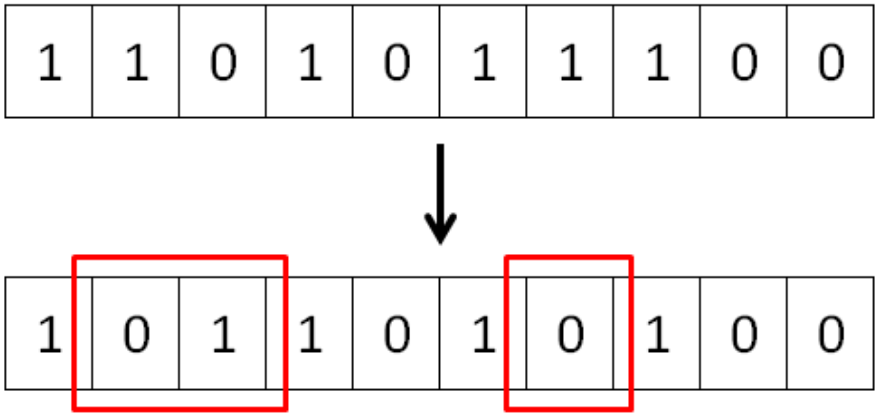

Figure 3. A visual example of the mutating process

Breeding involves two chromosomes and generates a new chromosome that contains a combination of the data from each parent chromosome. Like mutation, a random portion of each chromosome is selected. The new chromosome will then have characteristics of both parents, hopefully taking the best parts of each. The boxes highlight where the second chromosome was altered in Figure 4.

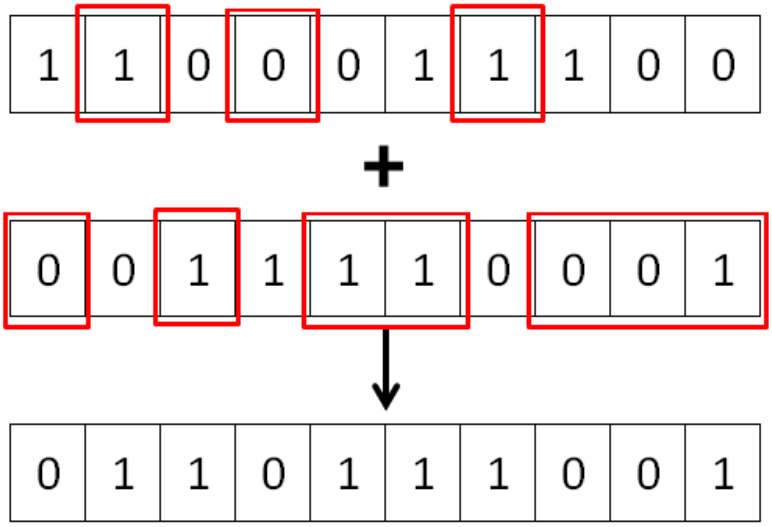

Figure 4. A visual example of the breeding process

Both mutating and breeding are done based on the idea that the resulting chromosomes are at least slightly better than their parent chromosomes, however this is not always true due to the randomness involved in each process. Still, over time chromosomes will tend to improve more often than not and the heuristic values will slowly but steadily improve.

Strengths and Weaknesses

The strength of the genetic algorithm stems from its simplicity. It is simple to quickly define a chromosome and heuristic function and start looking for a solution. Additionally, the heuristic function gives the user incredible

flexibility to make up complicated constraints, as the only requirements are to take a chromosome as input and return a heuristic value. However, the simplicity has a major drawback; the algorithm has difficulty finding near-optimal solutions when the solution space is large. This issue stems from the randomness involved in the breeding and mutating process. The algorithm has no idea whether the changes it is making are good or not, and can only stumble upon the correct solution by chance. This reliance on chance can be good if you are lucky enough to get the right chromosome early, but most of the time this is not the case. The problem is compounded when many of the chromosomes are invalid solutions, as they will all have a heuristic value of 0 so the algorithm has difficulty finding the best chromosomes to mutate. We now return to the original problem to apply the genetic algorithm.

Genetic Algorithm Model

Instructor-Class Assignment: Method 1

Formulation. The initial attempt was simply creating a chromosome with a binary value for each instructor-section pair; 0 means the instructor does not teach the section, 1 means the instructor does teach the section. The heuristic function was not much harder to create, using all the restrictions applicable to the instructor-class assignment.

The heuristic function used in this method is:

$$h(\vec{x}) = \begin{cases} 0 & \text{when } \vec{c} \text{ fails a hard constraint} \\ \sum_{i \in I} \sum_{c \in C_i} (p_{ic} + r_{ic}) & \text{when } \vec{c} \text{ meets all hard constraints} \end{cases}$$

where I is the set of instructors, C_i is the set of classes taught by instructor i, p_{ic} is the preference value for instructor i teaching class c and r_{ic} is the number of times instructor i repeats teaching class c. The heuristic measures how much of a preference instructors have for the classes they teach, along with how many sections of the same class they teach. An example chromosome, using the small example from Table 1, may be 001110. A visualization of the chromosome is displayed above in Figure 1, along with a description of the information contained in the chromosome. This chromosome doesn't fail any hard constraints so it has a non-zero heuristic value. It gets 4 from the preference values, since Instructor 1 teaches Math 209 and Instructor 2 teaches Math 151 and Math 208. As each class has only a single section, it gains no more value from professors having repeat sections. So the chromosome has a heuristic value of 4, out of a maximum of 5.

Results/Discussion. Running this algorithm for several days yielded no good or even valid solutions, which determined that the algorithm would have to be revised for it to produce a solution. The lack of valid solutions is demonstrated by a best heuristic value of 0, showing the chromosomes always failed. The problem turned out to be two-fold; a large solution space and a poor chromosome representation that meant most chromosomes represented invalid solutions.

With 116 sections and 38 instructors, the chromosome was 4408 bits long, which means 2^{4408} possible chromosomes. This is already an incomprehensible number of possible solutions to search through, but the problem is compounded since by far the majority of solutions were invalid.

Consider the simplified situation of assigning an instructor to only one section, ignoring all constraints other than having a single instructor teach the class. There are 2^{38} possible chromosomes in this situation, as all 38 instructors either teach the section or not. However, there are only 38 valid solutions, even less when taking qualifications into account. This makes up a minuscule subset of the solution space, so randomly choosing chromosomes means it is incredibly unlikely that the genetic algorithm finds one of the valid solutions. Then in the full problem the size grows exponentially larger and more constraints are added in, and the genetic algorithm has little to no chance to converge to an optimal solution.

Instructor-Class Assignment: Method 2

Formulation. For the second attempt it was clear that blindly creating a chromosome would not suffice, and finding a more complex but shorter chromosome was necessary. The solution to this problem was to use the structure of the solutions to our advantage. We know each section must be taught by exactly one instructor who is qualified to teach the class, so the portion of the chromosome relating to that section should which of the possible instructors teaches that section. Specifically, the chromosome is an index in the list of instructors qualified for the class. In this case, the chromosome ended up being 555 bits long, a drastic reduction from the previous attempt.

Consider creating a section of the chromosome related to describing the instructor of a single section with preference values shown in Table 4. For Math 151 there are only two instructors who can teach the class, and Math 309 has three possible instructors. The chromosome can then be reduced to an index from 1 to 2 describing which of the two possible instructors will teach Math 151, and an index from 1 to 3 for Math 309. There are then only 6 possible chromosomes in this construction, compared to 1024 possible chromosomes using the method 1 construction. Clearly it will be much easier to maximize over only 6 schedules. Since the chromosome is defined in binary, returning an index is slightly complicated but comes down to converting a binary string to an index. The full chromosome is constructed by stringing together the portions for each section.

Table 4. Preferences for sample chromosome construction

Instructor	Math 151	Math 309
Instructor 1	1	1
Instructor 2	2	0
Instructor 3	0	2
Instructor 4	0	2
Instructor 5	0	0

The heuristic for this method also gets simpler. Because of how a chromosome defines a solution, we are guaranteed that every chromosome matches one section to exactly one instructor, every section is taught, and is taught by a qualified instructor. This means we don't have to check these requirements in the heuristic. Additionally, the hard restriction of 3 or 4 classes per instructor is changed into a soft restriction based on credits to let the algorithm explore more potential solutions. This gives the algorithm more flexibility to move classes around without getting severely penalized by the hard constraint. Changing a hard constraint into a soft constraint can help the algorithm search for possible solutions, but may result in some problems in the schedule. For example, in this case some instructors were given 13 or 14 credits of classes, which is more than the typical value. Depending on how big of a problem this presents, someone may then have to make manual changes to the schedule output by the algorithm.

The heuristic function for this case is slightly different because one of the hard constraints was changed into a soft constraint. The heuristic function is:

$$h(\vec{x}) = \begin{cases} \sum_{i \in I} \sum_{c \in C_i} (p_{ic} + r_{ic}) & \\ \quad - \sum_{i \in I} \max(i_{credits} - 12, 0) & \text{when } \vec{c} \text{ meets all hard constraints} \\ 0 & \text{when } \vec{c} \text{ fails a hard constraint} \end{cases}$$

where I is the set of instructors, C_i is the set of classes taught by instructor i, p_{ic} is the preference value for instructor i teaching class c, r_{ic} is the number of times instructor i repeats teaching class c, and $i_{credits}$ is the number of credits taught by instructor i. It is identical to the previous function but with an added term to measure how many credits are taught by each instructor above 12 so they aren't given too many classes to teach.

Results/Discussion. Running this version of the algorithm proved to be much more successful; running for 6 hours resulted in a schedule needing very little alteration to be useful. In contrast to the last method, the algorithm was able to generate new chromosomes that gradually increased their heuristic values. The package used to run the genetic algorithm is actually a minimization algorithm, so the algorithm is set to minimize the negative of the heuristic value so the optimal value is the smallest value. Figure 5 shows the decrease over time. The reduction made to the length of the chromosome, and the drastic increase in ratio of valid schedules to invalid schedules made the genetic algorithm much more successful. Most instructors teach preferred classes, and repeat sections are usually taught though not as much as is theoretically possible. These two parts of the heuristic may improve with longer running times. The genetic algorithm made nearly all instructors teach 12 of fewer credits, only three taught more. This balance is difficult for the algorithm to find as it isn't always as simple as swapping which instructor

teaches the class. Usually, the solution involves swapping pairs of classes, which is difficult for the algorithm to figure out. This balance is easier for a human to do when only a few instructors need to be changed, and can be done after the algorithm does most of the work. Figure 5 shows how the heuristic value changed over each generation. Note the graph is of the negative of the heuristic values, so a decrease in value on the graph corresponds to an increase in heuristic value and an improved chromosome.

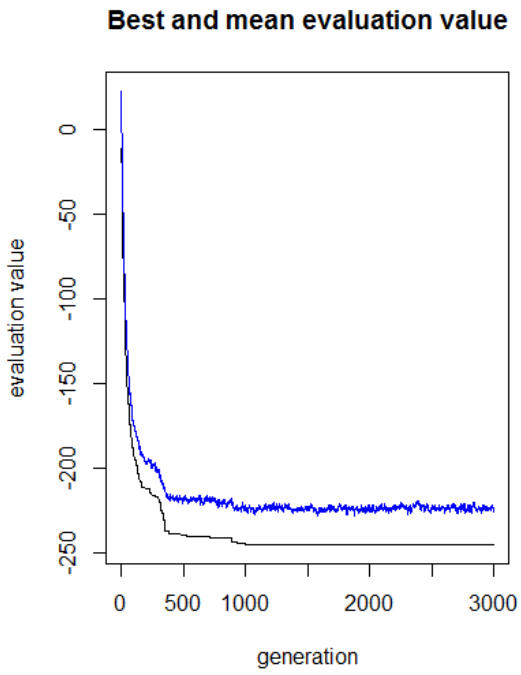

Figure 5. A graph of the best and mean heuristic values over time

Class-Time Assignment

Formulation. Once classes are assigned an instructor, they need to be assigned a time and a room to complete the schedule. This part has fewer restrictions and preferences, and may be easier for the algorithm. For simplicity, the room restriction was abstracted into the number of classes taught at each time, under the assumption that rooms can be arbitrarily organized from the list of classes taught at each time. Note that there is no restriction for time preferences, but time preferences are taken into account with the integer programming algorithm. To simplify the work the genetic algorithm has to do, the problem was organized as a combination of blocks of time each instructor has to place into 8 time slots. These blocks represent any single class the instructor must teach, and the time slots where the instructor does not teach.

Blocks. Blocks are designed so they guarantee that if an instructor teaches the same class, they will teach it back to back in consecutive time slots, and instructors will always have a time slot where they teach no class between different classes. This means no matter how the blocks are organized, they will always satisfy both hard restrictions, and the soft restriction that instructors prefer to teach the same class back to back. The genetic algorithm is then only concerned with minimizing the number of classes and spreading each section's classes across all time slots.

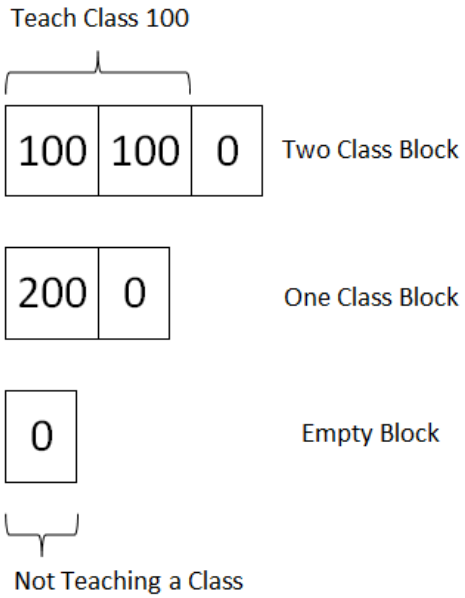

Figure 6. Construction of each type of block

A block consists of a series of time slots, where each time slot is given the class number of the class that is taught at that time or a 0 if no class is taught at that time. There are three structures for the blocks that are constructed depending on the number of classes that belong to the block. Repeat sections of the same class are paired in a block so the classes are next to each other with an empty slot afterwards, for a total length of three time slots. Leftover classes are placed so the class is in one slot and an empty slot afterwards. Finally, empty space is needed to fill any extra time slots in the schedule not taken by the larger blocks with classes, so the last type of block is a single empty time slot.

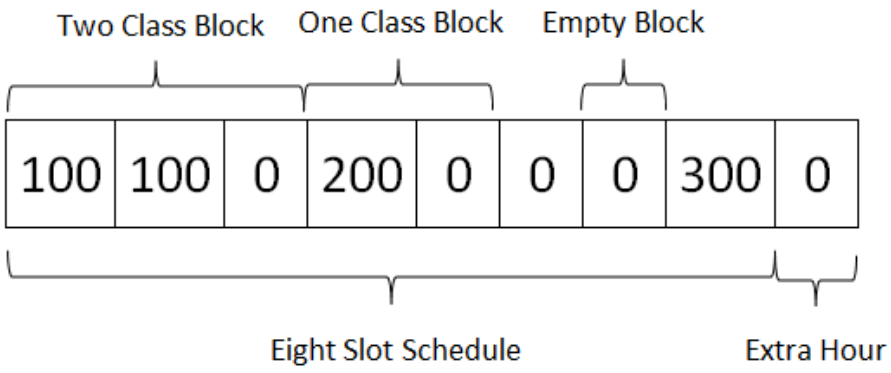

Figure 7. An example for how a series of blocks can be placed into an instructor's schedule

Each instructor's schedule of classes is a combination of the blocks that contains the classes they teach as determined by the first algorithm and empty space to fill out their schedule. The total length of the blocks will be one more than the actual length of the schedule, because each block ends with an empty slot that isn't necessary at the end of the day. For example, if the schedule has 8 time slots that classes can be placed in, the blocks will fill 9 time slots so classes can be placed in the first 8 slots and the last is always empty so it can be cut without losing any classes. This makes sure that classes can be placed into the last time slot of the day even though each block ends with an empty time slot.

Representing Blocks in a Chromosome. The genetic algorithm consists of a chain of binary codes, one for each instructor, that correspond to an index of a combination of blocks to make a daily schedule for each instructor. As mentioned above, the block construction guarantees the hard constraints are met and one of the soft constraints as well. The number of rooms is determined from the number of classes taught during one time slot. The spread of the classes across the schedule is not quantified, but is verified afterwards to make sure the classes are spread out enough. The heuristic function is simplified to:

$$h(\vec{x}) = \begin{cases} 0 & \text{when } \vec{c} \text{ fails a hard constraint} \\ \sum_{t \in T} c_t & \text{when } \vec{c} \text{ meets all hard constraints} \end{cases}$$

where T is the set of all time slots, and c_t is the number of classes in time slot t. The only soft constraint is minimizing the number of rooms used, calculated by adding up the number of classes taught at each time slot.

Results. This algorithm ran for only a few minutes before reaching a nearly optimal arrangement of the classes. When the 116 sections are optimally packed into 8 time slots they use at most 15 rooms at any given time. The genetic algorithm met that goal after only a few hundred iterations. The classes were also very well distributed over all time periods, even without attempting to optimize the spread, due to the genetic algorithm's tendency to randomize the arrangement of each instructor's classes. The success of this portion of the problem compared to the instructor-class assignment was likely due to a combination of a well-defined chromosome that reduced the number of restrictions, as well as reducing the remaining restrictions to a little bit of randomization. The chromosome definition allowed the genetic algorithm to ignore several difficult to satisfy restrictions, like having a gap between classes and placing two of the same class directly after each other. These are very unlikely to happen by random, but when they are guaranteed by design the algorithm doesn't waste time going through schedules that don't work. The room requirement was relatively easy for the genetic algorithm because it doesn't require intricate structure like placing classes next to each other, only needing some random arrangement of each instructor's classes so the classes are spread out enough so not too many are at the same time. This also means there are many more chromosomes that meet this constraint optimally. Randomness is inherent in the genetic algorithm, so it will tend to find optimal chromosomes quickly since it usually only requires a few good random mutations.

Integer Linear Programming

Introduction to Integer Linear Programming

Integer Linear Programming solves for optimization problems by solving for the target function under the constraints with integer value results. This scheduling problem requires integer values because for example, an instructor can't teach a fraction of a class or be assigned to a fraction of a time slot. The first problem of assigning instructors' classes had integer solutions since a lecturer could teach 0,1,2,3 or 4 sections of a class and a professor could teach 0,1,2 or 3 sections. The problem of assigning classes to times had two solutions so the values in the solution of the target function were either a 0 or 1 since an instructor was teaching a class at that time or not.

Integer Linear Programming Form

Integer linear programming solves an optimization problem like the one modeled below by maximizing the target function, $f(\vec{x})$, subject to the constraints, $h(\vec{x})$.

$$f(\vec{x}) = \sum_{j=1}^{n} c_j x_j$$

$$h(\vec{x}) = \begin{cases} \sum_{j=1}^{n} a_{ij} x_j \ \leq b_i \text{ , for } i = \{1, 2, \ldots, n\} \\ x_j \ \geq \ 0 \text{ , for } j = \{1, 2, \ldots, n\} \\ x_j \text{ is an integer, for } j = \{1, 2, \ldots, n\} \end{cases}$$

The constraints in $h(\vec{x})$ are formed by creating lists of inequalities and through defining the optimization problem to have integer values. We built our model of the problem with a target function and list of constraints and then utilized the *lp_solve* package for the R programming language in order to solve our model. The *lp_solve* package used the simplex method in order to maximize the target function, which we then were able to convert and create a final schedule [5].

Simplex Method

Before we explain how the target function and constraints worked together, we are going to look at how the simplex method solves integer programming problems. The simplex method was created in 1946 by George Dantzig in order to mechanize the planning process for the US Air Force where Dantzig worked. The simplex method works by mapping all of the constraints to an n-dimensional space, with n being the number of variables in the target function, and then searching through the space within all of the constraints [6].

Consider a target function with only two variables that may look like the following example:

$$f(\vec{x}) = (1)(x_1) + (2)(x_2)$$

Then the potential solutions for this target function lie in two dimensional space where both x_1 and x_2 are greater than or equal to zero. We can create constraints that reduce the size of the solution space, the area where the potential solutions lie, by creating linear inequalities like those below:

$$h(\vec{x}) = \begin{cases} f(\vec{x}) \leq -2x_1 + 20 \\ f(\vec{x}) \leq -4x_1 + 35 \\ x_2 \leq 15 \end{cases}$$

The solution space lies in the area bounded on the bottom by the x_1 axis, on the left by the x_2 axis and then below all of the inequalities in the graph. In each of the three graphs below, there is a shaded region under one of the inequalities showing where the potential solutions to that inequality lies. The solution space is in the intersection of all three of the shaded areas, seen in the resulting graph.

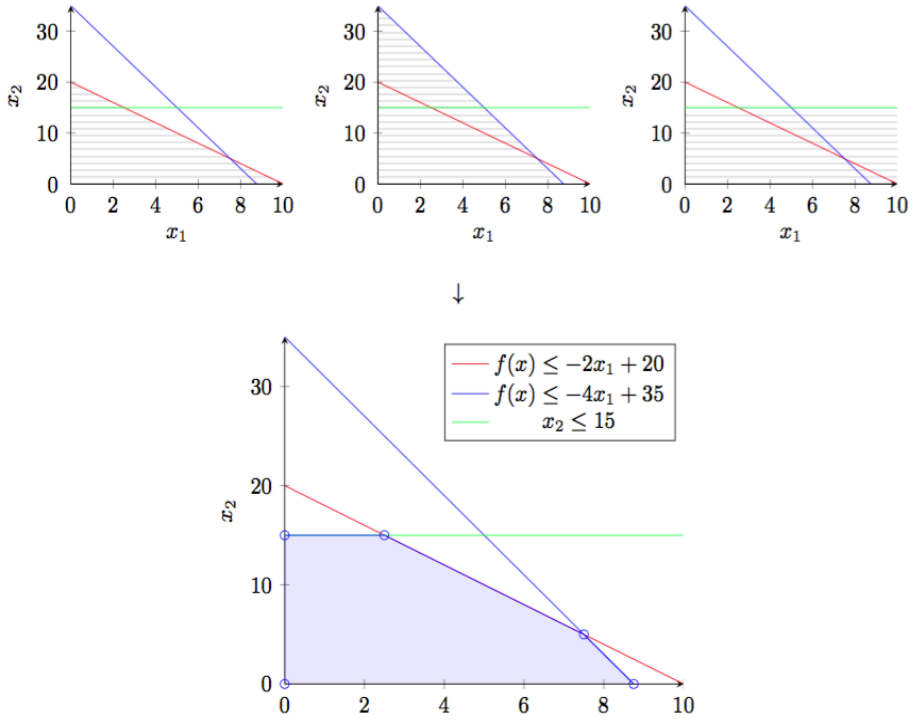

Figure 8. Graph of the solution space resulting from the inequalities in $h(\vec{x})$

Since all of the solutions lie in the solution space, the simplex method then just needs to check the points within this area to find the best solution. This solution space however could end up being extremely large for some problems and searching through every possible solution would become unreasonable, even for a computer. The simplex method relies on a property of this graph that which says the maximal value in this space lies on the boundaries of this space, specifically at the intersection of any of the constraints, marked by the circles on the graph [7]. So in order to find the maximum value the simplex method now just has to search through these intersection points. Another property of this graph is that when at one of these intersection points, if it doesn't correspond to the optimal solution, then there is an edge on that point connecting to another point that leads to a more optimal solution [7]. Using this property, the simplex method starts at an initial point and moves along edges to higher value points until it reaches the optimal solution.

Creating the Model for Integer Linear Programming

Target Function. We will be using a modified example of the data and a modified constraint in order to demonstrate how integer programming works. Below are two tables, the first containing a set of sample data for the the time preferences and the second, the variables to solve for, corresponding to the instructor solutions. The target function, $f_1(\vec{x})$ was created by multiplying corresponding values from the tables together and summing the result.

Table 5. Sample Time Preferences Data

Instructor	Math 151	Math 207	Math 208
Instructor 1	1	2	0
Instructor 2	1	2	2
Instructor 3	0	1	2

Table 6. Sample Time Preferences Solutions

Instructor	Math 151	Math 207	Math 208
Instructor 1	x_1	x_4	x_7
Instructor 2	x_2	x_5	x_8
Instructor 3	x_3	x_6	x_9

$$\downarrow$$

$$f_1(\vec{x}) = (1)(x_1) + (1)(x_2) + (0)(x_3) + (2)(x_4) + (2)(x_5) + (1)(x_6) + (0)(x_7) + (2)(x_8) + (2)(x_9)$$

The coefficients represented the weights on each of the values and the model would choose each x_i value in order to maximize $f(\vec{x})$. Assigning a positive value to an x_i with a coefficient of 0 results in no negative effect to the target function meanwhile it creates an invalid schedule since an impossible event was assigned. In order to prevent this, all of the zero values were converted to -9999 which allowed the model to be biased towards not violating the hard constraints. The model could be biased towards choosing preferred values in a similar manner by converting the 2's to a higher value, in this case 100. Below is the conversion of the above target function, $f(\vec{x})$, to the modified target function $f_2(\vec{x})$.

$$f_1(\vec{x}) = (1)(x_1) + (1)(x_2) + (0)(x_3) + (2)(x_4) + (2)(x_5) + (1)(x_6) + (0)(x_7) + (2)(x_8) + (2)(x_9)$$

$$\downarrow$$

$$f_2(\vec{x}) = (1)(x_1) + (1)(x_2) + (-9999)(x_3) + (100)(x_4) + (100)(x_5) + (1)(x_6) + (-9999)(x_7) + (100)(x_8) + (100)(x_9)$$

Constraints. Each x_i in the target function can be either 0, which corresponds to the instructor not teaching at that time or 1, which means the instructor is teaching at that time. To create a constraint that doesn't allow an instructor to teach more than 1 class in this three hour time period we have to make sure that the row sum for each row in the instructor solutions matrix must be less than or equal to 1. This corresponds to there being only one class taught by each instructor so the sum of the x_i s corresponding to each instructor is at most 1, which can be seen below.

$$h(\vec{x}) = \begin{cases} x_1 + x_4 + x_7 \leq 1 & \rightarrow \text{Instructor 1} \\ x_2 + x_5 + x_8 \leq 1 & \rightarrow \text{Instructor 2} \\ x_3 + x_6 + x_9 \leq 1 & \rightarrow \text{Instructor 3} \end{cases}$$

Then if we look at the variables in the modified target function, $f_2(\vec{x})$, corresponding to instructor 1 and the constraint corresponding to instructor 1, we can see how the target function and constraints interact. To maximize

the target function we need to choose 0 or 1 for each x_i such that the highest value for the target function is found while the constraint hasn't been invalidated.

$$\text{maximize: } (1)(x_1) + (100)(x_4) + (-9999)(x_7)$$
$$\text{subject to: } x_1 + x_4 + x_7 \leq 1$$

To maximize the modified target function, choose: $x_1 = 0,\ x_4 = 1,\ x_7 = 0$. Then, the maximum value for the target function is $(1)(0) + (100)(1) + (-9999)(0) = 100$, and $x_1 + x_4 + x_7 = 1$ which means the constraint is still valid. Then the schedule has been optimized for instructor 1, so when solving for the full target function all of the constraints are considered at the same time in order to optimize the schedule for all of the instructors.

One of the harder constraints to create was to force two of the same class to be taught back to back by the same instructor. This constraint initially seemed like it could be created by using an interesting application of the time preferences, however that failed. It was then decided to break up the assignment of times for classes being taught more than once by one professor into its own problem and then combine it with the classes only taught once after they had both been assigned. Tables 5 and 6 can be converted in order to solve for these double classes by converting them to Tables 7 and 8 below.

Table 7. Sample Time Preference Data (Doubles)

Instructor	8:50 & 9:55	9:55 & 11:00
Instructor 1	1	2
Instructor 2	1	2
Instructor 3	0	1

Table 8. Sample Time Preference Solutions (Doubles)

Instructor	8:50 & 9:55	9:55 & 11:00
Instructor 1	x_1	x_4
Instructor 2	x_2	x_5
Instructor 3	x_3	x_6

The time preference table was reduced by removing the second to last column, this made it so that every class was assigned based on the starting time except for the last class which was assigned based on the ending time. This meant that instructors who didn't want to teach before a certain time were still able to ensure that preference and instructors that didn't want to teach at the last available time were able to avoid that. The same procedure was done with the new instructor solutions table. Thus the target function for the double classes, call it $g(\vec{x})$ can be represented as follows:

$$g(\vec{x}) = (1)(x_1) + (1)(x_2) + (1)(x_3) + (0)(x_4) + (2)(x_5) + (2)(x_6)$$

Then $g(\vec{x})$ was optimized via constraints that didn't allow two classes to be taught back to back, in order to prevent overlapping the classes and to minimize the number of rooms used. Once $g(\vec{x})$ was optimized, then the results were expanded back into the original sized matrix. This was achieved by appending a new column to the end and then adding a 1 to every cell to the right of a cell with a 1, to represent there being a class taught at that time. This can be seen below with a sample solution to $g(\vec{x})$, seen in Table 9, that was converted into the single class schedule seen in Table 10.

Table 9. Sample Time Preference Solution (Doubles)

Instructor	8:50 & 9:55	9:55 & 11:00
Instructor 1	1	0
Instructor 2	0	1
Instructor 3	0	1

→

Table 10. Sample Time Preference Solution (Singles)

Instructor	8:50 am	9:55 am	11:00 am
Instructor 1	1	1	0
Instructor 2	0	1	1
Instructor 3	0	1	1

Once the classes being taught twice were all assigned, they were removed from the list of classes that needed to be assigned, and the times were marked with a 0 in the time preference matrix in order to represent that it was impossible to schedule another class for that instructor at that time. The single classes were then assigned in a similar manner and the two schedules were merged at the end.

Integer Linear Programming Model Methodology

The robustness of the *lp_solve* algorithm allowed for the constraints to be created very exactly and the model was able to be solved for almost perfectly according to these constraints. Due to this, it was decided that every lower level class with more than one section being offered should be taught in tandem with the same class by the same instructor at back to back times in the schedule. This was easy to implement by simply assigning two of these classes to each instructor and then when assigning rooms, the classes were placed back to back as demonstrated in the previous section. This method was viewed as beneficial to the schedule since it optimized two of the soft constraints given for the schedule.

When deciding how to schedule the rooms, instead of picking specific rooms to assign each section, we chose to assign each section a time slot corresponding to a room being needed at a certain time. This meant that if there were 20 classes being taught at 12:05, then there would need to be 20 rooms also available for that time and they could be assigned later. Then in order to minimize the number of rooms needed throughout the day, the number of classes taught at each time slot was simply minimized by setting it less than or equal to some chosen constant. In theory, it would be possible to limit the max number of rooms to only 14 per time slot however when compared to the current schedule, it created a very different distribution of classes. While many instructors are able to teach in the first and last time slots, there were only 10 classes being taught between the two of them in the real schedule so it was decided to limit the number of classes being taught at those times in the integer programming model as well in order to better replicate the original schedule. It is possible to assign a unique constant to each time slot in order to limit the number of rooms needed throughout the day to different values, however it made more sense to pick one constant for throughout the day until those numbers were provided.

Strengths and Weaknesses

Integer programming was very efficient in finding an optimal schedule and was able to create a schedule that didn't invalidate any of the hard constraints and optimized over the soft constraints very well. Modeling the constraints as linear inequalities proved to be difficult and in order to create a final schedule we needed to solve four separate models and combine their results. Another weakness of the integer programming model is that it was only ever able to find one optimal schedule. The schedule that it found proved to be very effective however having multiple optimal schedules to choose from would be beneficial for the math department. Finally, there is no guarantee that integer programming will always find an optimal solution in a reasonable amount of time, due to the fact that this problem is NP-hard. As mentioned before, this means that there is a point at which the solution space gets large enough to where the algorithm won't find a solution in a reasonable amount of time and determining this point would be difficult. We never encountered this issue however so we were able to find these optimal results.

COMPARING THE RESULTS

Genetic Algorithm

Overall, the genetic algorithm produced a schedule that is at least as good as the actual schedule that was used by the math department, and in some aspects is better. The generated schedule had a total instructor preference value of 176, compared to 180 for the real schedule, which were computed by feeding the two schedules into the heuristic function. The generated schedule faltered when it came to giving instructors new sections for classes they already teach, and getting these repeated sections to be taught back to back. Repeat sections will always be a fault in schedules created with the genetic algorithm, because the randomness involves makes it difficult for any one schedule to get every possible section next to another of the same class. This leads into the problem with creating back to back sections. The generated schedule guarantees that whenever it has two sections from the same class, it puts them back to back. However, if an instructor has 3 of the same class, only two can be placed after each other and the third is left alone. The real schedule was able to give several instructors 4 of the same class, which allows for two pairs of back to back classes. The genetic algorithm was able to make improvements to the number of rooms used and the distribution of classes among the instructors, scoring better than the real schedule in these benchmarks. The genetic algorithm had no instructors teaching a single class, and over 80 percent of the instructors taught 3 classes, creating a more equitable distribution of the class load. The genetic algorithm successfully created a schedule that is equitable and sometimes better than the real schedule while only taking two hours to complete.

Integer Linear Programming

The integer linear programming model created a schedule that outperformed the current schedule in almost every measurable way. First, the ILP schedule was able to give every instructor with teaching preferences only their preferred classes, meanwhile for the current schedule, there were 3 instructors teaching classes they didn't prefer to teach. The ILP schedule also ensured that every single instructor taught at least one pair of classes and that pair was always back to back in the schedule. In fact, since professors only taught 3 classes, this meant that every professor was assigned one pair of the same class and then a third class therefore reducing their course preparation load to at most two different courses. The lecturers also benefited since they were all assigned strictly pairs of classes and thus they were also assigned at most two different courses. The assignment of pairs of the same course to be taught back to back was clearly handled much better by the ILP schedule and it results in less course prep and room changing for every instructor. In the ILP schedule, the number of rooms needed was reduced to a max of 16 while still maintaining the instructor's time preferences as well as accounting for wanting to teach fewer classes at the very beginning and end of the day. Finally, the number of sections taught per instructor was distributed better in the ILP schedule since the fewest number of sections an instructor taught was 2, meanwhile most instructors taught 3 classes. Overall the ILP schedule outperformed the current schedule in almost all of the benchmarks, while being calculated in about 10 seconds.

Head to Head Comparison

Below is a table comparing a few key benchmarks of each schedule that were based on the constraints given to us by that math department that were listed previously in the Introduction section.

Table 11. Comparison of benchmarks between the actual schedule and the two generated schedules.

Benchmark	Actual Schedule	Genetic Programming Schedule	Integer Programming Schedule
Execution Time	weeks to months	2-6 hours	about 10 seconds
Max Rooms	19	16, could go lower	18, could go lower
Time Preferences Met	8	7	9
Number of Non-Preferred Classes Being Taught	3	34	0
Back to Back Repeat Classes	36	22	40
Number of Instructors Teaching Too Many Classes	3	3	0
Number of Instructors Teaching at an Unavailable Time	All schedules performed equally well with no scheduling at unavailable times.		

The results of this comparison show that the schedule created via the integer programming method proved to be more efficient and more successful than both the genetic programming generated schedule and the actual schedule. One likely reason that the integer programming model performed so much better than the genetic programming model in assigning back to back repeating sections is because the integer programming model was built around giving instructors two repeating classes back to back and the genetic programming model was only able to achieve that effect through randomness. The schedule has a definite structure that the integer programming model was able to exploit, the genetic programming model on the other hand results in very little structure which seemed to be a detriment in this specific optimization problem. Overall both schedules performed well and both models are more efficient than creating the real schedule by hand which provided marginal benefit over the genetic programming model and none over the integer programming model.

CONCLUSION AND FUTURE WORK

In conclusion, we both learned about two powerful optimization methods and how to utilize them in a real life application to create viable schedules for the UW-La Crosse math department. We ran into a few issues along the way and both had to end up scrapping our original approaches and find better, more efficient methods, however that was part of the learning process. We plan to demonstrate the results and effectiveness to the math department in the hopes of them using our code to create future schedules. If the math department is interested, then we will work to refine our code and create a GUI so that the programs are more user friendly. We would also create a standardized format for the input files and train a member of the math department on the program so that they could use the program without our assistance.

ACKNOWLEDGEMENTS

We would like to thank our faculty advisors, Dr. Song Chen and Dr. Chad Vidden, for their guidance throughout our work on the project and the writing of this paper.

REFERENCES

[1] R Core Team. **2015**. R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna, Austria.
[2] Mitchell, M. **1998**. An Introduction to Genetic Algorithms. MIT press.
[3] Dianati, M. Song, I. and Treiber, M. An Introduction to Genetic Algorithms and Evolution Strategies.
[4] Ballings, M. and Willighagen E. **2015**. genalg: R Based Genetic Algorithm. R package version 0.2.0.
[5] Konis, K. and lp_solve. **2014**. lpSolveAPI: R Interface for lp_solve version 5.5.2.0. R package version 5.5.2.0-14.
[6] Dantzig, G.B. **1981**. Reminiscences about the origins of linear programming. Stanford University, Department of Operations Research.
[7] Murty, K.G. **1983**. Linear Programming. John Wiley & Sons.