An Integer Programming Solution to the University Class Scheduling Problem

---

A Thesis Presented to

The Honors Tutorial College

Ohio University

---

by

Megan Thomas

2009

Table of Contents

Chapter 1

Introduction

This thesis presents methods for solving the University Class Scheduling Problem (UCSP) by Integer Programming. Integer Programming is a branch of Operations Research that solves intricate real-life problems by modeling them as a set of mathematical equations. The UCSP is an example of such a complex system and is a prototypical example of a problem that can be solved using mathematical modeling.

In the UCSP, professors list their course preferences and are designated two or more courses based on their input and availability. Then the courses are scheduled to time periods during school hours. The model becomes increasingly complex because of many scheduling constraints. Some examples of constraints are professors' course preferences, preferred lecture times during the day, and scheduling multisection classes.

The Integer Programming model for this thesis will represent each of the above-mentioned constraints and other constraints using linear inequalities or equations. Concepts from disciplines such as applied mathematics, computer science, and systems engineering are combined to construct the model. The goal of the model is to find a feasible class assignment schedule for a given department while maximizing teacher-assignment satisfaction.

The model is written on the mathematical modeling language AMPL. The rest of this chapter gives an introduction to Integer Programming, its applications, solution methods and a short introduction to AMPL.

1.1 - Linear and Integer Programming

Linear Programming is an area of applied mathematics that deals with special types of

optimization problems. A Linear Program is an optimization problem in which a linear objective function subject to linear equality or inequality constraints is either maximized or minimized [5]. The variables of a Linear Programming (LP) model can be discrete or continuous. An Integer Programming (IP) model is a special case of LP in which all or some variables can only take integer values.

To achieve a goal in an optimization problem, quantified decisions are made. *Decision variables,* whose values we wish to determine, will form the solution to the model. These variables will be optimized through the use of an *objective function*. The decision variables $\{x_1, \dots, x_n\}$ are subject to various *constraints*, which are restrictions in the form of linear equations or inequalities. One constraint common to IP models is that the variables must take integer values. While nonlinear IP problems exist, this thesis will only be using linear IP models, which have easier solution methods [7].

The general form of an IP optimization model looks like this:

minimize (or maximize) $f(x_1, \dots, x_n)$	(objective function)
subject to $g_i(x_1, \dots, x_n) \leq b_i$	(functional constraints)
$x_1, \dots, x_n \leq 0$, integer	(set constraints)

A *solution* to such an IP model is an assignment of values to the decision variables $\{x_1, \dots, x_n\}$. A *feasible solution* is one which satisfies all of the constraints of the IP model. An *optimal solution* for a minimization problem is a feasible solution which achieves the lowest value in its associated objective function. An Integer Program might have more than one solution, that is, there may be more than one solution with the same associated optimal value.

1.2 - Applications of Linear Programming

Linear Programming has many useful applications in the real world, including scheduling, inventory management, network design, facility location problems [5] [7]. Some of them are discussed briefly below.

Employee scheduling, as in the University Class Scheduling Problem, is one example of a real world LP application. Companies such as United Airlines have also used LP techniques to solve employee scheduling problems. The number of flights arriving or departing on any one day should be maximized; this should not be at the expense of the safety of the passengers and staff of the airplanes. Other types of scheduling problems exist, such as finding an optimal schedule of tasks for an auto repair shop which satisfies as many customers as possible.

Other examples of LP applications include inventory management problems. In this instance, LP readily helps companies make decisions balancing costs, profit, and demand for different items that they may produce.

LP may also be used for network design. One example of a network design problem is the minimum spanning tree problem, which is applicable to many companies, especially those which transport data. In this case, each *node* of the network must be connected to the others with minimum total distance [5].

Another real world application for LP is facility location. When considering where to build a new facility, a company must consider many constraints such as price and proximity to other facilities and customers. An LP model can be formulated in this situation to help companies make critical decisions involving resources.

1.3 - Solution Methods for Linear Programs

LP models have been thoroughly studied and have established algorithms for solving them, such as the Simplex Method and the Interior Point Method [5]. The first method, the Simplex Method, was originally developed by George Dantzig in 1947. It has been named one of the top ten algorithms having "the greatest influence on the development and practice of science and engineering in the 20th century" by the journal *Computing in Science & Engineering Magazine* [8]. The Simplex Method is based on the fact that optimal solutions for LP are along the boundary of the feasible region. The Simplex Method involves initializing a Corner Point Feasible solution (CPF) and then traveling along the boundary to adjacent CPF solutions of the feasible region and testing for optimality. Though the Simplex Method has exponential running time in the worst case, it is remarkably efficiently in practice. See Figure 1 for the structure of the Simplex algorithm.

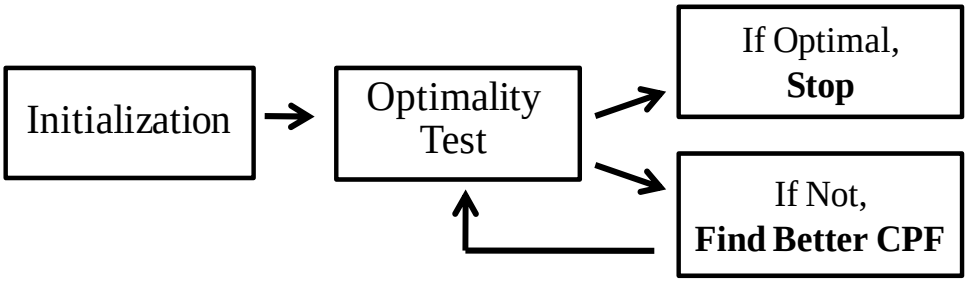

Figure 1. The Structure of the Simplex Algorithm

The Interior Point Method was developed in 1984 by Narendra Karmarkar, and has been proven to be able to solve larger LP problems than Simplex, as well as solving established problems faster [5]. Like Simplex, the Interior Point Method is an iterative algorithm which begins with a trial solution. Contrary to Simplex, it optimizes by traversing trial solutions on the interior of the feasible region.

1.4 - Solution Methods for Integer Programs

IP models are harder to solve than their LP counterparts, and thus the IP solution methods are less time efficient. Since there are efficient solution methods for LPs, one approach to solve an IP problem could be to solve its continuous LP relaxation and then round the values of the variables to integers. While this is an intuitive approach to solving an IP problem, it is in no way guaranteed to obtain an optimal solution. In fact, it is often far from optimal, and should not be considered a valid solution method for IP models. However, it is a technique used as part of other main solution methods, discussed below.

The main solution methods for IP problems are Branch-and-Bound and Cutting Planes. The underlying concept of Branch-and-Bound is to *divide and conquer* [5]. The original problem is divided into smaller and smaller sub-problems, until it can be conquered. *Branching* partitions the problem into subsets which cover the entire feasible region. Then the *bounding* procedure computes a bound for the best value of each subset by solving the continuous LP relaxation for that subset. Any subset whose bound is less than the incumbent, or currently best, integer solution (in a maximization problem) is discarded. This is called *fathoming.* Branch-and-Bound is guaranteed to return an optimal solution after a finite number of iterations. But since each iteration involves solving several LPs, Branch-and-Bound might be inefficient for large instances of Integer Programs.

Another technique for solving IP problems is the use of Cutting Planes. This method adds new functional constraints that do not remove feasible integer solutions, thus the original IP problem remains the same. At the same time, these new constraints cut off some fractional solutions from the original feasible region, making the feasible region of the continuous LP

relaxation smaller. This tighter formulation makes the search for an optimal IP solution more efficient.

Unfortunately, Cutting Planes are not guaranteed to give an optimal integer solution. Thus, a powerful combination of Branch-and-Bound and Cutting Planes, called Branch-and-Cut, was developed to give an efficient algorithmic approach to solving IPs. The use of Cutting Planes on the sub-problems rising in Branch-and-Bound gives tighter bounds and thus accelerates the solution process. Branch-and-Cut is widely used to solve large IP problems. More about these techniques can be found in [3], [5].

Another method for solving IPs is the use of *meta-heuristics*. A *heuristic* is a procedure which is likely to discover a good feasible solution very fast, but does not guarantee an optimal solution. A meta-heuristic is a general solution method that provides both a structure and a strategy for developing a specific heuristic method to fit a particular type of problem [5]. Examples of meta-heuristics include Tabu search, Genetic algorithms, and Simulated Annealing.

These heuristics are mainly based on physical phenomena. These approaches are used for very quickly getting a good integer solution, but do not guarantee an optimal solution. Some of the heuristics, such as approximation algorithms can guarantee that the solution is within a certain factor of the optimal solution, however. Normally, a rigorous mathematical analysis is required to prove the existence of the approximation factor. Heuristics can also be used to provide a starting point for Branch-and-Bound, as a good incumbent integer solution can greatly reduce the number of subproblems, which increases efficiency. For further discussion of heuristics see [9], [10].

1.5 - AMPL

For this thesis, the programming language AMPL was used to write the model. AMPL is compatible with several IP solvers, which use a mix of Branch and Bound and Cutting Plane techniques. AMPL is noted for its similarity to customary algebraic notation [3], which can make it more intuitive to use. The programming language itself is flexible and general, and the command environment is designed to communicate with a wide variety of solvers [3]. It is used for solving many of the different applications of LP, discussed in an earlier section, on a larger scale than would be possible without a computer's aid.

Next will be a short discussion on the construction of a basic model in AMPL. This discussion is needed to understand the model in Chapter 2. To define the input data to an LP, AMPL uses structures called *sets* and *parameters.* A *set* is used for defining a group of objects. To declare a set name, the keyword "set" is used.

```
set P;
```

For example, if the LP is about a production system, P could denote the set of possible products.

The members of set P might possess numerical attributes relevant to the LP. To define these attributes AMPL uses a structure called *parameter.* Parameters are declared using the keyword "param." Parameters may take the form of a single scalar value, or a collection of values which are indexed by a set (or several sets). Below, the indexing for parameter c is given by { j in P}, so that a value is associated with each product in P.

```
param b;
param c { j in P};
```

For example, if P is the set of products, then c[j] might be the unit profit of product j. Keyword “var” is used to declare variables. As with parameters, variables may take the form of a single scalar value, or a collection of values which are indexed by a set (or several sets).

```
var X {j in P};
```

For example, the variable X[j] might denote the number of each product j that the company is going to produce. These values will be determined by the solver.

Objective functions are given an arbitrary name, such as “profit.” An example of a declaration of an objective function is listed below. Keywords in the phrase below include *maximize,* which tells the solver to maximize the objective function.

```
maximize profit: sum {j in P} c [j] * X [j];
```

Here, c[j] * X[j] is the profit from product j. Keyword “sum” is used to take the sum of the profits from all products j in P. Thus, the objective function maximizes the total profit for the production system.

Lastly, a constraint is declared with the keyword *subject to*. An example is provided below.

```
subject to Limit {j in P}: 0 <= X [j] <= u [j];
```

In this example, u[j] represents the maximum capacity for production for each product j in P. Thus, the constraint ensures that the production amount of each product j in P is greater than or equal to 0 but no greater than the maximum capacity for j in P. The actual constraint appears after the “:” . The expression before the colon, “Limit{j in P}”, provides that there is one of that type of constraint for each j in P

For further discussion of modeling with AMPL see [3].

1.6 - Overview of Thesis

The IP model for the UCSP was broken into smaller problems that could be tackled more easily before trying to obtain a final solution. The two smaller problems are the Teacher

Assignment, which matches professors to courses, and the Time Assignment, which matataches the professor-course pairs to times of the day. Thus, the programs are meant to be run in a sequence to obtain an optimal solution. The separate parts of the model could be run multiple times with different data. For example, the scheduling is intended to be run first with permanent staff and later with graduate students, since decisions on admittance of graduate students often makes the scheduling of their teaching schedules a last minute process for universities.

After the two sections were completed, there was the option of either integrating both sections into a larger model or keeping both separate. There are advantages and disadvantages to both ways of types of organization. The advantage of a larger program is that everything is done at once, and if there is a solution to the complete problem then it is instantly displayed. However, larger and more complex programs have a greater risk of being infeasible. Also, the running time of such a program could be slowed down because of larger data sets.

Separate models require that the programmer submit the output from the initial model as input to subsequent sections of the program. In this thesis, each time a solution for the Teacher Assignment model is found, it is then submitted to the Time Assignment section in order to run the second model. This approach has advantages as well as disadvantages. The drawback to this approach is that multiple steps must be taken to reach the final solution. However, separating the larger problem into individual pieces allows the program, and thus the department, more flexibility in the course scheduling process. For example, last minute time conflicts could easily be solved by running the time assignment model again. Separate sections allow the programmer to make small changes to the models more easily and to resolve potential conflicts more efficiently. Therefore, the multiple sections approach is the most practical for the UCSP and was used in this thesis.

The decision of which constraints to include in each model is very important. It is possible for some constraints be incompatible with other constraints if too many preferences are taken into account. The model should be as realistic as possible, but not at the cost of making the model infeasible. In the end, the most important constraints were included in the final version of the model, and the models have been tested using realistic-sized data sets. In Chapter 2, the UCSP model will be explained in detail and Chapter 3 will discuss the experimental results of the model.

Chapter 2

IP Model for University Class Scheduling Problem

2.1- Teacher Assignment Model

2.1.1- General Discussion

The goal of the Teacher Assignment model is to find satisfactory pairings of professors and classes which will fill all of the requirements of the department. One requirement is that the model must be able to assign each professor a specific number of classes to instruct. To obtain a working schedule, the correct number of sections for each class must also be filled.

Ideally the professors should be happy with the courses they are assigned. Each professor submits a list of three courses ranked in order of their teaching preferences. The integer program not only finds a feasible solution of teacher pairings, but it also maximizes the number of professors who are assigned to their favorite classes.

It is important to note that graduate students are assigned classes after the full-time professors. The graduate students help to fill out the lower level courses that have multiple sections, and they are not generally given a high priority in the scheduling of classes. In fact, an administrator may not know which graduate students are attending the university or what their schedules will be until the last minute.

One way to deal with this situation is to assign leftover courses to the graduate students by hand after the permanent professors have class assignments. Alternatively, the integer programming model could also be run a second time on the remaining courses for the graduate students. This would be done by changing the input in the data section- the classes assigned to full-time professors would be taken out of the data set the second time through, and the set of professors would be changed to the names of the graduate students.

A detailed discussion of the AMPL model on the Teacher Assignment model is given in

the next five subsections.

2.1.2- Sets

```
#Given sets of professor and class names
set PROFESSORS;
set LOWER;
set UPPER;
set CLASSES:= LOWER union UPPER;
```

The sets included in the model are professors and the classes available for that quarter. A problem that arose in the development of the programs was what to do with multi-section classes. Upon study of historical mathematics class schedules, it seems that almost all the multi-section classes were lower level classes. The upper and lower level sections of classes in this model were dealt with separately in order to simplify later constraints, especially the constraints in the time assignment model.

2.1.3- Input Parameters

```
# The number of classes each professor is required to teach
    param avail{j in PROFESSORS}, default 2;
```

There are three types of input parameters in the teacher assignment program. The first type of input parameter is the number of classes the professors are required to teach. The default workload is two courses. Other amounts of courses could be entered for specific professors, for example, if a teacher were on early retirement or if he or she was only working part time.

```
# A listing of each professor's favorite classes
param preferences{i in CLASSES, j in PROFESSORS}, default 7;
```

The second type of input parameter is a crucial part of the decision process. At Ohio University professors are allowed to indicate several courses, ranked in order of their preference. A professor's most favorite class is assigned the value 1. His or her second favorite course is assigned the value 2, etc. The default value for the professor's classes is chosen to be 7, which provides weight to the classes that are not a professor's favorite. This type of input parameter is important because one of the goals of the model is to find not only a solution that assigns courses to professors, but one which satisfies as many professors' course preferences as possible.

```
# The number of sections of lower level classes
param lower_sections{l in LOWER};
```

The third type of input parameter is the number of sections of lower level classes that need to be filled. Only courses which have multiple sections are included in this type of parameter, given the way the sets were created. The nature of the lower level course decision variable (see the following section) allows professors to be assigned to 0, 1, or 2 sections of any multi-section course.

Note that the specific values for the sets and parameters are given in input data sets which will be discussed in Chapter 3.

2.1.4- Decision Variables

There are two types of decision variables in the Teacher Assignment of the model.

```
 # Decision variable
# If an upper-level class is assigned to a professor
# Then the value is 1, otherwise 0
        var chosen_upper{u in UPPER, j in PROFESSORS} binary;
```

The first is a binary assignment variable for the upper level classes called *chosen_upper*. Each upper level course is only taught once per quarter, professors class pairings for this variable are given the value zero if the professor does not teach that class and a one if it has been assigned to her.

```
#Decision variable -
#If a lower level class is assigned to a professor,
#The variable can either be 1 or 2
#Otherwise variable is 0
        var chosen_lower{l in LOWER, j in PROFESSORS} integer, >= 0, <=2;
```

The second type of decision variable, *chosen_lower* can take integer values between 0 and 2, because this variable assigns professors to lower level, multi-section courses. A professor may teach one lower level course or none at all, just as in the assignment of upper level courses. The reason that this type of decision variable is kept separate from the *chosen_upper* is that professors may be assigned to two sections of the same lower level course. For example, it is not uncommon for a professor to be assigned two sections of Calculus I, but the binary nature of the variable *chosen_upper* would prevent this sometimes necessary situation from being allowed in the model. While having more possible values for the *chosen_lower* assignment variable makes it less preferable to the binary variable in *chosen_upper*, it was necessary to make the model

more realistic by including as many schedules as possible in the solution set for the model.

2.1.5- Constraints

```
#Constraint - Each professor is required to teach a certain number of classes
    subject to required_to_teach{j in PROFESSORS}:
        sum{l in LOWER} chosen_lower[l,j] + sum{u in UPPER} chosen_upper[u,j] =
        avail[j];
```

The first constraint, *required_to_teach* provides that each professor is assigned the correct number of classes. The number of classes from the parameter avail, which gives each professor's workload, must be equal to the sum of all of the upper and lower level classes to which the professor is assigned. This is an important logical constraint for the model.

```
#Constraint - Each upper level class needs exactly one teacher
    subject to filling_upper{u in UPPER}:
        sum{j in PROFESSORS} chosen_upper[u,j] = 1;
```

The second constraint, *filling_upper*, concerns upper level classes. It provides that each upper level class is assigned to exactly one professor. Thus, each upper level course is assigned some professor and no two instructors could be assigned to teach a class that only has one section per quarter.

```
#Constraint- Filling lower-level classes
    subject to sections{l in LOWER}:
        sum{j in PROFESSORS} chosen_lower[l,j] <= lower_sections[l];
```

The next constraint, *sections*, concerns filling of multiple sections of lower level courses. This inequality provides that the sum of all the teachers assigned to lower level courses should be less than or equal to the total number of sections for that class. The reason this constraint is not held to equality is that many times lower level class sections are assigned to graduate students. As was mentioned earlier in the thesis, there are simply too many sections of lower level courses to fill with upper level faculty. The courses that do not have all sections filled by the model will be filled in at a later time by hand with incoming graduate assistants.

```
#Constraint - Each individual professor should get at least one preferred class
    subject to prof_preferences{j in PROFESSORS}:
        sum{u in UPPER}preferences[u,j]*chosen_upper[u,j] + sum{l in
        LOWER}preferences[l,j]*chosen_lower[l,j] <= 9;
```

The final constraint, *prof_preference*, addresses the need for professors to be assigned to their preferred courses. As stated earlier, each professor is given the opportunity to rank his or her favorite courses. The top favorite course is given the value 1, the second favorite course is given the value 2, and the third favorite course is given the value 3. Recall that all other courses are given the default value of 7 for that professor. The constraint ensures that each professor is assigned to at least one class that he or she likes. As explained below, this is accomplished by multiplying the value of the preference parameter given to each course by the decision variable associated with that class.

Note that if a course was not assigned to that professor, then the preference parameters for those courses would all be multiplied by zero. This implies that those courses will not affect the overall value of the inequality. Thus, only the two classes which have been assigned to that

specific professor will affect the value of this constraint. The decision variable for those classes will be either the value 1 or 2, depending on whether the professor is assigned to one or two sections of that class. The value 9 on the right hand side of the inequality was chosen to make certain that each professor is assigned to at least one of his or her top two favorite courses. This is because any combination of the top preferred courses will certainly be less than 9. Also, if a professor is assigned to a non-favorite class, then that professor must also be assigned to either his or her first or second choice of classes as only 1 + 7 and 2 + 7 will make the inequality hold.

Also note that two situations regarding non-favorite classes will be rejected by the model. If the professor were teaching a third favorite and a non-favorite class, or two non-favorite classes, the inequality would not hold. The sum of the professor's preferences would be 10 or 14- both of which are clearly greater than 9. Thus, one can see that each professor must be assigned to at least one of his or her favorite classes.

On a final note, the value 9 was chosen for this inequality so that each professor could be kept as happy as possible by being assigned to at least one of his or her favorite courses. This number could be changed to tighten or ease the constraint on preferred courses. For example, if the model could find an optimal solution with a tighter constraint, such as 5, then all professors would only teach preferred courses. Or, if no solution could be found for the value 9, then the inequality could be relaxed to 10 or completely removed. Such decisions depend on the complexity of the attempted schedule and on the preferences of the professors themselves.

2.1.6- Objective Function

#Objective Function- Minimize total sum of preferences

minimize total_preferences:

```
sum{u in UPPER, j in PROFESSORS}preferences[u,j]*chosen_upper[u,j] + sum{l in
LOWER, j in PROFESSORS}preferences[l,j]*chosen_lower[l,j];
```

The objective function, as mentioned earlier, minimizes the sum of the preferences for the group of professors as a whole. As the values for favorite courses are closer to 0, minimizing the preferences provides that the faculty as a whole are as happy with the scheduling decisions. If the professors are satisfied with their courses then there is a greater chance that students will be satisfied as well.

Note that the expression of the objective function is similar to the left hand side of the inequality of the final constraint. The difference is that the final constraint finds a feasible combination of courses for each professor separately based on his preferences, whereas the objective function minimizes the sum of preferences for all professors together. Essentially, the objective function maximizes overall faculty course satisfaction while the final constraint provides that the goal is not achieved by satisfying only some professors and ignoring the preferences of others.

For example, if only the objective function were included, some teachers could be made very happy by being assigned their most favorite courses and other teachers could be assigned none of their favorite courses. As long as that situation made the sum of teacher preferences minimal, it could occur without the inclusion of the final constraint. If only the final constraint were included, a feasible solution could be that every professor was assigned to his second favorite course and a course that they do not like. While this is certainly feasible, it is not optimal. Only both the final constraint and the objective function together provide an optimal solution for teacher course satisfaction by requiring individual and overall satisfaction.

2.2 - Time Assignment Model

2.2.1 - General Discussion

The time scheduling portion of the model is the next step to completing the UCSP. After the solution to the Teacher Assignment model is obtained, it is used as input for the time assignment model. The chosen professor-class pairs will now be assigned a time period during the day. At many universities, classes meet for four (or three) times a week at the same time each day. Because of this standard four credit hour schedule, there was no distinction made between days of the week for the model. It is assumed that if a class is scheduled at eight o'clock in the morning then it will be held at that same time four days a week.

There are plenty of special constraints to consider for the time assignment model. For example, a department might allow its mathematics professors to specify a four hour time window in which they would prefer to teach. This is related to the fact that some universities have unions which do not allow their professors to teach classes that are separated by too large of a period of time during the day. Also, teachers may have preferences as to whether they wish to teach back to back classes, or whether they wish to have breaks in between. For the basic time assignment model, we are trying to find a feasible solution that satisfies all the constraints rather than trying to maximize an objective function as in the class scheduling section.

The details of the model are discussed in the next four subsections.

2.2.2 - Sets

#Given sets: class names, professors and time of day

set UPPER;

set LOWER;

set CLASSES:= LOWER union UPPER;

set TIMES;

set PROFESSORS;

The sets in the Time Assignment model are similar to those in the Teacher Assignment model. Classes are still divided into upper and lower levels in order to differentiate between the multiple sections of lower level classes.

The new set given is TIMES, which divides the academic day into time periods. In the data section, the set TIMES is listed as the following:

set TIMES:= 8 9 10 11 12 13 14 15 16 17 18;

This convention allows teachers to be assigned to a time of the day without having confusion between A.M. and P.M.

Finally, we have the set of professors. Professors' names are still used as an indexing set in many of the constraints and parameters, thus they are also included in the time assignemnt model.

2.2.3 - Parameters

#Parameter - Output from the Teacher Assignment model

#Professor-course pairs which have been assigned are given value 1

#All other professor-course pairs are given the default value 0

param inp{j in PROFESSORS, i in CLASSES}, default 0;

The parameter *inp* is based on the output of the teacher assignement model. This value could be 1 or 2 depending on if the professor was assigned to teach one or two sections of the

same course. The default value is 0, because a professor will not be assigned to the majority of classes. Thus, the user of the model only has to enter the list of classes which each professor teaches by entering inp[j , i] = 1 if professor j was assigned to teach class i in the solution of the teacher assignment model. These professor-course pairs will be assigned to time periods in this model.

#Parameter - What time of the day the teacher would prefer to teach class

param time_pref{j in PROFESSORS, t in TIMES}, default 0;

Another parameter is called *time_pref* and it allows professors to choose a four hour time block in which they would like to teach. If time_pref [j,t] = 1, then professor j has expressed preference for teaching in the time window [t, t + 4]. For example, some professors enjoy early classes and they may choose the 8 a.m. time block. This would require that the program schedule all of his or her classes to start between 8 a.m. and 11 a.m. Other professors prefer to start later and may choose an afternoon time block.

#Parameter - Professor preferences for back to back classes or not

#1 for back to back, 0 for not back to back, and 2 for no preference

param back_pref{j in PROFESSORS};

The third parameter is called *back_pref* and it is assigned based on whether or not the professor wants to teach his or her classes back to back. Professors who wish to have breaks between classes will be assigned the value 0, professors who want consecutive classes are assigned the value 1, and professors with no preference are assigned the value 2.

#Parameter - Number of sections of lower level classes

param lower_sections{l in LOWER};

The fourth parameter, *lower_sections*, is the number of sections for each lower level class, as seen before in the teacher assignment model.

#Parameter - The number of rooms available at any one time of the day

param rooms;

The final parameter is the number of rooms that are available in the building in which the classes are taught. Certainly, at any given time the number of classes cannot be more than the number of rooms in the building, and there will be a constraint preventing this situation.

2.2.4 - Decision Variables

#Decision variable - Binary assignment of times and classes

var assign{j in PROFESSORS, i in CLASSES, t in TIMES : inp[j,i] > 0} binary;

One of the variables in the time assignment model is a binary assignment of times to professor-course pairs. Professor-course pairs being taught at a given time will be assigned the value 1 and all other times will be given the value 0 for that pair.

Not only should a model be logically correct, but it should also be as small and time-efficient as possible. This variable includes a crucial time-saving measure that improves the efficiency of the time assignment model. Notice that the variable *assign* is only defined for those professor-time pairs that have value 1 or 2 for the parameter *inp*. Not defining variables for those pairs which have *inp* value 0 (that is, a professor not teaching that class) greatly reduces the number of variables in this model. Also, each constraint that includes the variable *assign* will

include this input restriction. This prevents the constraints from considering pairs for which the decision variable is not defined. Now the model includes as few variables as possible and thus improves overall efficiency. This model will be discussed in more detail in Chapter 3.

#Decision variable – Assigns value 1 to professors who are assigned to teach back to back #courses

```
var back_to_back{j in PROFESSORS, t in TIMES: back_pref[j] = 1} binary;
```

The decision variable *back_to*_back concerns the assignment of consecutive courses. The decision variable *back_to_back[j , t]* will receive value 1 if professor j is assigned to teach consecutive courses at times t and t + 1. This is actually an auxiliary variable which is necessary for writing the corresponding back_to_back constraint. It is only defined for those professors who have indicated back to back teaching preference. This reduces the number of variables in the Time Assignment model.

2.2.5 - Constraints

Basic Scheduling Constraints

#Constraint - A professor can only teach one class at a time

```
subject to prof_one_at_atime{t in TIMES, j in PROFESSORS}:
    sum{i in CLASSES : inp[j,i] > 0}assign[j,i,t] <= 1;
```

The constraint *prof_one_at_atime* provides that each professor may only teach a single class at a time. This is a fairly straightforward constraint because a teacher cannot be in two places at once. Recall that this restriction: "inp[j,i] > 0" appears in any constraint having the variable *assign*, ensuring that the constraint only includes professor-course variables which are

defined in this model.

```
#Constraint - Upper level classes should be taught once per day
    subject to one_time_period{u in UPPER}:
        sum{t in TIMES, j in PROFESSORS : inp[j,u] > 0}assign[j,u,t] = 1;
```

The next constraint, *one_time_period*, is a standard logical constraint for upper level classes. It requires that each upper level class with a single section to be assigned to exactly one time period per day.

```
#Constraint - The number of classes assigned in a time period is limited by the number of rooms
    subject to room_constraint{t in TIMES}:
        sum{i in CLASSES, j in PROFESSORS : inp[j,i] > 0} assign[j,i,t]<= rooms;
```

This constraint concerns the number of rooms available in the building holding the classes. Certainly there should not be more classes assigned to a time period than there are rooms available to hold the classes. Room assignment is the final part of the class scheduling problem. It was decided that it was not necessary to write an IP model for room assignment as it can easily be done by hand. The only concern with assigning a class to an arbitrary room would be class size. A lecture class of 200 people should not be held in a classroom for 30. It could be done by simple inspection.

Lower-level Course Constraints

In this subsection, there are two time constraints concerning the assignment of lower level classes.

```
#Constraint - Professors assigned to lower-level classes
#Must teach the right number of sections
        subject to sections{j in PROFESSORS, l in LOWER}:
                sum{t in TIMES : inp[j,l] > 0}assign[j,l,t] = inp[j,l];
```

This constraint provides that each professor teaches the right number of sections of a lower-level class. The left hand side of the constraint is the number of time periods that professor j is assigned to teach sections of class l, while the right hand side of the equation is the number of sections of class l assigned to professor j by the teacher assignment model. Note that if a professor is assigned to teach two sections of the same course, then this constraint would require that they are scheduled at two different times.

```
#Constraint - Sections of a lower-level class should be scheduled at different times
        subject to dif_time_lower{l in LOWER,t in TIMES}:
                sum{j in PROFESSORS : inp[j,l] > 0}assign[j,l,t] <= 1;
```

The next lower level constraint, *diff_time_lower*, is a practical convention that makes students' scheduling easier. The constraint requires that for any pair of a lower level class and time period, no more than one section can be held at that time. This ensures that the sections of class j will be held at different times of the day. Thus, if a student's schedule could not fit a particular section of a class because of a time conflict, then there would be other times available as well. Many of the lower level classes have many sections because they are requirements for different colleges and this constraint is a very practical way of ensuring that as many students as possible can take the course that they need to graduate

Blocking Constraint

This constraint provides the four- hour time windows for each professor as specified in the parameter *blocking.*

.#Constraint - Teachers give classes within a four-hour preferred time block

subject to blocking{j in PROFESSORS, t in TIMES : time_pref[j,t] = 1 and t<=14}:

sum{i in CLASSES : inp[j,i]>0}(assign[j,i,t] + assign[j,i,t+1] + assign[j,i,t+2] + assign[j,i,t+3]) = sum{i in CLASSES : inp[j,i] >0}inp[j,i];

The constraint, *blocking*, takes into account professors' preferences for time periods for instruction. Recall, that if a professor were to specify the 10 a.m. time block, then all of his or her courses would be scheduled between 10 a.m. and 2 p.m. The left hand side of the equation is the number of courses that professor j is assigned to at times t, t + 1, t +2, and t +3 where t is the beginning of the professor's preferred four hour time block specified by the value 1 for the parameter *time_pref[j , t]*. The right hand side of the equation is the number of courses assigned to professor j. Thus, the constraint requires that the classes that a professor teaches must be during his preferred four-hour time block. The restriction t <= 14 is included to ensure that the model does not begin the time blocks any later than 2pm.

Back to Back Teaching Constraints

The next several constraints concern whether or not professors prefer to teach courses back to back or not.

```
#Constraint – Determines the teaching time for teachers who want to teach back-to-back courses
    subject to back_assignment{j in PROFESSORS: back_pref[j] = 1}:
        sum{t in TIMES : t <= 16}back_to_back[j,t] = 1;
```

This constraint, *back_assignment*, does not immediately change the solution to the UCSP in any way. It does, however, set up the variable *back_to_back* for use in the following constraint, *back_constraint*. Here, all the professors who indicated that they would like to teach consecutive courses have the binary decision variable *back_to_back* set to the value 1 for exactly one time period during the day.

```
#Constraint - If a back_to_back is 1 then professor j is assigned to teach at times t and t + 1
#Note - If back to back is 0, then nothing is enforced
    subject to back_constraint{j in PROFESSORS, t in TIMES : back_pref[j] = 1 and
    t <= 16}:
        sum{i in CLASSES : inp[j,i] >0}assign[j,i,t] + sum{i in CLASSES : inp[j,i]
        >0}assign[j,i,t+1] ≥ 2*back_to_back[j,t];
```

The constraint *back_constraint* makes use of the decision variable *back_to_back* that was just a value 0 or 1. Recall that for each professor who wants to teach consecutive classes, the variable back-to-back was chosen to be 1 for exactly one time period. For each professor and time pair, if *back_to_back[j , t]* = 1, the constraint requires that at time t, which is the time chosen for the preferred time block for that teacher, professor j must be assigned to teach at both time t and time t + 1. For example, if for professor Thomas the variable back-to-back has value 1 for 9 a.m., then the inequality requires that the number of courses that professor Thomas instructs

at 9 a.m. and 10a.m. is greater than or equal to 2. If *back_to_back[j , t]* = 0, the constraint requires that the number of courses that the professor teaches at times t and t + 1 is greater than or equal to 0. Essentially, this does not enforce any change in the solution to the model. This constraint guarantees that each professor who wants to teach back to back courses will teach consecutively at some point during his or her preferred time block.

The constraints *back_assignment* and *back_constraint* complement each other. Together they provide that consecutive classes are scheduled for the time period which has variable *back_to_back* equal to 1.

```
#Constraint - Professors who don't want to teach back to back courses
    subject to not_back{j in PROFESSORS, t in TIMES : t <= 16 and back_pref[j]= 0}:
        sum{i in CLASSES : inp[j,i] >0}assign[j,i,t] + sum{i in CLASSES : inp[j,i]
        >0}assign[j,i,t+1] <= 1;
```

In this constraint, only professors for whom *back_pref[j] = 0* are considered. That is, only those professors who wish to have a break in between classes will be included in this constraint. The constraint *not_back* provides that for these professors the number of courses taught at time t and t + 1 must always be less than or equal to 1 for each time t. This means that for every grouping of back to back times, say 9 a.m. and 10a.m., or 10 a.m. and 11 a.m., a professor could only teach at most one course.

2.2.6 - Objective Function

There is no objective function for the basic model of the Time Assignment model. The goal is to find a feasible solution only. However, there are situations where it could be advisable

to add an objective function. For example, if the Time Assignment model fails to deliver a feasible solution, then certain professor preferences may need to be relaxed. In that case some of the preference constraints would be relaxed, and instead the objective function would try to satisfy as many teacher preferences as possible.

Chapter 3

Experimental Results

In Chapter 2, the logic of each constraint of the IP model for the UCSP is discussed in detail. However, it is also important to check the correctness of the model computationally. The model was tested on different data sets throughout the process of writing the model to test new constraints as they were added. Finally, the model was tested on a realistic data set based on the faculty of the mathematics department of Ohio University. In all data sets tested, the output of the IP model for the UCSP has been logically correct in pairing teachers, courses, and times, as well as correct in respecting the preferences of all professors listed.

Time efficiency was another important consideration when writing the model. Recall from Chapter 2 that the way to achieve a smaller running time was to decrease the number of binary variables in the model. Thus, once the model has been checked for logic, it must also be checked for a reasonable running time. The IP model for the UCSP has been tested on data sets of varying sizes. The running time was very small for the initial data sets that were meant to test the various constraints. Later, when the model was tested on a realistic-sized faculty data set, it was still remarkably efficient. In practice, the running time of the program for a realistic size data set is less than 0.1 seconds.

Chapter 3 is organized the following way. Sections 3.1 and 3.2 give the results for small and large examples correspondingly. Section 3.3 discusses issues related to time efficiency.

3.1 Small Example

Input - Teacher Assignment

In this section, a small example is given to demonstrate the format of the input into the program and its output. The AMPL syntax of the data is explained, and a table format is used for a better visual understanding of the output.

Recall that the input is given by sets and parameters. The elements of each set used in the model are explicitly listed in the input data section. Examples of sets below are class titles and professors' names.

```
set PROFESSORS:= Thomas, Kreuzer, Schoenefeld, Veleta, Irwin;
set LOWER:= math113, math1115,  math250;
set UPPER:=  math300, math340, math443, math450;
```

Parameters can define single values associated with each element of a set as seen below in *lower_sections* and *avail.* The number listed in *lower_sections* for each class denotes the number of sections for that particular class. The parameter *avail* lists the number of courses each professor should be assigned to teach.

```
param lower_sections:=
math250  2
math113  2
math115  3;
```

```
param   avail:=
Thomas          2
Kreuzer         2
Schoenefeld     2
Veleta          2
Irwin           2;
```

Recall that the parameter *preference* associates a value with each professor-course pair based on that professor's teaching preference. A professor's top three courses are given values 1,

2, or 3, with one being the most favorite. All other professor-course pairs for that professor are given value 7. In the input data, parameter *preference* is indexed by two sets, PROFESSORS and CLASSES. The value associated with that professor-course pair is listed immediately after the course title in the same row. For example, professor Irwin has given math340 a value of 1. All courses not explicitly listed in parameter *preference* have been given the default value of 7.

param *preferences*:=

[*,Irwin]	math340	1	math250	2	math113	3
[*,Thomas]	math113	1	math115	2	math250	3
[*, Kreuzer]	math443	1	math250	2	math340	3
[*,Schoenefeld]	math115	1	math113	2	math250	3
[*,Veleta]	math250	1	math450	2	math300	3;

These sets and parameters are all of the input data needed for the small example of the Teacher Assignment model. In the next section, the output of this example is given and explained.

Output – Teacher Assignment

The output of the Teacher Assignment model represents the values of decision variables. For the Teacher Assignment model, the set of variables "chosen_upper" is assigned value 1 if the professor is teaching that course. The set of variables "chosen_lower" can take value 1 or 2 depending on how many section of that course the professor is teaching. Each professor-course pair that was not assigned received value 0 for these decision variables, and are not listed here.

chosen_upper['Irwin','math340']	1
chosen_lower['Irwin','math250']	1
chosen_upper['Kreuzer','math443']	1
chosen_lower['Kreuzer','math250']	1
chosen_upper['Veleta','math450']	1

chosen_upper['Veleta','math300'] 1
chosen_lower['Thomas','math113'] 2
chosen_lower['Schoenefeld','math115'] 2

The value of each variable for an assigned professor-course pair is the number of sections of that course that the professor will be teaching. For example, professor Schoenefeld is teaching two sections of math115.

Input – Time Assignment

Below, new parameters for the Time Assignment model are listed. The sets PROFESSORS, UPPER, and LOWER, as well as the parameter *lower_sections,* are included in this data set but are omitted in this section.

The set TIMES is simply the times from 8 a.m. to 5 p.m. The parameter *rooms,* is a restriction on the number of rooms available during a single time period for instruction. The parameters listed below follow the same conventions as in the input of the Teacher Assignment model.

set TIMES:= 8 9 10 11 12 13 14 15 16 17;

param *rooms*:= 10;

Recall that parameter *input,* below, lists the assigned professor-course pairs generated by the Teacher Assignment model.

param *inp*:=
[Irwin,*] math340 1 math250 1
[Kreuzer,*] math443 1 math250 1
[Veleta,*] math450 1 math300 1
[Thomas,*] math113 2
[Schoenefeld,*] math115 2;

Also recall the value assigned to each variable in *back_pref* is the preference for back-to-back courses. If a professor does want to teach back-to-back courses, that professor receive value 1, those professors who do not wish to teach back to back receive value 0 and those have no preference receive value 2 and do not have back-to-back constraints enforced.

param *back_pref*:=
Thomas 0
Kreuzer 1
Schoenefeld 2
Irwin 2
Veleta 1;

Lastly, parameter *time_pref* associated value 1 to the beginning of each professor's preferred four-hour time block. In the example below, professor Thomas wishes to teach courses between 8 a.m. and 12 p.m. so *time_pref[Thomas, 8]* is assigned value 1.

param *time_pref*:=
[Thomas,*] 8 1
[Kreuzer,*] 12 1
[Schoenefeld,*] 10 1
[Irwin,*] 8 1
[Veleta,*] 12 1;

Output – Time Assignment

The output of the Time Assignment model is similar to that of the Teacher Assignment model. For the variable "assign," the professor, course, and time period that have been assigned are given value 1. All other combinations are given value 0 and are not listed below. For example, professor Thomas is assigned to teach math113 at 8 a.m. as well as at 10 a.m.

assign['Thomas','math113',8]" 1
assign['Thomas','math113',10]" 1

assign['Schoenefeld','math115',10]" 1
assign['Schoenefeld','math115',11]" 1

assign['Irwin','math250',9]" 1
assign['Irwin','math340',8]" 1

assign['Kreuzer','math250',13]" 1
assign['Kreuzer','math443',12]" 1

assign['Veleta','math450',13]" 1
assign['Veleta','math300',12]" 1

Lastly, for easier understanding, the same information as above is presented in table form.

Professors	**Times**					
	8 a.m.	9 a.m.	10 a.m.	11 a.m.	12 p.m.	1 p.m.
Thomas	math113		math113			
Schoenefeld			math115	math115		
Veleta					math300	math450
Irwin	math340	math250				
Kreuzer					math443	math250

3.2 Number of Variables and Time Efficiency

The time efficiency of IP solution methods largely depends on the number of integer variables. The number of LP subproblems solved by Branch-and-Bound increases exponentially as a result of the increase in the number of variables. In this section, the number of variables and the running time for the UCSP model will be discussed.

var chosen_upper{u in UPPER, j in PROFESSORS} binary;

var chosen_lower{l in LOWER, j in PROFESSORS} integer, >= 0, <=2;

In the Teacher Assignment model, there are two sets of decision variables, listed below as a reminder. The variable *chosen_upper* is indexed on two sets. If the number of upper level courses is m and the number of professors is n, then the number variables for *chosen_upper* is m * n. Similarly, if the number of lower level courses is k, then the number of variables created for *chosen_lower* is k * n.

In the large data set used to test the model, the number of upper level courses is 27 and the number of professors is 22. So the number of variables for *chosen_upper* is 594. Also, as the number of lower level courses is 10, the number of variables for *chosen_lower* is 220. Solvers today are routinely solving IP problems with over well one thousand variables [5]. The solver used for this thesis easily handled a data set with 814 variables. For a bigger department with about 40 to 50 professors, the number of variables would be no more than 2000 which would still be solvable by today's solvers.

In the Time Assignment model there are also two sets of variables, *assign* and *back_to_back*.

var assign{j in PROFESSORS, i in CLASSES, t in TIMES : inp[j,i] > 0} binary;

var back_to_back{j in PROFESSORS, t in TIMES: back_pref[j] = 1} binary;

Variable *assign* is indexed by three sets, and with n professors, u classes and v times, the total number of variables would be n*u*v if a variable were defined for every possible combination. With 22 professors, 37 courses and 10 times defined, there could be a total of 8140 variables in the large example. This is a significantly large number for today's best solvers. Fortunately, the restriction "inp[j,i] > 0" greatly reduces the number of variables. This restriction guarantees that only the professors-course pairs that were matched from the Teacher Assignment model are used to create variables in the model. In the large example given in a later section, the number of

professor-course pairs used as input for the Time Assignment model is 45, reducing the total number of variables for *assign* from 8140 to 450.

For the variable *back_to_back*, the number of variables depends on the number of teachers who wish to teach back to back courses. Even if every professor chose to instruct back to back courses, this would only add 220 variables in the case of the large example.

Thus, the number of variables for both the Teacher Assignment or Time Assignment models is within the capabilities of modern computer solvers. Especially due to the use of variable reducing techniques in the case of the Time Assignment model, the UCSP should be solvable using this model for even the largest departments.

Conclusion and Future Direction

The goal of this thesis was to find an Integer Programming solution to the University Class Scheduling Problem. More specifically, to find a feasible class assignment schedule for a given department in a university while maximizing teacher-assignment satisfaction. The model, which was broken into the separate Teacher Assignment and Time Assignment models has been shown to be logically correct. Initial results have produced feasible results for realistic-sized data sets.

There is room for improvement in the model, and future work could be done in several areas. One concept that was not explored in this thesis is the possibility of keeping future quarter or semester schedules as close to previous quarters as possible. Since many professors will not change their teaching preferences from year to year, it could be desirable to change future schedules as little as possible. Constraints could be added or the objective function changed to track changes and include as few as possible.

Another possible future direction would be to track which teachers are assigned to teach their most favorite classes and which teachers are not satisfied as fully as possible with their teaching schedule. Rotating the sum of the teacher preferences that are satisfied for each teacher could be a fair way of maintaining overall faculty satisfaction.

Also, though it did not occur with any data sets tested for this thesis, it is possible that teacher preferences could make the Time Assignment model infeasible. If this were to happen, it would still be necessary to find a feasible solution. One possibility would be to turn the teacher preference constraint into an objective function. This would require that the solver satisfy as many teacher time preferences as possible, but it would allow more feasible solutions to be considered.

Lastly, though it is not related to a Mathematics thesis, it would be useful to create a Graphical User Interface so that the model could be put into practice by an administrator. At this point in the solution process, one would have to be familiar with Integer Programming and AMPL to use these models. It would be wonderful if this thesis could be useful in the real world and not solely as an exercise in Integer Programming.

In conclusion, this thesis has been an enjoyable and challenging problem that has been largely sucessful. The goal of solving the UCSP has been achieved with the IP models included here. With more effort, this thesis could be of real world use to universities.

Appendix

A.1 - Teacher Assignment

Teacher Assignment Model

```
#Given: sets of professor and class names
set PROFESSORS;
set LOWER;
set UPPER;
set CLASSES:= LOWER union UPPER;

#number of classes each professor needs to teach, with default value!
param avail{j in PROFESSORS}, default 2;

#list of each professors fav classes - now with data-saving default!
param preferences{j in PROFESSORS, i in CLASSES}, default 7;

# Number of sections of lower level classes
param lower_sections{l in LOWER};

#if upper-level class is assigned to a professor

#then the value is 1, otherwise 0
var chosen_upper{j in PROFESSORS, u in UPPER} binary;

#if lower level class is assigned to a professor
#variable can either be one or two
#otherwise variable is 0
var chosen_lower{j in PROFESSORS, l in LOWER} integer, >= 0, <=2;


#Constraint - upper level classes happen once and need exactly one teacher
subject to filling_upper{u in UPPER}:
   sum{j in PROFESSORS} chosen_upper[j,u] = 1;


#Constraint- assign as many lower level classes as possible
subject to filling_lower{j in PROFESSORS}:
  sum{l in LOWER} chosen_lower[j,l] <= avail[j];

#Constraint - each professor needs to teach the right number of classes
subject to required_to_teach{j in PROFESSORS}:
   sum{l in LOWER} chosen_lower[j,l] + sum{u in UPPER} chosen_upper[j,u] = avail[j];
```

```
#Constraint- all sections of lower classes should be filled
subject to sections{l in LOWER}:
  sum{j in PROFESSORS} chosen_lower[j,l] <= lower_sections[l];


#Constraint - each class should have exactly one professor
subject to one_prof_per_class{u in UPPER}:
  sum{j in PROFESSORS} chosen_upper[j,u] = 1;


#Constraint - each prof gets classes they like
subject to prof_preferences{j in PROFESSORS}:
  sum{u in UPPER}preferences[j,u]*chosen_upper[j,u] + sum{l in
LOWER}preferences[j,l]*chosen_lower[j,l] <= 9;


#Objective function- minimize total sum of preferences
minimize total_preferences:
  sum{u in UPPER, j in PROFESSORS}preferences[j,u]*chosen_upper[j,u] + sum{l in
LOWER, j in PROFESSORS}preferences[j,l]*chosen_lower[j,l];
```

Teacher Assignment Data

```
data;
set PROFESSORS:= Aftabizadeh, Aizicovici, Arhangelskii, Barsamian, Chapin, Eisworth,
Gulisashvili, Huynh, Just, Kaufman, Klein, Lin, Melkonian, Mohlenkamp, Pavel, Savin, Shen,
Uspenskiy, Vinogradov, Vu, Wolf, Young;

set LOWER:= math163A, math163B, math211, math250, math251, math263A, math263B,
math263C, math263D, math266B;

set UPPER:= math147 math300, math306, math307, math308, math314, math330A, math330B,
math340, math344, math410, math412, math441, math446, math449, math450C, math451,
math452, math460C, math470, math480B, math615, math645C, math649, math660C,
math670C, math680C;

param lower_sections:=
math163A    7
math163B    1
math211     1
math250     8
math251     1
math263A    4
math263B    5
math263C    3
math263D    2
```

```
math266B      2;

param preferences:=
[Aftabizadeh,*]     math163A    1     math263B    2     math263A    3
[Aizicovici,*]      math645C    1     math340     2     math460C    3
[Arhangelskii,*]    math680C    1     math211     2     math306     3
[Barsamian,*]       math163A    1     math330A    2     math263A    3
[Chapin,*]          math441     1     math344     2     math163A    3
[Eisworth,*]        math480B    1     math263B    2     math263C    3
[Gulisashvili,*]    math670C    1     math263D    2     math263C    3
[Huynh,*]           math263B    1     math263C    2     math307     3
[Just,*]            math266B    1     math340     2     math263B    3

[Kaufman,*]         math147     1     math340     2     math250     3
[Klein,*]           math300     1     math330A    2     math330B    3
[Lin,*]             math450C    1     math452     2     math250     3
[Melkonian,*]       math308     1     math250     2     math306     3
[Mohlenkamp,*]      math649     1     math446     2     math344     3
[Pavel,*]           math470     1     math163A    2     math449     3
[Savin,*]           math410     1     math263B    2     math266B    3
[Shen,*]            math250     1     math211     2     math446     3
[Uspenskiy,*]       math263D    1     math263C    2     math263B    3
[Vinogradov,*]      math441     1     math250     2     math450C    3
[Vu,*]              math410     1     math263A    2
[Wolf,*]            math615     1     math410     2
[Young,*]           math660C    1     math344     2;

param     avail:=
Eisworth            4;
```

A.2 - Time Assignment

Time Assignment Model

```
#Given sets: class names, professors and time of day
set UPPER;
set LOWER;
set CLASSES:= LOWER union UPPER;
set TIMES;
set PROFESSORS;

param rooms;
#Output from previous problem- given
param inp{j in PROFESSORS, i in CLASSES}, default 0;

# Number of sections of lower level classes
param lower_sections{l in LOWER};

#What time of the day the teacher would prefer to teach class
param time_pref{j in PROFESSORS, t in TIMES}, default 0;

#Professor preferences for back to back classes or not
#1 for back to back, 0 for not back to back, and 2 for no preference
param back_pref{j in PROFESSORS};

var back_to_back{j in PROFESSORS, t in TIMES: back_pref[j] = 1} binary;

#Binary assignment of times and classes
var assign{j in PROFESSORS, i in CLASSES, t in TIMES : inp[j,i] > 0} binary;

#Constraint- Teachers give classes within
```

```
#a four hour preferred time block
subject to blocking{j in PROFESSORS, t in TIMES : time_pref[j,t]>0 and
t<=14}:
  sum{i in CLASSES : inp[j,i] >0}inp[j,i] = sum{i in CLASSES : inp[j,i]
>0}(assign[j,i,t] + assign[j,i,t+1] + assign[j,i,t+2] + assign[j,i,t+3]);

#Constraint - A professor can only teach one class at a time
subject to prof_one_at_atime{t in TIMES, j in PROFESSORS}:
  sum{i in CLASSES : inp[j,i] > 0}assign[j,i,t] <= 1;

#Constraint - Lower level classes must have the right amount of professors
#assigned
subject to lower_time_periods{l in LOWER}:
  sum{t in TIMES, j in PROFESSORS : inp[j,l] > 0}assign[j,l,t] = sum{j in
PROFESSORS}inp[j,l];

#Constraint - Upper level classes should be once per day
subject to one_time_period{u in UPPER}:
  sum{t in TIMES, j in PROFESSORS : inp[j,u] > 0}assign[j,u,t] = 1;

#Constraint - Professors professors assigned to lower classes
#Must teach the right amount of sections
subject to sections{j in PROFESSORS, l in LOWER}:
      sum{t in TIMES : inp[j,l] > 0}assign[j,l,t] = inp[j,l];

#Constraint - The number of classes assigned in a time period is limited by the number of rooms
subject to room_constraint{t in TIMES}:
  sum{i in CLASSES, j in PROFESSORS : inp[j,i] > 0} assign[j,i,t]<= rooms;

#Constraint - Lower sections should be scheduled at different times
subject to dif_time_lower{l in LOWER,t in TIMES}:
```

```
  sum{j in PROFESSORS : inp[j,l] > 0}assign[j,l,t] <= 1;

#Constraint - 1 for teachers who want back to back
subject to back_assignment{j in PROFESSORS: back_pref[j] = 1}:
  sum{t in TIMES : t <= 16}back_to_back[j,t] = 1;

#Constraint - If a teacher is assigned a time and he also wants back to back
#classes, then he must also be assigned to the following time period
#Note - If back to back is 0, then nothing is enforced
subject to back_constraint{j in PROFESSORS, t in TIMES : back_pref[j] = 1
and t <= 16}:
  sum{i in CLASSES : inp[j,i] >0}assign[j,i,t] + sum{i in CLASSES : inp[j,i]
>0}assign[j,i,t+1] >= 2*back_to_back[j,t];

#Constraint - Professors who don't want back to back can't teach the time period
#immediately following the one they are assigned
subject to not_back{j in PROFESSORS, t in TIMES : t <= 16 and back_pref[j]
= 0}:
  sum{i in CLASSES : inp[j,i] >0}assign[j,i,t] + sum{i in CLASSES : inp[j,i]
>0}assign[j,i,t+1] <= 1;
```

Time Assignment Data

```
data;
set LOWER:= math163A, math163B, math211, math250, math251, math263A, math263B,
math263C, math263D, math266B;

set UPPER:= math147 math300, math306, math307, math308, math314, math330A, math330B,
math340, math344, math410, math412, math441, math446, math449, math450C, math451,
math452, math460C, math470, #math480B, math615, math645C, math649, math660C,
math670C, math680C;

set PROFESSORS:= Aftabizadeh, Aizicovici, Arhangelskii,
Barsamian, Chapin, Eisworth, Gulisashvili, Huynh, Just, Kaufman, Klein, Lin,
Melkonian, Mohlenkamp, Pavel, Savin,
Shen, Uspenskiy, Vinogradov, Vu, Wolf, Young;
```

```
set TIMES:= 8 9 10 11 12 13 14 15 16 17;

param rooms:= 30;
param lower_sections:=
math163A      7
math163B      1
math211       1
math250       8
math251       1
math263A      4
math263B      5
math263C      3
math263D      2
math266B      2;

param inp:=
[Aftabizadeh,*]     math163A     2
[Aizicovici,*]      math460C     1     math645C     1
[Arhangelskii,*]    math680C     1     math211      1
[Barsamian,*]       math330A     1     math163A     1
[Chapin,*]          math441      1     math451      1
[Eisworth,*]        math263B     2     math263C     1
[Gulisashvili,*]    math412      1     math670C     1
[Huynh,*]           math263B     1     math307      1
[Just,*]            math266B     2
[Kaufman,*]         math147      1     math340      1
[Klein,*]           math300      1     math330B     1
[Lin,*]             math450C     1     math452      1
[Melkonian,*]       math306      1     math308      1
[Mohlenkamp,*]      math446      1     math649      1
[Pavel,*]           math449      1     math470      1
[Savin,*]           math263B     2
[Shen,*]            math250      2
[Uspenskiy,*]       math263D     2
[Vinogradov,*]      math250      2
[Vu,*]              math263A     1     math410      1
[Wolf,*]            math314      1     math615      1
[Young,*]           math344      1     math660C     1;

param time_pref:=
[Aftabizadeh,*]     8     1
[Aizicovici,*]      8     1
[Arhangelskii,*]    10    1
[Barsamian,*]       12    1
[Chapin,*]          12    1
[Eisworth,*]        12    1
```

```
[Gulisashvili,*]    10    1
[Huynh,*]           10    1
[Just,*]            12    1
[Kaufman,*]         12    1
[Klein,*]           12    1
[Lin,*]             8     1
[Melkonian,*]       10    1
[Mohlenkamp,*]      8     1
[Pavel,*]           12    1
[Savin,*]           10    1
[Shen,*]            12    1
[Uspenskiy,*]       12    1
[Vinogradov,*]      8     1
[Vu,*]              12    1
[Wolf,*]            12    1
[Young,*]           10    1;
param back_pref:=
Aftabizadeh         0
Aizicovici          2
Arhangelskii        0
Barsamian           1
Chapin              0
Eisworth            1
Gulisashvili        0
Huynh               1
Just                1
Kaufman             0
Klein               0
Lin                 1
Melkonian           2
Mohlenkamp          2
Pavel               1
Savin               2
Shen                0
Uspenskiy           1
Vinogradov          1
Vu                  0
Wolf                1
Young               2;
```

Time Assignment Output

Professors					**Times**				
	8a.m	9a.m	10a.m	11a.m	12p.m	1p.m	2p.m	3p.m	4p.m
Aftabizadeh	163A			163A					
Aizicovici	460C	645C							
Arhangelskii			680C				211		
Barsamian							330A	163A	
Chapin					451				441
Eisworth							263C	263A	263B
Gulisashvili			670C				412		
Huynh				307	263B				
Just								266B	266B
Kaufman					340				147
Klein					330B				300
Lin			452	450C					
Melkonian			308	306					
Mohlenkamp	649			446					
Pavel							470	449	
Savin			263A	263B					
Shen					250				250
Uspenskiy								263D	263D
Vinogradov			250	250					
Vu					410				263A
Wolf							615	314	
Young			660C		344				

References

[1] Aloul, F., and A. Wasfy. 4th IEEE GCC Conference, Nov. 2007, Bahrain. Solving the University Class Scheduling Problem Using Advanced ILP Techniques. American University of Sharjah. 30 Oct. 2008 http://www.aloul.net/Papers/faloul_sch_gcc07.pdf.

[2] Boland, Natasha, Barry D. Hughes, Liam T. Merlot, and Peter J. Stuckley. "New Integer Linear Programming Approaches for Course Timetabling." Computers and Operations Research. 35 (2008): 2209-233.

[3] Fourer, Robert, David M. Gay, and Brian W. Kernighan. AMPL- A Modeling Language For Mathematical Programming. Davers: Boyd and Fraser Publishing Company, 1993.

[4] Gunawan, Aldy, Kein M. Ng, and Kim L. Po. "Solving the Teacher Assignment-Course Scheduling Problem by a Hybrid Algorithm." International Journal of Computer, Information, and Systems Science, and Engineering 1 (2007).

[5] Hillier, Frederick S., and Gerald J. Lieberman. Introduction to Mathematical Programming. New York: McGraw-Hill, 1995.

[6]Savelsbergh, Martin W., R. N. Uma, and Joel Wein. "An Experimental Study of LP-Base Approximation Algorithms for Scheduling Problems." INFORMS Journal on Computing 17 (2005): 123-36.

[7] Taha, Hamdy. Integer Programming- Theory, Applications, and Computations. New York: Academic Press, 1975.

[8] Barry, Cipra. "The Best of the 20th Century: Editors Name Top 10 Algorithms." Applied Mathematics. SIAM News. 1 Mar 2009 <http://amath.colorado.edu/resources/archive/topten.pdf>.

[9] Hochbaum, Dorit S, ed. Approximation Algorithms for NP-Hard Problems. Boston: PWS Publishing Company, 1997.

[10] Michalewicz, Zbigniew, and David B. Fogel. How to Solve It: Modern Heuristics. 2nd ed. Berlin: Springer-Verlag, 2004.