Robin M. Baur
Scientific Computing

Final Project
Due 28 April 2006

The Freshman Scheduling Problem: An Exercise in Discrete Optimization?

1 The Problem, Introduced

Suppose that at a certain university, the students in the incoming freshmen class have no choice in their first-semester courses, which are instead pre-assigned by the university. Knowing which students are enrolled in which courses, the university then desires to assign the freshman courses to the available time slots in some optimal way. We call this the *freshman scheduling problem*.

What constitutes optimality, of course, will vary widely between institutions which employ this pre-scheduling process. One university, for instance, may consider it non-optimal to make students late to class by putting their courses in rooms too far apart. Another might consider it non-optimal to schedule courses on Fridays. Yet another might find it optimal to schedule as many of its courses as possible for 6 a.m. Saturday morning.

In this discussion of optimality, it is useful to note two things. First, in almost all cases, it is possible to turn a desire stated as an optimality into one stated as a non-optimality, and vice versa (e.g., it is either optimal to schedule time-adjacent classes in nearby rooms or it is non-optimal to schedule them too far apart). It turns out that it is more convenient from an implementation and data storage perspective to consider these desires from the perspective of non-optimality, as will be discussed in Section 4.2. Second, note that there is a distinction between something a university wishes to discourage, such as the examples given so far, and something the university wishes to expressly forbid or make impossible, such as scheduling a professor to teach two different classes at the same time. We will call the former cases non-optimalities, and the latter cases we will call constraints.

The freshman scheduling problem is very similar to some existing problems in industry and computer science, in particular the fleet scheduling problem common to airlines and the postal service, who must place the vehicles in their fleet (student analogues) at particular places at particular times (classroom and timetable analogues) in such a way as to minimize their costs, travel time, and vehicle downtime. The fleet scheduling problem is actually a more elaborate problem than the freshman scheduling problem, because fleet scheduling naturally incorporates the traveling salesman problem (in which the goal is to travel to each of a set of destinations for minimal cost), which is known to be NP-complete. (For more information on NP-completeness and NP-complete problems, see [5]) Although the complexity of these problems makes it inefficient or impossible to arrive at an optimal solution by exact means or by hand, both the fleet scheduling problem and the freshman scheduling problem can be formulated such that they are solvable using a method known as linear programming. The freshman scheduling problem can also more naturally be formulated using a related method known as quadratic programming.

2 Quadratic Programming

2.1 The Mechanism

In its most basic description, the goal of quadratic programming is to minimize some *objective function*

$$f(x) = \frac{1}{2}x^T Hx + g^T x$$

subject to some set of constraints $Ax \leq b$, where g is a vector the same length as x, H is a square matrix with the dimension of each side equal to the length of x, and A and b are a matrix and a vector, respectively, with dimensions such that the multiplication works out appropriately. Note that here we are making use of the mathematical convention of dropping the vector bars from vectors; we shall be careful to point out which objects are vectors and which scalars, and to always refer to components of these vectors using an index, e.g. x_i.

The $x^T Hx$ (quadratic form) term in the objective function shows us the origin of the term quadratic programming; not surprisingly, *linear programming* is the special case of quadratic programming in which $H = 0$. We can further restrict the problem by requiring that the variables x_i in the solution vector x must be integers, and even further by requiring that they be binary, either 0 or 1.

Since the solutions we seek will be used to create a timetable for the semester's courses, we expect that all our variables ought indeed to be either 0 or 1 — either some class is taught is some time slot (1) or it is not (0). We cannot have .273 of a class offered in one time slot and the other .727 offered in a different time slot. So we would like, if possible, to use an integer programming method to solve this problem.

Unfortunately, since integer programming is NP-complete, the algorithms for solution of integer programming problems have complexity which is exponential in the number of variables in the problem. As an illustrative example of why such a solution method might be exponential time, note that for a binary integer programming problem there are 2^N possible binary strings of length N; one solution method for binary integer problems involves searching a tree whose paths from root to the bottom layer of leaves describe these 2^N strings. While the algorithm is not necessarily obliged to search the entire tree for the optimal solution, the amount of the tree it may need to search is large enough that the problem is roughly $O(2^N)$.

2.2 Why QP For This Problem?

Why should we use quadratic programming for this problem, even though we think we might be able to formulate it as a simpler case of the fleet scheduling problem discussed in Section 1, which we know is commonly solved using linear integer programming methods? (See [4] for such a solution.)

Essentially, the reasoning boils down to the goals of keeping the number of variables as small as possible and of keeping the computational cost as low as possible. Formulating

the problem as a linear integer programming problem would require the introduction of a large number of dummy variables in order to keep the objective function and constraints linear. Computationally, the non-integer quadratic programming method is much cheaper than the binary integer programming method. Given the restriction that all variables must satisfy $0 \leq x_i \leq 1$, our possible solution space (in quadratic programming terms, the *feasible region*) is a cube in dimension N, and hence is bounded. Furthermore, because of the way our objective function will be formulated (see Section 4.2 below), the matrix H will satisfy $x^T Hx \geq 0$ for all x in the feasible region. Such a problem is said to be *convex*, and it is known that there are polynomial time algorithms to solve this specific subclass of quadratic programming problems [3]. Although the solution algorithm we will use is known not to be polynomial time, formulating the problem as a quadratic problem allows for easy transition to the use of a polynomial time algorithm if it is desired.

As an aside, the impetus to maintain the condition $x^T Hx \geq 0$ is another reason to formulate the optimality and non-optimality conditions strictly in terms of one or the other type, since as we discussed in Section 1 we can often convert an optimality statement into a non-optimality statement and vice versa. By ensuring that the terms in the objective function uniformly describe either optimality or non-optimality but not some combination of the two, we ensure that all the terms in the objective function are positive. If we have some terms describing non-optimality which are positive, then any terms describing optimal behavior of the schedule would have to be negative (removing non-optimality from the schedule), and we would then run the risk of causing the problem to become non-convex.

3 The Problem, Revisited

Let us now lay out the problem as it stands given the discussion above, for the sake of transparency.

3.0.1 Problem Statement

- Given a set of S students, P professors, C courses, and T time slots, and given knowledge of each student's enrollment in the courses and of the professors' teaching assignments, how is it best to assign the courses to time slots?

3.0.2 Assumptions

We make the following assumptions and employ them throughout the remainder of the paper.

1. Each course must be assigned to some time slot.

2. Each course is taught by at least one professor.

3. Each course has its own classroom, and this room is capable of accommodating an arbitrary number of students. (We make this assumption for the sake of simplicity; it is possible to add in the condition that courses must be assigned to rooms as well as to time slots.)

4. Each course is to be assigned a unique index — multiple sections of the same course are considered for the sake of scheduling to be different courses. (This could be relaxed, with some nontrivial effort in implementation, to allow multiple sections of a class to be merged into a single section, since we have assumed room capacity is essentially infinite.)

5. Assuming we have knowledge about the distances between the courses (or rooms), it is slightly non-optimal to make students late for class by scheduling their courses such that the distance they must walk from one class to the next is larger than they can traverse in the allotted period between classes.

6. Similarly, it is moderately non-optimal to assign courses in such a way that a professor must arrive late to class.

7. It is moderately non-optimal to assign courses to time slots in such a way that students are obliged to be in two courses at the same time. (The known enrollment data allows us to determine whether this is the case.)

8. It is *highly* non-optimal to assign courses such that professors are scheduled to teach two courses at the same time.

We will describe below in Section 4.2 how each of these assumptions is implemented.

4 Implementation

It turns out that MATLAB's Optimization Toolbox has solution routines which accommodate many forms of linear and quadratic programming. The routine `quadprog` accommodates quadratic programming of the type described in Section 2.1 above, with the additional benefit of being able to accept upper and lower bounds on the values of the variables and incorporate these bounds into the constraints without any work on the part of the user.

4.1 Under the Hood

A full description of the methods used by MATLAB to solve quadratic programming problems is given at [2]. In brief, the algorithm involves converting the quadratic objective function into a set of linear objective functions using Lagrange multipliers, at which point the solution can be obtained by standard methods of solving large systems of linear equations.

Mathematically, as stated in the documentation cited, the algorithm constructs a set of equations

$$\begin{aligned} \nabla f(x) + \textstyle\sum_i \lambda_i \nabla G_i(x) &= 0 \\ \lambda_i &\geq 0, \end{aligned}$$

where $f(x)$ is the objective function, $G(x)$ gives the values of the constraint functions at x, and the λ_i are the Lagrange multipliers. We have argued that our problem is convex; the documentation also indicates that for convex problems, solving this system of equations is necessary and sufficient for finding not just a local minimum, which is the best that many algorithms can do, but also a global minimum of the objective function on the feasible region. (Such a minimum exists because the objective function is bounded below by 0.)

All of this occurs "under the hood" and the documentation is no more forthcoming than this; presumably MATLAB's standard procedures for solving linear systems apply. The mechanism by which this solution method occurs is not particularly important for the remainder of this discussion, but is presented here for completeness.

4.2 Data Structures

There are in essence two sorts of data structures used in this scheduling program: those created by the user, and those created by the program to accommodate our specified constraints and non-optimalities.

4.2.1 User-Generated/Random

Recall that we have C courses, T time slots, P professors, and S students. Since we want our solution to be a vector of variables describing the time slots in which courses occur, we use a vector of length TC. The first C entries are 1 if course k is offered in time slot 1 and 0 if it is not. The second C entries are 1 if course k is offered in time slot 2, and so forth. In general, the entry x_{iC+k} is 1 if course k is offered in time slot $i-1$, for a total of TC entries. At the very end of the program, after all necessary computations have been performed on the vector of solution variables, the vector is reshaped into a matrix with the courses assigned to rows and the time slots assigned to columns, so that the visual format is more user-friendly.

We randomly generate a sparse $S \times C$ matrix with entries 0 or 1 and use this as the enrollment information for the program. A 1 in position (i, j) indicates that student i is enrolled in course j. Similarly, we generate a sparse, symmetric, traceless $C \times C$ matrix whose (i, j)th entry is 1 if course i and course j are too far apart to walk between in time for classes — symmetric because we assume the distance from i to j is the same as the distance from j to i, and traceless because we assume a class cannot be too far from itself.

Finally, we generate a $P \times C$ matrix describing which professors are teaching which courses. The construction of this matrix is slightly more involved than the others; in those

cases it was acceptable to have the random number generator generate a blank row, corresponding to a student enrolled in no classes (on academic leave, from an upperclassman perspective, though for incoming freshmen it might make more sense to say that this student chose to withdraw from the school before attending) or a classroom which is conveniently located near all the others. But for the matrix of teaching assignments, it is not permissible to have any of the columns blank, because this would correspond to a course that potentially had students enrolled in it, but did not have a professor to teach it! So to generate this matrix, we first set up a matrix of zeros to allocate the necessary memory; then for each column in the matrix, we choose a random number between 0 and 1 (uniformly distributed), multiply by the number of professors P, and take the ceiling of the result. This gives a random integer p between 1 and P, inclusive. We set the pth entry in the column to 1 and leave the others 0, indicating that professor p is teaching the course specified by that column. With exceeding rareness, it is possible that the random number generator could yield 0; but this is so rare that we need not concern ourselves with the possibility (perhaps the professor for the course went on emergency leave).

4.2.2 Optimization and Constraint Related

To construct the constraint equations specifying that each course must be taught at least once, we write

$$x_{0\cdot C+k} + x_{1\cdot C+k} + \cdots + x_{(T-1)\cdot C+k} \geq 1,$$

each term corresponding to the "amount" of course k that is taught in a particular time slot (or, ideally, simply whether course k is being taught in that time slot at all, if the scheduler gives an integer value for all variables). From these equations, one for each k with $1 \leq k \leq C$, we can construct a set of constraint inequalities in the form $Ax \geq b$ (though in order to conform to the structure required by MATLAB, we actually use $-Ax \leq -b$).

To construct the matrix H for the quadratic term of the objective function, we include one term that accounts for how many students are double-booked for a given time slot. If we assume courses k and l are both offered in time slot i, the coefficient of $x_{iC+k} \cdot x_{iC+l}$ in the objective function is

$$\sum_{s=1}^{S} \mathrm{e}(s,k) \cdot \mathrm{e}(s,l),$$

where $\mathrm{e}(s,k)$ is the entry in the enrollment matrix for student s and course k, being 1 if the student is enrolled in course k and zero otherwise. So the coefficient on this cross-term is simply the number of students who are double-booked in the two courses. In theory, when we enter this number into the $(iC+k, iC+l)$th slot in the matrix H, we ought to divide it by 2 because of symmetry, but we can also accomplish this rescaling by changing the weights of the other non-optimalities relative to this one, so we let it be.

To account for the slight non-optimality associated with making students late for class, we consider the term $x_{iC+k} \cdot x_{(i+1)C+l}$, which corresponds to putting courses k and l into adjacent time slots. We then use the distance matrix we generated randomly to create the coefficient

$$\sum_s \mathrm{e}(s,k) \cdot \mathrm{e}(s,l) \cdot \mathrm{dist}(k,l),$$

which corresponds to the number of students who have to get from one course to the next multiplied by the entry in the distance matrix for courses k and l. If the entry is 0, the classrooms are nearby and there is no associated non-optimality with scheduling these courses in back-to-back time slots; if the entry is 1, we get a contribution to the non-optimality equivalent to the number of students who are going to be late to course l because of the distance they have to walk. Since making people late is less unfortunate, relatively speaking, than making them miss class entirely, we apply a scaling factor to this term when we enter it into the matrix H. By default it is set to .1, but a user can edit the code to give it any (non-negative) value.

We can attach a similar term for making professors late to class by replacing $\mathrm{e}(s,k)$ with $\mathrm{profs}(p,k)$ in the above term, simply drawing from the teaching assignments matrix instead of the

Finally, to account for the severe non-optimality of double-booking a professor (which is as close as we can come to constraining double-booking not to occur), we include an addition to the coefficient of the term $x_{iC+k} \cdot x_{iC+l}$, namely

$$\sum_s \mathrm{profs}(p,k) \cdot \mathrm{profs}(p,l);$$

this is identical to the term we added for student overlap except that we draw information from the teaching assignments matrix instead. We also attach a large scaling factor to this part of the coefficient to make double-booking professor comparatively far more unfavorable than double-booking students (on the theory that an entire class is not brought to a grinding standstill if one student cannot make it, but the class cannot meet without a professor). By default this scaling factor is set to 10, but a user can edit the code to give it any (non-negative) value.

The description above of the terms we put in the objective function matrix H should serve as a template for anyone wishing to refine the objective function. Additionally we feed to `quadprog` a basic lower bound vector of zeros and an upper bound vector containing only ones, which is how the solver constructs its feasible region. This accounts for all the data structures associated with the program.

5 Results

5.1 Observations

First and most curiously, we notice that in most simulations, despite the fact that we are using a method whose solutions are not restricted to be binary integers, we see one of two behaviors. Either the scheduler suggests a solution which is in fact comprised of binary integers, or it recognizes that any placement of the courses in the available time slots will result in the same amount of non-optimal behavior, in which case it suggests a solution in which $1/T$ of each course should be put in each time slot. (For more on fractional course placement, see Section 5.2.3.)

Other than this curious behavior, the solver performs remarkably well (and the integer solutions are an added bonus). Because the only set of constraints is that each course must be offered at least once, the program essentially cannot fail to satisfy the constraints; it can just produce a solution which is highly non-optimal. For instance, if we specify a situation with 5 students, 3 courses, 2 time slots, and only 1 professor, it is quite willing to spit out a solution, with the additional caveat that the non-optimality of this solution is high, since there is no way to arrange such a schedule in such a way that the professor is not double-booked. This is discussed in more detail in Section 5.2.1.

5.2 Drawbacks and Limitations

5.2.1 Quadratic Constraints?

Even in quadratic programming, the constraints must still be linear equalities or inequalities, at least in MATLAB's implementation of `quadprog`. There is no way to specify a quadratic constraint term; any quadratic terms must be incorporated into the objective function. Unfortunately, many of the conditions we would like to use as constraints turn out to involve quadratic terms.

For instance, we would like to constrain the problem so that no professor is scheduled to teach two different classes in the same time slot. But this inherently involves checking the status of two different variables, x_{iC+k} and x_{iC+l}, to see if the professor is scheduled to teach courses k and l in time slot i. Under the current formulation of our variables, any term in the constraint would then have to involve the product of these two variables, since only if they are both 1 (indicating the professor is slated to teach both classes at that time) do we consider the situation a problem. So we cannot formulate this constraint linearly, and therefore must instead take it into account by including it in the objective function, using a large scaling factor relative to the other terms in the objective function so that it is comparatively more important and can still retain some of its constraint-like functions.

This is still undesirable from two perspectives: first, it does not actually prevent a professor from being double-booked, despite the fact that in reality this would cause the schedule

to be infeasible (or would require the department to do some hasty course reassignments). Second and relatedly, it violates our aesthetic rule that, structurally, things which are forbidden should be formulated as constraints, while things which are discouraged should be formulated as terms in the objective function.

It is reasonable at this point to ask why we do not restructure the problem to include more variables which allow us to write these constraints as linear constraints and avoid the above difficulty. The reason is that while using these expressions as terms in the objective function is structurally and aesthetically unfortunate, the penalties in computational cost for restructuring the variables are likely to be considerable. The current construction of the matrix for the quadratic term of the objective function involves a number of nested `for` loops (for lack of inspired programming ideas; suggestions for modification are welcome.), and if the matrix were to become any larger, the amount of time spent creating this matrix would be astronomical, even if most of the new entries were to remain 0. For small numbers of variables (on the close order of 20), the creation of this matrix is already the most time-consuming part of the solution process.

5.2.2 Computational Constraints

Aside from the fact that the creation of the coefficient matrix is time-consuming because of the inefficient `for` loop construction, the storage of this matrix is an incredible problem. Although the matrix is exceedingly sparse, `quadprog` currently has no special routine for dealing with sparse matrices, and if it converts the coefficient matrix back to full form in the process of doing computations on it, the memory usage will quickly become prohibitive for even a rather small problem.

To give a sense of scale, for a problem with $S = 20$, $C = 10$, $T = 5$, (without professorship constraints built in yet), it took perhaps two to three minutes to construct the coefficient matrix, and the optimizer converged almost instantly. For a problem with $S = 100$, $C = 50$, $T = 30$, it took approximately six to seven minutes to construct the coefficient matrix, and had taken half an hour to work on the optimization before the computer on which it was running needed to be rebooted. The timescale of the construction of the matrix is in part set by the number of students, so if this is high, this portion will be computationally expensive; but the product TC sets the timescale for the actual solution of the problem, and since this is the portion of the scheduler which uses an exponential-time algorithm, this is in general the rate-limiting factor in these computations.

5.2.3 Fractional Students?

Although the quadratic programming method is valuable in terms of cheap computations and simple implementation, it has the troubling characteristic that it is still a non-integer method and has the capability of giving results that indicate that some fraction of a class should be offered in one time slot and the rest in another, which is exactly what we said

we did *not* want when formulating this problem. Although this seems dire, all is not lost. We might, for instance, interpret this as an indication that it is preferable to put the course in the time slot which has the larger fraction, though we could choose to pay some cost in non-optimality to put it in the other time slot if there is some other preference which our objective function is not equipped to take into account.

Also, the behavior mentioned above in Section 5.1 in which the quadratic program gives an integer solution in almost all cases despite not being restricted to integer solutions is comforting — perhaps there is some hidden behavior that prevents it from giving non-integer solutions other than the single solution which indicates all timetables are equally good. This would be the case if (except in the case of all timetables being equivalent) the objective function were along a portion of the quadratic that happened to be monotonic with respect to each variable separately, over the entire feasible region. In this case, the extremum of the objective function for the feasible region would occur in one of the corners of the N-dimensional cube that forms the feasible region, and of course the corners have binary coordinates.

6 Further Work

6.1 The Upperclass Scheduling Problem

We posed this problem as a situation in which all students are pre-assigned to courses with, by the time the problem is given to the solver, no choice in the matter. While this might be realistic for freshmen in their first semester, or for high school students, it is rare for upperclassmen to be asked to choose their classes for an upcoming semester with no knowledge of whether those classes will, for instance, be scheduled early in the morning when they would rather be asleep. The problem of choosing an optimal timetable in this situation then becomes more difficult than the freshman scheduling problem. Instead of knowing precisely the pairwise enrollment overlap of the courses, we are forced to rely on other information to determine when it is likely that two courses will overlap. We might use statistical data from previous registration information to determine these probabilities, or we might suppose that it is more likely that two courses in the same major will be more likely to overlap than two courses in very different majors. This would require the input of many more optimization terms and constraints, plus the application of some ingenuity and statistical elbow grease to identify the most meaningful terms to include in the objective function.

6.2 Refinement of Constraints

The optimization terms and the constraints we have included in the implementation so far are only the barest minimum of the constraints and optimization terms that could be

implemented. Entering new terms into the coefficient matrix H will not take up significantly more time than it already takes, since the time-consuming portion of the process is cycling through the for loop, not the data entry within the for loop. Including constraints that require the introduction of dummy variables, while also within the bounds of possibility, is more likely to cause computational slow-down. What follows is a list of possible refinements to the existing program, with commentary.

6.2.1 A Sampling of Possible Options

- Specify that we must optimally place courses in rooms as well as time slots. This requires another RC entries in the solution vector, where R is the number of available rooms.
- For additional excitement, specify that some courses can only be placed in certain subsets of the rooms available — this corresponds to, for instance, ensuring that a laboratory course actually meets in a laboratory. Relatedly, we might also specify that rooms have a maximal capacity and use the student enrollment information to ensure that we do not overfill a classroom.
- Allow, as briefly mentioned earlier, for two sections of the same class to be taught at the same time, effectively absorbing them into one larger section.

This is of course only a very small sample of the possible additions that could be made to the objective function or to the structure of the program. One of the advantages of keeping the data-structure-generating pieces of the program separate from the file which actually solves the optimization problem is the level of modularity it affords — provided the optimization terms are kept separated by source (e.g. we don't mix professor double-booking and student double-booking into a single coefficient that we can't pick apart), it is easy to add or remove terms and add or remove data structures effectively on a whim.

7 Conclusions

Although this program is relatively simplistic, its extensible nature makes it suitable for a large class of small scheduling optimization problems. We specify "small" optimization problems because the computing time limitations are such that it is not reasonable to attempt to optimize over scales much larger than $TC \simeq 10^3$. Fortunately the fact that the number of available time slots T is generally relatively small (even on the entire 5C, there are perhaps 30 time slots) helps keep the problems manageable, but in general, we recommend the scheduling program primarily for, e.g., scheduling by individual departments. Unfortunately, this is not a terrific improvement on the current way of scheduling which is done primarily by the human brain, since the number of departmental course offerings is small enough and the

professors familiar enough with the curriculum that a professor could probably arrange the courses much more efficiently than this program, with much more subtlety. However, for a professor new to the department, or perhaps even more for a high school employee tasked with scheduling classes, this could be an efficient tool.

The behavior that integer solutions appear even when not constrained to be so is curious and unexplained. If indeed the objective function turned out to be monotonic with respect to each coordinate, that would lead to this behavior, but we do not know that this is the case, nor that if it is the case now it will necessarily remain so if the optimization problem is refined in some of the ways detailed above.

References

[1] `quadprog` program documentation, in MATLAB documentation. The MathWorks, Inc. Available at: `http://www.mathworks.com/access/helpdesk/help/toolbox/optim/ug/quadprog.html`

[2] Constrained Optimization documentation, in MATLAB Optimization Toolbox documentation. The MathWorks, Inc. Available at: `http://www.mathworks.com/access/helpdesk/help/toolbox/optim/ug/f51152.html`

[3] Nick Gould. Quadratic Programming: Theory and Methods. Presentation slides. 3rd FNRS Cycle in Mathematical Programming. Han-sur-Lesse, Belgium, February 2000. Available at: `ftp://ftp.numerical.rl.ac.uk/pub/talks/nimg.hansurlesse.3II00.ps`

[4] Glenn W. Graves, Richard D. McBride, Ira Gershkoff, Diane Anderson, Deepa Mahidhara. Flight Crew Scheduling. *Management Science*, Vol. 39, No. 6 (June 1993), pp. 736–745.

[5] Wikipedia contributors. Complexity classes P and NP. Wikipedia, The Free Encyclopedia. Available at: `http://en.wikipedia.org/w/index.php?title=Complexity_classes_P_and_NP&oldid=50385094`