Assigning and Scheduling Teachers and Students

Keshane Gan
Advisor: James Aspnes
CPSC 490
Department of Computer Science
Yale University

May 5, 2016

1 Introduction and Motivation

Every school year, the Yale University Guild of Carillonneurs recruit new members to join their group to play a musical instrument called the carillon. During this process, each member of the Guild teaches several students individually how to play the carillon (through a process they call Heel) before those students audition for membership. These lessons are scheduled once a week over nine weeks. Since the founding of the Guild, the responsibility of scheduling these lessons lay in the hands of one Guild member.

The popularity of the Guild has made this responsibility more time consuming and stressful. Several hours are usually spent with one or two additional helpers trying to schedule teachers and students. Several hard [H] and soft [S] constraints must be satisfied:

- [H] Both the teacher and the student have to be available for the lesson time.
- [H] Because there is only one practice instrument available, only one lesson may be taking place at a time.
- [S] Each lesson is 30 minutes long and are assigned to half-hour slots between 8:00 AM and 12:00 AM. For various reasons, some of these slots may have to be made unavailable for lessons.
- [H] Students must be taught by teachers at least one college class year higher. For example, freshman may be taught by a sophomore, junior, or senior, but a sophomore can only be taught by a junior or senior. For the cases when a graduate or professional school student is taking lessons, a Guild member also in the graduate or professional schools should teach the student. The Guild may not have such a member, so a college senior may also be the teacher.

- [H] Students must not be taught by a Guild member whom he or she personally knows.
- [S] Teachers should be given students with a variety of musical experiences.
- [S] All teachers should have roughly the same number of students.

A couple of other important considerations is that the number of members in the Guild has historically been between 15 and 20 and the number of students in Heel has been between 50 and 100. Automating the scheduling process would allow the responsible Guild members to focus their energy and efforts on other important aspects of coordinating Heel. I aim to provide them that automation. Note: for brevity and clarity, members of the Guild who act as the teachers will be referred to as *Guildies* and the students who are taking lessons to learn carillon will be called *Heelers*.

2 Background

The nature of this problem resembles several others that are widely known in the optimization field, namely the assignment problem and the interval scheduling problem.

2.1 Assignment Problem

In the assignment problem, p people must be assigned to j jobs (for now assume $p = j$). There is a specific cost for each person performing a specific job. The goal is to assign people to these jobs so that the total cost of those jobs is minimized.

In [1], Munkres presents an algorithm to find this minimum cost. In his discussion, he represents all the costs in a matrix. This matrix in turn can be represented as a complete bipartite graph. Each element in the matrix can be discussed as the weight of an edge between a vertex in the set of rows P and a vertex in the set of columns J. The goal is to turn this bipartite graph into a perfect matching between the two sets of vertices. Using a matrix makes this algorithm easier for computation. For a matrix, the goal is thus to essentially find an independent set of 0s (created by simple subtraction operations) in the matrix. This can be completed in $O(n^3)$ time, which is much faster than the naïve $O(n!)$ solution (where n is either the people or the jobs). Throughout this paper, Munkres's solution will be referred to as the Hungarian algorithm (named after earlier work by two Hungarian mathematicians, though Carl Jacobi had already solved it in 1890).

In the current problem, figuring out the cost of a Guildie having a particular Heeler is not straightforward. Some of our constraints cannot be represented solely in a one-dimensional cost. In addition, there are many more Heelers than Guildies. The solution presented by Munkres cannot handle this many-to-one mapping.

2.2 Interval Scheduling

Interval scheduling seeks to assign the most tasks in a given time frame, with the parameters that the tasks can be arbitrary lengths and the constraint that no more than one task may run at one time.

A greedy algorithm provides the solution to this problem. On each iteration, the task that will finish earliest should be assigned. All other tasks whose intervals intersect the assigned task must be discarded.

As opposed to the assignment problem, the interval scheduling problem is slightly easier in the current case. Assuming we already have Guildie-Heeler pairs, each of those pairs will likely have multiple times that it is available together. Thus, a pair need not be removed from the set of pairs if it does not end up being scheduled in an interation since there will be more chances for it. In addition, all lessons last the same time, so finding the one with the earliest finishing time is unnecessary. A straightforward comparison of the start time is sufficient, but if we wanted to be exact, the pairs could be sorted on their finishing/start time *and* the number of other available times they have (as a tie breaker). This would help sure the pairs that have fewer available times are allowed to be scheduled as soon as possible.

While the aforementioned problems form the core of the program, their implementations are not. There are many subtle and not-so-subtle wrinkles that have been (and still need to be) solved.

3 Implementation

In this section, I will give a broad overview of my implementation and the reasons behind my decisions. The program was written in Python 2.7 with `print` functions imported in from the future and uses Numpy. The main module is `match.py`. To run, `python match.py <file of Guildie data> <file of Heeler data>`. The comments in the source code are mostly helpful in explaining the details.

3.1 Generating Data

To gather data, I used Google Forms. The service provides a convenient way of viewing the data and exporting it into a tab-separated file. The link to the form is here. All Guildie and Heeler data are stored in `.tsv` files.

Unfortunately, I was not able to collect enough data to use as test data for my program. I *was* able to collect information from 13 current Yale students. I treated these students as the Guildies and extrapolated their data to generate a set of fake Heelers. The code for this can be found in `gen_data.py`.

For each time slot on a particular day, I used the percentage of Guildies who responded that they were available as the probability that a Heeler would be available. To add a little more variability to imitate the different schedules of real Heelers, this probability was given a variance equal to $p(1-p)$, where p is

the probability. For a given day, each Heeler also had a limit on the number of times he or she could be available. This limit is determined by a normal distribution based on the number of times the Guildies were available for that day.

Musical experience was not collected on the Google Form because every responder would have a different view of their own experience. In addition, it is difficult to judge experience quantitatively. In practice, the intent is for Heelers to respond with their *background* (in text) in music and for the collector to assign a number from 1 to 10 as their musical experience. These values are used solely for ordinal purposes, not cardinal purposes, so their exact values do not matter. For the tests, a normal distribution of experiences was assigned (with mean of 7 and standard deviation of 2).

Instead of unintelligible, random strings, Heeler names come from United States Census data from 1990. Those files can be found in `names/`.

During the actual processing phase, there are two broad stages: assignment and scheduling. Guildie-Heeler pairs are first created before they are scheduled. I will describe the assignment stage first.

3.2 Person and Pairing Classes

In `match.py` a `Person` class is defined. This represents either a Guildie or a Heeler, and its attributes correspond to the data collected in the `.tsv` files. The availability of a person is stored in a two-level dictionary indexed by a string of the day of the week and a string of the time; the value is 1 if the person is available at the indexed time, 0 otherwise. The parser reads the files, creates a `Person` for each line of the file, and stores that `Person` in a list. The main driver of the program maintains two lists: one for Guildie `Person`s and one for Heeler `Person`s.

Then all combinations of pairs between Guildies and Heelers are created through the use of the `Pairing` class. This class defines a pair and holds information about the combined availability of the Guildie and Heeler (essentially an `AND` of their availabilities) and the cost of that pair. A perfectly representational cost, as mentioned previously, is difficult to achieve given the constraints. After some heuristic experiments, I settled on the following:

- +1 for every time slot that both Guildie and Heeler are not available
- +1000 for having no overlapping availabilities at all
- +500 for a Guildie (excluding seniors) being in an equal or lower class as the Heeler
- −100 for a graduate Guildie teaching a graduate Heeler
- +20 for a senior Guildie teaching a graduate Heeler
- +1000 for a Guildie-Heeler pair who know each other personally

The general rationale for these numbers is to discourage selecting pairs that do not have many available times since a pair with only a small availability would have smaller chances of finding an open lesson slot. Time is the strictest constraint, so the penalties must be high. Enforcing a lower class Guildie to teach a higher class Heeler is important for maintaining authority during lessons, but, unlike the time constraint, is not absolutely necessary for *having* a lesson. Having a Guildie teach a Heeler whom he or she knows can lead to favoritism and strained relationships, so this situation merits a high cost. To emphasize, these numbers do not have any statistical basis, but they did work well on the tests.

3.3 Matching

In `hungarian.py`, the Hungarian algorithm is implemented as the class `Hungarian`. It closely follows the outline discussed by Munkres. The steps closely resemble a finite state machine, possibly looping several times as the edges of the bipartite graph are "pruned" before satisfying the exit condition. The Hungarian algorithm always finds the same assignment for the same set of inputs, but there are subtle conditions which make this optimum assignment not unique. In my implementation, iterations through the matrix start at $(0, 0)$ (as is natural). But this means that it finds the zero costs earlier in a row and earlier in a column. If a set of inputs has the same cost for more than one pair, then reordering the columns or rows could lead to different assignments that are still optimum. This is important to keep in mind.

Preprocessing is done before a cost matrix is fed into `Hungarian`. To eliminate most of the problems of assigning h Heelers to g Guildies when $h > g$, we partition the Heelers into g size parts. This way, we can assign each Heeler-partition separately to the Guildies and create cost matrices with equal numbers of rows and columns. In addition, this process evenly spreads the Heelers among the Guildies, so no one Guildie will have too many or too few Heelers. However, h is likely not to be divisible by g, so we may get to the point when $h < g$. We do not partition the Guildies like we did for the Heelers because we would end up assigning the Heelers to the first partition of Guildies and not allowing the rest of the Guildies to try to be assigned. Instead, we pad the rest of the columns (Heelers are the columns, Guildies the rows) with a dummy value. In this current case, that dummy value is 100000, an extremely high value that a regular cost cannot reach. The intention behind this is to discourage the best Guildies for the real Heelers from being matched with a dummy Heeler, although in certain cases the initial row and column subtraction operations in the Hungarian algorithm may annull this intent.

It is at this preprocessing step that we also try to satisfy the range-of-musical-experience constraint. Before the Heelers are partitioned as described above, they are sorted according to their musical experience. Thus, for every partition of the Heelers, all the Guildies are matched with Heelers who have around the same musical experience. This method allows each Guildie the opportunity to be assigned to a Heeler with a lot of musical experience, a Heeler with medium

amount of experience, and a Heeler with little to no experience. Since the order of the Guildies always stays the same, Heelers are shuffled within their partition to ensure that one Guildie is not biased to consistently receive a Heeler with less than or greater experience than the other Guildies receive. This shuffling takes advantage of the quirk of reordering inputs mentioned earler in this section.

3.4 Scheduling

After Guildies and Heelers are assigned to one another, they must be scheduled. `scheduler.py` contains the appropriate functions.

In `scheduler.schedule()`, we iterate through each day's time slot and assign a pairing if that pairing is available at that time. One bug/feature I noticed is that lessons tended to get scheduled earlier in the time frame, i.e. earlier in the week and in the morning, because of the nature of scheduling lessons at the earliest time possible. To hold back this early bias, the order of the time slots are shuffled for each day. Since the possibility of failing the scheduling is small with so many possibilities (see Interval Scheduling), this risk is safe enough to take. Unfortunately, shuffling the order of the days does not have the same effect since the `scheduler.schedule()` only iterates through the days of the week once per call. Lessons will tend to bunch up on the days that are first iterated through and will be sparse on the last day. Another way to hide the bias is to limit the number of lessons that can be assigned per day. Thus, the function will be forced to move on to the next day instead of overloading the current day. One drawback of this approach is that some pairings may now not be able to be scheduled.

However, it may be desirable not to have lessons on a particular day or during a particular time of day, so whether this issue is a feature or a drawback is left up to the user.

3.5 Handling Failures

There is always the possibility that all the assignments are not able to be scheduled due to various limitations, whether it be the scheduler, the number of time slots, or busy schedules. A mechanism exists to deal with this possibility.

First, the residual Heelers are reassigned to Guildies (through the same assignment with minimum cost as before). Then, an attempt is made to schedule this pairings set in the existing schedule without the tweaks mentioned in the Scheduling section. These aforementioned steps are repeated a certain number of times (arbitrarily set to 10) until all the Heelers have lessons.

If the loop fails, then a pseudo brute force option is used. For each day's empty time slot, all combinations of Guildies and Heelers are tested at that time, and given that the cost is not exceedingly large and the pair is available, a lesson is assigned to that time slot. The reason that this is not a pure brute force solution is that we schedule the first pairing that works - we do not try all possible permutations of all the remaining pairings. This pure brute force

solution was attempted, but crippled my computer with its $O(n!)$ complexity (see `match.py: brute_force()`).

If even the pseudo brute force does not work, the program returns the schedule that it has so far and notes the Heelers that could not be scheduled.

Acknowledgements

I would like to thank Darien Lee and Alex Carillo for their support and company on my all-nighters. I would also like to thank Megan Brink for her encouraging words. Special thanks goes to Christopher Shriver for the idea and motivation for creating this. Thank you Professor Aspnes for starting me on the right track on this project.

References

[1] J. Munkres, "Algorithms for the Assignment and Transportation Problems," *Journal of the Society for Industrial and Applied Mathematics*, vol. 5, no. 1, pp. 32 - 38, Mar. 1975.