A Local-Search Approach to Timetable Scheduling

Katherine Sullivan
University of Maryland, College Park
May 2021
katiesul@umd.edu

ABSTRACT

In this paper, I present an implementation to assist in a real-life problem —assigning prospective graduate students to faculty of the University of Maryland Computer Science Department. This timetabling problem is an NP-Hard problem along with other similar problems such as the examination timetabling problem and the high-school timetabling problem. The implementation uses local search heuristics and is evaluated using randomly generated schedules as well as real-life Visit Day data.

1 INTRODUCTION

This research assists in a real-life problem relevant to the University of Maryland Department of Computer Science: assigning prospective graduate students to faculty for meetings on visit days. The goal was to replace the manual assignment that the Assistant Director of Graduate Education does each year with a program that can try many combinations of schedules in order to find one as optimal as possible much more efficiently than a human could.

The constraints of the schedule are as follows. There are several half-hour time slots on visit days for students to meet with professors, and depending on their interested area of research, students can submit their preferences on which faculty they would like to meet. To ensure students have a good experience, an optimal schedule honors as many of these preferences as possible, keeping in mind that many professors have limited availability. Students may attend tours, which limits their availability. There are also some soft limits such as trying to give each student about five to six meetings and also trying to distribute the meetings among the professors relatively equally.

The problem of assigning graduate students to faculty for visit days can be considered a timetabling problem. *Timetabling problems* seek to schedule events in the most optimal way possible according to specified objectives. In these problems, there are often different types of constraints and multiple variables including time, location, and groups of people. Various types of timetabling problems have been studied. One example includes the high-school timetabling problem. While high -school timetabling can vary between different schools and countries, in general, the object is to assign teachers to specific classes at specific times in specific classrooms [5]. Some obvious hard constraints which cannot be violated would include ensuring that no teacher is assigned to teach two different classes at the same time, or ensuring that no two classes are assigned to the same classroom at the same time. Some soft constraints, which can be violated if necessary but which are preferable to follow, could include scheduling teachers so that they have appropriate amounts of planning periods and scheduling teachers so that they do not have to travel long distances around the school to teach consecutive class periods.

With all of the different variables and constraints, it is no surprise that these problems are NP-Hard, meaning that it is not feasible to determine an optimal solution efficiently and the optimal solution is in fact often unknown. Luckily, timetabling problems are highly researched and various methods have been proposed and used to deal with them.

2 RELATED WORK

McDiarmid (1972) solves a timetabling problem in which teachers are assigned to classes and have to satisfy a requirement matrix A = (a_{ij}) indicating the amount of times a particular teacher has to "meet with" a particular class; at the same time, the goal is to minimize the total amount of time and space required in the schedule [1]. McDiarmid reduces this timetabling problem to a graph problem: the problem is represented as a bipartite graph G with the set of teachers T and the set of classes C as vertex sets and an edge set with A_{ij} edges joining T_i and C_j. The timetabling problem can be solved by finding the chromatic index $I(G)$ and an $I(G)$-coloring of the graph [7]. McDiarmid uses a standard graph matching algorithm, the Hungarian algorithm, to find the $I(G)$-coloring of the graph and thus solve the timetabling problem.

While this method seems promising, in the course of my review of prior work, I determined that reducing the problem to a graph problem was too difficult or perhaps not even feasible. This is because my problem involves more variables including graduate students' preferences of which faculty to meet as well as potential time conflicts from both faculty and students, making my problem more complex than the problem from McDiarmid's work.

Heuristics are another set of techniques often used in timetabling problems. A heuristic is like a "rule of thumb" approach to solving a problem. This type of approach is often useful for approximating an optimal solution to a problem, especially when there are a combinatorially large amount of possible solutions that cannot all possibly be enumerated. Unsurprisingly, timetabling fits this category, and heuristics are a common approach to solving such problems. Examples can be seen in the International Timetabling Competition,a competition held to encourage further research into the field of complex timetabling problems. A glance over the winners' PDFs from the lesson scheduling track and exam scheduling track of the 2007 competition shows that the winning solutions were based on local search heuristics [4].

Local search is a technique where one starts with an initial feasible (but not optimal) solution to the problem and tries various neighboring states, or solutions that are very closely related to the current solution (for instance, in a timetabling problem, a neighboring solution could be swapping two professors' lectures from the current solution). To determine if the move is accepted, part of the problem definition should include a cost function, which defines the optimality of a solution. For a timetabling problem, the cost function could involve measuring how many people received their preferences. Cost functions can also include penalties or rewards: for instance, conflicts could incur a negative penalty.

There are many different types of heuristic techniques and even combinations thereof. Moscato and Schaerf (1998) write about several different techniques and "propose the combination of local search with other solution paradigms, such as genetic algorithms and constructive heuristics" [2]. In a 1999 paper, Schaerf extends her work by coming up with an algorithm combining heuristic techniques and applies the results to real schools [6]. In particular, Schaerf used a mixture of Tabu Search and the randomized non-descent method (RNA), a variant of local search in which a neighboring state may be accepted even if it does not strictly improve the cost function (to allow for potentially getting over a "plateau" in the cost function space). Azimi (2005) tested different heuristics (Simulated Annealing, Tabu Search, Genetic Algorithm and Ant Colony System) to solve the Examination Timetabling Problem and found that their novel combinations of the heuristics were more effective than the heuristics used individually [3].

The problem in my research is similar to high-school timetabling in that there are staff, students, timeslots, and preferences involved. Based on the promising research on using heuristics for timetabling, and given this similarity, I elected to use a heuristics approach in my algorithm.

3 DESIGN

First I discuss the goals I considered when designing the algorithm. Then I talk about different types of algorithms that I considered in the design process of the program.

3.1 Goals

In designing the program to solve the visit day problem, I had several goals in mind:

- **The program should be easy to use.** Since the program will be used by others who have not written the program and who perhaps may not have a Computer Science background, the program should be simple to run.
- **The program should be time and space efficient.** Even though the number of students attending a visit day will be of moderate size (on the order of hundreds), and there is also a limited number of faculty, efficiency is important to keep in mind. The program should be much more efficient than the prior manual process of creating a schedule.
- **The schedule should follow all hard constraints and attempt to accommodate as many soft constraints as possible.** Hard constraints include ensuring that no students or professors are assigned to be in two places at once; that no professors or students are assigned to have a meeting during a time they indicated they were unavailable; that students have 1-on-1 meetings with professors; and that students have at least some minimum amount of meetings. Soft constraints include attempting to give students meetings with as many of their preferred faculty as possible, weighting first preferences higher than second preferences, second preferences higher than third preferences, and so forth.

In order to make the program easy to use, I wrote the program in Java and documented my code. I made the program available to the Assistant Director on a GitHub repository and I included a detailed documentation PDF on how to run the program and view the results. I used .CSV files for the input and output files of the program, which can easily be created and edited in programs such as Google Sheets or Microsoft Excel. In the documentation, I showed how to use a Google Form to automatically create a Google Sheet of results from a form. This makes for a feasible workflow where the Assistant Director can send out a Google Form survey to the graduate students, have them fill out information such as their preferences, and then compile all of this information in a Google Sheet (during the manual process, the Assistant Director was already doing this anyway). The Assistant Director can also send out a Google Form survey to the faculty to get their availability information (during the manual process, the Assistant Director was having faculty individually email him their availability; a Google Form would streamline this process while also collecting the data in a convenient spreadsheet form for the program).

3.2 Initial Schedule Heuristic

To begin, the program creates an initial schedule that is as optimal as possible using a simple algorithm. I considered a few different heuristics for this. First is a naive heuristic (see Algorithm 1), which serves as a baseline to compare against other potentially better heuristics. The naive heuristic simply shuffles the list of students and loops maxNumberofPreferences times, attempting to assign the current student a meeting with their *i*th preference.

Algorithm 1 Initial Schedule Heuristic 1 (Naive Baseline)

```
1: Let students = shuffled list of all students
2: Let n = maximum number of preferences
3: for s in students do
4:     for i from 1 to n do
5:         Attempt to assign a meeting with s's ith preference
6:         if s has no i+1th preference then
7:             Remove s from students
8:         end if
9:     end for
10: end for
```

Next is Heuristic 2 (see Algorithm 2). Heuristic 2 attempts to balance out which students get their preferences. It shuffles the list of students each round and if a student got their ith preference in the ith round, they are added to a secondaryStudents list; if not, they are added to a primaryStudents list. Each round, the algorithm looks at the primaryStudents first for scheduling meetings and then the secondaryStudents list (in the first round, the shuffled list of all students make up the primaryStudents list while secondaryStudents list is initially empty). In this way, students who did not get their (i-1)th preference should be more likely to get their (i)th preference as they will be considered first. Additionally, the algorithm shuffles both the primaryStudents and the secondaryStudents before starting the next round, so that the same student will not be at the top of the list every round.

Algorithm 2 Initial Schedule Heuristic 2 (Priority Lists)

```
1: Let primaryStudents = shuffled list of all students
2: Let tempPrimaryStudents = []
3: Let tempSecondaryStudents = []
4: Let secondaryStudents = []
5: Let n = maximum number of preferences
6: for i from 1 to n do
7:     for s in primaryStudents do
8:         Attempt to assign a meeting with s's ith preference
9:         if s has an i+1th preference then
10:            if attempt was successful then
11:                Add s to tempPrimaryStudents
12:            else
13:                Add s to tempSecondaryStudents
14:            end if
15:        end if
16:    end for
17: end for
18: for i from 1 to n do
19:     for s in secondaryStudents do
20:         Attempt to assign a meeting with s's ith preference
21:         if s has an i+1th preference then
22:             if attempt was successful then
23:                 Add s to tempPrimaryStudents
24:             else
25:                 Add s to tempSecondaryStudents
26:             end if
27:         end if
28:     end for
29: end for
30: Clear primaryStudents and secondaryStudents
31: Copy over tempPrimaryStudents to primaryStudents
32: Copy over tempSecondaryStudents to secondaryStudents
33: Shuffle tempPrimaryStudents and tempSecondaryStudents
34: Clear tempPrimaryStudents and tempSecondaryStudents
```

Another heuristic I considered is Heuristic 3 (see Algorithm 3). Heuristic 3 sorts the students by increasing order of the number of preferences they listed. The idea is that if there is a student with only 2 or 3 professors, for example, they should be looked at first before students with 5 preferences, because if the student with less preferences does not get any of theirs, the optimality of the schedule would likely decrease. In other words, the scheduler should attempt to give everyone at least one or two of their preferences, and in order to make this more likely to happen, this heuristic takes this strategy. The heuristic still uses randomization as it shuffles the students within buckets of the same number of preferences.

3.3 Local Search Heuristics

As mentioned, I elected to use heuristics in my algorithm based on the literature review. In particular, I chose to implement local search heuristics. Local search algorithms begin with a starting state (for example, an initial schedule in our case) and make small changes (for example, swapping two appointments in the schedule) to see if the optimality of the solution improves, where the optimality is determined by the cost function. There are different variants of local search. Some local search heuristics only accept a move if it strictly increases the cost function. Others also accept a move if the cost function remains the same or increases. Still others accept decreases in the cost function as they may allow the local search to get over a "plateau" in the search space.

Algorithm 3 Initial Schedule Heuristic 3 (Ascending Number of Preferences)

```
1: Let listOfArrays = []
2: Let n = maximum number of preferences
3: for i from 1 to n do
4:     Create a list of students with exactly i preferences
5:     Shuffle the list
6:     Append the list to listOfArrays
7: end for
8: for i from 1 to n do
9:     for s in listOfArrays[i] do
10:        Attempt to assign a meeting with s's ith preference
11:        if s has no i+1th preference then
12:            Remove s from listOfArrays[i]
13:        end if
14:    end for
15:    for i from 1 to n do
16:        Shuffle listOfArrays[i]
17:    end for
18: end for
```

For this program, I implemented a simple local search heuristic. To find a neighboring state of the current schedule in the program, I wrote a method called randomMove (see Algorithm 4). The method selects two random professors and swaps two random timeslots (essentially cells in the schedule) and then calculates the "satisfaction level," or optimality, of the current schedule. I include penalties for if the professor is already meeting with the student at another timeslot, or if a student or professor are now scheduled for a time they indicated they were not available after the swap. This is to help discourage the algorithm from selecting swaps like these that could lower the optimality of the schedule.

4 EVALUATION

In this section, I discuss the cost function which is used to quantitatively measure the optimality of schedules created by the algorithm. Then I discuss the results of experiments run on the different initial schedule heuristics as well as the local search portion of the algorithm in order to evaluate their effectiveness.

4.1 Cost Function

The cost function I defined for the problem is a simple one. The program checks if students have received their preferences. Assuming there are n preferences, if a student received their 1st preference, this results in $+n$ to the cost function; if the student received their 2nd preference, then $+(n-1)$; and so forth. There are also penalties which decrease the cost function if the swap would schedule a student and/or professor at a time when he/she is unavailable, or if the swap would schedule a student/professor to meet with the same

Algorithm 4 Random Move

```
1: Save the students' and professors' current state
2: Let prevSatisfaction = the current satisfaction
3: Select two random distinct professors
4: For each professor, select a random timeslot
5: Swap the meetings at each timeslot between the professors (professor 1 will meet with who professor 2 is meeting at professor 2's timeslot and vice versa) ▷ This can include having no meeting, in which case the swapped professor would not have a meeting at that spot after the swap
6: Apply penalties for if the professor is already meeting with that student or if the student or professor is not available at that time
7: With the penalties applied, let currSatisfaction = the current satisfaction
8: if currSatisfaction < prevSatisfaction then
9:     Revert students' and professors' state ▷ move not accepted
10: end if
```

person more than once. The penalties are meant to discourage the local search part of the algorithm from making moves that would break these schedule constraints. Because the cost function is based on how "satisfied" the students are with the meetings they receive according to their preferences, I often refer to the optimality of the schedule as the "satisfaction" level.

4.2 Initial Schedule Heuristic

To evaluate the three different heuristics, I conducted a trial of 100 rounds. In each round, I used the random schedule generator to test each of the 3 initial schedule heuristics on the same 100 randomly generated schedules. In each round, I recorded the currentSatisfaction/maxSatisfaction (the current cost function at the end of the initial schedule divided by the maximum value the cost function can attain for the given schedule) as well as the average percentage of preferences that students received. Table 1 reports on the averages of these two metrics across the 100 rounds. The heuristic with priority lists came out slightly on top over the naive heuristic while the ascending number of preferences heuristic was much less satisfactory; therefore, I opted for Heuristic 2.

4.3 Random Schedule Generator

In order to evaluate my algorithm, I wrote a generator function (see Algorithm 5). The function randomly generates professor availability files and student preference files and then runs the algorithm to create a meeting schedule.

To create realistic schedules, I looked at the data provided from the 2019 Visit Day as my guide. There were 70 different faculty listed in the 213 preferences made by 83 different students. The majority of the faculty were only listed a handful of times; however, 10 were listed greater than 5 times; 1 was listed 10 times; and 1 was listed 15 times. In real life, some faculty members are likely to be preferred more than others due to various factors such as popular research areas or being more well-known. Therefore, I designed the pool of generic student names to have 85 different student names and the pool of generic professor names to have 80 professor names.

Algorithm 5 Random Schedule Generator

```
1: Let students = shuffled list of all students
2: Let n = maximum number of preferences
3: for p in testProfessors do
4:     Let t = number of time slots
5:     Let n = a random number between t and (t - 5)
6:     Select n random distinct time slots and append to p's availability
7: end for
8: for s in testStudents do
9:     Let n = a random number between 1 and 5
10:    Select n random distinct professor names and append to s's preferences
11: end for
12: Run the scheduler algorithm on the randomized inputs
```

I also weighted several professor names to have a higher chance of being selected for preferences than other professor names by adding them in the pool a greater amount of times, roughly trying to follow the distribution in the 2019 Visit Day data.

4.4 Local Search Heuristics

To evaluate the local search portion of the algorithm, I ran 100 trials using the Priority Lists heuristic to create an initial schedule, as that was the most promising based on the section 4.2 results. The experiment took 882.646 seconds to run, or about 8.83 seconds per round. On average, the local search portion of the algorithm led to a 0.3066% increase in optimality of the schedule. While this may seem small, it should be noted that the average maximum satisfiability of the schedules in the experiment was 931.67 which is a large number. The average optimality of the schedules in the experiment after the local search portion was 88.49%.

4.5 Real Life Visit Day Evaluation

The Assistant Director gave me the information files from the 2019 Visit Day in order to evaluate the program on realistic data and learn the format of the data. I ran the scheduler on this data. With a maximum satisfaction value of 895, the initial schedule heuristic outputted a schedule with a score of 870, which was improved to 873 with the local search heuristic. This gives a 97.5% optimal schedule. The program ran in 8.724 seconds. These results give even more confidence in the ability of the program to output a close-to-optimal schedule in fractions of the time it would take a person to do it by hand. Additionally, students or professors may have to change their availability or preferences at the last minute. If this happens, the Assistant Director can simply edit the input files to the program and re-run it again with the new data. This would be a painstaking process if done manually.

This year, due to COVID-19, Visit Day was made online. As a result of this, students did not have to fly or travel into Maryland for the event, and thus many more students participated than in prior years. In total, there were 134 students, much more than in 2019 (in 2020, there was no event held due to the pandemic). While I did not see the data myself to perform experiments on the optimality, the Assistant Director used the documentation I provided with the

Table 1: Initial Schedule Heuristic Evaluation Results

Heuristic	Average Satisfaction	Average % of Preferences Received
Naive Baseline	.8450	.8449
Priority Lists	.8826	.8782
Ascending Number of Preferences	.2770	.4612

program to run it himself and come up with a schedule. He said that the program worked successfully. In his words, "Doing twice the work manually on such a short timeframe would not have been possible."

5 DISCUSSION

I discuss the results of the evaluations, as well as ethical considerations and future work in this research area.

5.1 Heuristic Results

From Table 1, we see the results of the different initial schedule heuristics. Interestingly, the Priority Lists heuristic is only slightly better than the Naive Baseline, while the Ascending Number of Preferences heuristic is much less optimal. This could be because the Number of Preferences heuristic prioritizes students with less preferences. Then when the algorithm looks at students with more preferences, the professors in their first few preferences may not have slots left for meetings. Since the cost function prioritizes first preferences over second preferences and so forth, and because these later students are more likely to only receive their later preferences, they are not contributing to the optimality of the schedule as much. For the Ascending Number of Preferences heuristic, students only received about 46% of their preferences on average. This supports the idea that this heuristic is less "balanced" among the students where students with less preferences are more likely to get all or most of their preferences and students with more preferences are less likely.

It is interesting that the Priority Lists heuristic is better than the Naive Baseline, but not by a very large amount. Perhaps the Naive Baseline is not such a naive heuristic after all. It could be that many heuristics will work decently well for creating an initial schedule as long as they are not unreasonably unbalanced like the Ascending Number of Preferences heuristic.

The Priority Lists heuristic prioritizes students who did not get their earlier preferences in the next round, so there is more "balance" between the students in this regard (i.e., in contrast to the Ascending Number of Preferences heuristic, it is less likely that some of the students will receive all or most of their preferences while some will not). This is supported by looking at the "Average % of Preferences Received" column in the table: in the Priority Lists heuristic, students received about 88% of their preferences on average in contrast to 85% with the Naive Baseline. These results support the idea that the Priority Lists heuristic is an improvement over the Naive Baseline approach.

For the local search heuristic results, we saw about a .3% average increase from the initial schedule. While this is a small number, it could possibly be explained by the initial schedule heuristic being a good heuristic for which there may not be a lot of improvement. Additionally, in the local search portion of the algorithm, the program tries over 10,000 swaps. I mainly chose this number so that the experiments would not run too long (recall that the experiment took almost 900 seconds). If I increased the amount of swaps the program is allowed to make while still keeping the computation time reasonable, perhaps the algorithm would find even larger improvements.

5.2 Ethics

In order to assist in my research, the Assistant Director gave me access to files from Visit Day 2019. The files included students' preferences, professors' availability, and the schedule he came up with. When handling these files, as they contained personal data, I anonymized the names (I used generic names such as Jane Doe and John Smith). The GitHub repository containing the code did include some of these test files, but as mentioned, they were anonymized, and I received permission from both my research advisor and the Assistant Director before pushing any of these files to the repository.

5.3 Future Work

It would be interesting to test this scheduling algorithm on later as well as other prior Visit Days. Future Visit Days could include even more students, so more research evaluating the efficacy of this algorithm on larger groups of professors and/or students would be useful.

For simplicity, I left out some aspects of scheduling that could make the program more complex. For instance, if a professor is very popular, one possibility could be allowing for a professor to meet with a few students at once instead of only having one-on-one meetings. Additionally, while I implemented the possibility of putting in tours (students can sign up for various tours of different parts of the Computer Science department on Visit Days, which would take up room in the schedule) I did not write the program to test the result of moving tours around on the schedule. For instance, say that there is an HCI Lab tour at 2:00. What if the tour were moved to 3:00 —would it allow for a more optimal schedule?

Another aspect of the scheduling to study could be the distribution of timeslots between the professors. Some professors or students could prefer to have their meetings spread out, while some could prefer to have them close together. This could be implemented in the cost function as penalties.

Finally, it would be intriguing to try to devise a brute force algorithm for this scheduling problem. I did not attempt to write a brute force algorithm given the problem's complexity, but if this were to be done in the future, even if done for a few schedule examples, it would help provide more of a baseline for how optimal the schedules produced for this algorithm are.

6 CONCLUSION

I presented an implementation to a solution for a real-life problem faced by the Department of Computer Science at the University of Maryland: matching prospective graduate students with faculty members on Visit Days. I elected to write a heuristics-based algorithm and evaluated the algorithm using both randomly-generated schedules and real-life scheduling data from Visit Days.

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. David Mount, and the Assistant Director, Tom Hurst, for assisting me in this research. Special thanks to Nelson Padua-Perez for coordinating this CMSC 499A research opportunity.

REFERENCES

[1] C. J. H McDiarmid. 1972. The solution of a timetabling problem. *IMA Journal of Applied Mathematics* 9, 1 (1972), 23–34.

[2] Pablo Moscato and Andrea Schaerf. 1998. Local search techniques for scheduling problems. *Notes of the tutorial given at the 13th European Conference on Artificial Intelligence, ECAI* (1998).

[3] Zahra Naji Azimi. 2005. Hybrid heuristics for Examination Timetabling problem. *Appl. Math. Comput.* 163, 2 (2005), 705–733. https://doi.org/10.1016/j.amc.2003.10.061

[4] Queen's University of Belfast. [n.d.]. *International Timetabling Competition - The Finalists.* http://www.cs.qub.ac.uk/itc2007/winner/finalorder.htm

[5] Andrea Schaerf. 1999. Local search techniques for large high school timetabling problems. *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans* 29, 4 (1999), 368–377.

[6] Andrea Schaerf. 1999. Tabu Search Techniques for Large High-School Timetabling Problems. *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on* 29 (08 1999), 368 – 377. https://doi.org/10.1109/3468.769755

[7] Dominic JA Welsh. 1971. Combinatorial problems in matroid theory. *Combinatorial mathematics and its applications* (1971), 291–306.