Room Scheduling via Graph Coloring

Lexie Gaines-Smith, Justyn Pham, Michael Millian, Connor Bottum

---

WORK SUMMARY

Input Sets: Our program requires three external spreadsheets of data: historical data describing each course and where it has been taught for the past 5 years; a list of courses being offered in the upcoming semester (we used the last semester, 1213S, from the historical spreadsheet as a testing sample), and a list of rooms on campus with relevant specifications. In addition, because our program implements a multi-phase process, it generates several intermediate spreadsheets which it then needs during the next phase.

All based on real-world data was given to us by Lillian Mosgofian and the Registrar's Office during Thanksgiving break (leaving very little time to adapt the code to the complex data set).

Input Files: proto-coursehistory.csv, workingCourseList.csv, proto-roomslist.csv
Output will be printed to: gen-LFsolutionslist.csv

Constraints: The constraints we explicitly check for are: no two courses can be held in the same room at the same time; room capacity must be able to hold the enrollment number of students (or the listed capacity, in case the course is capped and overenrolled); lab classes must be held in lab-specific classrooms. Other constraints such as technological needs are implicitly contained in the data we receive or can be specified by the user in our multi-phase process.

Evaluation/Objective Function: Our objective function is the number of courses in our "overflow" list, i.e. courses that are assigned colors greater than *room* (the total number of rooms available). We chose this objective function because we are most interested in a "complete" solution, meaning solutions that use at most *room* number of colors. It is possible that a complete solution is not an optimal solution, meaning a solution that uses the smallest number of rooms. For the problem at hand, a complete solution is satisfactory, but optimal solutions offer benefits which we discuss, such as identifying extra classrooms

which may be converted into offices or other spaces.

Approaches: When we were looking through potential algorithms to attack the room scheduling problem with, we quickly settled on a deterministic approach to graph coloring. Alternative algorithmic philosophies modeling the problem as genetic species or cooling system all appear as unnecessarily complicated - to interpret and to code - without any of the requisite improvement in solution quality or run-time they would need to demonstrate to be competitive with a deterministic approach. We decided to use Sequential Coloring Algorithms described by Frank Thomson Leighton in A For Large Scheduling Problems, first randomly ordered (RND) and then with a largest first ordering (LF). We did not have time to implement the recursive largest first (RLF) algorithm described in his paper.

In our first iteration for the solution to this problem, we tried to mock-up data for courses and rooms. This data involved listing preferred buildings for each department and drawing maps between buildings to represent how courses should overflow from one building to another. Under this set of constraints for the problem, we were able to find complete solutions for all the courses in our data set, taking into account the constraints of the course and room (size/capacity; type, e.g. seminar, lecture, lab; technological needs/capabilities) After a short time, we decided that this approach was unwise, because it represented a false set of constraints - a set that we made up rather than those the college actually needed. But courses have been assigned to rooms every year, so we decided to use this historical data as the baseline for solving our problem. This approach would have the added benefit that many of our constraints - namely technological constraints - would be rolled into the existing data; courses that needed access to certain technologies will already have been taught in the rooms that can accommodate them.

When we met with Lillian to discuss the constraints of the room scheduling problem, it became clear that while there are some constraints that our program must take into account (e.g. room capacity versus course size and room type versus course type, mostly relevant to labs) there were others that were difficult - if not impossible - to implement without polling each professor about in which rooms they could possibly teach each course every semester, such as instances in which a professor desires to teach in very specific rooms (or not teach in others) for a variety of personal reasons. In practice, the option of polling each professor is likely impossible because, according to Lillian, professors would fail

to correctly use any spreadsheet or poll that was sent out in some of the following ways: not providing all the requested information, providing unrequested information (e.g. paragraph descriptions in a yes/no field), or reformatting the layout of a spreadsheet. These incorrect uses would certainly make the collected data unparsable by a machine, and would require many hours of human labour to correctly re-format. One possible solution to this problem would be to build an online form using html select, radio, and checkbox elements, but we did not have access to the Amherst website to attempt this.

Without the ability to reliably poll the faculty about their room preferences, we have to do our best to create this data ourselves, but the very specific constraints of our problem described above means that there must be human control over the possible inputs. We designed our code to run in a three-phase process in order to allow maximal user interface. The first phase takes in historical data about the courses that have been taught at Amherst. We decided to go back as far as five years to encompass any courses that might be taught only once per four years and to get some variety in where courses have been taught. It is possible that using a larger set of historical data reaching further back might provide more varied results, but we concluded that reaching back more than five years would include data that was phased out for a reason.

Using this historical data, we do a bit of manipulation to match up the old two-digit course numbers with the newer three-digit course numberings and produce three sets of data: a list of all the rooms each department has used, a list of all the rooms each professor has used, and a list of all the rooms each course has used. We print each of these to a separate spreadsheet so that the user can read through it and edit out any errant data (e.g. removing rooms that no longer exist or adding new rooms). Also during the first phase we require the course list for the upcoming semester, so that we can print out a control spreadsheet. The control spreadsheet is a copy of this list of the upcoming semester's courses and the professors teaching them followed by three columns labeled "Courses; Professors; Departments". These three control columns should be filled with a "1" or left blank, representing the bitwise inclusion of the three lists we just generated into the union used during phase two. This control spreadsheet is designed to be edited by the user and streamlines certain decision processes including "Professor X always teaches in the same room. Courses with Professor X should only use list *p*." and "Course X is a new course, it should look at list *d*, which is the largest list.".

The purpose of phase two is to generate a preferred rooms list for each course. The preferred

rooms list is a list of rooms that a given course can possible go in. The input files for phase two are: the same course list for the upcoming semester used during phase one, and the four output files generated during phase one. An older approach utilized the formula $d \bigcap (c \bigcup p)$, where *d* is the historical list of rooms used by departments, *c* is the historical list of rooms used by the course, and *p* is the historical list of rooms used by the professor. In practice, this ended up being no different than $c \bigcup p$. The final approach reads through the course list for the upcoming semester and refers to the control spreadsheet and uses specified bits in the formula $d \bigcup (c \bigcup p)$ to decide which of the three sets *d*, *c*, and *p* to include in a union, which is the preferred rooms list. This preferred rooms list is then printed to a file next to the course name and list of professors, ending phase two. The preferred rooms spreadsheet allows the most user control over which rooms each course may go into, as the use may add or remove single classrooms for each course, tailoring any of the specific criteria mentioned previously which are difficult or impossible to represent computationally. The purpose of phase one, then, is to act as a large sift, reducing the amount of human work needed between phase two and phase three, but not eliminating human work entirely due to the specific nature of the criteria.

Phase three is where the most human time will be saved. The room scheduling problem is NP-Hard, which means that a human solving it with imprecise and ill-defined heuristics will have quite a hard time. In practice, it takes a long time on the order of days, according to Lillian. Our LF algorithm finds a 91 percent complete solutions composed of 26 skipped courses and 16 overflows in a matter of seconds. The graph for LF to use is set up in the following way. We load in the same list of courses being offered in the upcoming semester used during the first two phases, the list of known rooms on campus with specifications on each, and the list of preferred rooms generated during phase two after it is edited by the user. We construct two types of nodes: room nodes and course nodes. First, edges are drawn between each pair of rooms, because rooms cannot “occupy” each other. This step guarantees that a complete solution will use at most *room* number of colors to color all of the courses. Second, edges are drawn between all courses which have overlapping times, so that none of them may have the same color, i.e. none of them may use the same room. Third, edges are drawn from each course to every room not in it’s preferred rooms list. This inversion allows a course to share a color with any room in its preferred rooms list. During this last step, we check each room in the preferred rooms list

and also draw an edge from the course if the capacity of the room cannot accommodate the size of the class or if the type of the room and the course do not match in the case that the type of either is listed as “lab”. The final output of phase three is a spreadsheet which lists all of the courses, together with their assigned rooms.

Solutions: Our LF algorithm solved near completely leaving only 46 courses of 461 without room assignments. The majority of the courses left unassigned were lab courses which typically occur in rooms that are scheduled by the department and not the registrar. The reason our program was unable to schedule these courses is because some departments prefer not to declare the lab rooms they own (assumedly so that other courses cannot be scheduled there), so these lab rooms do not exist as nodes in our graph. If these lab rooms were declared and entered into our data set, we estimate this set of unscheduled courses would be handled successfully. For the remaining courses that could not be solved, we hypothesize based on the experiments and conclusions of Frank Thomson Leighton that if we had time to implement RLF, our solutions would be much closer to complete.

Additionally, there are six rooms that remain unassigned as they do not fit any of the unassigned courses needs. These six rooms offer several different potential uses. The empty rooms should allow for an RLF or LFI (largest first with interchange - another algorithm described by Leighton) algorithm to find complete solutions (especially LFI, based on how it works), or for easily identifiable shifts to be made by hand to correctly schedule the courses which our algorithm does not solve. Alternatively, in the case that our algorithm solves completely, the unused rooms represent unneeded rooms on campus which may be turned into offices or other spaces.

Our solution is returned in the form of a spreadsheet with the course name in the first column and the room to which it has been assigned in the second.

Runtime: Phase 1 runs for ~1000ms, Phase 2 runs for ~700ms, Phase 3 runs for ~900ms

Conclusions: Due to time constraints, we were unable to implement the RLF algorithm, which we predicted would yield the best results. Our RND algorithm finds unacceptable solutions (Overflow to

244, or 41 percent complete), while our LF algorithm finds reasonable solutions in a very short time. Our experiments were not able to find complete solutions using LF, in part because of a set of errant data where lab courses needed to be scheduled into lab rooms which were not reported to the registrar so we did not have as room nodes in our graph, leaving the preferred rooms list for these courses empty. We also made no use of the control spreadsheet during phase one or the preferred rooms list during phase two because we could not determine how to simulate this stage of the process. With the proper set of data about lab rooms on campus, the overflow list would be 16. Given this small number of overflow courses, we predict that with proper use of the use interface we designed, LF will be able to find complete results.

We also did not have time to implement the command line interface for the non-cs user. The interface for our program would be very straightforward. The different phases are controlled by a switch statement, so main() simply needs to take an integer parameter from the user specifying which phase they wish to run. We feel that it is not ideal for the user to explicitly pass the spreadsheets that they wish to use when they run the program because the names of the files would be too long. Instead, we would provide documentation detailing which files need to exist in the same folder as the java files and what they should be called. Then, on execution of our code, we would need to throw informative exceptions if any of the needed files for the desired phase were not found.

We would also have liked to write more general exceptions regarding badly formatted spreadsheets and other predictable errors with informative messages to provide the most straightforward experience for the user.

How To Run Our Code: Open Driver.java and change the int phase to the desired phase. Make sure all the required .csv files for that phase exist in the same folder. Run java Driver.

Alternatively, if the user interface is built, move the required .csv files into the folder with the java files and run java Driver <desired phase number>.

Contributions: The coding method implemented by our team was very group-focused. We found times where all members could meet together in lab and have input to the decisions being made. During planning stages, we would discuss and diagram ideas and solutions, with all members offering ideas,

critiques, and cautions. During coding stages, often we would all huddle around one machine and guide the current typist in making decisions on specific implementation. This method was chosen when changes to one central file or method were critical to moving forward (i.e. debugging, of which there was a lot), and it was easiest to work with everyone's contributions simultaneously rather than to have some people write code which would likely be broken or incompatible with the critical task at hand. In other cases, when it was possible, e.g. when the work to be done was relatively small and self-contained or modular enough to be worked on separately, divided work into teams of two or even worked on individual tasks, but being in the same room meant that we could quickly address potential compatibility concerns and get second opinions on issues such as data structure choice. By the end of the project, after many rounds of writing, rewriting, deleting, and debugging, it is unlikely that there is a portion of the code that was not explicitly handled and approved by each member.