Club Sandwich

Club Sandwich is a tool for automatically scheduling clubs.

Last updated `2022-03-12`.

Overview

This tool tries to solve two headaches in scheduling:

1. Figuring out which clubs should be on which days.
2. Figuring out which students should be in which clubs.

This tool does just that. It aims to give every student as many of their choices as possible by strategically scheduling the clubs to avoid conflicts, and then giving students turns to get into the clubs in a way that prioritizes equal chances of getting their choices. It creates a report and a number of CSV "views" — you can see the schedule in terms of

- students, showing which clubs they're in
- clubs, showing which students are in them
- teachers, showing which clubs they teach
- days, showing which clubs are run in parallel

Additional features

There are many additional features that turned out to be necessary in our use case, including:

- Test different arragements of clubs to days, and for each arrangement, test different orders of giving students their choices.
- Limit the size and number of groups allowed for each club.
- Determine which clubs have enough interest to run and filter them out, promoting a student's remaining choices to fill the gap.
- Automatically add more groups to a club when it's overpopulated and balance the members across groups.
- Save reports of the votes for each club, before and after filtering eligible students.
- Merge multiple clubs into one, or split one club into multiple based on grade, gender, or hand-picked names.
- Pre-select students for a given club (such as Choir). Choose whether this is the exhaustive list or if it's still open to others.
- Enforce whitelists, blacklists, and mutual exclusions between clubs.
- Tracking teachers and avoiding over-scheduling them on the same day.
- Offer multiple scheduling options to be compared by humans.
- Show statistics on each option (students receiving their choices, club membership, grade and gender balancing, and more).
- Validate all options to ensure there are no conflicts.

The algorithm

In broad strokes, the scheduling algorithm goes like this:

Survey data

Process data on club options and student choices.

Additional student data

Align students with the master list from the school to eliminate inconsistencies in the survey. Also add any students who did not fill in the survey (they are just placed in Study Halls).

Calculate statistics on the school population in order to know the expected proportions of grades and genders across clubs.

Additional club data

Process additional data: splits, merges, whitelists, blacklists, exclusions, renamings, preselections.

Schedule clubs

Create various arrangements of clubs across the available days.

For any club that is forced to take a certain day or days, place it on those days. For example, Choir must run on all three days.

For the remaining clubs, calculate their "repulsion factor".

▶ Click to read about repulsion

Order the clubs so that those with the highest repulsion factors come first.

For each club, place it on the day with the lowest repulsion factor that will therefore create the fewest conflicts.

If there are any ties (or near ties), branch into two possibilities.

Schedule students

Shuffle the students into a random order.

For each student, try to place them in their first choice. If they can't get in, try to place them in their second choice. Continue until they get into a club. Then, move to the next student.

When everyone has gotten one choice, go for a second round. Go in order such that students who got the fewest and least preferred choices go first on the next pass.

After everyone's 5 choices have been considered, set any remaining slots to Study Halls.

Try multiple possibilities

Try this again *n* times (where *n* is supplied by the user), with a different initial random ordering of students each time. Also try *m* branches of the initial placement of clubs on days if there were ties, as noted when discussing repulsion above.

For each option, a few statistics are calculated:

- How many students got into their 1st choice? 2nd? etc.
- How many students had to end up in Study Halls for lack of being able to get their other choices?
- How wide is the gap between the most populated clubs and the least populated clubs?
- How balanced are the clubs in terms of gender and grade composition?

A score is then derived from these statistics. The top 1 or more options (as chosen by the user) are retained. Hence, the best options remain in the running until they are replaced by better ones.

Output report

After all possibilities have been tried, the top ones are saved in five views, as noted above, and are also stored in a format that can be reopened and manually tweaked and have all its views re-output later.

Input needed

In order to make this all run, some input is needed. Detailed formats are specified in `input/_input_specifications.csv`, but here's the overview:

- A student survey file. This lists students along with their top 5 choices.
- A master student file. This lists students' full names along with their grade, gender, attendance status, and any days they are unavailable.
    - A tool called `prepare_linkups.py` is provided compares the master student file and the student survey file and outputs a linkups file. Since it may be impossible to link up all students (e.g. they gave a different name on the survey), it also creates a consumable file that must be examined by hand to reconcile any differences.
- A clubs file. This lists clubs, along with their teacher; which days they can run; lower and upper limits on membership; how many groups can or must be run; grade and gender restrictions; and whether the club has open or closed membership.

In addition, these files can optionally be supplied:

- A merges file. This lists clubs to merge with other clubs.
- A splits file. This lists clubs that should be split into multiple branches. It can list grade, gender, or hand-picked names to go into each branch, as well as replacing any or all scheduling information from the original branch (e.g. it was to be one group running two days, but we have determined it should be two groups running on two separate days).
- A renames file that provides the ability to rename clubs on output.
- An preselects file. This lists students who are preselected for clubs, along with a number of groups they should be in, and any days they have to attend or cannot attend (if any are known in advance).
- An exclusions file that lists clubs that are mutually exclusive.
- A whitelists file that lists students who are whitelisted for individual clubs. A whitelist differs from being preselected in that preselected students are automatically placed into the club, while whitelisted students would still have to have chosen the club to be placed in it.
- A blacklists file that lists students who are blacklisted for individual clubs.

Merges, splits, and nice name files in particular are used for adjusting the offerings after the survey results are known.

Workflow

Given the above, here are the steps you would ideally carry out to use the tool:

1. Determine which clubs will be run. Additional data that can be omitted but may be helpful:
    - The teacher, if already decided
    - Days it can or can't run, if already decided
    - Upper and lower limit for number of students
    - How many groups you plan to run (or leave it undecided)
    - Any grade or gender filters
2. Offer a student survey in which students pick their top 5 choices.
3. Run the student linkup script on a master student list, then go through the consumable file to manually align any students whose names couldn't found. Add status and unavailable days (for internships, etc.) for students this is known of.
4. (Optionally, but for better results) Run in process votes only mode and review the filtered vote data. Decide which clubs will not be run based on lack of interest. Decide other scheduling info for the others: minimum and maximum groups or set a specific number, how many days each group should meet, and whether you need to limit how many of the group can run per day (e.g. they need a specific room or equipment).
5. Run the main mode and review the different views of the outcome, in particular the club and student views for data entry or importing.

Future plans

Technical design fixes

- Splits are currently handled as separate clubs. It would solve some problems if they were instead handled within an individual club, e.g. as named instances, or as 'branches'. In particular this would allow a choice for a club to be treated as one choice, with the club handling the split, rather than the school having to determine which split a student goes into (and, if they are eligible for more than one branch, either having to choose randomly or use up more than one choice). It would also unify the problem of split names filling the same role as instance keys (visible in 'Cosmetology 1 A' and 'Cosmetology 2 A', for instance).
- Student names are currently case-insensitive when preparing linkups, but case-sensitive during processing, resulting in potential errors.
- Numerous `TODO` items in the codebase.

New and improved features

- Add a teachers.csv input file with the teacher's name, maximum days they're available, and specific days they're available.
- Right now, all merges happen before all splits. It would be good to have a protocol whereby they could be carried out in a predefined sequence. For example, you might want to split a club into A and B; merge B with C; and then split C into D and E. This is currently impossible.
- Clubs that have too few votes to run are currently eliminated before distribution. A powerful feature would be to identify clubs that have too few *after* distribution, and reassign those students into other clubs that are themselves low but closer to the threshold of being able to run. In other words, cannibalize the least popular clubs so that other marginal ones can run.
- Splitting after the fact, i.e. when intances are balanced. This came into play when we decided to turn the Study Halls into grade-based groups after generation.
- A better, more generalized swap tool. (I have a highly manual-effort one that I used for the above Study Hall sorting.)
- Update the text interface to allow choosing the constants for create_schedule (n worlds, n student configurations, n best to keep).
- Pluralize teachers per club (including n_teachers_needed to determine whether all need to be engaged or others are free!), and, in preparation for course scheduling, number of clubs per day for a teacher, rather than just whether a day is used.

Course scheduling

One idea that came up during development was whether the program could be used for general course scheduling in the future.

I feel that it's possible to reuse much of the program, and in particular the placement of clubs in periods to avoid overlap would be a valuable problem to automate. There would certainly be some technical hurdles involving a significant rewrite or adaptation of parts. Also, the concepts are not quite the same and an general solution for both courses and clubs might not be possible.

Some of the changes would include:

- The day must be expanded into 4 periods. All 4 must be filled (note that spare and prep would be treated as courses; each teacher must have one prep).
- On the other hand, days are simplified (all courses run every day, and in exactly one period; there is no pluralization of timeslots).
- There is no catch-all course parallel to the catch-all club of Study Hall.
- Days have to be reindexed to account for all 5, and choices have to be made variable instead of fixed at 5.
- Students have not only choices but also requirements for courses; this means preselections should be at least partly algorithmized rather than just on a student-by-student basis.
- To accomplish the above, we would probably need a record of which courses a student has taken and which ones they need.
- Rather than all students being in the same category, there are Grade 9 - 12 students.
- Courses do need to be split into course codes, e.g. TGG3M/TGG4M are two codes but (for the purposes of scheduling) one course.
- Some courses can accommodate multiple grades, but not all. There needs to be a way of overcoming the restriction for individual students, too.
- Teachers should have a set of courses they are eligible to teach (perhaps even split into 'must', 'willing', and 'last resort'). Mirror to make it clubwise too.
- Some teachers may need preps at a particular time, or to overlap with other teachers' preps.
- Some courses (and therefore some teachers) need to be mapped to specific rooms.

No doubt there are more factors taken into consideration by those who make the schedule each year.

In addition, the human factor of ensuring that class compositions make sense (whether by grade, friend cohorts, etc.) is much more central than with clubs.

I remain on the fence as to whether the effort required to adapt this tool for course scheduling would result in better schedules than are currently crafted with intention. I would need to understand the decision process and logistical process of course scheduling in more depth before knowing how much we can meaningfully represent in code.

Development notes

When first approached to create an algorithm to put students into clubs, I assumed it would be a couple hours' work. I was only envisioning one of the last steps of the algorithm, and simplified at that: go through the students, place them in their first choice; reverse order, place them in their second choice; and so on. Then I had the idea of also randomizing club placements to days and trying each option, which I also thought would take a couple hours at most.

I consulted with a couple of Grade 12 students with strong math skills to help with some preliminary calculations. There were 3 days, about 40 clubs (from the early interest survey), and about 420 students. The question was, how many different ways are there to place 40 clubs on 3 days? After much math and some trial and error, we figured out roughly that it would be about C(14, 40) x C(13, 26) — the last multiplicand being C(13, 13), which is 1. (Variables could of course be substituted for these numbers.) However, some configurations are redundant if we consider that, for the purpose of which clubs overlap, we don't care which permutation of the actual days each "block" of clubs ends up on. Trial and error revealed that the total should be divided by a factor of 2 if the number of clubs is divisible by 3, otherwise by a factor of 6.

For 40 clubs, this meant about 150 quintillion options for randomization. Even for 30 clubs, this meant 925 billion options. I naïvely began optimizing. One major issue was the total being 2 or 6 times higher than necessary unless we removed duplicates, which meant storing the options in RAM, which is clearly impossible. I did a number of heuristic analyses and finally derived a formula to predict duplicate options (as a function of the index of iteration when going through combinations in a predictable order) and was able to eliminate memory usage. Now it was only a question of speed of generation. After much fine-tuning of the Python, I used a tool to compile it to C and was able to generate 66 million options per minute. This still meant about 230 hours of processing for 30 clubs, though, so it was not plausible. Instead, I converted the exhaustive generation to a random generation and hoped that doing, say, 1 million iterations would yield a good one as scores floated to the top. By the end this dream of generating all options was even more clearly futile: the number of clubs went up (to about 60), and the actual processing of the distribution for each option meant that only about 3,600 could be tested per minute.

I worked on other aspects for a while and got a working prototype together which served as a proof of concept for the tool and I was given the go-ahead to develop it

after test data (using the early interest survey) suggested we could hit 80% or more students getting their first choice. We worked out the many factors we would need to consider for club distribution.

An idea occurred to me. What if we could place clubs on days strategically instead of randomly? What would be a good strategy? I reflected that each of student must be in one club each day. Hence, the days should be balanced in terms of the number of "votes" for the clubs on them. This appears to be a variation of the knapsack problem (also cf. here) and leads to an algorithm for choosing day a given club should be on: Sort clubs to place those with the most votes first. For each club, choose the day with the lowest total number of votes so far, and place it on that day. Let ties branch and be yielded in random order. I implemented this and the results instantly became much better and more stable.

Not long after that, I thought of a better algorithm. After thinking about strategies for placing clubs some more, one of those "so obvious it's invisible" insights appeared: Two clubs picked by the same student should not be on the same day. If they were, the student could not be in both. What if we went through all 420 students and, for each club, tallied how many times a student picked it alongside each other club? The total score could be called its "repulsion factor", because those clubs would "repel" each other onto different days. The factor could also be weighted by the priority of the pick (e.g. 1st vs. 5th choice). Then, use repulsions exactly as vote totals had been used: Sort clubs by total repulsion with all other clubs, then place each on the day that repels it least. Ties still branched. I implemented this and the results improved still more. I was now satisfied with club distribution.

The next challenge was to accommodate the additional requirements and feature requests that were coming in. This is when I really began to realize how messy scheduling is. Up till now, the process had been fairly mechanical, but now many different scenarios arose, including:

- Teachers who wanted preselected students.
- Teachers who wanted to only accept certain students, but not to force them in if they didn't want to be in the club.
- Teachers who wanted students to be in the club some days but not others so that they could still choose other clubs.
- Students who were preselected for multiple clubs that might end up running on the same day; that is, preselected students that forced clubs to take separate days.
- Students who were in internship and could not attend clubs some days, but who had still filled in the survey with all five choices.
- Students who had signed up for clubs they were ineligible for.
- Clubs that had too few students to run and could donate their students to other marginal clubs to make them viable.
- Clubs that had so many students we could run more than one group.
- Clubs that needed to be merged into other ones for lack of interest in two similar offerings.
- Clubs that needed to be split into junior and senior divisions.
- Clubs that needed to be split into two groups based on another student who was leading them, and who wanted to select some of their members, and randomize others.
- The need for students who weren't in enough clubs to be placed in Study Halls and for those Study Halls to be balanced in number (and, later, to be split by grade).

As I handled these, mostly through extra parameters in the input files, I made a major overhaul to the modelling of clubs, dividing them into *clubs* in the abstract and *instances* in the sense of a specific group of students in that club meeting on a specific set of days. This required heavy refactoring but gave me much more flexibility. By the end, however, I did end up pushing this model to its limit (which in development means shoehorning features in through less-than-ideal means, much as you might fix part of a wall in your house that ultimately will have to be knocked down).

Towards the end, I had another idea. I had de-randomized the placement of clubs to days; could I also de-randomize the order in which students were assigned to clubs? They were being randomly shuffled on each attempt, and the number of shufflings of 420 students is easy to calculate: 420 factorial = 1.1 e 921. So even a million iterations was a drop in the ocean. Was there a strategic way to decide which students should get first pick?

I had a few ideas, and one was a factor I termed "reactivity". I observed that a student who picks an unpopular club can be placed in it at any time, because it will never be full; whereas a student who picks a popular club for their first choice has a good chance of not getting in, and so should get an early shot at their second choice. I implemented this, but didn't have the time to refine it, and I was somewhat unsure about this insight in the first place. In any case, the results were no better than random.

However, a related improvement did make it in. Up till now, after each round of giving students a choice, I "snaked" the list, so that those who had picked last would pick first next round. I altered this step of the algorithm such that after each round, the students would be sorted by who had gotten the fewest picks, and first pick on the next round be given to these students. While the initial order would still be random, the algorithm would adjust to target the least successful students on each pass. The scores improved again and became even more stable (suggesting that the initial randomness is compensated for to some degree).

The longer the algorithm runs, the higher the scores it generates, so when it came time to run to get our schedule, I left it running overnight to target 1 million variations. The resulting schedule gave 94% of students their first choice, 73% their second choice, and 70% their third choice; 7% of students were in one Study Hall they didn't choose, 1% were in two, and 2% were in three. This is a low enough number that manual swapping should be feasible.

The codebase as it stands is about **1,900 lines of code** and **1,200 lines of documentation**. If we include discarded components (such as the random club assignments described above), it adds some 1,200 lines of code and 100 lines of documentation. In addition, this readme is about 4,000 words.

I did not track my hours but they can be estimated by counting the time between git commits (i.e., check-ins of code into the version control system). This might overshoot the reality by counting times I was up from my computer during a work session, so I've limited them to commits within 1 hour of each other. Also, it might undershoot the reality by not counting the time between sitting down and the first commit of a work session. The total time based on this calculation is **45 hours**.

Credits

Created by Luke Sawczak for TDChristian, December 2021 – March 2022.