Optimizing meetings in duplicate bridge schedules

Thilo Kielmann, April 16, 2021 (updated after meeting)

Schedules

During a duplicate bridge tournament, participating pairs each play against multiple other pairs. A tournament is organized in multiple rounds. In each round, a pair plays against an opposing pair. Which pair is playing against ("meeting") which other pair, and in which round, is determined by a so-called *schedule* (also referred to as a *movement*). A schedule also determines other things like at which table the meeting happens, and which chairs the pairs sit on (north-south vs. east-west), and also which hands they play. But these details are not relevant for this study. Within a schedule, pairs are identified by a number, ranging from 1 to the number of pairs in total.

Typical schedules are for 8 up to 18 (or 20) pairs and tournaments normally have 6, 7, or 8 rounds. Finding good schedules is non-trivial such that they provide equal chances to all participants. Here, we simply take them as given. Schedules come in families with common structures and properties, but different numbers of pairs. Usually, one uses such a family for a tournament and selects the proper schedule for the number of pairs taking part.

Schedules are always made for even numbers of pairs. Should there be an odd number of pairs, each round one of the pairs gets a so-called waiting table, which is just the table where they would play if there would be another pair. Not surprisingly, people hate having to wait instead of play.

Schedules are commonly stored in a very simple (ASCII) file format, described here:
http://www.pjms.nl/movementfile_standard.html

Sections

Because schedules are limited to fixed and small numbers of pairs (and because of limitations with the pre-dealt hands), tournaments are usually played in multiple *sections*. This way, larger groups like 50 pairs can be split into smaller sections for which schedules exist.

Another effect of using sections is to group pairs into pools of roughly equally strong playing abilities. Sections within a tournament are usually labelled with letters, like A, B, C, … with the A section having the strongest players and so forth.

Competitions (playing seasons)

Schedules take care that, during a tournament, a pair meets any other pair at most once. (There are exceptions, but we do not care about them for this study.) Tournaments, however, are not played in isolation. Instead, a bridge club organizes weekly tournaments that together form a competition. At the end, the overall winners can be determined as the ones who consistently scored best. We can consider all tournaments during a season (typically september until some day in summer) as a long competition.

During a competition, it is important to balance the number of meetings that each pair has with the possible opponents. Ideally, a pair should play equally often against each opponent pair. Also, the total number of waiting tables should be fairly distributed among all pairs.

The tournament director can influence the balance of meetings and waiting tables by assigning pair numbers for a tournament differently. Remember that the schedule determines the opponents against who a pair is playing, and hence the opponents against who they are not playing. For a given (section of a) tournament, the director can use permutations of pair numbers for which a schedule makes certain pairs meet each other or not.

Finding the best permutation is a nice optimisation problem. It needs to be solved in real time (within tens of seconds) as players are waiting to get started while the director is setting up the tournament. This calls for heuristics solutions as simply trying all permutations is not feasible. The total number of permutations for a section is *n!* with *n* being the number of pairs in the section. Typical values for *n* are in between 12 and 16.

As sections represent groups of players of the same playing skills, there also is a mechanism of promotion and degradation for pairs who consistently play better or worse than the other pairs in their section. In modern club bridge, the strength of players in a club is tracked by so-called numerical *rankings*, where each tournament's result is taken into account. After each tournament, the rankings of all (participating) players are updated, determining a sequence from the strongest to the weakest pair. These rankings are used to split the pairs playing the next tournament into multiple sections. In practice, this means that after each tournament, some pairs may promote or degrade between sections.

Defining the optimization problem

While bridge is played by pairs, in modern duplicate bridge, the composition of the pairs and also which pairs play at a given weekly tournament changes during a playing season. People go on vacation, they have other duties, or become ill. This typically leads to the situation that the set of participants that actually plays can only be determined last minute (literally) before the start of a tournament. Many/most pairs will show up, but others will not play. Also, some pairs will be composed of two ad-hoc partners in cases in which one player cannot make it but the other would still like to play. This setting requires that meetings from past tournaments are recorded on a per-player basis (rather than per pair). Also the rankings are tracked on a per-player basis.

The situation:

We are dealing with the situation in which the tournament director is when setting up the tournament: There is a list of pairs who have signed up to play. And there is the history of meetings from the past. Goal: set up the tournament such that all pairs are distributed among the sections, while getting pair numbers by which their meetings will be optimized.

Given:

- Players represented by their numerical id (1 or higher), each having a ranking value indicating their playing strength.
- A history from a running season in which for each player in the club it is recorded how often he or she has met (played against) each other player. Also, how often he or she has had a waiting table so far.
- A registration file, consisting of pairs of player ids. These are the player combinations that will play the tournament for which we aim to distribute pairs over sections, and optimize their meetings by different assignments of pair numbers within their sections.
- A family of schedule files, indicating which pair in a section is meeting which other ones.

Preliminaries:

- For a given pair of history and registration files, distribute the registered pairs over the given number of sections, based on their ranking values. For details, see below.
- For each section, compute a matrix of previous meetings among the pairs. Meetings are counted per player. So for pairs (A,B) and (C,D), we add up the number of meetings among A-C, A-D, B-C, and B-D to compute the total number of previous meetings among them.
- For each section, record the number of previous waiting tables of each pair, also adding up the waiting tables of both players. This will be a vector of values.

Optimisation goal:

For each section find a permutation of pair numbers for the pairs in this section such that the value of the fitness function (see below) will be minimal among the known (or tried out) permutations.

Evaluating a permutation:

Compute a matrix of meetings and a vector of waiting tables:

- For the meetings, start from a copy of the “previous meetings matrix” and use the schedule file for the respective number of pairs (and rounds) such that for all meetings that happen in the new tournament the number of meetings among two pairs will be incremented by 4. (two players each meeting two other players)
- For the waiting tables, start with a copy of the vector of “previous waiting tables”. For each pair having a waiting table increment their value by 2. (both players have a waiting table)

Compute the fitness function for the permutation:

- The fitness is the sum of the meeting factor and the waiting factor.
- The meeting factor is the sum over all entries in the matrix, where each entry is taken to the third power.
- The waiting factor is the sum over all entries in the vector, where each entry is taken to the sixth power.

Needless to say, the waiting factor only applies when a section has an odd number of pairs. In this case, a pair has a waiting table if, in a particular round, it should meet the pair with the number that is absent. For the sake of finding a good permutation, also the pair number of the absent pair can and should be chosen freely. (It is not correct to assume the absent pair always has the highest possible number.)

The fitness function aims at penalizing excessive repetitions. The “best” permutations try to avoid that players have to play “against the same folks” over and over again. The function does not actively penalize when two pairs meet (hardly) at all. Also, taking the waiting tables to the sith power, repeated waiting tables get a stronger penalty, compared to repeated meetings with other players.

Overall fitness

The fitness function is defined per section. For a given tournament, all sections need to be optimized, but it is not obvious how to integrate the results per section into an overall value. Anyone a suggestion? The meeting values across the sections are not normalized to each other. Is there a way we can normalize them (post mortem) such that we can add them up properly? There is a way to compute a (hypothetical) best case for the fitness function to which we can relate the solutions we find. Should we include this here?

File formats

In the Google drive, we have the following types of files.

1. Schedules. We have two schedule families, both with schedules from 8 up to 18 pairs (per section) All schedule files are in the format linked to above.
    a. 6 rounds (multiplex)
    b. 8 rounds (teamplus)
2. Data (two separate sets from 2020 and 2021), consisting of a meeting history and two related registration files (pairs…) with player ids belonging to the history file

The registration files have a simple format. Each line consists of two positive integer numbers, separated by a comma. The numbers represent player ids from the history file.

The history files also have a line-by-line format. Each line consists of the following, separated by commas:

- Player id (integer, 1 or larger). Beware: there are gaps in the numbers!
- Player ranking (positive float), representing the current playing strength relative to the other players
- Meeting history with all players (array of non-negative integers, separated by commas and framed by square brackets
    - The first entry (index 0) is the number of times the player had a waiting table

    - All other entries (index 1 .. max) is the number of times the player has played against ("met") the players with the player id corresponding to the index in the array.
    - Many entries are 0. That is because sections are created by grouping pairs of equal strength (based on the player rankings).

Generating test cases from the data

We have two data sets (one from 2020 and one from 2021). For each data set, we have two registration files, one with an even number and one with an odd number of pairs. For both cases (odd and even) two cases should be investigated, namely a tournament with 6 rounds (using multiplex schedules) and one with 8 rounds (using teamplus schedules).

The 2020 data is to be used for tournaments with 4 sections.
The 2021 data is to be used for tournaments with 3 sections.

Building the sections of a tournament:

Using the registration file, you first compute the rankings of all pairs, which are the sums of the rankings (in the history file) of the two players. Sort the pairs by their rankings.

Determine the numbers of pairs per section. Sections should be roughly equal in size. If the total number of pairs is even, each section must have an even number of pairs. If the number of pairs cannot be divided by the number of sections, equally distribute as many pairs as possible. Then add two pairs (each), starting from the strongest section A (with the highest rankings) towards the weaker sections, until all pairs have been distributed.
Example 46 pairs, 3 sections: first shot is 14-14-14 pairs. Then make it 16-16-14 pairs.

With an overall odd number of pairs, one section must get an odd number. You choose the section in which the players so far had the lowest average number of waiting tables.

The number of pairs per section must not differ more than two from the largest to the smallest section. If adding the "odd" pair to a section that has more than other sections, you may be forced to redistribute pairs.

Example 47 pairs, 3 sections. You begin with 16-16-14 pairs (see example with 46 pairs). If you find that the players in the lowest section C had the fewest waiting tables in the past, you are lucky. You simply add the additional pair to the lowest section: 16-16-15. But, if you find, for example, that the waiting table goes to section A, you would end up with 17-16-14, so you must redistribute to 15-18-14, and then finally to 15-16-16.

Once the numbers of pairs per section have been determined, you assign the pairs based on their rankings. The strongest pairs go to the highest section A, the next ones to B etc.

All sections must have between 8 and 18 pairs, because otherwise we have no schedules available. If you end up with fewer or more pairs for a section, something is wrong. The data should not require this. If a section has an odd number of pairs, we use the schedule for the next higher even number.

There may be corner cases like pairs with equal ranking or subtle differences in the number of previous waiting tables. In such cases, you are free to implement your distribution one way or the other. However, make sure that you are always doing it deterministically, such that all attempts at optimisation use the same distributions.