A Tiling Algorithm for High School Timetabling

Jeffrey H. Kingston

School of Information Technologies
The University of Sydney 2006 Australia
jeff@it.usyd.edu.au

Abstract. This paper presents a tiling algorithm for high school timetabling. The meetings are grouped into small, regular clusters called tiles, each of which is thereafter treated as a unit. Experiments with three actual instances show that tiling, coupled with an alternating path algorithm for assigning resources to meetings after times are fixed, produces good, comprehensible timetables in about ten seconds.

1 Introduction

As a recent survey makes clear [1], the problem of automatically constructing timetables for high schools remains far from solved. This paper offers a new approach based on grouping meetings together into small clusters called *tiles*. Although there are some drawbacks in doing this, there are significant advantages: the resulting timetable is comprehensible to the timetable planner; assignment of teachers to classes is simplified; and run times are reduced to about ten seconds, freeing the timetable planner to explore alternative scenarios quickly.

After a description of the high school timetabling problem (Section 2), this paper offers an overview of tiles and the tiling algorithm (Section 3). Sections describing the phases of the algorithm follow. Results are presented for three instances taken from a high school in Sydney, Australia (Section 8).

2 Specification

High school timetabling problems vary from place to place. Some high schools are timetabled like universities, for example. The problem described here is the one occurring in Australian high schools.

In high school timetabling problems, the meetings are timetabled on a weekly or fortnightly cycle. Time is partitioned into *periods* of equal length. One common pattern is a week of 40 periods, each 40 minutes in length, spread over five days. Adjacent periods may be adjacent in time, or separated by a meal break or by the end of a day.

Unlike university timetabling, where each student follows an individual timetable, high school students are grouped together into *student groups* (often called *classes*, but that word can also mean the meetings they attend, and is so used here). The members of one student group (typically 30 students) follow the same timetable and may be treated as a unit. Four or five student groups, containing all the students of a certain age group, make one *form* (also called a *year*). Australian high schools have six forms, called Year 7, Year 8, and so on to Year 12.

A typical high school has 50 or more teachers. They are partitioned into *faculties*: subject areas, such as English, History, Mathematics, and so on. Some teachers have qualifications spanning more than one faculty, and some meetings (such as Sport) are taught by teachers in several faculties. Within faculties the teachers have further specialties, and our model allows any number of *capabilities* (such as Drama, or Senior History) to be defined and granted to arbitrary subsets of the teachers. A teacher may have many capabilities, not just one. Teachers have *quotas* specifying their workload. A common example would be a teacher whose quota is 30 periods per week (out of the maximum of 40), with a preference for at most 7 periods on any one day (out of the maximum 8). Head teachers or teachers with other duties have smaller quotas, and there are part-time teachers who are available only on nominated days.

Rooms are like teachers with unlimited quotas. Although most rooms are ordinary classrooms, there are specialised rooms such as Science laboratories (possibly also usable as ordinary classrooms) and Computer laboratories. Again, our model allows any number of these capabilities to be defined and granted to arbitrary subsets of the rooms. Rooms could also be unavailable at certain times, for example for maintenance, although there are no cases of this in our data.

Student groups, teachers, and rooms (collectively, *resources*) play fundamentally the same role in timetabling: each must be assigned to meetings, avoiding clashes. However, there are differences in detail. Student group resources attend something at every period of the week, and are always preassigned to meetings – there is never a request to *choose* a student group, as there is with teachers and rooms. Teachers differ from rooms in that it is important for the same teacher to attend all of the periods allocated to each class meeting. One does not want Ms. Smith to take some English class for three of its six periods, and Mr. Brown to take it for the other three. That would be a *split assignment*, and it is permissible but undesirable.

A *meeting* is an entity in which a set of resources meet together at a set of times. The times and resources may be *preassigned*, meaning fixed in advance to particular values, or they may be left open to the solver to choose. Meetings may request any number of times, and may request blocks of adjacent times not separated by breaks (these are called *double periods*, *triple periods*, etc.). It is preferred for these times to be spread evenly through the week, and that undesirable times (such as the last period on any day, when the students are tired and restless) should not be concentrated in one meeting. Student group resources are always preassigned. In our data, teachers and rooms are usually *not* preassigned. Instead, meetings request resources with given capabilities: a History teacher, a Science laboratory, or whatever.

In traditional class-teacher timetabling [6], each meeting contains one teacher and one student group, but our meetings typically request more resources than this. For example, in the junior years the students in each form may be grouped by ability for Mathematics, meaning that the Mathematics classes of that form must run simultaneously, leading to one large meeting requesting four or five student groups, Mathematics teachers, and rooms. In the higher years there are *electives*: sets of meetings planned to run simultaneously so that students can choose one. There may be seven or eight teachers, with varying capabilities, plus rooms in such meetings.

Occasionally there are *composite classes*, in which students from different forms study a specialised subject together. Although the students follow different curricula, the common subject matter makes such a class feasible. Composite classes are created when the school wants to offer some subject as an elective, but there are too few interested students in any one form to justify it. Their effect on timetabling is to cause two electives from different forms (those containing the specialised option) to be merged.

Our data also contain several kinds of staff meetings, requesting various preassigned subsets of the teachers, but no student groups or rooms. These meetings are equivalent to free time for the purposes of calculating teachers' daily and weekly quotas, but they can be involved in clashes like other meetings.

The objective is to assign times, teachers and rooms with the desired capabilities to the slots, avoiding clashes and not overloading any teachers. These two requirements are hard constraints and they dominate the problem. If necessary, a resource assignment may be split between two teachers as described above, and as a last resort the smaller of the two fragments may be occupied by a teacher not qualified for the requested capability. Meetings must receive the number of times they request, but the block structure and spread through the week of these times are soft constraints.

3 Overview

In this section we define tiles and present an overview of our algorithm. Following sections then explain its phases in more detail.

Consider a typical meeting, such as the English class of student group 7C in the *bghs98* instance. This meeting requests 6 times including two double periods, student group resource 7C, one English teacher, and one ordinary classroom. We can think of this meeting as a 3×6 rectangle:

7C
1 *EnglishTeacher*
1 *OrdinaryClassroom*

Its *width* is the number of times requested, although sometimes we give a sequence of numbers for the width, defining the required block structure. For example, width 2 2 1 1 requests six times including exactly two double periods. Its *height* is the number of resources required, although again we often give more detail – a list of the resources required – and call that the height.

Several meetings may be grouped together into a larger entity of a specific width and height, which we call a *tile*. For example, suppose we decide to run the English classes of the five Year 7 student groups simultaneously. This produces a tile of width 6 and height 15, containing the five Year 7 student group resources, five English teachers, and five ordinary classrooms.

Figure 1 contains two other examples of tiles. The students are grouped by ability for Mathematics, so the five Mathematics classes must run simultaneously and are combined into one large meeting in the input data. The adjacent History classes do not have to run simultaneously, but fitting them neatly alongside Mathematics forces them to. The second tile illustrates a construction, well known to manual timetablers, called the *runaround*. There are only two Music teachers and two Music rooms, so the five Music classes cannot run simultaneously. By interleaving them among other meetings as shown, the tile demands only one of each at any one time.

Our recipe for producing comprehensible timetables may now be stated: first place all of the meetings neatly into tiles, then timetable the tiles. For simplicity we require that timetabled tiles either overlap exactly in time or not at all. In our test instances there are 40 times in the week, and the timetable is planned around six classes each 6 periods wide, plus four periods of Sport and optional religious instruction, so it is natural to build six 6-period tiles and one 4-period tile in each form. The 4-period tiles must align with each other, but the six-period ones may be timetabled together in whatever way utilizes resources best. This common set of widths, or *major columns*, could be inferred, but it forms such a basic part of the timetable planner's thinking that we have chosen instead to require it to be given as part of the input data. Figure 3 shows a timetable following this pattern.

Our algorithm proceeds in four phases, each the subject of a following section. First, specific times during the week are assigned to the major columns (Section 4). Next, tiles are created and meetings inserted into them and timetabled within them (Section 5). Third, the tiles are timetabled against each other by placing them into the major columns (Section 6). Finally, specific resources are allocated to the meetings' resource slots (Section 7).

4 Column Layout

The first step in our algorithm is to assign times to each of the major columns. Figure 2 gives a typical example of what is wanted: each column spread evenly through the week, with its blocks of times not interrupted by meal breaks.

This is an easy problem in practice so we will be brief. A tree search is used which first attempts to give each column one period on each of a set of days that is spread evenly through the week. Once this is achieved the search continues downwards, with columns making requests to days for their single periods to be exchanged for larger blocks, until every column has the block structure it requires. Each day maintains a small bin packing of the blocks it has promised to columns into the intervals between meal breaks, solved exhaustively as each request arrives.

Columns may have preassigned times, propagated from classes. For example, if some class requests Mon1 and Mon2, then a preassignment of these times to some column will occur. Time preassignments are rare and we assume that they do not overconstrain this problem.

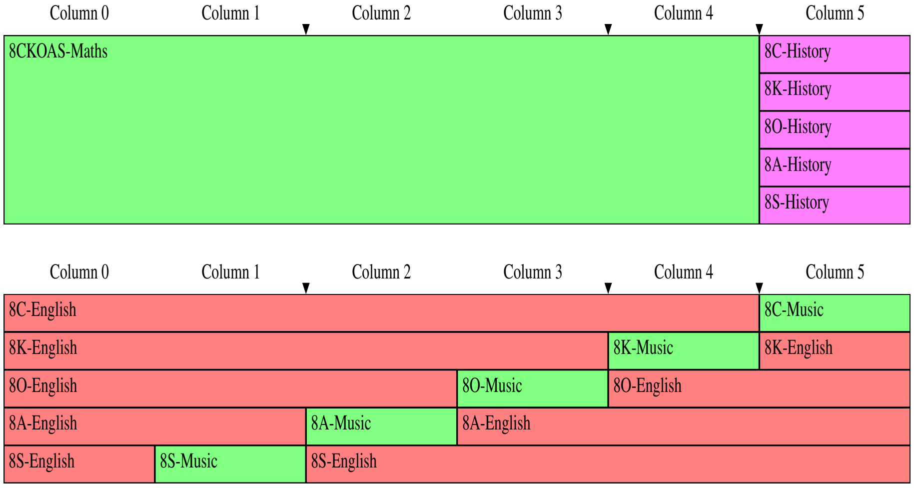

Fig 1. Two examples of tiles from the *bghs98* instance. Each has width 2 2 1 1, as marked by the wedges

	Day 1	Day 2	Day 3	Day 4	Day 5
Period 1	Column 1	Column 4	Column 5	Column 5	Column 3
Period 2	Column 1	Column 4	Column 5	Column 5	Column 3
Period 3	Column 0	Column 3	Column 1	Column 4	Column 0
Period 4	Column 0	Column 3	Column 1	Column 4	Column 1
Period 5	Column 2	Column 2	Column 0	Column 2	Column 2
Period 6	Column 3	Column 2	Column 0	Column 2	Column 6
Period 7	Column 4	Column 5	Column 3	Column 0	Column 6
Period 8	Column 5	Column 6	Column 4	Column 1	Column 6

Fig 2. A typical layout of a week of 40 times into six columns of width 2 2 1 1 plus one of width 3 1. Double lines indicate meal breaks

In practice this algorithm finds layouts like the one in Figure 2 with virtually no backtracking. It has laid out two-week cycles with ease. At present it does not try to equalize the number of morning and afternoon times granted to each column (a common soft requirement), but this could easily be added in a final stage which permutes blocks of times within days.

5 Tile Layout

The second phase of our algorithm builds, for each form, a set of tiles holding the meetings of that form. For example, the Year 8 meetings from the *bghs98* instance have widths

2 2 1	*English*	2 2 1	*Mathematics*	2 2 1	*Science*
2 2	*Languages*	2 2	*Health*	1 1 1	*Geography*
1 1 1	*History*	2	*Art*	2	*Sport*
2	*Technology*	2	*Design*	1 1	*Music*

and must be packed into six tiles of width 2 2 1 1 and one of width 3 1. We wish to minimize the number of meetings split across two tiles, creating a bin packing problem. This particular example is unusually difficult; more commonly, the meeting widths stay quite close to the column widths. The range of possibilities may be gleaned from Figure 3.

Some of these entries represent single meetings (e.g. Mathematics) while others represent a set of five meetings, one for each student group (e.g. English). For comprehensibility, even in this second case we prefer to place all the meetings for one subject into the same tile.

Our current algorithm uses a tree search which first searches for a packing that does not split any meetings over two tiles. If that fails, it splits the smallest meeting in two in all possible ways and tries again. If that fails it splits the two smallest meetings in two in all possible ways, and so on. It usually works quite well (Section 8) but during development has occasionally entered on a long, fruitless search. We plan to replace it with a heuristic method which we believe will do just as well in practice, so instead of describing our current algorithm further we now present some considerations of importance to any tile layout algorithm.

Although we place all the meetings for a single subject within one tile, it is not always possible for them to run simultaneously there, because resources may not be sufficient, and this leads to the runarounds already mentioned (Section 3). The key quantity is the *minimum runaround width*, the minimum number of times that a set of meetings must spread through if its demands are to be satisfied. For example, five one-period Music classes must spread through at least three times if two Music teachers are all that are available. In general, the minimum runaround width is the maximum, over all resource demands made by the meetings, of

$$\left\lceil \frac{\textit{Width of one meeting} \times \textit{Number of occurrences of demand}}{\textit{Number of resources available to satisfy demand}} \right\rceil$$

Each set of meetings is classified as *vertical* (meaning that the meetings must run simultaneously, as in the case of Mathematics where the input data demands it), *runaround* (meaning requiring a runaround, because the minimum runaround width is greater than each meeting's width), or *easy*, meaning that either layout will work. For example, in Figure 1, Mathematics is vertical, Music is runaround, and the others are easy.

For a set of meetings to be timetabled within one tile it is of course necessary, to begin with, that the total width of all meetings containing any one student group resource should not exceed the tile width. Beyond this, there must be room for the runaround meetings to spread out in. When several sets of runaround meetings lie in the same tile, they interleave with each other but still occupy width equal to their total width, so this total must be at least as large as every minimum runaround width. Easy meetings may be co-opted into the runaround, as English is in Figure 1, to help achieve this *runaround condition*, which is sufficient as well as necessary for a timetable to exist, provided that the resource demands of the different sets of meetings do not interact.

Some tiles contain meetings from several forms, and consequently when a tile layout algorithm begins it may find that some of the tiles it is given are not empty. It must check for each of its sets of meetings whether resources are sufficient to permit them to enter such tiles. In particular, when some major column's width is unique, as is the case for the width 4 column of the *bghs98* instance, we only ever create one tile for that column, and that tile holds meetings from every form.

There is an artificial *fixed form* of *fixed tiles* holding meetings with preassigned times. Each fixed tile is permanently assigned to a particular major column, and since the times assigned to columns are known at this point, any meetings with preassigned times can find their way via the columns to the fixed tiles they belong in. These fixed tiles also record resource unavailabilities at particular times, converted in the usual manner into artificial meetings occupying those resources at those times. If a form contains a meeting with one or more preassigned times, that meeting will have already been assigned to the corresponding fixed tile when the form's tile layout begins, and that fixed tile will be one of the tiles handed to the tile layout algorithm.

To summarize, the aim of a tile layout algorithm is to assign its sets of meetings to tiles, avoiding violations of the runaround condition and resource sufficiency problems, minimizing the number of meetings that are split in two, and paying attention to block structure. Some of the meetings it is given may have already been assigned to some of its tiles, and these preassignments must be respected. We believe that a heuristic algorithm that assigns the widest meetings first, keeping vertical and runaround meetings apart as far as possible, looking ahead to avoid resource sufficiency traps, and splitting one meeting in a best-fit manner whenever it gets stuck, will do all this very well.

After meetings are allocated to tiles, they are timetabled within them so as to satisfy resource limits and time block structure requests as far as possible. For single-form tiles this is a small search problem easily solved to optimality; for large multi-form tiles, see the remarks at the end of Section 6.

Although it has not occurred yet in our data, it is quite possible for the minimum runaround width of some set of meetings to exceed the tile width. In that case the runaround must spread over more than one tile, or perhaps it could trigger *part-form tiling*, where the student group resources of one form are partitioned into two parts, each of which is then treated as a separate form. At present our algorithm always partitions the Year 7 and Year 8 forms into two part-forms, using a simple clustering algorithm to decide which student groups to place in each partition. These decisions could be automated, or optionally taken from the timetable planner. Most of the meetings in the higher forms are vertical, so there is nothing to gain from part-form tiling those forms.

6 The Main Timetabling Phase

After all the meetings have been allocated to tiles, and timetabled within them, the next step is to timetable the tiles against each other; that is, to assign the tiles to columns. We call this the *main timetabling* phase.

Underlying any main timetabling algorithm will be a test, probably called many times, for determining whether a given set of tiles is *compatible*: able to run simultaneously without exceeding resource limits. We consider this compatibility testing problem first.

The simplest way to test a set of tiles for compatibility is to merge their meetings into one large tile of the same width and timetable it. This is likely to be too slow when many calls on the test are made.

Three faster compatibility tests have been tried. Each gives an upper bound on the number of unassignable tixels that will result from placing the tiles into the same major column (a *tixel* is one resource at one time). All three methods assume that the individual tiles have been timetabled, and never redo these individual timetables.

The simplest method uses a worst-case measure of the demand for resources made by each tile. If demand varies at different columns of the tile, we take for each type of demand the maximum over the columns. For example, if the five classes do Mathematics simultaneously for some of the tile's times, and History simultaneously for the rest, then the tile's demand will include five Mathematics teachers *and* five History teachers, but only five rooms.

Now form a bipartite graph with one left-hand node for each resource of the instance, and one right-hand node for each resource demand made by each tile in the set being tested. Edges join demand nodes to all resources qualified to satisfy that demand. Find a maximum matching in this graph. The desired upper bound is the number of unmatched demand nodes, multiplied by the common tile width to give a result in tixels.

The second method is a refinement of the first, in which the demand nodes are weighted by the number of tixels that would be deficient if the node remained unmatched. For example, suppose a tile requires two Computer laboratories at two of its times, one Computer laboratory at two of its times, and no Computer laboratories at the other two times – a total of six tixels altogether. This would be represented by two Computer laboratory demand nodes, the first weighted 4 and the second 2. The maximum matching is now required to minimize the weight of the unmatched nodes. For example, if one of our two Computer laboratory nodes was unmatched, it would be the weight 2 one, reflecting the fact that withholding one laboratory from this tile would cost 2 unassigned tixels. If both were unassigned the cost would be 6 tixels. The total weight of unmatched nodes gives a more refined measure of incompatibility, and finding maximum node-weighted matchings is not much harder than finding unweighted ones.

The test we currently use is a yet further refinement. It is considerably slower than the first two, but still fast enough for our purposes. Each tile is assumed as before to be timetabled in one fixed way. The test works by combining the tiles one by one into a larger tile, so let us suppose that k tiles have been taken and we now wish to add in the $(k + 1)$st.

For each column (individual time) of the combined tile, find the demands made on resources by that column. Do the same for each column of the tile to be added. Take the first column of the combined tile and the first column of the tile to be added, merge their demands together into one bipartite graph, find a maximum matching, and count the number of unmatched nodes. This is the number of unallocated tixels that would result if these columns were aligned. Do this for every combination of one column from the combined tile and one from the incoming tile – if the tile width is W, a total of W^2 tests.

Now build a complete bipartite graph whose left-hand nodes are the columns of the combined tile, and whose right-hand nodes are the columns of the incoming tile. Weight the edge connecting a pair of nodes by the outcome of the test on the corresponding columns. Find a maximum matching of minimum cost in this graph, giving a permutation of the columns of the incoming tile that minimizes the number of unallocated tixels. Permute the timetable of the incoming tile according to this matching, and merge the tile into the combined tile. Repeat until all tiles are merged. The total weight of the last min-cost matching will then be an upper bound on the number of unallocatable tixels if these tiles are timetabled together.

This last test has the advantage when tiles with tall, thin demands meet. For example, in the *bghs98* instance there is a Year 9 tile that requires (among other things) five Physical Education teachers simultaneously for two of its six times, and a Year 10 tile that also requires five Physical Education teachers for two simultaneous times. There are five Physical Education teachers altogether. The first test would rate the incompatibility of these two tiles at 30 tixels (five teachers times six times), the second at 10 tixels (five teacher nodes of weight two each), while the third recognizes that there are no unallocatable tixels at all.

Several algorithms for the main timetabling phase have been implemented and tested, including a set covering algorithm (which generates many sets of compatible tiles and then tries to select some sets which contain every tile exactly once), various tree search algorithms, and an augmenting path algorithm inspired by the algorithm of Section 7. In no case did any of these other algorithms outperform the algorithm about to be described.

Take each form in turn and assign its tiles to suitable columns, beginning with the fixed form and ending with the part-form forms, which have small height and so make good fillers of cracks. Suppose that k forms have been allocated to columns in this way and we now wish to allocate the $(k + 1)$st form. Test each tile in the $(k + 1)$st form for compatibility with each column. To be considered even minimally compatible the tile must have the same width as the column and not have been previously assigned to any other column (a tile may be in multiple forms, in which case it will be assigned a column along with the first of its forms that is timetabled, and must not be assigned to some other column afterwards). Build a bipartite graph in which the left-hand nodes are the columns and the right-hand nodes are the tiles of the current form. Edges join tiles to those columns with which they are minimally compatible. These edges are weighted by the number emerging from the compatibility test of this tile with this column. Find a minimum cost maximum matching in this graph; its cost will be an upper bound on the number of unassignable tixels created by adding in these tiles. Make the assignments of tiles to columns indicated by this matching and proceed to the next form. An example of a timetable created by this algorithm appears in Figure 3.

The algorithms for testing tiles for compatibility and for the main timetabling are essentially the same, only operating at different scales. Both algorithms are weighted versions of the meta-matching algorithm of [2], without the look-ahead tests employed there.

Should some kind of tree search seem indicated in future, one interesting one we have tried is based on finding an alternating cycle of minimum cost in the min-cost flow at each level. Applying this cycle gives the maximum matching of second-minimum cost, giving two matchings of each form so that a small binary tree of alternative assignments may be searched.

After the tiles' columns are fixed, each column receives a final timetabling which attempts to give its meetings the block structures they require, while minimizing unallocated tixels. This is a general timetabling problem, and despite being limited to one column it can still be challenging. Our current algorithm embarks on a long tree search which is terminated early to keep running time down. It is slow and unreliable and needs to be replaced by an algorithm based on (but not limited to) the matchings found when constructing the column, so we will say no more about it here. Instead we offer a method of reducing the problem size which should be useful to any algorithm for timetabling multiple forms within one column.

Consider a meeting that happens to stretch the full column width. Its resource demands naturally affect the feasibility of timetabling the column, but because they are constant at every position, they cannot influence any meeting in the column to choose one position over another.

Next consider the set of all meetings in a column containing a given student group resource, and suppose that together they stretch to the full column width (as is almost always the case). A resource demand common to all these meetings is effectively a full-width demand with the properties just outlined. For example, if student group 7C attends English for one part of a tile and Geography for the rest, and both classes require an ordinary classroom, then that requirement might as well lie in a single full-width meeting.

When such requirements are subtracted out it is often possible to recognize that the timetabling problem for some forms is independent of other forms. For example, if Year 7 is attending English and Geography while Year 10 is attending Science and Music, then the two forms may be timetabled independently. We frequently find that the number of forms that must be timetabled together is reduced to three or four using this analysis.

7 Resource Allocation

After times are assigned to meetings, the last major phase of our algorithm is to assign resources. The method to be used here could be applied to any situation in which resources are to be assigned after the times of meetings have been fixed, although, as we will see, it relies to some extent on the coherence provided by tiling.

At first sight resource allocation may seem trivial, since time assignment is supposed to guarantee that for each time, resources are sufficient to cover all the slots of all the meetings running at that time. However, merely using the resources provided by that guarantee would produce large numbers of split assignments.

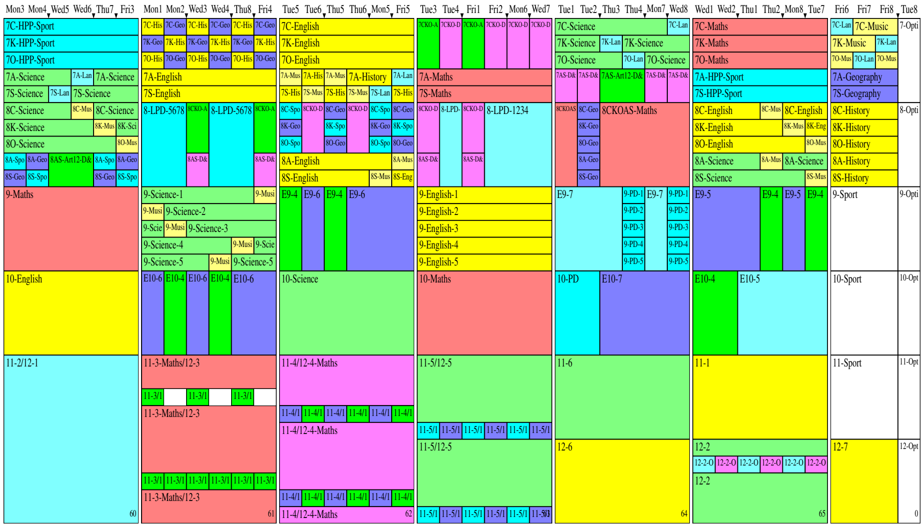

Fig 3. A timetable for the *bghs98* instance described in Section 8, created by the algorithm of Section 6. Each row contains the meetings attended by one student group. Each narrow column represents one of the 40 times of the week; these are grouped into 7 major columns, within which traces of the tiles that were placed in those columns are clearly visible. The tile at bottom left is an example of a multi-form tile; it spans Years 11 and 12. One can also see part-form tiles for Year 7 at the top, one set for student groups 7C, 7K, and 7O, the other for 7A and 7S

A tree search algorithm was tried initially for this problem. The teachers were taken one faculty at a time and assigned in all possible ways. Despite full propagation of constraints and grouping of equivalent resource slots to avoid searching symmetrical situations, this method was never able to search the full tree, and often the best solution it could find in a reasonable time (several minutes) was easily improved by a small chain of exchanges.

This experience suggested a switch to an alternating path algorithm, as used in bipartite matching. Choose a currently unfilled slot of maximum width. See if there is a teacher able to fill this slot (the teacher must be free at its times, and adding the slot to the teacher's current load must not overload the teacher). If so, assign that teacher and move on to the next widest unfilled slot. If not, see if there is a teacher who would be able to fill this slot if only some one of the slots that teacher is currently teaching was taken away and given to some other teacher able to fill it. If so, make the indicated chain of two assignments and one deassignment, and move on. If not, look for a longer chain of three assignments and two deassignments, and so on. In searching for these chains, possible assignments and deassignments are marked *visited* when they are first considered. No already visited assignment or deassignment may be revisited in the course of one search. This prevents loops, and ensures that only a limited amount of time is spent assigning any one slot. If the search fails, then the slot is left unassigned for the time being.

The chain of assignments and deassignments is actually carried out by the algorithm as the search proceeds. At any moment where the state is *feasible* (where no teacher is overloaded), the total weight of the assignments in that state is compared with the total weight of the best solution found so far, and the best solution is replaced with the current one if the current weight is greater. The weight of an assignment is the number of times in the slot being assigned (so that large slots are favoured over small ones). If quota overflows are allowed the amount of any overflow is subtracted from the weight, to discourage overflowing even though it is permitted. Split assignments (see below) are also discouraged by being given smaller weights.

A few failed slots can be retrieved by a second pass, attempting again to assign slots that failed the first time around. This achieves nothing in the traditional applications of the alternating path method, where the first pass produces an optimal result, but in our more complex situation it does produce an occasional extra assignment. After that, a third pass is made in which split assignments and partial assignments are permitted, as we now describe.

Enhancing the basic algorithm to include split assignments is quite straightforward: a split assignment merely affects the quotas of two teachers rather than one. The problem with split assignments is not one of definition, it is their large number. Consider a slot occupying T times, and for which there are R qualified resources. Then there are R possible ordinary assignments (one for each qualified resource), but $R(R-1)(2^{T-1}-1)$ split assignments: choose an arbitrary non-trivial subset of the times, assign an arbitrary qualified resource to that subset, assign some other qualified resource to the remaining times, and divide by two to correct for counting every assignment twice. For typical values such as $R = 10$ and $T = 6$ this is already in the thousands and growing rapidly. Introducing thousands of objects in order to model something that we would rather not use anyway does not seem cost-effective; we have not tried it.

As mentioned earlier, we choose not to introduce split assignments at all initially, to give the algorithm a chance to show what it can do without them. When they are eventually introduced, they are in the form of *assignment factories* rather than the assignments themselves. These factories are lists of qualified resources and subsets of times, from which assignments can be generated. When a factory for some meeting is present, it will be asked to produce a single assignment which would terminate the search at that meeting. It then searches its lists for a pair of resources that can split the current slot between them while *both* remaining not overloaded. Any such assignment is added to the pool of assignments and competes with them. If not used it is removed again. Partial assignments are handled in the same way.

This algorithm has proved to be fast and almost perfect (Section 8). It does not need to assign the teachers faculty by faculty as the tree search method did. Some detailed examples of its results appear in Figures 4 and 5. We offer the following explanation for its success in a context where the guarantees it usually operates under are absent.

The literature contains occasional references to an exact network flow algorithm for resource allocation (for example, [1] cites [4] on this point). It is easy to find such a network for our problem in the special case where all meetings require the same number of times, each pair of meetings either clashes completely or not at all, and no split assignments are allowed. Then limits on the number of *times* a teacher may teach may be converted into limits on the number of *classes* a teacher may teach, by dividing

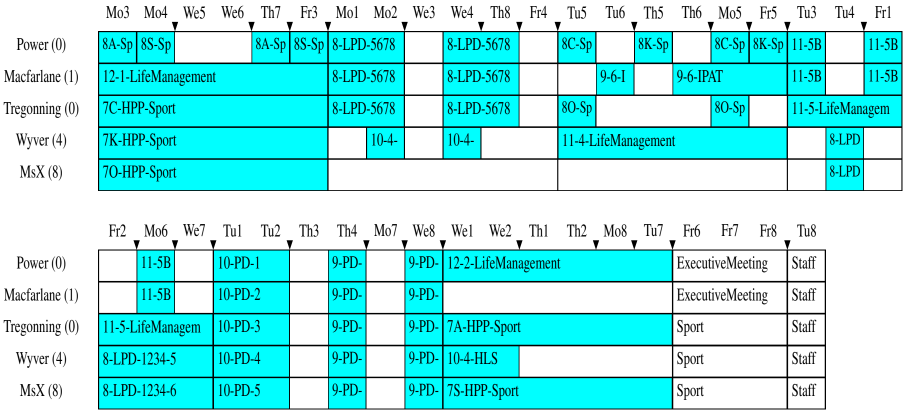

Fig 4. Part of the teacher allocation for instance *bghs98*, showing the teachers and classes for Physical Education. This allocation is perfect: all classes are covered, there are no split assignments, and as many teachers as possible are teaching Sport outside the faculty. The number in parentheses after each teacher is the remaining unused portion of the teacher's quota. This faculty is lightly loaded

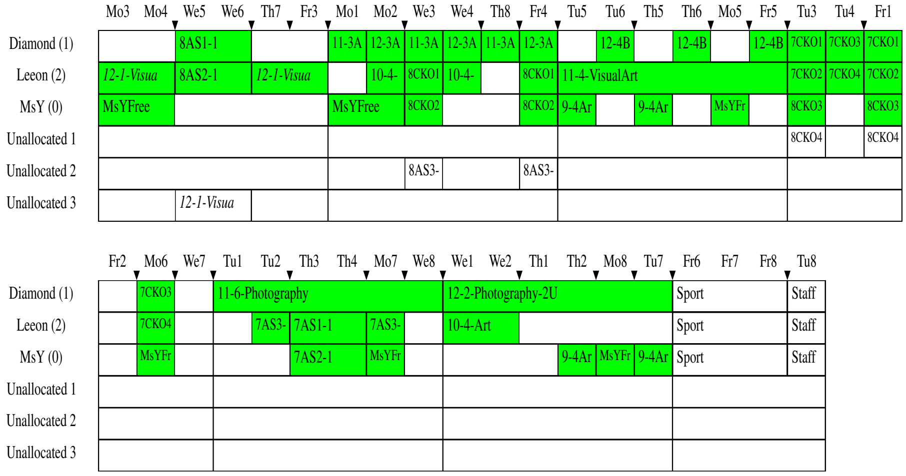

Fig 5. Another part of the teacher allocation for instance *bghs98*, showing the teachers and classes for Art. This allocation is much less perfect: there is one split assignment, shown in italics, partly unallocated, for which a correcting alternating path exists that the algorithm has failed to find; plus two other unallocated classes, caused by assigning four simultaneous Art classes when there are only three Art teachers. If these meetings' times were moved, the Art teachers would need to be assigned less Sport in order to take them, since the total remaining unused teachers' quota is only 3 times. This suggests that it would have been better if some Art classes had been timetabled over Sport, rather than classes from some other subject in which the teachers are more lightly loaded

by the common meeting length and rounding down. Each path in the network begins with an edge from the source to a node representing one teacher, of capacity equal to the number of classes that teacher may teach, then proceeds via an edge of capacity 1 to a node representing that teacher's availability at a particular set of times, thence to each node representing a class at that set of times that the teacher is qualified for, and from there to the sink with capacity 1.

It is easy to verify that the usual max-flow algorithm on this network is equivalent to our algorithm above. There is also a matroid intersection formulation whose equivalence is even more intuitive. Thus, in this special case, our algorithm is optimal.

When we move to the general problem, two sources of NP-completeness appear: when meetings vary in length, fitting them into teachers' quotas is a bin packing problem; and when they clash in arbitrary ways, the clash graph (in which nodes are meetings and edges join pairs of clashing meetings) changes from a set of disjoint cliques to an arbitrary graph, producing a node colouring problem [3]. Thus the alternating path method cannot be optimal in the general case, but we argue now that it stands an excellent chance of doing well nevertheless.

The algorithm sorts the meetings by decreasing number of times, assigning the meetings with the most times first. In bin packing terms this is the 'first fit decreasing' heuristic, for which there are quite good performance guarantees [5]. In the common situation where the largest meeting size is equal to the column width, and no meetings of that size are split across two columns, the algorithm will be optimal while assigning these large meetings.

The node colouring intractability is mitigated by the use of tiles. Most meetings occupy a single tile, so the clash graph is close to the set of disjoint cliques of the tractable special case. Meetings split across two tiles spoil the disconnectedness, and meetings that occupy less than the full column width may not clash with other short meetings in their column; but these cases are in the minority.

8 Results

This section analyses the performance of our algorithm on three instances taken from a high school in Sydney, Australia. A statistical description of these instances appears in Table 1. These instances contain staff meetings, which are not yet included in our solutions except when their times are preassigned. Staff meetings do not affect teacher quotas, they merely make the teachers involved unavailable when they are running.

Run times for the four phases and in total are given in Table 2. Total times were checked against wristwatch time. Both the tile layout and main timetabling phases include laying out meetings within the tiles created by those phases, and this operation in its current defective state (Section 6) dominates the cost of both these phases, so speed improvements can be expected here in future. The times given for resource allocation include a stopgap attempt at room allocation using the teacher allocation algorithm. When a dedicated room allocation algorithm is installed, resource allocation time should decrease by two to three seconds.

The quality of the solutions found for the three instances is summarized in Table 3. The algorithm always assigns the correct number of times to each meeting, never

Table 1. Statistical description of the three instances tested, showing the number of times in the week, meetings, teachers, rooms, and student groups. The last three lines show, for each of the three resource groups, the number of tixels of demand for that resource group as an absolute number and as a percentage of the number of tixels of supply for that resource group. (A tixel is one resource at one time.) For students and rooms the tixel supply is just the number of resources times the number of times in the week, but for teachers it is less owing to teachers' quota limits. The demand for student groups is less than 100% because a few final year students attend marginally less than full time

Instance description	*bghs93*	*bghs95*	*bghs98*
Times in the week	40	40	40
Meetings	148	146	152
Teachers	53	52	56
Rooms	46	48	45
Student groups	23	27	30
Teacher demand (tixels)	1489 (95.3%)	1378 (95.4%)	1408 (96.6%)
Room demand (tixels)	1295 (70.4%)	1306 (68.0%)	1357 (75.4%)
Student group demand (tixels)	872 (98.0%)	1041 (99.3%)	1197 (99.8%)

Table 2. Run times in seconds for the three instances tested. The tests used a 1.2GHz Pentium IV running Redhat Linux 5.1. Run times are as reported by the Linux *time* command, which is accurate to one second

Run time (seconds)	*bghs93*	*bghs95*	*bghs98*
Column layout	0.0	0.0	0.0
Tile layout	2.0	7.0	1.0
Main timetabling	2.0	6.0	5.0
Resource allocation	6.0	6.0	6.0
Total time	10.0	19.0	12.0

Table 3. Solution quality for the three instances tested. The table shows both the absolute number of each possible kind of defect, and the number as a percentage of the number of meetings, room tixels, teacher slots, or teacher tixels as appropriate

Solution quality	*bghs93*	*bghs95*	*bghs98*
Meetings with at least one time layout problem	19 (13.8%)	63 (43.2%)	63 (41.4%)
Room tixels unassigned during time assignment	33 (2.5%)	12 (0.9%)	26 (1.9%)
Teacher slots split by resource assignment	13 (2.9%)	31 (6.7%)	21 (4.8%)
Teacher tixels unassigned during time assignment	4 (0.3%)	12 (0.9%)	10 (0.7%)
Teacher tixels unassigned during resource assignment	11 (0.7%)	25 (1.8%)	20 (1.4%)
Total teacher tixels unassigned	15 (1.0%)	37 (2.7%)	30 (2.1%)

introduces student group clashes, and prefers to leave teacher and room slots unassigned rather than introducing teacher and room clashes. So the possible defects are time layout problems (wrong number of double periods, meeting spread over too few days, etc.), missing teacher and room assignments, and split teacher assignments.

The number of meetings with some kind of time layout problem is quite high at present (over 40% in two instances), but this is less serious that it may seem. Most of the problems concern assigning one more or less double period than was requested, and often there would not be a strong preference about this in any case. We hope to capture better data concerning time layout preferences in future, including optional alternatives and priorities, and this plus the planned new algorithm for distributing meetings to tiles (Section 5) should reduce the number of time layout problems to an acceptable level.

We have not yet written a dedicated room assignment algorithm, so the table only reports the number of tixels for which rooms are not available after the main timetabling phase. Since room constancy is not required this is probably the exact number of unassigned rooms that will occur in the end (the only possible problem being the need for room constancy in double periods). These unassignable room demands are for specialised laboratories whose demand is very tight. This problem is quite common in high schools and is not of major concern, since, given its low relative frequency, it is not difficult to ensure that no class meets in an inappropriate room for more than one of its times, and the teacher would organize the classroom material accordingly. Our remarks below about reducing unallocated teacher tixels also apply to rooms.

The number of split teacher assignments seems to be close to optimal now. In other experiments, not reported in detail here, in which teachers were allowed to take just one more period than their quota, but with a penalty if this occurred, the number of teachers who exceeded their quota was quite modest (between 10 and 20), and the number of split assignments typically halved. This, along with hand analysis, provides good evidence that split assignments are mainly needed to pack classes into teachers' quotas, and thus are inevitable.

This leaves the problem of assigning qualified teachers to classes. Although as a last resort an unqualified teacher may be assigned, this is considered much worse than assigning an inappropriate room, and our absolute numbers of unassigned slots are at present too high for our timetables to be usable.

Unassigned teacher tixels are created in two ways. First, during the main timetabling, a decision may be made to run tiles simultaneously that results in the demand for teachers with a certain capability at some time exceeding the number of teachers qualified for that capability available at that time. The number of tixels affected by defective main timetabling in our instances is quite small (4, 12, and 10), and close examination shows that some of them are caused by defective layout of meetings within large tiles. Our new algorithm (Section 6) should correct that problem. Beyond that, it will be necessary to break open the tile structure to swap fragments of classes containing unassignable teachers to other times where teachers are available. Hand analysis of our current solutions indicates that this will succeed most of the time. Any of the meetings contributing to the excessive demand may be moved, and we would naturally choose to try to move small classes rather than large electives.

The second chance to create unassigned teacher tixels comes during resource assignment, when the resource allocation algorithm is unable to assign a teacher or split teacher to a slot, even if there are teachers free. Examination of the data suggests that many of these problems arise from various imbalances in the supply of teachers.

For example, some faculties are lightly loaded, so their teachers should take some classes from outside the faculty. But the number of such outside classes is very limited (in our instances, essentially only Sport), so care must be taken to ensure that the faculty's own classes are not scheduled at the same time as these other classes. Our algorithm is currently quite blind to the need for this, although we can diagnose the situation well by comparing supply with demand for each faculty. On occasions during development we have observed solutions in which the right decisions were made fortuitously, and these contained significantly fewer unassigned tixels than reported in our formal results.

9 Conclusion

The work reported in this paper is ongoing, and our results at the time of writing are not perfect. Nevertheless, they show that it is possible to construct high school timetables of high quality in about ten seconds. Our key innovations are tiling and an effective resource allocation algorithm based on alternating paths. The alternating path algorithm might find application in other resource allocation tasks, although it does rely on tiling to ensure that the graph colouring problems it faces are not too severe.

Things to do immediately include getting staff meetings into tiles, replacing the tile layout algorithm, replacing the algorithm for timetabling the meetings within a single tile, writing a room allocator, and writing code for breaking open the tile structure and swapping small parts of meetings that failed to receive their needed resources to better times. Ideas for detecting and correcting resource supply imbalances could be developed further.

When all this is done our timetables should be good enough to show to high schools. Australian high schools are just now receiving broadband Internet connections, so exciting prospects are opening up for delivering timetabling across the Internet. Fast response time will be important.

References

1. M. W. Carter and Gilbert Laporte. Recent developments in practical course timetabling. In Edmund Burke and Michael Carter (eds.), *Practice and Theory of Automated Timetabling II (Second International Conference, PATAT'97, University of Toronto, August 1997, Selected Papers)*, pages 3–19. Springer Lecture Notes in Computer Science 1408, 1998.

2. Tim B. Cooper and Jeffrey H. Kingston. The solution of real instances of the timetabling problem. *The Computer Journal* **36**, 645–653 (1993).

3. Tim B. Cooper and Jeffrey H. Kingston. The complexity of timetable construction problems. In *First International Conference on the Practice and Theory of Automated Timetabling*. Napier University, Edinburgh, UK, 1995.

4. J. S. Dyer and J. M. Mulvey. Computerized scheduling and planning. *New Directions for Institutional Research* **13**, 67–86 (1977).

5. Garey M. R. and Johnson D. S.. *Computers and Intractability: a Guide to the Theory of NP-completeness.* Freeman, San Francisco, 1979.

6. D. de Werra. An introduction to timetabling. *European Journal of Operational Research* **19**, 151–162 (1985).