Resource Assignment in High School Timetabling

Jeffrey H. Kingston

the date of receipt and acceptance should be inserted later

Abstract This paper explores one aspect of the high school timetabling problem, namely the assignment of resources, such as teachers and rooms, to meetings after times are assigned. Several algorithms, with run times of just a few seconds, are presented and tested on real-world data. The best of these is currently in operation within KTS, a free, public web site for high school timetabling created by the author. A large bipartite matching model, called the *global tixel matching*, is used to preserve optimality of one key measure of quality as resource assignment proceeds.

1 Introduction

High school timetabling is an NP-complete problem with an extensive literature [5]. Some recent contributions may be accessed via the PATAT conference series [2,3]. This paper explores one aspect of the problem, namely the assignment of resources, typically teachers and rooms, to meetings after their times are assigned. A detailed specification appears in Sect. 2. The prior problem, of assigning times to meetings, has been treated by this author previously [12,13].

At the centre of this paper lies the *global tixel matching*, a large bipartite matching model which provides a valuable check on the implications of one assignment (Sect. 3). Similar models were used by Mulvey [14] when assigning rooms, and Cooper and Kingston [7], whose model included workload limits. This paper extends this earlier work with a more complete model of workload limits, and optimizations (Sect. 4) which open the way for run times of just a few seconds, fast enough for use in timetabling web servers such as this author's KTS system [11,12]. Several such algorithms, all currently implemented in KTS, are presented (Sect. 5) and tested on six instances taken from Australian high schools (Sect. 6).

Jeffrey H. Kingston
School of Information Technologies
The University of Sydney, NSW 2006, Australia
`http://www.it.usyd.edu.au/~jeff`
E-mail: jeff@it.usyd.edu.au

2 Specification

The KTS specification of the high school timetabling problem will be used here. Time is modelled as a sequence of intervals of equal length, called the *cycle*, some of which are adjacent in time while others are separated by breaks. Resources come in sets called *resource groups*, of which there are usually three, namely *Students*, *Teachers*, and *Rooms*. Resources from the *Students* group are always preassigned, so only teachers and rooms need assignment.

Resources may have *workload limits*, which limit the number of times that a resource may attend meetings on any day, or over the whole cycle. KTS offers hard limits (which may never be exceeded) and soft limits (which may be exceeded with a penalty), set individually for each resource on any subsets of the times of the cycle. Resources may also be unavailable at nominated times.

There may be any number of meetings, each demanding any number of times and resources of various *types* (a Science laboratory, an English teacher, and so on), either preassigned or left for the solver to choose. The resources assigned to a meeting meet together during all of its times, so are unavailable for other meetings at those times.

The assigned times of a meeting fall naturally into *time blocks*: sets of adjacent times not spanning a break. For example, denoting the first time on Mondays by $Mon1$, etc., the set of times $\{Mon1, Mon2, Wed5, Wed6, Fri4\}$ consists of three blocks, $\{Mon1, Mon2\}$, $\{Wed5, Wed6\}$, and $\{Fri4\}$, unless a break intervenes.

When assigning rooms, it is desirable to assign the same room throughout any one time block, since it is disruptive for the students to change rooms part-way through the block, but there is no requirement for the same room to be used for all the blocks of one meeting. When assigning teachers it is desirable for the same teacher to be assigned for all the times of one meeting.

Accordingly, a *resource slot* is defined to be a place in a meeting to be occupied (preferably) by a single resource. For example, a meeting in which student group $7A$ studies Science at the five times given above would have one slot preassigned student group resource $7A$, one slot requiring a teacher qualified to teach Science, and three slots requiring a Science laboratory, one for each time block.

The usual case, where one resource is assigned to the entire slot, is called a *satisfactory assignment*. All other cases are *unsatisfactory assignments*, classified by KTS into three kinds: *split assignments*, in which the slot is shared between two resources (one covering some of its times, the other covering the rest); *incomplete assignments*, where at most one resource is assigned to the slot, but for less than all of its times, leaving the rest uncovered; and *dreadful assignments*, where the slot is split between three or more resources, or incomplete and split between two or more resources. This terminology is a little unfair to split assignments, which although undesirable are common in manually generated timetables.

The *high school resource assignment problem* is to assign the resources of one resource group to as many resource slots as possible, after all times are assigned. The hard constraints are that resource slots must receive resources of the appropriate types, preassignments must be respected, clashes must be avoided, and hard workload limits and unavailabilities obeyed. KTS never violates these constraints, preferring instead to leave some resource slots incompletely assigned. The soft constraints, in order of decreasing importance, are that the number of dreadful assignments, incomplete assignments, split assignments, and soft workload limit overloads should be minimized. In accordance with the philosophy of KTS, in this paper good but not necessarily optimal solutions are sought that can be found in a few seconds.

The problem has two main variants, *room assignment* and *teacher assignment*, whose sole difference is that room slots contain fewer times, and these times always form a single block. This makes room assignment easy in practice, although formally it is exactly list colouring of interval graphs, which is NP-complete even when the cycle contains just two times [4].

KTS offers a few other features. It is assumed that these are used relatively infrequently, so can be handled simply. A resource slot may have *preferred* resources, to be used if possible. One resource may be declared to *follow* another, meaning that it should be assigned to every meeting containing that other resource, wherever a suitable slot for it exists. For example, a teacher may have a 'home room' that follows the teacher in this way. And a slot may be marked *unsplittable*, meaning that a split assignment is not acceptable there.

Teachers are usually partitioned into *faculties*, such as English, Mathematics, and so on. Most teacher slots also have a faculty (the main exception being Sport, which can be taught by any teacher). A slot with a faculty may accept teachers from other faculties, but teachers from its own faculty are slightly preferred.

The resource assignment problem in high schools resembles staff rostering problems, which also assign resources after times are fixed. However, staff rostering problems typically have complex constraints on the overall timetable of each resource, such as that a nurse not work more than two night shifts per week, or that each leg of a bus driver's schedule end at the place where the next leg begins. In contrast, the corresponding constraints in high schools are much simpler, as we have seen.

European high school problems [8,9] often have teacher utilizations of 50% or less; that is, teachers typically attend meetings for at most half of the times of the cycle. This naturally leads to a requirement that teachers' timetables be *compact*: have few gaps within each day. This author's experience is based on Australian high schools, where teacher utilizations are typically 75%. Compactness is not a requirement in these schools, and it will not be considered here.

3 The global tixel matching

A timetabling problem is a kind of market, in which resources are demanded by meetings and supplied to them. The unit of supply is one resource at one time, which will be called a *supply tixel*. The term 'tixel' has been coined by the author by analogy with the 'pixel', one unit of a graphical display.

Each meeting demands a certain number of tixels of certain types. For example, a typical meeting called *7A-English*, in which student group *7A* studies English for 6 times per cycle, demands 18 tixels: six tixels of student group resource *7A*, six tixels of teachers qualified to teach English, and six of ordinary classrooms. This meeting is said to contain 18 *demand tixels*.

Each demand tixel in each meeting requires a supply tixel to be assigned to it. Each supply tixel can only be assigned to one demand tixel, since to assign it to two is to introduce a timetable clash, which is not permitted. This immediately implies that underlying the problem is an unweighted maximum bipartite matching, which will be called the *global tixel matching*: each demand tixel is a node, each supply tixel is a node, and an edge joins demand tixel d to supply tixel s whenever s is suitable for assignment to d. For example, a tixel demanding student group resource *7A* would be connected to the supply tixels for resource *7A* (one supply tixel for each time in the cycle). A tixel demanding an English teacher would be connected to each supply tixel of each teacher qualified to teach English.

When an assignment is made to a meeting, the sets of edges connected to its demand tixels (their *domains*) shrink. For example, the six tixels demanding resource *7A* in meeting *7A-English* are initially connected to all the supply tixels for *7A* (one for each time of the cycle), but after time assignment each becomes associated with a particular time, and is connected to just one supply tixel, the one for *7A* at that time.

The number of unassignable tixels in this graph (that is, the number of demand tixels not touched by a maximum matching) is a lower bound on the number of unassignable demand tixels in any solution, given the decisions already made. KTS uses the matching to diagnose supply problems (shortages of laboratories, etc.) before solving begins. However, this paper is concerned with its use during resource assignment, after all times are assigned, the basic idea being to allow only assignments that do not cause the number of unassignable tixels to increase. The matching itself provides an assignment which is optimal with respect to the number of tixels assigned; but this is not useable directly, because it has many split assignments.

Teachers typically have workload limits which form an important constraint. These may be modelled by introducing additional demand tixels, as follows.

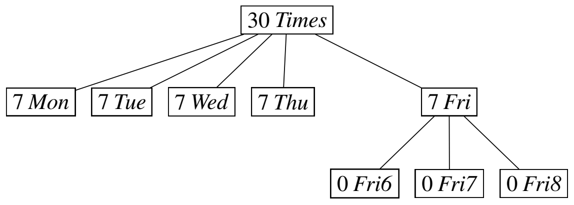
Suppose that resource r is unavailable at time t. This could be modelled by deleting the supply tixel representing r at t. However, it turns out to be more convenient to introduce another kind of demand tixel, the *unavailability demand tixel*, linked in the matching graph only to the supply tixel representing r at t. This prevents this supply tixel from being assigned elsewhere.

More generally, a resource may have a hard workload limit on some subset of the set of times of the cycle. For example, suppose that there are 8 times on Friday, but that resource r is allowed to be occupied for only 7 of those times. This is modelled by introducing another type of demand tixel, the *workload demand tixel*. In this example there would be one workload demand tixel linked to all of r's Friday supply tixels. Satisfaction of this demand ensures that r remains free for at least one time on Friday.

A resource may have limits on several subsets. For example, it might be limited to 7 times on each day. Resource unavailability adds more subsets: to say that resource r is unavailable at time t is equivalent to imposing a workload limit of 0 on the set of times $\{t\}$, so the unavailability demand tixel is a kind of workload demand tixel.

It is necessary to consider the relationships between these subsets. For example, if one of the Friday times is an unavailable time, then a workload limit of 7 on Friday times is satisfied automatically. In general, it is possible to model a collection of workload limits for various subsets of the set of times, provided that these subsets satisfy the *subset tree condition*: each pair of subsets is either disjoint, or else one is a subset of the other.

For example, suppose the cycle has five days of eight times each, and that resource *r* has workload limits requiring it to be occupied for at most 30 times altogether, at most 7 times on any one day, and to be unavailable at times *Fri*6, *Fri*7, and *Fri*8. These limits form a tree:



A postorder traversal of this tree may be used to deduce that workload (including unavailability) demand tixels for *r* are needed for one *Mon* time, one *Tue* time, one *Wed* time, one *Thu* time, one *Fri*6 time, one *Fri*7 time, one *Fri*8 time, and 3 arbitrary times. Each node contributes a number of tixels equal to the size of its subset minus the size of its workload limit minus the number of tixels contributed by its descendants, or none if this number is negative.

When the subsets do not satisfy the subset tree condition, an exact model using tixels seems not to be possible. An example would be if, in addition to the above limits, there were limits on the number of morning and afternoon times. Of course, assignments which violate such limits can always be prevented, simply by checking them, but their omission from the matching could give an unduly optimistic assessment of the state of resource assignment. Fortunately, such overlapping limits do not seem to occur in practice.

To support Australian practice, KTS allows a resource slot to attract a smaller workload than its associated number of times. Such slots are said to have a *special workload*. They are treated as unsplittable, since it is not clear how to divide a special workload between two resources. Like overlapping resource limits, special workloads cause inexactness in the global tixel matching. The details of how KTS handles this problem are not of general interest and have been omitted from this paper.

4 Optimizations

Even after time assignment has reduced the domains of most demand tixels, the global tixel matching remains large and potentially very slow. This section introduces optimizations which significantly reduce its cost. Familiarity with the theory of unweighted bipartite matchings, including the augmenting path method [1], is assumed.

Although the global tixel matching is essential for finding subtle problems with proposed resource assignments, it is a clumsy tool for finding obvious ones, such as clashes and hard workload limit overloads. Accordingly, each resource holds an array of integers, one for each time of the cycle and one for each subset of the times subject to a workload limit, whose value is the number of times from the subset during which the resource is currently attending meetings. The first step in evaluating a proposed assignment of a resource to a slot is to increase these totals by amounts determined by the times associated with the slot. If any exceed a hard limit, the assignment may be abandoned without touching the global tixel matching. This test also checks any hard limits not reflected in the matching because they did not satisfy the subset tree condition (Sect. 3), and calculates soft workload limit overload penalties.

The global tixel matching module stores the demand and supply nodes, plus a current matching, which is always valid (that is, its edges lie in the domains and have no common endpoints) but not necessarily maximal. A list of all the currently unassigned demand nodes is kept separately from the main list of all demand nodes. A record is kept of whether or not the current matching is known to be up to date (that is, maximal).

The operation of changing the domain of a demand node is easily implemented efficiently. First, change the domain; then, if the node is currently assigned to a supply node that is not present in the new domain, deassign the node and add it to the currently unassigned demand node list. Finally, mark the matching as not up to date. An up to date matching must be marked not up to date even if the node remained assigned, since its new domain could permit it to match with a different supply node which opens the way for some other, currently unmatched demand node to match with its current supply node.

The matching may be brought up to date at any time, by doing nothing if it is already up to date, and otherwise attempting an augment out of each element of the unassigned demand node list, removing those elements which assign, then marking the matching as up to date. For example, whenever an enquiry is received asking how many unassigned nodes there are, the matching is first brought up to date, then the number of unassigned nodes is returned.

This arrangement already offers some optimization: it amortizes the cost of bringing the matching up to date over all domain change operations since the last time it was brought up to date. However, the time taken could be on the order of the total number of unassigned demand nodes multiplied by the size of the graph. In realistic cases one would expect a reasonable balance between supply and demand, and hence few unassigned demand nodes, but this cannot be guaranteed. So the following three optimizations attempt to mitigate this unfortunate dependence on the total number of unassigned demand nodes.

The first optimization introduces a variable called *ulower*. It is always defined and contains a lower bound on the number of unassignable nodes in the current state. Its initial value is 0, always a valid lower bound. Whenever the matching is brought up to date, *ulower* is set to the number of unassigned nodes at the end of the operation.

Since the current matching is always valid, the number of nodes on the unassigned demand node list is always an upper bound on the number of unassignable nodes in the current state. When *ulower* equals this number, the matching must be up to date.

Each domain change operation supplies a parameter which indicates whether the new domain is known to the caller to be a subset of the old domain, or known to be a superset, or not known to be either. If the new domain is not a subset of the old domain, then the change might allow one more node to match than previously, so *ulower* is decreased by one (unless already 0). If the new domain is a subset of the old domain, then the change cannot allow any more nodes to match than previously, so *ulower* is not changed. If the new domain is a superset of the old, then there is no need to check whether any current assignment lies in the new domain: it must.

When bringing a matching up to date, if augments reduce the number of unassigned nodes to *ulower*, the operation may terminate early: *ulower* is a lower limit on the number of unassignable nodes, and that limit has been reached, so there is no point in trying any more augments. As a further heuristic, augmentation is tried first at those nodes which entered the unassigned list most recently. These are more likely to match than nodes that have been lurking in the list for a long time; those may well be permanently unassignable.

The second optimization uses operations called *MarkBegin* and *MarkEnd*. These always occur in matching pairs, possibly nested. *MarkBegin* brings the matching up to date and returns the number of unmatched nodes. This value is stored by the caller and passed to the matching *MarkEnd* operation, which informs the module that all domain changes since the corresponding *MarkBegin* have been undone. *MarkEnd* sets *ulower* to this value (if it was the number of unassignable nodes when the corresponding *MarkBegin* ended, then since the graph has returned to that state, it must be the ideal lower bound now), then brings the matching up to date.

As the prime example of these optimizations in action, consider testing whether a resource may be assigned to a slot without increasing the number of unassignable nodes. The operation begins with *MarkBegin*, followed by one domain reduction operation for each demand node of the slot, to a domain containing just the resource in question. Then comes a test of the number of unassignable nodes, followed by restoration of the initial domains, and *MarkEnd*.

The call to *MarkBegin* brings the matching up to date and sets *ulower* to the number of unassignable nodes. The domain reduction operations will not change *ulower*, but they will usually add their nodes to the unassigned list. The test of the number of unassignable nodes will bring the matching up to date again. It will try to augment the newly added nodes first, and if all of them succeed it will terminate early, not touching any older unassigned nodes. Bringing the matching up to date at the end will require no augments at all, since the restored domains include the current assignments. The implementation is less efficient when not all domain changes are reductions (special workloads, whose detailed handling is beyond the scope of this paper, cause this), or when the test fails. Simple failures due to clashes and hard workload overloads have already been filtered out by the array of integers described at the start of this section, however, so most tests should succeed.

A typical run was carried out (*BGHS98* using the room assignment and resource packing teacher assignment algorithms of Sect. 5) and various measurements were made which show that these optimizations are useful. For example, only 2.2% of teacher domain changes were neither to a subset nor a superset, and 71.0% of teacher graph updates were terminated by the lower bound either before, or at the same time as, the unassigned nodes list was exhausted, and hence would cost the same even if the number of long-term residents of the unmatched demand nodes list were larger.

The third optimization improves the running time of sequences of unsuccessful augments. The augment operation requires a Boolean *visited* flag in each supply node, indicating whether the current augment has previously visited the node, in which case it will not revisit it, avoiding cycles in the search. It is standard practice to implement this flag by an integer *visit_num* variable, initialized to zero, rather than a Boolean; a node is visited if this number is equal to the visit number of the current augment. Before each augment, this current visit number is increased by one, re-initializing all the nodes to unvisited for free.

Consider two augment operations not separated by any domain change operation, and suppose that the first operation is unsuccessful. Then there is no need to increase the visit number before starting the second operation, because nodes visited by the first operation are known not to lead to unmatched supply nodes, hence there is no need to search through them again. This optimization is applied during the sequence of augment operations required to bring the matching up to date: the visit number increment is omitted before those operations which have an unsuccessful predecessor. This reduces the maximum cost from the number of augments times the graph size to the number of successful augments (plus one) times the graph size.

5 Resource assignment algorithms

This section presents one room assignment algorithm and three teacher assignment algorithms. All four algorithms utilize the techniques of Sect. 4.

At several points in the algorithms there is a need to sort a collection of slots so that the more difficult ones to assign come before the less difficult. The definition of 'more difficult' is the same in all cases; it is to apply the following rules in order until a decision is reached: unpslittable slots are more difficult than splittable ones; wider slots (i.e. with more times) are more difficult than narrower ones; slots with preferred resources are more difficult than slots without; slots with fewer preferred resources are more difficult than slots with more; slots with fewer qualified resources are more difficult than slots with more; slots from wider meetings are more difficult than slots from narrower ones; and slots whose first time comes earlier in the cycle are more difficult than slots whose first time comes later. This last rule encourages a sweep through the cycle in the last resort.

Another common need is to evaluate the cost of assigning a given resource to a given slot, and again a uniform rule is followed. Only assignments of resources of the right type that do not cause clashes or hard workload overloads are permitted; these are called *acceptable* assignments. The quality of the current state of resource assignment is described by a cost vector $(c_1, c_2, c_3, c_4, c_5, c_6)$, and a best assignment is one which produces a lexicographically minimum value of this vector. The first component, c_1, is the number of unassignable tixels in the global tixel matching. The second, c_2, is the number of currently unassigned tixels. The third, c_3, is the number of assignments split between two or more resources, with assignments split between three or more resources weighted more heavily. The fourth, c_4, is the number of soft workload overloads. The fifth, c_5, is the sum over all assigned slots of a number which is low when the resource assigned is from the same faculty as the slot, or if the slot has no faculty it is low for resources from lightly loaded faculties. The sixth, c_6, is the sum over all resources with workload limits of the square of the resource's current workload. The last two components encourage even sharing of workload.

Except when assigning unsplittable slots, the algorithms only carry out assignments which cause no increase in the first component (called *strictly acceptable* assignments); and they eventually assign as many tixels as are assignable according to the global tixel matching. In other words, these algorithms are optimal in terms of the number of tixels assigned, except near unsplittable slots.

As mentioned earlier, it is assumed that preferred and follows resources and unsplittable slots occur relatively infrequently, so may be handled simply. These are assigned in three initial phases which are the same for all four algorithms. Phase 1 assigns slots with preassigned resources, in order of decreasing difficulty as defined above. If any fail to assign, owing to clashes or workload limits, the resulting gaps remain unassigned and become defects in the solution. Phase 2 assigns slots with preferred resources, again in decreasing difficulty order, choosing the best strictly acceptable assignment for each slot. A failure here causes the slot to be treated as though it had no preferred resources; it will be assigned later, along with the other unsplittable or splittable slots as appropriate. Phase 3 assigns the unsplittable slots, again in decreasing difficulty order, choosing the best acceptable assignment. Strict acceptability is not enforced here, because a failure causes the slot to remain unassigned in the solution.

KTS requires the relation 'resource group G_2 contains a resource that follows a resource from resource group G_1' between resource groups to have no cycles. It topologically sorts the resource groups by this relation, and assigns the resources of each resource group in this order. (Typically, this just means that rooms are assigned after teachers.) This allows follows requirements to be converted into preferred resources and handled as such.

It remains to assign the bulk of the resource slots – the ordinary ones. This is where the four algorithms differ.

Room assignment is an easy problem in practice, and a simple algorithm, called Phase 4R, suffices. Assign the remaining unassigned slots one by one in decreasing difficulty order, as follows. As explained earlier, each request for a room in a meeting is broken into several room slots, one for each time block of the meeting. Although these slots are not required to have the same room, for regularity it is good to encourage them to. So the first step is to check whether any other room slots from the same room request have already been assigned. If so, take the resources assigned to those slots and find the best strictly acceptable assignment from among them. If there is such an assignment, use it; otherwise try again using the full set of qualified resources. If there is still no strictly acceptable assignment and the slot has width at least 2, break it into slots of width 1 and add them to the end of the list of slots for assignment later during this phase; otherwise the slot remains unassigned. This is essentially the same as an algorithm presented previously by this author [12].

Teacher assignment is more challenging than room assignment, because the slots are typically wider and may contain several time blocks. The three teacher assignment algorithms all have four more phases. In Phase 4, full-width (that is, satisfactory) assignments are made to most of the remaining unassigned slots. In Phase 5, alternating paths of full-width deassignments and assignments are found which further increase the number of full-width assignments. In Phase 6, split, incomplete, and dreadful assignments are found for the remaining unassigned slots. Finally, Phase 7 tries to improve these unsatisfactory assignments, and especially to reduce the number of dreadful assignments. These phases will now be described in detail.

Phase 4, the full-width assignment of most of the remaining unassigned slots, is the only phase in which the three teacher assignment algorithms differ. The three alternative algorithms for Phase 4 are called the *slot oriented*, *time sweep*, and *resource packing* algorithms.

The slot oriented algorithm assigns teacher slots one by one. Sets of indistinguishable slots (slots with identical assigned times, workload, and type of resource required) are grouped together into *slot sets*. These occur quite frequently, since the KTS time assignment algorithm tries to encourage this kind of regularity. For each slot set, the algorithm keeps an approximate list of resources strictly acceptable to the slots of that slot set. Every strictly acceptable resource is in the list, but some of the resources may have become unacceptable recently, owing to complex and essentially unpredictable interactions within the global tixel matching, without the slot set becoming aware of this. A priority queue of slot sets is maintained, sorted so that the slot sets with the smallest *excess* (approximate number of strictly acceptable resources minus number of currently unassigned slots) are at the front of the queue, and, among slots of equal excess, wider slots precede narrower ones.

At each step the algorithm dequeues the slot set at the front of the queue, and tests each resource in its list for strict acceptability. If there are no strictly acceptable resources, the unassigned slots of the slot set are set aside for later phases to work on. If there is at least one strictly acceptable resource, the one which yields the best assignment is chosen and the assignment is made. The chosen resource's acceptability to all slot sets it is currently listed in is re-tested; being less available now, it may have become unacceptable to some of them. It is removed from any such slot sets, and their priority and place in the priority queue are updated. Finally, the selected slot set's priority is re-calculated and it is reinserted, unless all its slots are now assigned.

The time sweep algorithm sweeps through the times of the cycle in a particular order likely to work well. (As part of time assignment, KTS partitions the cycle into *columns*, which are sets of times that are preferred for assignment to meetings, and so are likely to occur together in many meetings. The order used is defined by taking each column in turn, and visiting its times in chronological order.) It mimics the kinds of algorithms frequently used with interval graphs; Carter [6] used a similar algorithm for room assignment in university course scheduling, but without the global tixel matching. As each time t is reached, a bipartite graph is constructed, with one demand node for each unassigned slot containing t, and one supply node for each resource available at t. An edge joins a slot to a resource whenever the corresponding assignment is strictly acceptable. The edge is weighted by an integer based on the usual cost vector, minus the slot width (to favour wider slots). A maximum matching of minimum weight is used to guide the assignment of resources to as many of these slots as possible, the rest being set aside to receive split assignments later.

Each edge in the matching graph represents one strictly acceptable assignment, but this does not guarantee that the set of assignments indicated by the maximum matching will be strictly acceptable all together, because as each is made it alters the global tixel matching. So the assignments are attempted widest slots first, and each is re-tested for strict acceptability before it is applied. Any slots not touched by the maximum matching, or whose assignment fails this test, are left until the assignments indicated by the matching have all been tried. Then the best available strictly acceptable assignment is made to each of these leftover slots, and those that fail to assign are set aside for later phases to work on.

Finally, the resource packing algorithm takes each resource in turn and *packs* it: makes as many strictly acceptable assignments as it can to it. The resources are stored in a priority queue. At each step, that resource is dequeued and packed which has the smallest *total available workload*: the sum, over all slot sets which are currently not completely assigned and for which assignment of the resource is (approximately) strictly acceptable, of the workload associated with one slot of the slot set. If this number is low, the resource is in danger of being underutilized, which is the rationale for giving it priority. Part-time teachers usually come first in this ordering, which is desirable since they are notoriously difficult to utilize effectively. When the last slot in a slot set is assigned, the slot set is deleted and all its acceptable resources are informed, causing them to reduce their total available workloads and update their priority queue positions accordingly.

Each resource is packed using a simple branch-and-bound tree search through the list of acceptable slot sets, aiming first to maximize the workload assigned to the resource, and second to minimize soft workload overloads. Several optimizations are implemented. The slot sets are sorted by decreasing width. Whenever a slot set is assigned, some previously acceptable slot sets may become unacceptable; these are removed from the list. If the list cannot supply enough workload to surpass the best solution found so far, the search backtracks. If the resource's workload limit is reached exactly without any soft workload overloads, the search ends early. As usual, only strictly acceptable assignments are permitted.

A full search could take exponential time, so each subtree is forced back on a simple first-fit heuristic after a fixed number L of nodes is visited. The value $L = 100$ was chosen for this paper and the public version of KTS, because preliminary experiments showed that running time was approximately proportional to L (as expected), but quality did not improve significantly as L was increased beyond this value.

Phase 5 of teacher assignment, identical for all three algorithms, tries to increase the number of full-width assignments, and to reduce the cost vector generally, by finding alternating paths of full-width deassignments and reassignments. Starting at each slot in turn, if the slot is currently unassigned the algorithm takes each qualified teacher in turn and tries to either assign that teacher directly, or else to identify a single slot which, when deassigned from that teacher, would permit the unassigned slot to be assigned to that teacher. If successful, it then continues trying to reassign the deassigned slot, and so on, marking each slot as visited to ensure that the search does not cycle. If the slot is currently assigned, the algorithm begins with the deassignment of that teacher. All this is repeated until there is no further improvement.

This is the augmenting path method from unweighted bipartite matching and similar algorithms, used in a context where the optimality guarantees that usually accompany it are absent. A similar algorithm was used previously by this author [10], but without the global tixel matching, and consequently its results were not then optimal with respect to the number of tixels assigned. In this incarnation, only paths that decrease the cost vector (and consequently do not increase the number of unassignable tixels) are applied. It has proved expedient, however, to follow all paths whose de-assignments and assignments produce no clashes or hard workload overloads, and only check the cost condition at the end of the path, since there are strictly acceptable augmenting paths with initial segments which are not strictly acceptable.

Phase 6 finds split assignments for the unassigned slots, again in the same way for all three algorithms. It is a small problem, as the results of Table 3 show, so presumably it could be solved to optimality, using integer programming for example. KTS uses a straightforward heuristic method of little interest. It continues to make only strictly acceptable assignments (but now to fragments of slots), and it assigns as many tixels as the global tixel matching permits, ensuring that, apart from the usual problems caused by unsplittable slots, all three teacher assignment algorithms are optimal with respect to the number of tixels assigned. It concludes with another invocation of the alternating path method, now operating over the fragments of slots introduced by this phase as well as the original full-width slots.

Finally, Phase 7 tries to improve the split assignments, using weighted bipartite matching in the manner that has become known recently as very large neighbourhood search. Take any set of times occupied by one resource within one split assignment. (The method works for any subset of the times of the cycle, but the subsets chosen are the only ones where there is any prospect of improvement.) Make one demand node and one supply node for every resource in that resource's resource group. The demand node represents whatever that resource is doing at these times (it could be free, or occupied with several meetings at different times), and the supply node represents the resource. Join a demand node to a supply node whenever the corresponding resource could do the duties of the demand node instead of whatever it is doing now, and weight the edge by the cost of this in terms of its effect on soft workload overloads and split assignments. Find a maximum matching of minimum weight and use it to reallocate the work at these times. In practice, it was found to be not worthwhile to include resources that attend two or more meetings during the times selected, since the chance of any change to their assignments being beneficial is very small.

6 Results

The algorithms were tested on six instances from Australian high schools. A summary of these instances appears in Table 1. The instances, solutions, and statistics presented here may be inspected via the Visitor account of the KTS web site [11].

The running time of the room assignment algorithm was never more than 1 second, and resource packing was never more than 2 seconds. The other two algorithms were similar except for one 4 second and one 7 second run.

In the instances tested, there are no unsplittable room slots, so the room assignment algorithm is optimal with respect to the number of tixels assigned. Over all instances tested, it produced exactly one split assignment.

Table 2 reports the number of teacher slots left unassigned by the three algorithms after Phase 4. Table 3 shows this same number after Phase 5, and Table 4 shows the final number of unsatisfactory teacher assignments (almost all of which are split assignments). Compared with the author's previous attempt [12], which was based on

Table 1 Statistical summary of the instances tested, showing their number of times, student groups, teachers, rooms, and meetings.

Instance	*Times*	*Students*	*Teachers*	*Rooms*	*Meetings*
BGHS93	40	23	53	46	155
BGHS95	40	27	52	48	147
BGHS98	40	30	56	45	152
SAHS96	60	20	43	36	131
TES98	30	11	33	20	95
TES99	30	13	37	26	86

Table 2 The number of unsatisfactory teacher assignments after Phase 4 of teacher assignment, shown as absolute numbers and as percentages of the total number of teacher slots, for the three teacher assignment algorithms. Instance *SAHS96* has been omitted, because its teacher slots are all preassigned.

Instance	*resource packing*	*slot oriented*	*time sweep*
BGHS93	28 (6.2%)	**24 (5.3%)**	36 (8.0%)
BGHS95	**28 (6.1%)**	37 (8.0%)	42 (9.1%)
BGHS98	**30 (6.8%)**	34 (7.7%)	45 (10.2%)
TES98	**11 (6.0%)**	**11 (6.0%)**	13 (7.1%)
TES99	**19 (10.8%)**	**19 (10.8%)**	**19 (10.8%)**

Table 3 Unsatisfactory teacher assignments after Phase 5 of teacher assignment.

Instance	*resource packing*	*slot oriented*	*time sweep*
BGHS93	**19 (4.2%)**	22 (4.9%)	21 (4.7%)
BGHS95	25 (5.4%)	**24 (5.2%)**	26 (5.6%)
BGHS98	**21 (4.8%)**	26 (5.9%)	26 (5.9%)
TES98	**11 (6.0%)**	**11 (6.0%)**	13 (7.1%)
TES99	**19 (10.8%)**	**19 (10.8%)**	**19 (10.8%)**

Table 4 Unsatisfactory teacher assignments after Phase 7 of teacher assignment.

Instance	*resource packing*	*slot oriented*	*time sweep*
BGHS93	**25 (5.6%)**	26 (5.8%)	30 (6.7%)
BGHS95	**29 (6.3%)**	30 (6.5%)	**29 (6.3%)**
BGHS98	**30 (6.8%)**	33 (7.5%)	35 (7.9%)
TES98	**11 (6.0%)**	**11 (6.0%)**	13 (7.1%)
TES99	**19 (10.8%)**	**19 (10.8%)**	**19 (10.8%)**

time assignments of similar quality to those used here, there is a small improvement in the number of unsatisfactory assignments, a small improvement in run time, and a major improvement in the fact that the present algorithms are optimal with respect to the number of tixels assigned (except near unsplittable slots): the previous algorithm offered no such guarantee, and in some instances left unassigned nine or ten tixels more than optimal.

Phase 7 occasionally removes a split assignment altogether, but its principal purpose is to reduce the number of dreadful assignments. Table 5 shows that it does

Table 5 The number of dreadful teacher assignments (split between three or more resources, or incomplete and split between two or more resources), before and after Phase 7 (resource packing only).

Instance	*Before Phase 7*	*After Phase 7*
BGHS93	7 (1.6%)	2 (0.4%)
BGHS95	6 (1.3%)	2 (0.4%)
BGHS98	6 (1.4%)	1 (0.2%)
TES98	2 (1.1%)	2 (1.1%)
TES99	0 (0.0%)	0 (0.0%)

Table 6 Soft workload limit overloads, shown as absolute numbers and as percentages of the total number of soft workload limits (usually one per teacher per day).

Instance	*resource packing*	*slot oriented*	*time sweep*
BGHS93	17 (6.4%)	**12 (4.5%)**	17 (6.4%)
BGHS95	**7 (2.7%)**	13 (5.0%)	8 (3.1%)
BGHS98	**13 (4.7%)**	16 (5.7%)	15 (5.4%)
TES98	4 (2.4%)	**2 (1.2%)**	**2 (1.2%)**
TES99	**9 (4.9%)**	**9 (4.9%)**	**9 (4.9%)**

this quite effectively, but it typically replaces one dreadful assignment and one satisfactory assignment by two split assignments, increasing the total number of split assignments. Table 6 reports on soft workload limit overloads.

The numbers of dreadful assignments and soft workload limit overloads seem to small enough to satisfy school management, but the acceptability of these numbers of split assignments is not known. Detailed examination of the *BGHS98* resource packing solution shows that only two unsatisfactory assignments are due to the time assignment failing to make enough qualified resources available at all times, but that deeper problems with time assignment are causing others. In several faculties the results are provably optimal for the given time assignment. Disproportionately many split assignments are concentrated in one faculty (the largest, combining English and History, with 14 teachers), which has 13 split assignments, mainly owing to the presence of two part-time teachers and many irregular time assignments.

7 Conclusion

This paper has presented several algorithms for resource assignment in high school timetabling. A key component of them all is an efficient implementation of the global tixel matching. This provides an important guarantee of optimality with respect to the number of tixels assigned, except near unsplittable slots, while supporting algorithms with run times of just a few seconds. The room assignment algorithm, which has been presented previously, gives virtually optimal results. The teacher assignment algorithms also perform well, and resource packing especially appears to be approaching a standard acceptable to high schools. It is used in the public release of KTS current at the time of writing [11].

Examination of actual timetables shows that split assignments cluster around heavily loaded faculties, part-time teachers, and meetings with irregular time assignments. The first two problems are inherent features of the instances, but it might be possible to improve the time assignments, which should reduce the number of split assignments. To achieve resource assignments of very high quality, some time assignments may need to be redone. That is likely to be slow, requiring a local search through the combined space of time and resource assignments, so the present work will remain valuable for producing a number of split assignments low enough to permit such methods to be applied to their repair.

References

1. Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin, Network Flows: Theory, Algorithms, and Applications. Prentice Hall (1993)
2. Edmund Burke and Michael Trick (eds.), Practice and Theory of Automated Timetabling V (Fifth International Conference, PATAT2004, Pittsburgh, August 2004, Selected Papers). Springer Lecture Notes in Computer Science 3616 (2005)
3. Edmund Burke and Hana Rudová (eds.), Practice and Theory of Automated Timetabling VI (Sixth International Conference, PATAT2006, Brno, Czech Republic, August 2004, Selected Papers). Springer Lecture Notes in Computer Science 3867 (2007)
4. M. W. Carter and C. A. Tovey, When is the classroom assignment problem hard? Operations Research, 40, S28–S39 (1992)
5. M. W. Carter and Gilbert Laporte, Recent developments in practical course timetabling, Practice and Theory of Automated Timetabling II (Second International Conference, PATAT'97, University of Toronto, August 1997, Selected Papers), Springer Lecture Notes in Computer Science 1408, 3–19 (1998)
6. Michael W. Carter, A comprehensive course timetabling and student scheduling system at the University of Waterloo, Practice and Theory of Automated Timetabling III (Third International Conference, PATAT2000, Konstanz, Germany, August 2000, Selected Papers), Springer Lecture Notes in Computer Science 2079, 64–81 (2001)
7. Tim B. Cooper and Jeffrey H. Kingston, A program for constructing high school timetables, Proceedings 1st International Conference on the Practice and Theory of Automated Timetabling, Edinburgh, UK (1995)
8. Peter de Haan, Ronald Landman, Gerhard Post, and Henri Ruizenaar, A four-phase approach to a timetabling problem in secondary schools, Proceedings 6th International Conference on the Practice and Theory of Automated Timetabling (PATAT2006), Brno, Czech Republic, 423–425 (August 2006)
9. Frank Jacobsen, Andreas Bortfeldt, and Hermann Gehring, Timetabling at German secondary schools: tabu search versus constraint programming, Proceedings 6th International Conference on the Practice and Theory of Automated Timetabling (PATAT2006), Brno, Czech Republic, 439–442 (August 2006)
10. Jeffrey H. Kingston, A tiling algorithm for high school timetabling, Practice and Theory of Automated Timetabling V (Fifth International Conference, PATAT2004, Pittsburgh, PA, August 2004, Selected Papers), Springer Lecture Notes in Computer Science 3616, 208–225 (2005)
11. Jeffrey H. Kingston, The KTS high school timetabling web site (Version 1.6), http://www.it.usyd.edu.au/˜jeff (October, 2007)
12. Jeffrey H. Kingston, The KTS high school timetabling system, Practice and Theory of Automated Timetabling VI (Sixth International Conference, PATAT2006, Brno, Czech Republic, August 2006, Selected Papers), Springer Lecture Notes in Computer Science 3867 (2007)
13. Jeffrey H. Kingston, Hierarchical timetable construction, Practice and Theory of Automated Timetabling VI (Sixth International Conference, PATAT2006, Brno, Czech Republic, August 2006, Selected Papers), Springer Lecture Notes in Computer Science 3867 (2007)
14. John M. Mulvey, A classroom/time assignment model, European Journal of Operational Research, 9, 64–70 (1982)