The Solution of Real Instances of the Timetabling Problem

TIM B. COOPER AND JEFFREY H. KINGSTON
Basser Department of Computer Science, The University of Sydney 2006, Australia

This paper describes a computer program for high school timetabling which has completely solved an instance taken without simplification from a large and tightly constrained high school. A timetable specification language allows the program to handle the many idiosyncratic constraints of such instances in a uniform way. New algorithms are introduced which attack the problem more intelligently than traditional search methods.

Received ...

1. INTRODUCTION

The basic timetabling problem is to assign times, teachers, students and rooms to a collection of classes and other meetings in such a way that none of the participants is required to attend two meetings simultaneously. Real instances can be very large, with hundreds of participants and hundreds of meetings squeezed into a week of about 40 meeting times; and they have additional requirements of various kinds.

As Lions [16] observed 25 years ago, the ultimate test of a computer-generated timetable is "Will the school buy it?". Computer programs have had some successes: the problem of assigning students to sections of university courses after the times are fixed has been solved, practically speaking [8, 14] and the literature does contain occasional reports of a school adopting a computer-generated timetable. However, the authors have been unable to find any report of a computer program for the full problem which is passing the test set by Lions regularly and reliably.

The most popular approach [1, 2, 4, 5, 9] has been to search through the assignments remaining to be made to find the one that is most tightly constrained, then to make that assignment in the way which constrains subsequent assignments as little as possible and repeat. Following the traditional hand method, there is often a second phase in which previous assignments are exchanged in an attempt to open up possibilities for further assignments.

This approach is widely reported to make 95–98% of the required assignments at best before getting stuck and one often sees a statement that the residual problems were or could be removed by manual adjustment. For example: "Very often, as many as six runs have been needed to obtain a solution for a single daily problem, with the average number of runs being about two or three. Frequently analyst intervention to correct one problem was frustrated by the emergence of a new problem. .... There occurs a point at which it is better to turn the problem over to the school for the final touches" [16].

The literature also contains a number of more theoretical papers. Gotlieb [12] proposed a method based on detecting some subtle forced assignments and making a single arbitrary assignment when there are no forced ones. Csima [6] seems to have been the first to find a solution for an almost realistic special case which could be implemented in polynomial time. Even *et al.* [7] demonstrated this implementation and proved the general problem to be NP-hard, although graph colouring, which is well known to be closely related to timetabling [19], was proved NP-hard earlier by Garey *et al.* [10]. Several papers by de Werra [20, 21] propose network flow solutions to various subproblems and are closest to our own work. Kitagawa and Ikeda [13] have attempted to generalize and unify the many special requirements of real instances. The bibliography by Schmidt and Strohlein [18] has a clear account of developments up to 1979.

If a program is ever to solve timetabling problems regularly and reliably, it must produce 100% solutions, and it must be portable between institutions, implying that all special requirements must be notated in the input, not fixed within the program. The Sydney timetabling project is an attempt at building such a program; this paper describes our progress to date. We have developed a language for formally expressing real instances of the timetabling problem, and a novel program incorporating heuristic, network flow and tree searching techniques, which we call TT1. This program has completely solved one real instance, taken without simplification from a large high school in the Sydney area. We will refer to this instance as BGHS.

2. THE LANGUAGE AND PROBLEM SPECIFICATION

Real instances of the timetabling problem abound in peculiarities: teachers available only on certain days, Science laboratories which can also be used as ordinary classrooms, double periods (where a class occupies two adjacent time periods) and so on. As already mentioned,

such peculiarities must be notated in the input, so we have introduced a simple language, which we call TTL1, for specifying instances. In this section we define TTL1 and show how most of the requirements of real instances can be specified within it.

One puzzling aspect of the literature to date is the universal assumption that individual teachers will be preassigned to classes before the program takes over. Not only does this place a heavy burden on the human timetabler, it reduces the choices open to the program and hence its chance of finding a solution. TTL1 permits preassignments, but it does not require them.

A TTL1 instance consists of *groups* and *meetings*. Groups contain *elements* which are the participants in the timetable, and *subgroups* which indicate the functions that the elements perform. There may be arbitrarily many groups, each with arbitrarily many elements and subgroups.

In high school timetabling the groups are normally *Teachers*, *Students*, *Rooms*, and *Times*. For example,

group *Teachers* **is**

subgroups *English*, *Science*, *Computing*;

Smith **in** *English*, *Computing*;
Jones **in** *Science*, *Computing*;
Robinson **in** *English*;

end *Teachers*;

specifies a *Teachers* group with three elements (*Smith*, *Jones* and *Robinson*), two of which can perform the *English* function (teach English) and so on. Similarly,

group *Rooms* **is**

subgroups *Ordinary*, *ScienceLab*, *MusicRoom*;

*R*1 **in** *Ordinary*;
*L*1 **in** *ScienceLab*, *Ordinary*;
*L*2 **in** *ScienceLab*, *Ordinary*;
*M*1 **in** *MusicRoom*;

end *Rooms*;

expresses the requirement mentioned earlier that Science laboratories be available as ordinary classrooms. It would be simpler if each element had just one function, but overlapping functions are extremely common. They can be very confusing for the human timetabler, but, as we will see below, a computer program can use bipartite matching to solve the problems they pose very quickly and elegantly.

In typical instances the *Times* and *Students* groups simply name the available times and student forms with no subgroup structure. If the subject choices of individual students are to be taken into account, the *Students* group would list all the students, with subgroups corresponding to the subjects available.

These groups make it clear that the roles played by teachers, students and rooms are essentially the same, an observation which goes back to Appleby *et al.* [1] at least. Our program accordingly treats all these groups in the same manner, using the same code. Although the *Times* group has the same syntax as the others, it is distinguished by the fundamental requirement that no element of any group may attend two meetings at the same time and it seems that the implementation must treat *Times* as a special case. Elements of the special *Times* group will be called *times*; elements of the other groups will be called *resources*.

Meetings are collections of *slots* which are to be assigned elements of the various groups. For example, the following is a typical meeting from the BGHS instance:

meeting 10-*English* **is**
  from *Students* **select** *Year*10;
  from *Teachers.JunEnglish* **select** 5;
  from *Rooms.Large* **select** 5;
  from *Times* **select** 6;
end 10-*English*;

It contains one slot which must be filled by the *Year*10 resource from the *Students* group (representing all Year 10 students); five slots to be filled by resources from the *JunEnglish* subgroup of the *Teachers* group; five slots to be filled by large rooms; and six time slots. There may be any number of meetings, each containing any number of selections from any group or subgroup. Each selection may be a preassignment of a particular element, or it may be an integer, or it may be **all**, meaning every element.

Formally, the meaning is that the selected resources will all be occupied together for the selected times. In the case of the meeting above, we understand that the Year 10 students will be split into five groups, each meeting for six times with one of the five teachers in one of the five rooms; the meeting is an aggregation of 30 smaller meetings. It would be possible to disaggregate them to some extent, using five meetings of the form

meeting 10-*EnglishA* **is**
  from *Students* **select** *Year*10*A*;
  from *Teachers.JunEnglish* **select** 1;
  from *Rooms.Large* **select** 1;
  from *Times* **select** 6;
end *10-EnglishA*;

The major difference is that these five meetings are no longer constrained to run concurrently. A further disaggregation of each of these into six parts would fail to specify that the same teacher is required for all six times. In general, larger meetings lead to fewer meetings, a smaller solution space and less chance of success; smaller meetings are less constrained but may yield a chaotic timetable.

There are some subtle requirements related to aggregation. For example, the 10-*English* meeting above specifies that the same five rooms be used at all six times, but this is not required in fact. Another requirement, characteristic of university timetables, is that some classes are

offered at several different times and each student is allocated to one of those times. We have not addressed these requirements in TTL1.

We have found that TTL1 can express most of the requirements of high school timetabling, with a little ingenuity. For example, to say that a teacher may teach for no more than 30 of the 40 available times, we create a meeting occupying that teacher for 10 times:

meeting *SmithFree* **is**
  from *Teachers* **select** *Smith*;
  from *Times* **select** 10;
end *SmithFree*;

By selecting from a subgroup of *Times* we can constrain the choice of free times if we wish. The only omission at present (as far as high school timetabling is concerned) is control of the spread of classes through the week, including double periods, avoiding giving one class the last time in each day or too many times in the one day and so on.

Our timetabling problem is NP-hard in at least two different ways. First, the problem of avoiding clashes is well known to be a version of graph colouring, assuming that arbitrary selections are permitted [19]. Second, the allocation of teachers to classes in such a way that no teacher is overloaded is easily seen to be a version of bin packing, where each teacher is one bin. A polynomial solution exists for the special case where each meeting selects exactly one nominated class, one nominated teacher, and any number of times, the latter not pre-assigned [7].

The BGHS instance, which is the only one we have tested our program on thus far, contains four groups: *Times*, *Students*, *Teachers* and *Rooms*, with 40, 22, 53 and 46 elements, respectively. For each teacher there is a meeting like *SmithFree* above; there are meetings of the various faculties (English, Mathematics, etc.) one time per week; and there are 152 other meetings representing ordinary classes. Most of these meetings are aggregates, like 10-*English* above. Some aggregates express *electives*, where a decision has been made in advance that certain meetings will be run concurrently and students advised to choose one; others express *runarounds*, where the classes of 1 year cycle through scarce resources such as the unique Technics room and teacher. The latter could easily be disaggregated, but we see no harm in helping the program along a little by encoding solutions to small subproblems, following school practice.

3. INVARIANTS

The postcondition of the timetabling problem may be expressed as

> *All slots are filled and no resource is assigned to two meetings which share a time.*

It is natural to take the second part of this condition as invariant and assign times and resources to slots until the first part holds. We call this fundamental invariant the *exclusion invariant*.[1]

Our program maintains a rather subtle generalization of the exclusion invariant that we call the *resource sufficiency invariant*:

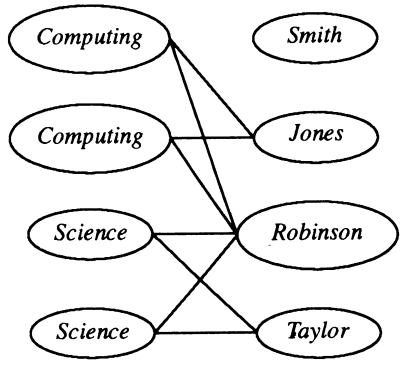
> *For each time* t, *the total resource requirements of the meetings scheduled for time* t *can be filled by the available resources.*

This is an extremely important invariant, which can be applied to many algorithms, and is used throughout our program. It allows the program to assign times to meetings, keeping within resource limits; and it can take into account the subgroup structure (e.g. the fact that a teacher may teach both English and History) without actually committing any resource to any slot. It strongly constrains the search path without ruling out any solutions.

In the unrealistic special case where each resource lies in exactly one subgroup, we may evaluate this invariant at time t by comparing the supply of resources in each subgroup with the demand for elements of that subgroup. However, in general, correct evaluation amounts to finding a matching in a bipartite graph, as we now show by example. Given the resources

Smith **in** *Mathematics*;
Jones **in** *Mathematics*, *Computing*;
Robinson **in** *Science*, *Computing*;
Taylor **in** *Science*;

suppose we ask whether a requirement for two *Computing* and two *Science* teachers at some time t can be met. The bipartite graph has one left-hand node for each resource slot, and one right-hand node for each resource. Edges join each resource slot node to the resources in its subgroup:



The invariant holds at time t if this graph has a *matching*, defined here as a subset of the edges such that each slot node is incident to exactly one edge, and each resource node is incident to at most one edge. There is no

[1] The opposite approach, taking the first part as invariant and exchanging assignments until the second part holds, has also been tried [3].

matching in this example, despite the presence of two Science and two Computing teachers.

Efficient algorithms for bipartite matching are well-known [17]. Most importantly, one can insert and delete nodes and edges, modifying a previous matching to reflect the changes, in linear time per change and usually much faster. This provides a quick test of the feasibility of assigning a particular time t to a meeting: add the meeting's resource requirements to a graph permanently associated with time t and attempt to augment the existing matching. Notice that although the matching defines an assignment of resources, that assignment is not actually made.

The remainder of this section is devoted to two invariants which address issues quite unrelated to resource sufficiency. Although these two invariants are not maintained at present by the TT1 program, we hope to add them in the future. In general, we contend that the addition of invariants is an important way to improve timetabling algorithms.

The essence of the resource sufficiency invariant is to check whether certain meetings can occur simultaneously; the particular times chosen are irrelevant to that invariant. We speak of a decision to run two meetings simultaneously as an assignment to those meetings of a common *prototime*; an actual time must be assigned separately to the prototime.

The assignment of actual times to prototimes is naturally done last, since it is usually only slightly constrained. However, some meetings do specify particular actual times: Sport on Thursday afternoons, or a meeting between a teacher and all the Monday times, included to leave that teacher free on Mondays. To avoid assigning prototimes in a way inconsistent with these actual time assignments, a *time consistency invariant* is needed:

> *There exists an assignment of actual times to prototimes consistent with all restrictions on time slots.*

This invariant would prevent the assignment of different prototimes to two time slots which were preassigned the same actual time and also the assignment of a common prototime to two time slots which were preassigned different actual times. A bipartite matching from prototimes to actual times may be used to evaluate this invariant. The nodes are fixed and edges are deleted as prototimes are assigned to time slots which constrain the actual times.

We have not implemented this invariant, because the identity mapping between real times and prototimes was sufficient for BGHS. However, it will be the foundation of future work on the spread of classes through the week.

Our second example of an invariant whose maintenance would improve the TT1 program is motivated by the following situation, which arose during the solution of BGHS. One time slot of a Drama meeting was assigned the same prototime as the English faculty meeting (attended by all English teachers), while another time slot was assigned the same prototime as the History faculty meeting. Unfortunately, every Drama teacher had to attend one of these other meetings, hence no single suitable teacher was available at all the times. This did not contradict the resource sufficiency invariant; resources were sufficient at each individual time. It would be possible to prevent this with a *resource constancy invariant*:

> *For each resource slot there exists a single resource which may be assigned to that slot for all the required times, without violating the resource sufficiency invariant at those times.*

One might hope to assert the existence of such assignments to all resource slots simultaneously; but that invariant encompasses the NP-hard bin-packing problem, so it cannot be evaluated efficiently. We do, in fact, partly implement this invariant in one part of our algorithm (Section 4.3).

4. ALGORITHMS

Below are the algorithms we tried and the ideas out of which our final solution is built. Note that the relationship they bear to each other is important, but they are all fairly general-purpose ideas that can be used in isolation as well.

4.1. Brute force assignment

This well-known algorithm explores all possible assignments to a nominated set of slots, which may include both time and resource slots. The only pruning of the search tree is that carried out by consulting the invariants.

Brute force search is useless as a general-purpose algorithm, but we still use it in one part of our program. We look for small, independent subproblems where it may be applied, such as the assignment of Science laboratories to Science classes whose times have been fixed.

4.2. Most-constrained assignment

In this very popular method, first the unassigned slot with the fewest available choices of assignment is found, then heuristics are used to select the most appropriate element for that slot, it is assigned and the method repeats. If a slot with no available elements is encountered, an attempt is made to remedy this by making a few exchanges of previous assignments. After this the slot is either assigned or abandoned and the method repeats.

This algorithm was the second to be implemented by the authors, who had some hopes of solving the entire problem with it. Our heuristic for deciding which slot to fill next was based on comparing the demand for the resources of each subgroup with their supply at each time (an idea which later evolved into the resource sufficiency invariant). When no element was free to fill some slot, we would perform an exhaustive search for a

chain of assignments and deassignments which would solve the problem. Owing to the combinatorial explosion it was not possible to explore chains longer than about five steps.

Today, we can find nothing positive to say about this algorithm. Blind to the difficult subproblems lying in its path, chaotic in its patterns of assignments, it stumbles against an impenetrable wall after about 90–95% of the assignments have been made. Its attempts to bypass the obstacles by exchanges are hopelessly ineffective, because it does not understand their nature.

4.3. Forced assignment

Forced assignments arise when previous assignments bring the system to a state where only one element may be assigned to some slot without violating any invariant. It is natural to make this assignment immediately, and doubtless many disasters are averted by doing so.

Forced assignments can be detected by assigning each possible element in turn to a slot and checking the invariants. When only one element is permitted, that is a forced assignment. Lions [15] found the forced assignments of Gotlieb's method [12] in this way, using bipartite graph matching for an efficient check.

Forced assignments can be implemented easily within brute force search, by placing them in the main line of assignments immediately after the assignment which caused them and backtracking over them without branching. Our program encounters forced assignments only during initialization and resource assignment, where no assignments are ever undone. This is important because, if care is not taken, a forced assignment might persist after its cause has been removed.

4.4. Meta-matching assignment

The failure of most-constrained assignment prompted a major re-think. We wanted algorithms which understood the nature of the obstacles to be overcome. *Meta-matching*, the first fruit of this reorientation, is a highly effective algorithm for assigning prototimes to meetings.

A set of meetings will be called a *time-disjoint meeting-set* if no two of the meetings may occur simultaneously. For example, the set of meetings attended by the Year 12 students must be time-disjoint, because these students cannot attend two meetings at the same time. Appleby *et al.* [1] pointed out some useful properties of these sets and used them in invariant checking.

Meta-matching is an efficient method for finding an assignment of prototimes to all the time slots of a time-disjoint meeting-set, consistent with the resource sufficiency invariant. It constructs a bipartite graph whose left-hand nodes are the time slots and whose right-hand nodes are all the prototimes (one for each actual time). Then it enquires of the resource sufficiency invariant whether the first prototime may be assigned to the first time slot of the first meeting. As described in Section 3, the resource sufficiency invariant answers this by adding a node for each resource slot of the first meeting to the bipartite resource graph permanently associated with the first prototime and attempting to augment the existing matching. If this assignment can be made, then an edge is drawn connecting the two nodes.

This process is repeated for each slot–prototime pair. If there are 40 times in the week and 40 meetings of 1 h, there would be $40 \times 40 = 1600$ invariant checks. In practice, however, the time slots of any one meeting are usually indistinguishable and this reduces the number of checks to a few hundred at most.

A matching of the resulting bipartite graph yields an assignment of prototimes to time slots. We call this a meta-matching because the matching is performed on a graph whose edges represent resource matchings.

Each prototime receives at most one assignment, so the resource sufficiency invariant must hold after all the assignments are made. Problems with other invariants can be prevented to some extent by testing each edge against them before including it in the graph.

Meta-matching is a generalization of a network model of de Werra [21], whose resource selections were limited to a single nominated class and teacher per meeting. It was invented independently by the present authors. We point out that the resource matchings could return a numerical quality measure, in which case the meta-matching would become a weighted matching. The method can be applied to any set of meetings, but since it yields only time-disjoint assignments, we reserve its use to time-disjoint meetings.

Although the assignment of times to meetings by meta-matching is a major step towards a solution, substantial problems remain. Pure meta-matching can produce a chaotic timetable and resources must still be assigned to slots. These problems are addressed in the following sections.

4.5. Time-coherent assignment

If meta-matching succeeds in finding one assignment of prototimes to meetings, there are usually many alternative assignments. Other assignment algorithms also have this property, and so the question arises of whether there is any reason to choose one assignment over another.

Heuristically, it seems clear that the assignment of prototimes to meetings should be *time-coherent*, i.e. meetings should either overlap completely in time, or not at all, as far as possible. A visual analogy is the idea of filling a box with blocks: more blocks will fit if they are lined up neatly rather than thrown in randomly. We have developed an algorithm for time-coherent assignment, and used it in conjunction with both most-constrained assignment and meta-matching.

The meta-matching version proceeds as follows. First, we check for the existence of a meta-matching in the usual way. Then, for each meeting of the time-disjoint set in turn (largest first), we search for a set of prototimes to assign to that meeting (briefly, a time-set) with the

following properties: (i) the time-set is permitted by the resource sufficiency and other invariants; (ii) if the time-set is assigned to the meeting, a meta-matching for the remaining unassigned meetings of the time-disjoint set can still be found; and (iii) the time-set is time-coherent in the following sense. Our strategy is to find a time-set satisfying (i) and (ii) which is a subset of (and preferably equal to) the union of the fewest possible number of existing time-sets. What this means in practice is that we usually find an existing time-set, used by many other meetings, to assign to this new meeting; failing that we try to break up a larger meeting or fuse together two or more smaller meetings; and failing anything of this sort (e.g. before any meetings have been assigned a time-set), we take the times generated arbitrarily by the meta-matching.

This method works very well in practice. An efficient implementation using bit vectors of prototimes ensures that the time complexity is proportional to the number of subsets of the set of previous time-sets tried, and pruning nearly always halts the search after pairs of existing time-sets at worst, since triples rate very poorly in any case. Continual reference to the meta-matching ensures that an assignment will be found if one exists at all and inspection of the output confirms that a highly time-coherent assignment usually results.

This algorithm was originally applied to most-constrained assignment, before meta-matching was developed, and it performed extremely well there (Section 6). Incidentally, there are pathological cases where enforcing time-coherence actually rules out all solutions (Appendix).

4.6. Beam-search set covering

We now present our method of assigning resources to meetings, assuming that all time slots have been assigned a prototime. Note that it is not necessary to assign actual times to prototimes before doing this: a resource may safely be assigned to two or more meetings if those meetings do not share a prototime.

If all our meetings contained only one time slot, the resource sufficiency invariant would trivially provide us with an assignment of resources to slots and our task would be over. However, in general, a meeting has several times and each of its resource slots must be occupied by the same resource at all times. For example, a Science meeting must be assigned the same Science teacher at all of its times, not a different teacher at each time. We call this the *resource constancy requirement*; it is related to the resource constancy invariant discussed in Section 3.

Even though we assume that the resource sufficiency invariant holds, or in other words that sufficient resources are available at each individual prototime, the resource constancy requirement makes resource assignment an NP-hard problem, related to bin packing and set covering.

Teacher constancy is a major practical problem in BGHS, because the teachers are utilized right to the limit imposed by their terms of employment (30 times per week). If, for example, all the meetings open to a teacher have exactly four time slots, then any assignment must leave two of the 30 times unused, since 4 does not evenly divide 30, and this wastage alone might render the instance provably unsolvable. The school solves this problem by having some meetings with one or two time slots that many teachers are qualified for (e.g. Sport) and also by violating teacher constancy in some junior classes.

Constancy is not a problem with the other resource groups. Room constancy is an artificial requirement imposed by TTL1, not by the school. Student constancy is trivial since the student resources are all preassigned. Incidentally, resource assignment can be carried out independently for each resource group once prototimes are assigned.

The difficulty that we anticipated in satisfying the teacher constancy requirement led us to view the resource assignment problem as one of utilizing each resource as completely as possible. Our algorithm, which we call beam-search set covering, accordingly takes one resource and assigns it to meetings which occupy as many times as possible.

We first find the set of all meetings to which the resource in question could be assigned. To qualify, a meeting must have an unassigned resource slot of an appropriate type and its prototimes must not include any when the resource is already assigned elsewhere (from preassignments or forced assignments). More generally, it must be possible to assign the resource to the slot without violating the resource sufficiency invariant at any prototime.

The problem now is to choose a disjoint subset of these meetings, including any meetings to which this resource is already assigned, having as many times as possible. This is easily seen to be equivalent to the NP-hard exact covering problem [11, p. 221]: for each prototime we have one set whose elements are the names of the meetings to which that prototime is assigned and we wish to choose a collection of these meetings that exactly covers all the prototimes.

We solve this problem approximately with a beam search of possible coverings i.e. a tree search with the number of active nodes at any moment restricted to the k most likely, for a small constant k. A variety of tree pruning rules is used: we discard meetings whose prototimes intersect with the prototimes of chosen meetings; we check that the remaining unassigned meetings contain enough times to improve on the current best; and we check that the sizes of these meetings are not too large to rule out such an improvement.

5. THE TT1 PROGRAM

This section describes the high-level structure of the TT1 program.

The first step is to read the input, set up the data structures and insert any preassignments. Failure to do a preassignment indicates that the instance has no solution.

Also in our initialization stage, we need to create the time-disjoint meeting-sets. If we have a set of meetings all preassigned the same resource, they must be a time-disjoint set. More generally, a set of meetings is time-disjoint if each one selects a number of elements from some resource group greater than half the size of the subgroup. This subsumes the previous case, since selecting a single nominated resource can be viewed as selecting every element of a subgroup of size 1. Any meetings not represented in any of these time-disjoint meeting-sets are put into sets by themselves.

The next step is to apply the time-coherent meta-matching algorithm to each time-disjoint meeting-set in turn, from largest to smallest. If the meta-matching succeeds, we make the indicated assignments of prototimes to time slots and proceed to the next largest time-disjoint set.

If the meta-matching fails, we try to bypass the problem as follows. First, we deassign one previously matched set chosen at random and try again to match the problem set. If this succeeds, we rematch the deassigned set, and if that succeeds we can continue. If it fails, we try the same procedure again using a different randomly chosen matched set and so on. If no single set deassignment solves the problem, we try all pairs of deassignments in a random order, then triples and so on.

Towards the end of a hard problem, we may see the same meeting-sets being assigned and deassigned several times. This is not evidence of going around in circles, because each time the meetings are assigned different (but always time-coherent) times.

If this procedure terminates successfully, the result is an assignment of prototimes to all the meetings which satisfy the resource sufficiency invariant. The next step is to assign resources.

If a resource subgroup (e.g. the Science laboratories) contains less than six elements, it is feasible to apply brute force assignment to the set of all slots which select that subgroup. This method is ideal when the subgroup is completely isolated (its elements are not lying in any other subgroups), but we use it on partly isolated subgroups as well.

The remaining resources are assigned by applying beam-search set covering to each in turn. After each is assigned to its set of meetings, we check for and make forced assignments of other resources.

Some resource slots may remain unassigned at the end of this step. Since the resource sufficiency invariant is maintained throughout, resources are available for all these slots; however, the resource constancy requirement cannot now be satisfied at any of them.

Finally, we assign real times to prototimes. TT1 does not address the issue of the spread of classes through the week, so at present we just take the identity mapping.

6. RESULTS

In this section we report the results of our programs. Our tests to date have been confined to the BGHS instance. Although other instances must be solved before we can claim the portability between institutions that is our goal, BGHS is an excellent test, being large, real and populated with a variety of tightly constrained subproblems.

Our implementation of the most-constrained method assigned 90% of all slots, rising to 92% when exchanges were added. Exchanges are relatively ineffective in the presence of large aggregate meetings. We tried several ideas but were unable to make any improvement until we added time-coherent assignment. The result was a dramatic improvement to 99% and in fact only four slots were left unassigned. We do not regard this as success, however, because the problem of completing the timetable from that point appeared to us to be intractable.

Meta-matching without time-coherent assignment assigned all but four meeting-sets, representing about 85% of the prototime slots, and would have immediately produced better results had we allowed it to split up meeting-sets, avoiding the deassignment of an entire meeting-set for the sake of one meeting. This result should be rated more highly than the corresponding 90% result from the most-constrained method, because the resource sufficiency invariant was maintained.

When time-coherent assignment was added, the 100% mark was reached in under 3 min of CPU time on a MIPS computer. Given the size of the instance, the heavy utilization of teachers and some special-purpose rooms, and the previous failure of most-constrained assignment with exchanges, we regard this as a major achievement. Relatively little deassignment occurred, and the deassignment of pairs of time-disjoint sets was quite rare.

The program then proceeded to find an assignment of resources which satisfied the resource constancy requirement at 95% of the teacher slots and 91% of the room slots, taking 9 min of MIPS CPU time. The resource sufficiency invariant was maintained throughout, so these results could be improved immediately to 100% by violating resource constancy. Room constancy is an artificial requirement in any case (imposed by TTL1, not by the school), and a small number of violations of teacher constancy is acceptable, and possibly inevitable (the school's solution has a similar level of teacher constancy). We hope to improve on this in the future; meanwhile the result is an acceptable timetable.

7. CONCLUSION

Despite the inherent intractability of the high school timetabling problem, we have developed a program which is capable of solving it in practice. Most notable are the use of bipartite matching to check the resource

sufficiency invariant, the time-coherent meta-matching algorithm for assigning prototimes to meetings and the beam-search set covering algorithm for resource assignment. We have also created a specification language which is able to express the complexities of real instances while itself remaining quite simple.

Efficiency is naturally an important consideration when dealing with NP-hard problems. However, we found it important to concentrate on polynomial-time algorithms that were not of too high degree, and exponential algorithms that were still feasible, rather than spend time optimising the code for constant-factor increases.

As it stands, the TTL1 language does not permit any expression of preferences, such as 'Room 21 is less desirable than the others' or 'Smith prefers to teach Economics'. One way to incorporate such preferences into the algorithms would be simply to always try the most preferred option first. A fully developed system of numerical preferences is also conceivable, provided it does not obscure the fact that with many constraints, a timetable is either acceptable or not acceptable.

We have a design for an extension to TTL1 for expressing the spread of classes through the week, which awaits implementation. After that it will be time to take our solution back to the high school for comment and to seek out other real instances.

In the longer term, we hope that the program will evolve until it can be used routinely to solve most instances of high school timetabling. We also plan to investigate the problem of assigning resources to one of several equivalent meetings (e.g. when a student has a choice of several laboratory times), with a view to extending our language and program to university timetabling.

We conclude with a lesson we have learnt about assignment algorithms that seems to us to be important. An assignment algorithm which is unaware of a tightly constrained subproblem lying in its path will often fail to solve that subproblem. However, an assignment algorithm specifically designed to solve that subproblem can actually benefit from the tight constraints it imposes. For example, the need for Year 12 meetings to be time-disjoint places a strong constraint on those meetings, which meta-matching uses to rapidly generate assignments for the entire set. Similarly, the need to utilize all teachers completely is a daunting obstacle, until set covering methods convert it into a powerful heuristic.

Acknowledgements

We thank Thomas Lapins-Silvirs for assistance with the literature search and Kandia Tagalakis for the BGHS instance data.

REFERENCES

[1] J. S. Appleby, D. V. Blake and E. A. Newman, Techniques for producing school timetables on a computer and their application to other scheduling problems. *The Computer Journal*, **3**, pp. 237–245 (1960).
[2] J. Aubin and J. A. Ferland, A large scale timetabling problem. *Computers and Operations Research*, **16**, pp. 67–77 (1989).
[3] R. J. Aust. An improvement algorithm for school timetabling. *The Computer Journal*, **19**, pp. 339–343 (1976).
[4] E. D. Barraclough, The application of a digital computer to the construction of timetables. *The Computer Journal*, **8**, pp. 136–146 (1965).
[5] J. Berghuis, A. J. van der Heiden and R. Bakker, The preparation of school time tables by electronic computer. BIT, **4**, pp. 106–114 (1964).
[6] J. Csima, *Investigations on a Time-Table Problem*. PhD thesis, School of Graduate Studies, University of Toronto (1965).
[7] S. Even, A. Itai and A. Shamir. On the complexity of timetable and multicommodity flow problems. *SIAM Journal on Computing*, **5**, pp. 691–703 (1976).
[8] J. A. Ferland and S. Roy, Timetabling problem for university as assignment of activities to resources. *Computers and Operations Research*, **12**, pp. 207–218 (1985).
[9] O. B. de Gans, A computer timetabling system for secondary schools in the Netherlands. *European Journal of Operational Research*, **7**, pp. 175–182 (1981).
[10] M. R. Garey, D. S. Johnson and L. Stockmeyer, Some simplified NP-complete problems. *Proceedings of the 6th Annual ACM Symposium on Theory of Computing*, 1974, 47–63. Also *Theoretical Computer Science*, **1**, 237–267 (1976).
[11] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, San Francisco (1979).
[12] C. C. Gotlieb, The construction of class-teacher timetables. *Proceedings of the IFIP Congress*, pp. 73–77 (1962).
[13] F. Kitagawa and H. Ikeda, An existential problem of a weight-controlled subset and its application to school timetable construction. *Discrete Mathematics*, **72**, pp. 195–211 (1988).
[14] G. Laporte and S. Desroches, The problem of assigning students to course sections in a large engineering school. *Computers and Operations Research*, **13**, pp. 387–394 (1986).
[15] J. Lions, Matrix reduction using the Hungarian method for the generation of school timetables. *Communications of the ACM*, **9**, pp. 319–354 (1966).
[16] J. Lions, The Ontario school scheduling program. *The Computer Journal*, **10**, pp. 14–21 (1967).
[17] C. H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, New York (1982).
[18] G. Schmidt and T. Strohlein, Timetable construction—an annotated bibliography. *The Computer Journal*, **23**, pp. 307–316 (1980).
[19] D. J. A. Welsh and M. B. Powell. An upper bound for the chromatic number of a graph and its application to timetabling problems. *The Computer Journal*, **10**, pp. 85–86 (1967).
[20] D. de Werra, Construction of school timetables by flow methods. *INFOR—Canadian Journal of Operations Research and Information Processing*, **9**, pp. 12–22 (1971).
[21] D. de Werra, An introduction to timetabling. *European Journal of Operational Research*, **19**, pp. 151–162 (1981).

APPENDIX

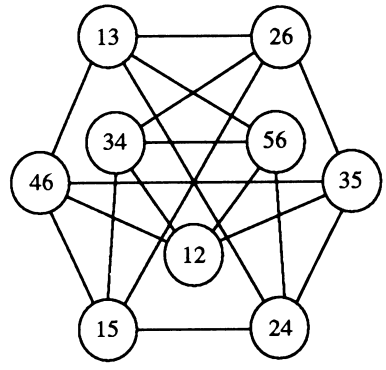
Time-coherence

We show here that enforcing time-coherence may exclude all solutions in some instances. We do this by exhibiting

a TTL1 instance with the following properties: (i) all meetings have the same number of time slots, and this number evenly divides the number of elements in the *Times* group (ii) a solution to the instance exists and (iii) no fully time-coherent solution exists.

Let the *Times* group have six times, called 1 to 6. Let the meetings be called *ab* for all *a* and *b* such that $1 \leq a < b \leq 6$; there are 15 such meetings. Let each meeting have two time slots, establishing condition (i). For all pairs of meetings *ab* and *cd* such that $\{a,b\} \cap \{c,d\} = \varnothing$, let both meetings select a resource constant called *abcd*. This ensures that these pairs of meetings will be time-disjoint.

This instance can be solved by assigning times *a* and *b* to meeting *ab* for all meetings; but this solution is not time-coherent. A fully time-coherent solution would be equivalent to a three-colouring of a graph which has one node for each meeting and an edge between each pair of meetings which select the same resource constant. It is easy to check that no such three-colouring exists and in fact the nine-element induced subgraph



has no three-colouring. This construction is due to Dr C. D. H. Cooper (private communication).