



The University of Sydney

A Program for Constructing High School Timetables

Technical Report Number 496

March, 1995

Tim B Cooper and Jeffrey H Kingston

ISBN 0 86758 960 4

**Basser Department of Computer Science
University of Sydney NSW 2006**

A Program for Constructing High School Timetables

Tim B. Cooper and Jeffrey H. Kingston

Basser Department of Computer Science
The University of Sydney 2006
Australia

6 March, 1995

Abstract

This paper describes TT2, a computer program for high school timetable construction. The program is a combination of tree search and techniques from operations research (matchings and flows). Two instances taken without simplification from real high schools have been solved.

Keywords: timetable construction, specification, implementation, operations research.

Contact: Dr. Jeffrey H. Kingston, email *jeff@cs.su.oz.au*.

1. Introduction

The high school timetable construction problem is to assign times, teachers, students, and rooms to a collection of meetings so that none of the participants is required to attend two meetings simultaneously. This basic requirement is often accompanied by others, such as that the times of a meeting should be spread evenly through the week, and so on. The problem is well known to be NP-hard [4].

This paper describes the specification language that we use for describing real instances of the timetabling problem, our current high school timetable construction program (which we call TT2), and the results we have obtained with it. Our approach may be characterized roughly as tree search, with move generation and pruning rules based on techniques adapted from operations research.

Many approaches have been tried, and the reader is referred to the annotated bibliography of Schmidt and Ströhlein [7] and the more recent bibliographies available on the Internet, such as [1]. We can mention here only work directly related to our own.

The first application of operations research techniques to timetable construction seems to have been by Csima [3], who was able to solve a special case in polynomial time. His methods were championed by Lions [5, 6]. The work of de Werra [8, 9] contains several network models, one of which has found its way into our work. Our search techniques, primarily beam search, are standard in the artificial intelligence literature [10].

2. Specification

In this section we present a specification of the timetable construction problem based on a *timetable specification language* called TTL. This language is formal yet flexible enough to

specify instances encountered in practice. An earlier version of TTL appeared in [2].

A TTL instance consists of a *time group*, some *resource groups*, and some *meetings*. Here is a typical time group:

timegroup *Times* **is**

Mon1; Mon2; Mon3; Mon4; Mon5; Mon6; Mon7; Mon8;
Tue1; Tue2; Tue3; Tue4; Tue5; Tue6; Tue7; Tue8;
Wed1; Wed2; Wed3; Wed4; Wed5; Wed6; Wed7; Wed8;
Thu1; Thu2; Thu3; Thu4; Thu5; Thu6; Thu7; Thu8;
Fri1; Fri2; Fri3; Fri4; Fri5; Fri6; Fri7; Fri8;

"[. . . : . . . ,][. . . : . . . ,][. . . : . . . ,][. . . : . . . ,][. . . : . . . ,]"

end *Times*;

It lists the names of the times available for meetings, followed by a specification of the way in which the times are distributed over the days of the week: brackets enclose days, colons signify breaks, and dots and commas stand for times, with comma times being considered undesirable. The above example specifies five days, with breaks after the fourth and sixth times except on Thursdays, and the last time on each day being undesirable.

Here is a typical resource group:

group *Teachers* **is**

subgroups *English, Science, Computing*;

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

end *Teachers*;

This group contains *resources* (*Smith, Jones, and Robinson*) which are available to attend meetings, and *subgroups* which are subsets of the set of resources defining functions that the resources are qualified to perform: teach English, etc. A resource may be in any number of subgroups. Typical instances have *Teachers, Rooms, and Students* resource groups.

After the groups come the meetings, which are collections of *slots* which are to be assigned elements of the various groups, subject to certain restrictions. For example, here is a typical meeting expressing five Science classes which meet simultaneously for six times per week:

meeting *10-Science* **is**

Year10;
5 Science;
5 ScienceLab;
6 Times: TwoDouble;

end *10-Science*;

There is one slot which must contain the *Year10* resource from the *Students* group; five slots

which must contain resources from the *Science* subgroup; five resources from the *ScienceLab* subgroup of the *Rooms* group, and six times from the *Times* group, which must satisfy the condition *TwoDouble*. This condition is defined to mean that the times must contain two pairs of adjacent times not spanning a break, and also that the times are to be spread over as many days as possible with at most one undesirable time in the set. There is a small fixed list of such conditions built into our program, which can easily be extended as required. Formally, the meaning is that the eleven selected resources will all be occupied together for the six times; in fact, it is clear that the Year 10 students will be split into five groups.

The problem is to find an assignment of times and resources to all the slots that satisfies the various conditions and is such that no two meetings with a resource in common share a time. In our experience, schools insist that all slots be filled with times and appropriate resources, so our program will fail rather than compromise on this. However we are willing to compromise in two respects: time conditions need not be perfect, and resource slots may be occupied by one appropriate resource at one time, and a different appropriate resource at another (this is called a *split assignment*). We try to minimise the number of these defects.

We have used the TTL language successfully, with some unimportant extensions, to specify high school instances. Real instances may have two hundred or more meetings altogether, and capturing them in full detail in TTL can take half a day or more of careful work with the school's timetable coordinator.

3. The TT2 program

This section describes the current version of our program, TT2, in detail. TT2 has evolved over the past two years from our TT1 program, which was the subject of an earlier paper [2].

3.1. Assigning times

The first major stage of the TT2 program assigns times to all meetings in such a way that the resources needed at any time do not exceed the resources available. This condition, which we have called the *resource sufficiency invariant* [2], is maintained throughout the execution of the program. We do not assign particular resources to meetings during this stage.

A set of meetings is called a *time-disjoint meeting-set* if no two of the meetings may share a time (for any reason). For example, the set of meetings attended by the Year 10 students must be time-disjoint, because these students cannot attend two meetings simultaneously.

The basic step of our time assignment algorithm is not the assignment of times to one meeting, but rather the assignment of times to all the meetings of one time-disjoint meeting-set. This exemplifies a recurring theme in our work: we try to identify large sets of assignments that strongly constrain each other, and perform them in a coordinated way.

When assigning times to the meetings of one time-disjoint meeting-set, the resource sufficiency invariant must be maintained. That is, for each time t , the resources needed by the meeting assigned time t must be able to be added to the resources needed by all meetings assigned time t in previous steps, without exceeding the resources available. In addition to this absolute requirement, we also wish to maximize the following measures of the quality of the partial solution:

Time conditions. The times assigned should satisfy each meeting's time conditions, specifying double times, an even spread through the week, and so on.

Time-coherence. The times assigned should cause the meetings to line up in time with meetings assigned previously, as far as possible. Our measure of time-coherence badness is the sum over all overlapping meetings of the number of times that lie in one meeting but not the other multiplied by a measure of the similarity between the resource requirements of the two meetings. Experience shows that time-coherence greatly improves the program's chances of finding a complete solution [2].

Freedom. The times assigned should allow other meetings, not yet assigned times, as wide a choice of times as possible. We measure this for each of these other meetings by dividing the number of times that it can be assigned by the number of times that it requires, and we sum these ratios.

Evenness. If the times assigned force all of the elements of some subgroup (for example, the Mathematics teachers) to attend meetings at some particular time, and hence none or few are required to attend at some other time, this unevenness can cause problems later when assigning resources. We prefer to leave most subgroups less than fully occupied at most times.

An assignment of times (to all the meetings of one time-disjoint meeting-set) which satisfies resource sufficiency alone can be found in polynomial time using the 'meta-matching' algorithm introduced in [2], whenever such an assignment exists. However, to take everything into account we use a randomized tree search in which meta-matching is used as a tree pruning rule, and time conditions, time-coherence, freedom and evenness contribute to the badness function.

This basic step of the TT2 time assignment stage, as just described, differs significantly from the corresponding TT1 step only in the badness function of the search. However, the way in which the whole sequence of basic steps is carried out has changed completely. We now use a beam search. We maintain a set of the n best solutions so far (the *beam*), for some fixed integer n . For each solution, we choose the unassigned time-disjoint meeting-set that appears to be most constrained, and perform a basic step on it. We allow the tree search to return up to m different assignments, for a second fixed integer m , if such can be found, and so the size of our beam expands to up to $n \cdot m$. This is then culled to n based on the badness function, although we also try to maintain diversity by excluding very similar solutions. All this is repeated until times have been assigned to every time-disjoint meeting-set.

If the beam ever becomes empty, we have failed and we can either quit or try again from the beginning with a different random number seed. In practice we always succeed, although with some imperfections in time conditions and not as much time-coherence as we would like.

We have said that this first stage assigns times to all meetings, but that is not quite true. Consider a meeting which contains exactly one specified resource, and no other resource slots. We call it a *single-resource meeting*:

```
meeting SmithFree is  
    Smith;  
    10 Times: TeacherFree;  
end SmithFree;
```

This meeting ensures that Smith has at least 10 free times each week, and the time condition requires that there be at least one each day.

Now it would not be sensible to decide first what times Smith is to be free, then subsequently look for meetings to assign Smith to that do not intrude on these times. Much better to assign Smith to meetings first, taking care to leave 10 suitable times free, and then assign those to *SmithFree*. Accordingly, single-resource meetings are left unassigned during the time-assignment stage.

3.2. The atomic invariant

Before explaining how we assign resources during the second major stage of the TT2 program, we pause to describe separately some ideas that will prove very useful in that stage. We will use the TTL instance

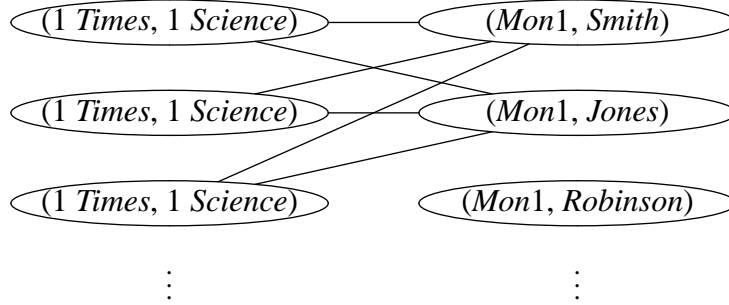
```
timegroup Times is  
  Mon1; Mon2; Mon3; Mon4; Mon5;  
end Times;  
  
group Teachers is  
  subgroups English, Science, Computing;  
  
  Smith in English, Science;  
  Jones in Science, Computing;  
  Robinson in Computing;  
end Teachers;  
  
meeting 8-Science is  
  3 Times;  
  2 Science;  
end 8-Science;  
  
meeting 8-English is  
  2 Times;  
  1 English;  
end 8-English;  
  
meeting 8-Computing is  
  2 Times;  
  2 Computing;  
end 8-Computing;
```

as a running example. It has one essentially unique solution, in which the English and Computing meetings have the same times, and the Science meeting has different times.

A *time-resource atom*, or just *atom*, is a pair consisting of one time and one resource, for example (*Mon1, Smith*). The instance above has $5 \times 3 = 15$ atoms altogether; each represents the availability of one resource at a particular time.

An *atom slot* is a pair consisting of one time slot and one resource slot from one meeting. For example, meeting *8-Science* above has 6 identical atom slots (*1 Times, 1 Science*), and the instance has $3 \times 2 + 2 \times 1 + 2 \times 2 = 12$ atom slots altogether. Each represents a demand for one resource for one time.

The *atomic graph* is a bipartite graph whose left-hand nodes are all the atom slots from all the meetings, and whose right-hand nodes are all the atoms:



We join each atom slot to each atom that may satisfy it. For example, we cannot join (1 Times, 1 Science) to (Mon1, Robinson) because although Mon1 is a suitable time, Robinson is not a Science teacher.

Now if the original TTL instance has a solution, then this graph must have a *matching*, defined here as a subset of its edges such that every atom slot node is incident to exactly one edge, and every atom node is incident to at most one edge. An algorithm which requires this graph to have a matching at all times is said to obey the *atomic invariant*. The bipartite matching problem is well known, and efficient algorithms exist for determining whether or not a matching exists, so this invariant can be checked quickly.

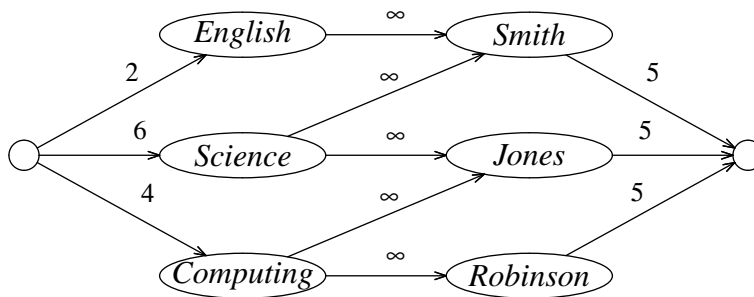
Initially, before any assignments except preassignments have been made, this invariant is able to detect some quite subtle inconsistencies, such as insufficient Mathematics and Computing teachers to cover the Mathematics and Computing meetings, taking teachers' free times, faculty meetings, etc. into account. In other respects it seems quite weak; for example, it is unaware that the times assigned to a meeting must be distinct.

As the first stage of the algorithm progresses, times are assigned to meetings and the atom slots change, for example from (1 Times, 1 Science) to (Mon3, 1 Science). This greatly reduces the number of suitable atoms, to those containing time Mon3 and a Science teacher in our example, so many edges are deleted, matchings become harder to find, and the atomic invariant becomes much stronger.

By the end of the time assignment stage, the only unassigned slots are resource slots in ordinary meetings, and time slots in single-resource meetings. In both cases at least one of the two components of each associated atomic slot is fixed. The atomic invariant subsumes the resource sufficiency invariant (but not meta-matching), since the atom slots containing a fixed time t test for resource sufficiency at t . At present TT2 does not check the invariant until the time assignment stage ends, but it is checked constantly thereafter.

3.3. Resource flow

TT2 contains a network flow model of resource assignment to subgroups, which we call the *resource flow graph*. Here is the graph for the instance from Section 3.2:



For each resource subgroup there is one left-hand node, and it has one incoming edge from the source labelled by the total number of times that the subgroup has to be taught; in the above example, there are 2 times of English to be taught, for example. For each resource there is one right-hand node with one outgoing edge to the sink labelled with the total number of times that the resource is available. (This will in general be less than the total number of times, owing to meetings such as single-resource meetings to which this resource is preassigned.) Edges labelled ∞ connect each subgroup node to every resource in that subgroup.

If the TTL instance has a solution, then this graph must have a *resource flow*, defined here as a network flow from source to sink of size equal to the sum of the labels of the edges leaving the source. Consequently the existence of a resource flow could be used as an invariant.

However, we do not use resource flow as an invariant, for the following reason. Take an atomic graph (Section 3.2), replace all particular times by the generic ‘1 Times’, coalesce identical nodes, add a source and sink, and use the node multiplicities as edge labels. The result is a resource flow graph. Thus the resource flow graph is just the atomic graph with information about specific times omitted, and it can never signal an inconsistent state unless the atomic invariant does so as well.

TT2 contains adaptations of standard network flow algorithms which find the maximum and minimum possible flow down each edge consistent with a resource flow. For example, in the graph given above, at most 2 units of flow can pass down the edge from *Computing* to *Jones*. We use this information when generating assignments of resources to meetings. It may happen, for example, that a Mathematics teacher is qualified for other subjects, but resource flow shows that almost all of the slots assigned to that teacher must nevertheless be Mathematics slots, and thus we would avoid generating assignments involving other slots.

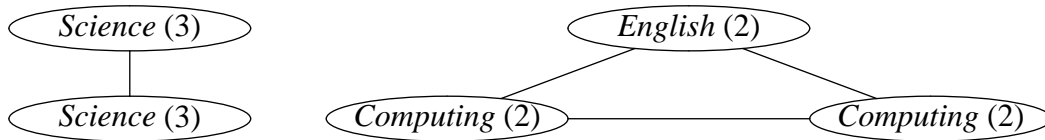
When a resource is assigned to a slot, the edge weights adjacent to the two affected nodes must be reduced by the number of times involved. This necessitates a recalculation of the flow maxima and minima. Since we assign a resource to its complete set of meetings in a single operation (Section 3.5), this recalculation is done only once per resource.

3.4. The clash clique invariant

Our resource assignment algorithm is based on a graph that we call the *clash graph*. In this section we describe the clash graph and an invariant arising from it called the *clash clique invariant*. These will be put to work in the next section.

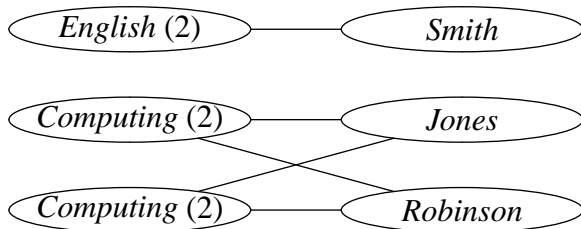
Suppose that times have been assigned to meetings. Take any set of resource slots requiring assignment, and build a graph which has one node for each resource slot, and an edge joining

two nodes whenever the two meetings from which the two slots come have at least one time in common. This is the *clash graph* for those resource slots. For example, here is the clash graph for the resource slots of the essentially unique solution to the instance of Section 3.2:



Within the node we write the resource subgroup of the slot and, in parentheses, the number of times in the slot's meeting. The numbers will not be used in this section. An edge between two nodes means that the two slots clash, so no resource may be assigned to both slots.

Now take any clique in the clash graph. (A clique is a set of nodes such that every two nodes in the set are connected by an edge.) No two slots in the clique may be assigned the one resource. Construct a bipartite graph with one left-hand node for each node in the clique, and one right-hand node for each resource in the instance. Join each slot node to the resources that may be assigned to the slot. For example, for the clique consisting of the English and Computing slots in the graph above, the bipartite graph is



Now if the TTL instance has a solution, this includes an assignment of resources to the slots of this graph. Because the slots clash and no resource may be assigned to any two of them, this assignment must be a matching. Thus we may use the existence of a matching in any bipartite graph derived from a clash clique as an invariant, which we call the *clash clique invariant*.

Ideally, we would maintain one bipartite graph for each maximal clique in the clash graph. Unfortunately, there may be exponentially many maximal cliques, in practice as well as in theory, so we use a heuristic method which looks for large cliques in likely places. For example, the set of all resource slots from meetings which contain a particular time t form a clique. It is easy to see that the bipartite graph for this clique is exactly what is required to test for resource sufficiency at time t . However, it is frequently possible to find other meetings which clash with all of these ones despite not themselves containing time t . For example, consider these three meetings:

```
meeting M1 is
  Mon1; Mon2;
  1 Science;
end M1;
```

meeting M2 is
 Mon2; Mon3;
 1 *Science;*
end M2;

meeting M3 is
 Mon3; Mon1;
 1 *Science;*
end M3;

Two Science teachers are enough to satisfy resource sufficiency at all times. The clique built from the meetings containing time *Mon1* contains the *Science* slots from meetings *M1* and *M3*, but since *M2* clashes with both of these meetings its *Science* slot may be added to the clique. Consequently the bipartite graph for this clique has three slot nodes, and no matching with only two teachers exists. If the clique from each particular time *t*, possibly extended in this way, is monitored by its own bipartite graph, then the clash clique invariant will always be at least as strong as the resource sufficiency invariant, and often stronger.

It is possible to detect a forced assignment by deleting one edge of any bipartite graph and seeing whether a matching is then impossible. Similarly, one can make an assignment and see whether a matching for the remainder is then impossible, in which case we have what may be called a forced non-assignment. This information can be propagated to other invariants; we have found a surprisingly large number of forced moves in this way.

Incidentally, the clash graph explains very clearly the importance of maximizing time-coherence during the time assignment stage (Section 3.1). Better time-coherence translates into fewer edges in the clash graph and a less constrained resource assignment problem.

3.5. Assigning resources

The second major stage of the TT2 program assigns all the slots left unassigned by the first stage: all resource slots, and all time slots in single-resource meetings.

We enter this stage with the atomic invariant holding (Section 3.2), which by this point has become very strong. The matching in the atomic graph guaranteed by this invariant provides an assignment to all remaining slots. We could make this assignment and quit immediately, except for one problem.

The sole problem is that this assignment does not guarantee that the same resource will be assigned to a slot for all the times of a meeting. For example, we might find that a Science slot is assigned Smith at one time and Jones at another, a defect known as a *split assignment*. The need to avoid or at least minimize split assignments we call the *resource constancy requirement*.

We measure resource constancy by the percentage of resource slots that are not split. In practice, 100% resource constancy (no split assignments) seems impossible to achieve. As a rule of thumb, we aim for 95% resource constancy with each split assignment shared between two resources, never three or more. Resource assignment thus has the character of a search through a space of feasible solutions delimited by the atomic invariant, looking for one with high resource constancy. Total failure is not possible here, as it was in the time assignment stage.

Each resource group may be assigned independently of the others. Although all resource

groups are formally equivalent, in practice they differ in their constancy requirements. Student resources are all preassigned, so there are no assignments to make. Room constancy, although formally required by the TTL language, is required by schools only during double and triple times. In practice, then, resource constancy means teacher constancy.

TT2 contains a simple resource assignment algorithm used for rooms. It assigns rooms to room slots one by one, checking that the atomic invariant is not violated. Any unassignable slots receive split assignments. Maintenance of the atomic invariant guarantees that this algorithm will never get stuck.

The remainder of this section explains the much more elaborate resource assignment algorithm that we use when constancy is important – in practice, for teacher assignment. This has been the most difficult problem that we have encountered in our work on timetabling.

Our overall structure is a beam search, as in the time assignment stage. In our instances it is vital that each teacher be utilized completely or almost completely, so our basic step is not the assignment of one resource to one slot, but rather the assignment of one resource to a whole set of non-clashing slots which utilize the resource as completely as possible.

The generation of suitable sets of slots for one resource proceeds as follows. We are looking for a set of nodes in the clash graph such that no two nodes in the set are connected by an edge (an *independent set* of nodes). The total size must equal or almost equal the number of times that the resource in question is available. Finally, it must be possible to assign all of the slots to the resource without violating the atomic invariant (which implies that any single-resource meetings containing the resource can be assigned times from those left over) and preferably without violating the clash-clique invariant either.

We find not just one independent set for each resource, but up to 50. Finding maximal independent sets is NP-hard, and so our method is basically exhaustive search, but we have several ways of reducing the cost. We try to partition the problem into faculties, so that we do not waste time examining English slots when assigning Mathematics teachers. Slots whose sets of times are identical (or sufficiently similar that they clash with the same slots, and with each other) are allowed to share one node, so that we do not waste time examining essentially identical slots. Resource flow (Section 3.3) is used to tell us that at least so many Mathematics times (or whatever) must be included in any independent set; and by tentatively deleting the resource in question from a clash clique, we can often prove that every independent set must include one slot from that clique.

If a particular node appears in every independent set that we find for some resource, we take the assignment of that resource to some slot in that node to be forced. Other forced assignments can be detected by the clash clique invariant. We check carefully for and make forced assignments at every step, since in our experience there are many of them.

At each step we choose for assignment the resource with the fewest independent sets. We try several independent sets, producing multiple partial solutions which fill our beam.

Because we only permit assignments of independent sets that do not violate the atomic invariant, total failure cannot occur. However, occasionally we may be unable to find any independent set of sufficient size for some resource whose assignment will not violate the invariants. This means that further progress from the current state requires a split assignment involving that resource. We prefer to abandon this whole line of assignments if this occurs, relying on the other members of our beam to carry on. But if the beam threatens to become

empty, we must do a split assignment.

At present, when a split assignment becomes necessary, the program prints out the relevant information and requires the user to decide which slot to split and how. We plan to automate this decision very soon. Provided our overall algorithm leads to few split assignments, it will not matter if the precise choices made are somewhat simple-minded.

4. Results

Our program is still being developed, so we have not attempted to solve a large number of instances yet. However, we have solved two instances taken without simplification from high schools in the Sydney area. We call these instances BGHS94 and JAM94.

BGHS94 has 40 times, 26 student resources (each typically representing a group of thirty to forty students), 50 teachers, 48 rooms, and 199 meetings, many of which are aggregate meetings like 10-*Science* in Section 2. Our best solution solves the instance, the only significant compromises being that 10 of the 114 double times requested (8%) are supplied by two singles instead, 16 teachers do not receive a free time on some day, and 28 of the 326 teacher slots (8%) are split between two teachers. No slots are split among three or more teachers. The CPU time for the time assignment stage was 91 minutes, with a beam width of 100; the time for resource assignment was dominated by the time taken by the user to decide which slots to split.

JAM94 has 40 times, 45 student resources, 97 teachers, 79 rooms, and 252 meetings. Despite the larger size it seems to be easier than BGHS94. Our best solution solves the instance. The only significant compromises are that 15 of the 125 double times requested (12%) are supplied by two singles instead, and 17 of the 580 teacher slots requested (2%) are split between two teachers. Once again, no slots are split among three or more teachers.

5. Conclusions and future work

The TT2 program has succeeded in constructing timetables for real high school instances in a reasonable amount of time. Its results are somewhat deficient in the area of time conditions, and its handling of split assignments must be automated, but these are relatively minor problems.

We plan to reorganize the program so that its components can be used in a more flexible, integrated way. This will allow us to detect and perform forced assignments during the first stage that are currently discovered only during the second. After this has been done, we expect that minor incremental improvements will be sufficient to reach our goal of producing timetables reliably. We have not yet seen our timetables adopted, but we are confident that that day is close, and we are in contact with a number of high schools that have expressed interest in working with us to solve their timetabling problems.

References

1. M. Carter. Timetabling literature. Available by anonymous *ftp* from *ftp.cs.nott.ac.uk* in directory *ttp*, January 1995.
2. Tim B. Cooper and Jeffrey H. Kingston. The solution of real instances of the timetabling problem. *The Computer Journal* **36**, 645–653 (1993).

3. J. Csima. *Investigations on a Time-Table Problem*. Ph.D. thesis, School of Graduate Studies, University of Toronto, 1965.
4. S. Even, A. Itai, and A. Shamir. On the complexity of timetable and multicommodity flow problems. *SIAM Journal on Computing* **5**, 691–703 (1976).
5. J. Lions. Matrix reduction using the Hungarian method for the generation of school timetables. *Communications of the ACM* **9**, 319–354 (1966).
6. J. Lions. The Ontario school scheduling program. *The Computer Journal* **10**, 14–21 (1967).
7. G. Schmidt and T. Ströhlein. Timetable construction—an annotated bibliography. *The Computer Journal* **23**, 307–316 (1980).
8. D. de Werra. Construction of school timetables by flow methods. *INFOR – Canadian Journal of Operations Research and Information Processing* **9**, 12–22 (1971).
9. D. de Werra. An introduction to timetabling. *European Journal of Operational Research* **19**, 151–162 (1985).
10. P. H. Winston. *Artificial Intelligence*. Addison-Wesley. Third edition, 1992.