Plenary Speakers

Cats and Rats

Dominique de Werra

École Polytechnique Fédérale de Lausanne

dewerra@dma.epfl.ch

Abstract. The basic class-teacher timetabling problem is examined with the additional constraints due to the (un−)availability of source teachers and/or classes at some periods. We consider a generalization of this problem which occurs as an image reconstruction problem in tomography.
Complexity issues are discussed for both types of problems and some solvable cases are presented which can be derived from the image reconstruction formulation. Reductions to canonical forms are also described. Some other types of unavailability constraints (for classrooms or for lectures) are also reviewed.

Keywords

graph coloring, timetabling, image reconstruction, cats, network flows, open shop scheduling, availability constraints, rats.

1 Introduction and Motivation

Let us imagine for a moment that some clever rats $c_1, \ldots, c_m$ have decided to follow a continuing education program offered by a small community of top cats $t_1, \ldots, t_n$. We observe that each cat t_j has to meet each rat c_i for r_{ij} nights of intensive study. If we do not distinguish the various topics on which the top cats are lecturing for the clever rats (like foxology, selective garbageism, rational mouseology, mammiferal sciences, etc.), the data can be summarised in an $(m \times n)$ array $R = (r_{ij})$ called requirement matrix.

Assuming that no top cat can teach two clever rats at the same time and no rat (clever or not) is able to get a lecture from more than one top cat at a time, we may ask ourselves how can one construct a schedule which will minimize the number of nights needed.

This basic model in theoretical timetabling has been almost useless in the real cases since most cats are obviously not available every night for intensive studies. Due to some survival requirements, they all have to exercise food searching (applied rational mouseology) and uncontrolled reproduction.

Considering the set H of hurricane nights of a month where lectures can take place (because top cats and clever rats have to remain inside), we can associate to each top cat t_j a subset $T_j \subseteq H$ of nights where it is available for giving lectures.

Having introduced this additional feature to make our model look more realistic, the question now is to determine whether a schedule can be found with the following:

1. all lectures in R are given within the set H of hurricane nights of the month.
2. no top cat (resp. no clever rat) is involved in more than one lecture during a night.
3. a cat t_j can give a lecture only during nights k in T_j

Such a problem can be formulated in terms of arrays in the following way: Given the set H of hurricane nights, let $h = |H|$; we define an $(m \times h)$ array A where row i is associated to the clever rat c_i and column k to night k.

For each c_i we consider integers $a(i,1), \ldots, a(i,n)$ with $a(i,j) = r_{ij}$, i.e., the number of lectures to be given by top cat t_j to c_i.

Similarly for each period k we consider integers $\alpha(k,1), \ldots, \alpha(k,n)$ defined as follows:

$$\alpha(k,j) = \begin{cases} 1 & \text{if night } k \in T_j \\ 0 & \text{otherwise} \end{cases}$$

This means that $\alpha(k,j)$ is 1 if top cat t_j is available during night k or 0 else.

A timetable is then given by array A where entry a_{ik} contains the name of the top cat t_j spending a night k of study with clever rat c_i (a_{ik} is empty if c_i has no night lecture during night k).

Then $a(i,j)$ is the number of occurrences of t_j in row i of A while $\alpha(k,j)$ is the number of occurrences of t_j in column k (0 or 1 depending upon the availability of t_j).

An example is given in Fig. 1

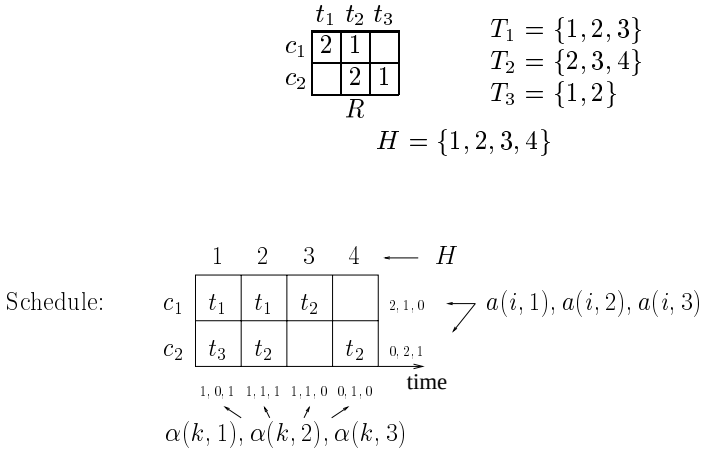

Fig. 1. An example of schedule

It turns out that this formulation is closely related to a problem of image reconstruction in tomography; we intend to explore here the basic timetabling model where unavailability constraints are present and exploit the analogy with image reconstruction in tomography. Hence the title: Constraints of Availability in Timetabling and Scheduling AND Reconstruction of Arrays in Tomographic Scanning, in short: CATS AND RATS.

The reader is referred to [3] for all graph theoretical notions not defined here and to [4] for a short introduction to complexity and for basic algorithms in scheduling.

2 Some reductions for the timetabling problem

For our purposes the basic timetabling problem with unavailability constraints will be defined by:

- a set $C = \{c_1, \ldots, c_m\}$ of classes
- a set $T = \{t_1, \ldots, t_m\}$ of teachers
- a set $H = \{1, 2, \ldots, h\}$ of periods
- an $(m \times n)$ requirement matrix $R = (r_{ij})$
- a collection $\mathcal{T}$ (resp $\mathcal{C}$) of subsets $T_j \subseteq H$ (resp. $C_i \subseteq H$) of periods of availability for each teacher t_j (resp. each class c_i).

A graph-theoretical model is often used to represent this problem $\mathrm{TT}(C, T, H, R, \mathcal{T}, \mathcal{C})$ of timetabling: it consists of a bipartite multigraph $G = (C, T, R)$ where each class c_i (resp. each teacher t_j) is associated to a node of G; furthermore nodes c_i and t_j are linked by r_{ij} parallel edges.

There is a one-to-one correspondence between solutions of TT($C, T, H, R, \mathcal{T}, \mathcal{C}$) and edge h-colorings of G where no two adjacent edges have the same color and the color c of each edge $[c_i, t_j]$ is in $C_i \cap T_j$.

We shall now show that the problem TT of timetabling can be reduced to some canonical form.

Figure 2 shows an example of TT with the associated bipartite multigraph G.

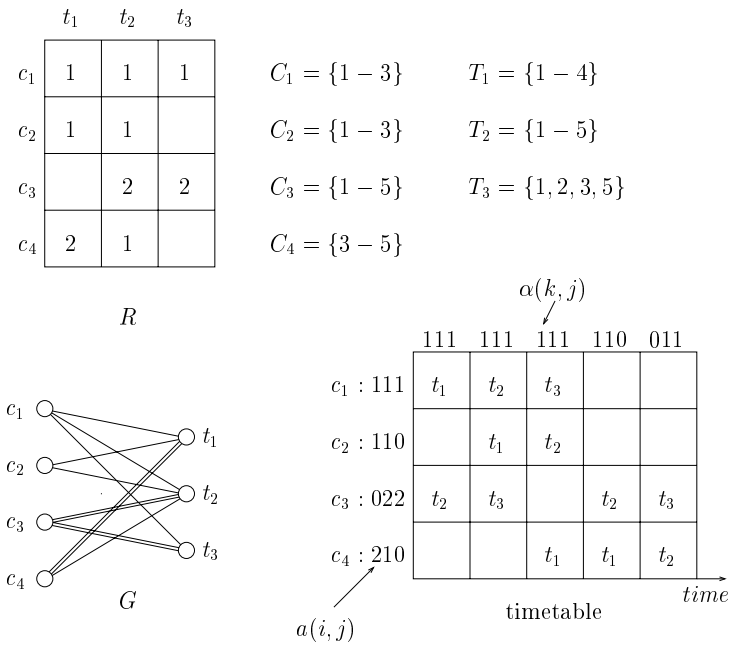

Fig. 2. An example of schedule

Observe that $\alpha(k,j) =$ 0 or 1 is the *maximum* number of occurrences of t_j in column k while $a(i,j)$ is the *exact* number of occurrences of t_j in row i. This is due to the fact that for some t_j we may have $|T_j| > \sum_i r_{ij}$, i.e., teacher t_j is not *tight*.

Property 2.1. *TT($C, T, H, R, \mathcal{T}, \mathcal{C}$) can be transformed into a restricted problem RTT($C', T', H, R', \mathcal{T}$) such that*

a. RTT has solution if an only if TT has one
b. $C_i' = H$ (i.e., no unavailability) for each class c_i'
c. $|T_j| = \sum_i r_{ij}'$ (teacher t_j' is tight) for each teacher t_j'
d. the sets C', T', R' of classes, teachers, lectures resp. satisfy

$$|C'| \leq |C| + |T|$$

$$|T^{'}| \leq 2\,|T| + |C|$$
$$|R^{'}| \leq |H|\,(|T| + |C|) - |R|$$

Proof. Let us construct problem RTT from TT as follows:

i. For each class c_i with $C_i \neq H$ introduce a new teacher t_i^+ with $|H| - |C_i|$ new lectures to be given by t_i^+ to c_i and set $T_i^+ = H - C_i$.

ii. Now for every teacher t_j which is not tight, i.e., with $d_j = |T_j| - \sum_i r_{ij} > 0$, introduce a new class c_j^+ and d_j new lectures to be given by t_j to c_j^+. These lectures should ideally occur in periods of T_j; to force this we introduce a new teacher t_j^* with $|H| - |T_j|$ new lectures to give to class c_j^+ and we set $T_j^* = H - T_j$.

iii. We rename $t_j^{'}$ and $c_i^{'}$ the teachers and classes of the new problem RTT; we set $C_i^{'} = H$ for each $c_i^{'}$ and $R^{'}$ is the new requirement matrix.

iv. One can verify that the resulting RTT has a solution if and only if TT has one. Furthermore all classes $c_i^{'}$ of RTT have $C_i^{'} = H$; in addition every teacher $t_j^{'}$ is tight.

v. We have introduced in the construction at most $|C|$ new teachers and at most $|C| \cdot |H| - |R|$ new lectures in *i*. Also in *ii* we have introduced at most $|T|$ new classes, at most $|T|$ new teachers and at most $|T| \cdot |H| - |R|$ new lectures. Here $|R| = \sum_i r_{ij}$ = number of lectures. Hence d holds.

□

This construction is illustrated in Fig. 3 for the example given in Fig. 2.

	t_1	t_2	t_3	t_1^+	t_2^+	t_4^+	t_3^*
c_1	1	1	1	2			
c_2	1	1			2		
c_3		2	2				
c_4	2	1				2	
c_3^+			1				1

$H = \{1-5\}$

$C_1 = \{1-3\}$ $\quad T_1 = \{1-4\}$

$C_2 = \{1-3\}$ $\quad T_2 = \{1-5\}$

$C_3 = \{1-5\}$ $\quad T_3 = \{1,2,3,5\}$

$C_4 = \{3-5\}$ $\quad T_1^+ = H - C_1$

$T_2^+ = H - C_2$

$T_4^+ = H - C_4$

$T_3^* = H - T_3$

Fig. 3. Construction of RTT for the example of Fig. 2

Let us now transform the problem RTT into an equivalent problem RTT* where each teacher t_j^* will have to meet at most once each class c_i^*, i.e., the requirement matrix $R^* = (r_{ij})$ will be such that r_{ij} is 0 or 1. For this purpose, we will use the graph theoretical model described above.

So we are given a timetabling problem $\mathrm{RTT}(C, T, H, R, \mathcal{T})$ where as before all teachers are tight and all classes c_i satisfy $C_i = H$. G is the associated bipartite multigraph. When $r_{ij} > 1$, nodes c_i and t_j are linked by r_{ij} parallel edges $[c_i, t_j]_1, [c_i, t_j]_2 \ldots$.

Property 2.2. *$RTT(C, T, H, R, \mathcal{T})$ can be transformed into a restricted problem $RTT^*(C^*, T^*, H, R^*, \mathcal{T})$ such that*

1. *RTT^* has a solution if and only if RTT has one*
2. *The requirement matrix R^* has $r_{ij} = 0$ or 1 for all i, j*
3. *The sets C^*, T^*, R^* of classes,teachers, lectures resp. satisfy*

$$
\begin{aligned}
|C^*| &\leq |C| + h^2 \min\{|C|, |T|\} \\
|T^*| &\leq |T| + h^2 \min\{|C|, |T|\} \\
|R^*| &\leq |R| + h^3 \min\{|C|, |T|\}
\end{aligned}
$$

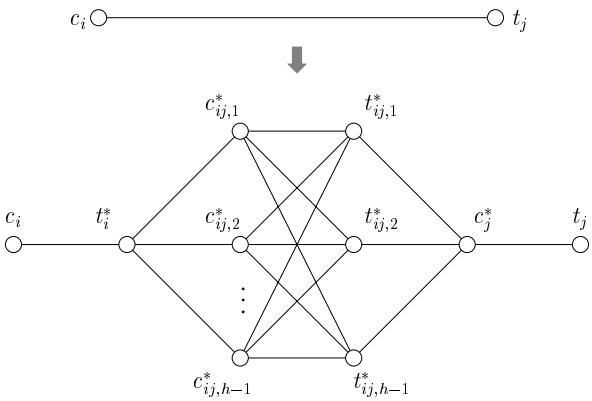

Fig. 4. Transformation of RTT into RTT^*

Proof. We shall describe the transformation directly in the associated graph G. To keep notation simple, let us consider an edge $[c_i, t_j]$ belonging to a family of parallel edges between nodes c_i and t_j. We replace $[c_i, t_j]$ by the following graph: as before let $h = |H|$ be the number of periods; we introduce h new teachers $t^*_i, t^*_{ij,1}, \ldots, t^*_{ij,h-1}$ and h new classes $c^*_j, c^*_{ij,1}, \ldots, c^*_{ij,h-1}$. For all these teachers t^*_x and all classes c^*_y we set $T^*_x = C^*_y = H$.

Then each one of these new teachers has to give one lecture to each one of these new classes (except t^*_i which gives a lecture to c_i instead of c^*_j).

Finally we replace the lecture of t_j to c_i by a lecture of t_j to c_j^*; so we have removed one lecture and introduced h^2+1 new lectures. The collection R of lectures has been increased by an amount of h^2.

The construction is illustrated in Fig 4. If we repeat this for all but one lecture in every family of parallel edges of G, we will get a simple bipartite graph G^* associated to a new problem RTT* with $r_{ij}^* \in \{0,1\}$ for all i,j.

It is easy to see that in any edge $h-$coloring of G^*, edges $[c_i, t_i^*]$ and $[c_j^*, t_j]$ will have the same color; so from any $h-$coloring of G^* we can derive an $h-$coloring of G and the converse is also true as can be verified easily.

Hence we will have a problem RTT* having a solution if and only if RTT has one.

The number of edges $[c_i, t_i]$ to transform is at most $h \min\{|C|, |T|\}$, so we have introduced

at most $h^2 \min\{|C|, |T|\}$ new teachers,
at most $h^2 \min\{|C|, |T|\}$ new classes

and at most $h^3 \min\{|C|, |T|\}$ new lectures. Hence 3. holds; since G^* is now simple, we have 2. □

With the reduction described above our timetabling problem RTT$^*(C, T, H, R, \mathcal{T})$ can be stated as follows:

We are given sets C, T, H of classes, teachers and periods respectively;

$|C| = m$, $|T| = n$, $|H| = h$. In addition we have an $(m \times n)$ requirement matrix R with $r_{ij} \in \{0,1\}$ for all i,j and a family $\mathcal{T} = (T_1, \ldots, T_n)$ of subsets $T_j \subseteq H$ giving the availabilities of the teachers t_j.

From these data, we define numbers

$$a(i,j) = r_{ij} \text{ for } i = 1, \ldots, m \text{ and } j = 1, \ldots, n$$
$$\alpha(k,j) = \begin{cases} 1 & \text{if period } k \in T_j \\ 0 & \text{else} \end{cases} \quad \text{for } k = 1, \ldots, h \text{ and } j = 1, \ldots, n\ .$$

We have to construct an $(m \times h)$ array A by introducing one of the symbols $t_1, \ldots, t_n$ (teachers) in the entries a_{ik} of A in such a way that for all $j = 1, \ldots, n$ the following holds:

1. t_j occurs exactly $a(i,j)$ times in row i of A (for $i = 1, \ldots, m$)
2. t_j occurs exactly $\alpha(k,j)$ times in column k of A (for $k = 1, \ldots, h$)

The array A will define a schedule in a unique way: a_{ik} contains t_j if and only if teacher t_j gives a lecture to class c_i at period k.

Notice that due to the fact that all teachers are tight, we have exactly $\alpha(k,j)$ occurrences of t_j in column k. In addition, we now have $a(i,j) \in \{0,1\}$ for all i,j (and $\alpha(k,j) \in \{0,1\}$ as before).

3 The basic image reconstruction problem

In this form the timetabling problem RTT* is very close to the image reconstruction problem which arises in tomography. We will now give its formulation. An image is usually decomposed into a collection of pixels of different colors. Basically it consists of an $(m \times n)$ array A where each entry a_{ij} contains a pixel of some color s chosen in a set of p colors.

In order to compress the data representing an image, one may for instance give simply the numbers $a(i,s)$ (resp. $\alpha(j,s)$) of pixels of color s occurring in row i (resp. column j) for $i = 1,\ldots,m, j = 1,\ldots,n$ and $s = 1,\ldots,p$.

The main question is to determine whether to given values $a(i,s), \alpha(j,s)$ for $1 \leq i \leq m, 1 \leq j \leq n, 1 \leq s \leq p$ corresponds an $(m \times n)$ array A having exactly $a(i,s)$ (resp. $\alpha(j,s)$) pixels of color s in row i (resp. column j)? Moreover the question of uniqueness of the reconstruction image is important. We shall concentrate here on the question of existence of an image. This problem is denoted by RP(m,n,p) (reconstruction of an $(m \times n)$ array from the values $a(i,s), \alpha(j,s)$ with p colors).

The similarity with the restricted timetabling problem is now obvious: The p colors correspond to the n teachers of RTT*, the rows i are associated to the classes c_i and the n columns to the h periods of the timetable to be constructed.

While $a(i,s)$ and $\alpha(j,s)$ are usually non negative integers in problem RP(m, n, p), they take values 0 or 1 in the timetabling problem RTT*.

As in [6], we call *unary* the colors s for which $a(i,s)$ and $\alpha(j,s)$ are 0 or 1 for all i,j. Problem RP(m,n,p) is denoted by RPU(m,n,p) when all p colors are unary; so there is equivalence between RPU(m,n,p) and RTT*. This fact can be exploited for deriving complexity properties of these problems.

At this stage, we should mention a slight difference between RP(m,n,p) and RTT*: while entry a_{ik} may be empty in the array A associated to a timetabling problem (at period k class c_i has no lecture), in the image reconstruction problem an entry a_{ij} which does not have a pixel of some color $s \leq p$ is said to have the "ground color" $p+1$. But we can easily forget it and work with the first p colors.

The analogy between RPU(m,n,p) and RTT* where $r_{ij} \in \{0,1\}$ for all i,j suggests the following: in RPU(m,n,p) the rows and the columns play a symmetric role; it is also the case for RTT* (but only when $r_{ij} \in \{0,1\}$ for all i,j). We can interchange the roles of classes and of periods. More precisely consider a teacher t_j characterized by

- a subset $T_j \subseteq H$ of periods k where (s)he has to give lectures
- a subset T_j^* of classes c_i to which t_j has to give a lecture

$$T_j^* = \{c_i | r_{ij} = 1\}$$

We may consider that H is the set of all classes; each teacher t_j is characterized by:

- a subset T_j^* of periods c_i where (s)he has to give lectures
- a subset $T_j \subseteq H$ of classes k to which t_j has to give a lecture

If RTT** is this last problem, then clearly RTT** has a solution if and only if RTT* has one.

This simple observation may be useful for deriving complexity results for RTT in general.

One should also remark that when $r_{ij} \notin \{0,1\}$ for some pairs c_i, t_j, then one cannot interchange the role of classes and of periods in RTT.

Let RTT$^{*}(C,T,H,R,\mathcal{T})$ denote the timetabling problem where all teachers are tight and $r_{ij} \in \{0,1\}$ for all i,j. We recall some complexity results:

Proposition 3.1 ([7]). *RTT$^{*}(C,T,H,R,\mathcal{T})$ is NP-complete even when* $|H| = 3$.

From this we derive immediately

Proposition 3.2 ([6]). *RTT$^{*}(C,T,H,R,\mathcal{T})$ is NP-complete even when* $|C| = 3$.

Notice that when $|T_j| \leq 2$ for each teacher t_j, then RTT$^{*}(C,T,H,R,\mathcal{T})$ can be solved in polynomial time [7] by transforming it to a $2-SAT$ problem. It follows that RTT$^{*}(C,T,H,R,\mathcal{T})$ can be solved in polynomial time if $|H| = 2$.

As above, we may also deduce

Proposition 3.3 ([6]). *RTT$^{*}(C,T,H,R,\mathcal{T})$ can be solved in polynomial time if* $|C| = 2$.

The following result has been obtained for RPU(m,n,p):

Proposition 3.4 ([6]). *RPU$(m,n,p=3)$ can be solved in polynomial time.*

Interpreting this in terms of timetabling gives the following:

Proposition 3.5 ([6]). *RTT$^{*}(C,T,H,R,\mathcal{T})$ can be solved in polynomial time if* $|T| \leq 3$.

One should recall here that while RPU$(m,n,p=3)$ can be solved in polynomial time, the complexity of RP$(m,n,p=2)$ is unknown.

Also for RTT$^{*}(C,T,H,R,\mathcal{T})$ with $|H| \geq 4$, the complexity is unknown.

4 More on the basic timetabling problem

We have shown how a basic timetabling problem TT$(C,T,H,R,\mathcal{T},\mathcal{C})$ could be transformed into a problem RTT where $C_i = H$ for each class c_i and where all teachers where "tight" ($\sum_i r_{ij} = |T_j|$ for each teacher t_j). We can also transform the basic problem RTT into other regular forms. We shall just mention the following:

Property 4.1. *TT$(C,T,H,R,\mathcal{T},\mathcal{C})$ can be transformed into a problem RTT$'(C',T',H,R',\mathcal{P})$ satisfying the following:*

1. *RTT' has a solution if and only if TT has one*
2. $|T'| = |T| + |C|$
3. $|C'| = |C| + |T|$
4. $\sum_i r'_{ij} = \sum_j r'_{ij} = |H|$ *for all* i, j
5. $C'_i = T'_j = H$ *for all* i, j
6. *some families of lectures* $c'_i - t'_j$ *have to be scheduled at some fixed periods.*

Proof. : Let $m = |C|, n = |T|$; we construct an $(m+n) \times (m+n)$ requirement matrix R' by inserting the initial matrix R in the upper left corner and the transposed matrix R^T in the lower right corner of R'. Then we may introduce in entries $(m+1,1), \ldots, (m+n,n)$ values $r'_{m+j,j} = |H| - \sum_{i=1}^{m} r_{ij} \geq 0 \quad (j = 1, \ldots, n)$ and similarly in entries $(1, n+1), \ldots, (m, n+m)$ we introduce values $r'_{i,n+i} = |H| - \sum_{j=1}^{n} r_{ij} \geq 0$.

We now have a requirement matrix R' where all row sums and all column sums are equal to $|H|$.

This corresponds to introducing a set of new classes $\overline{c}_1, \ldots, \overline{c}_n$ (corresponding to the initial teachers $t_1, \ldots, t_n$) and a set of new teachers $\overline{t}_1, ..., \overline{t}_m$ (corresponding to the initial classes $c_1, \ldots, c_m$).

We set $\overline{T}_i = C_i$ for $i = 1, \ldots, m$ and $\overline{C}_j = T_j$ for $j = 1, \ldots, n$.

Consider now a pair $\overline{c}_j, t_j$: by construction t_j has to give $r'_{m+j,j}$ lectures to $\overline{c}_j$. Let $d_j = |\overline{C}_j| - \sum_{i=1}^{m} r_{ij} \leq |H| - \sum_{i=1}^{m} r_{ij} = r'_{m+j,j}$.

We assign $r'_{m+j,j} - d_j$ lectures of t_j to $\overline{c}_j$ to the $r'_{m+j,j} - d_j$ periods of $H - \overline{C}_j = H - T_j$; this is the preassignement constraint $\mathcal{P}_j$. The remaining d_j lectures of t_j to $\overline{c}_j$ are not preassigned. We repeat this for all pairs $t_j, \overline{c}_j$, thus obtaining preassignments $\mathcal{P}_1, \ldots, \mathcal{P}_n$.

Similarly we consider each pair $c_i, \overline{t}_i$: we define $e_i = |\overline{T}_i| - \sum_{j=1}^{n} r_{ij} \leq |H| - \sum_{j=1}^{n} r_{ij} = r'_{i,n+i}$ and preassign $r'_{i,n+i} - e_i$ lectures of $\overline{t}_i$ to c_i to the $r'_{i,n+i} - e_i$ periods of $H - \overline{T}_i = H - C_i$; the remaining e_i lectures of $\overline{t}_i$ to c_i are not preassigned. This gives the preassignment constraint $\mathcal{P}_{n+i}$. Repeating this for all pairs $c_i, \overline{t}_i$ we define preassignment constraints $\mathcal{P}_{n+1}, \ldots, \mathcal{P}_{n+m}$. It is then easy to check that the new problem RTT' satisfies all statements 1-6. □

The construction is illustrated in Fig. 5 and timetables for RTT and RTT' are shown in Fig. 6.

In terms of the image reconstruction problem, we can interpret RTT$'$ as follows. We have an $(m \times n)$ array A and p colors (corresponding to the teachers): the rows correspond to classes and the columns to periods. All colors s satisfy $\alpha(j,s) = 1$ for all columns j: each color occurs exactly once in each column; in each row i, color s occurs exactly $a(i,s)$ times.

Finding such an array would be an easy problem (solvable by classical edge coloring techniques); but here we have a collection of preassignment requirements to take into account: some entries of A have already been assigned some color.

This is what makes the problem difficult. Notice that in these requirements each color is preassigned in only one row of A (see the timetable of RTT$'$ in Figure 6 where for instance color t_2 is preassigned only in the row associated to $\overline{c}_2$).

	t_1	t_2	t_3
c_1	2		1
c_2		1	2

$m = 2, \quad n = 3$

$H = \{1-4\}$ $\quad T_1 = \{1-3\} = \bar{C}_1$

$C_1 = \{1-4\} = \bar{T}_1 \quad T_2 = \{1-2\} = \bar{C}_2$

$C_2 = \{1-3\} = \bar{T}_2 \quad T_3 = \{1-4\} = \bar{C}_3$

Regularisation

	t_1	t_2	t_3	$\bar{t}_1$	$\bar{t}_2$	
c_1	2		1	1		$e_1 = 4-3 = 1$
c_2		1	2		1	$e_2 = 3-3 = 0$
$\bar{c}_1$	2			2		$d_1 = 3-2 = 1$
$\bar{c}_2$		3			1	$d_2 = 2-1 = 1$
$\bar{c}_3$			1	1	2	$d_3 = 4-3 = 1$

R'

$\mathcal{P}_1$: one lecture $\bar{c}_1 - t_1$ preassigned in $H \setminus \bar{C}_1 = \{4\}$

$\mathcal{P}_2$: two lectures $\bar{c}_2 - t_2$ preassigned in $H \setminus \bar{C}_2 = \{3, 4\}$

$\mathcal{P}_3$: -

$\mathcal{P}_4$: -

$\mathcal{P}_5$: one lecture $c_2 - \bar{t}_2$ preassigned in $H \setminus \bar{T}_2 = \{4\}$

Fig. 5. Construction problem RTT′

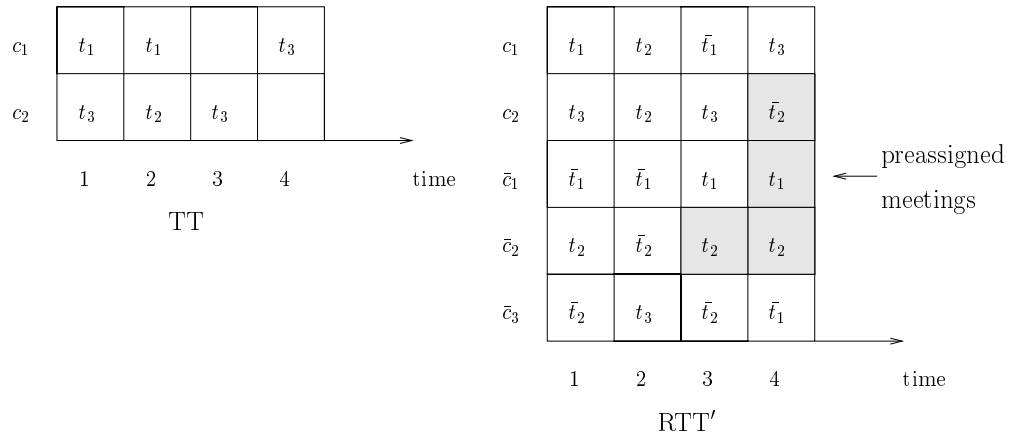

Fig. 6. A timetable for TT and the corresponding one for RTT′

We shall next state an additional result related to the complexitiy of RTT:

Property 4.2. *RTT$(C, T, H, R, \mathcal{T})$ is NP-complete even if every $T_j \subseteq H$ is an interval.*

Proof. It has been shown that the following problem is NP-complete [2]. Given a bipartite multigraph $G = (C, T, E)$ with maximum degree $\Delta(G) = 3$, does there exist an edge 3-coloring of G such that for each node $v \in T$ the colors are $1, 2, \ldots, d_G(v)$ (i.e, the first colors)?

It follows immediately that it is also NP-complete to decide whether in such a graph G the edges adjacent to node $v \in T_j$ have color 2 (if the degree is 1) or colors 2,3 (if the degree is two), or colors 1,2,3 (for degree three).

We consider the problem RTT$(C, T, H, R, \mathcal{T})$ with $|H| = 3$ and we may assume as before that all teachers are tight.

We will transform the above edge coloring problem into RTT where all T_j's are intervals in H:

consider a node $v \in T$ of degree 1 and introduce an edge $[v, c_{1v}]$ and an edge $[v, c_{2v}]$ as well as double edges $[c_{1v}, t_{1v}]_1$, $[c_{1v}, t_{1v}]_2$ and $[c_{2v}, t_{2v}]_1$, $[c_{2v}, t_{2v}]_2$ where $c_{1v}, c_{2v}, t_{1v}, t_{2v}$ are new nodes.

Setting $T_{1v} = \{2, 3\}, T_{2v} = \{1, 2\}, T_v = \{1, 2, 3\}$, we get a new bipartite graph with maximum degree 3 (the construction is given in Figure 7). We repeat this for all nodes $v \in T$ with $d_G(v) = 1$

Then we get a new bipartite graph G' with $\Delta(G') = 3$; we set $\mathcal{T}_v = \{2, 3\}$ for all nodes v of degree 2 in $\mathcal{T}$ for which it has not been defined yet.

All nodes in $\mathcal{T}$ have now degree 2 or 3.

G' corresponds to a timetabling problem RTT$(C, \mathcal{T}, H, R, \mathcal{T})$ where all teachers are tight; furthermore all T_j's are intervals $\{1, 3\}, \{2, 3\}$ or $\{1, 2, 3\}$. Furthermore it follows from the construction of G' that RTT has a solution if and only if

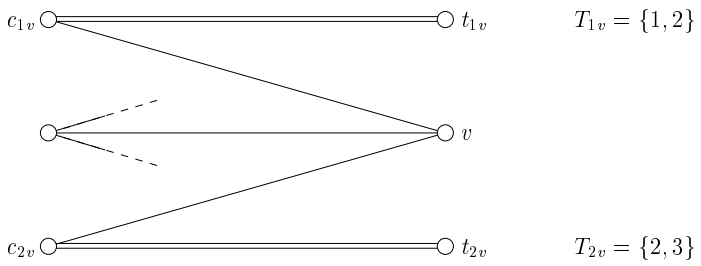

Fig. 7. Transformation of a node v of degree 1

G has an edge 3-coloring where the colors at each node $v \in T$ are $1, 2, \ldots, d_G(v)$. The result follows.

□

Again in terms of the image reconstruction problem, we notice that RP(m, n, p) remains NP-complete even if each color s is known to occur only in a subset of consecutive colors.

In some situation, problem TT($C, T, H, R, \mathcal{T}$) has to be solved while taking into account availability constraints related to the classrooms. We will also assume that $T_j = H$ for each teacher t_j, so that our problem is simply TT(C, T, H, R). The availability constraints are defined by values $\mathcal{H} = (h_1, h_2, \ldots, h_h)$ where h_k represents the number of classrooms (they are all identical) available for lectures at period k.

For TT($C, T, H, R, \mathcal{H}$) a solution is represented in the associated bipartite multigraph $G = (C, T, R)$ by an edge coloring $(M_1, \ldots, M_h)$ where the matching M_k has at most h_k edges, i.e., $|M_k| \leq h_k$ for $k = 1, \ldots, h$.

The following result is a consequence of properties stated in [8]:

Property 4.3. *TT($C, T, H, R, \mathcal{H}$) is NP-complete even if* $|H| = 3$.

The problem can also be reduced to a canonical form which will exhibit its relation with the other timetabling problems discussed earlier.

Consider TT($C, T, H, R, \mathcal{H}$) and apply the same transformation as in Property 4.1. Then we call $\mathcal{M}$ the set of all parallel edges of the form $[c_i, \overline{t}_i]$ and $[\overline{c}_j, t_j]$. Such a collection of node disjoint families of parallel edges is called a multi matching.

Property 4.4. *TT($C, T, H, R, \mathcal{H}$) can be transformed into problem RTT'(C', T', $H, R', \mathcal{H}, \mathcal{M}$) satisfying the following:*

1. *RTT' has a solution if and only if TT has one*
2. $|T'| = |T| + |C|$
3. $|C'| = |C| + |T|$
4. $\sum_i r'_{ij} = \sum_j r'_{ij} = |H| = h$ *for all* i, j
5. $C'_i = T'_j = H$ *for all* i, j

6. *In $\mathcal{M}$ at least $|T| + |C| - 2h_k$ edges (not specified in advance) receive color k (for $k = 1, \ldots, h$).*

Proof. We only have to establish 1. when 6. is given as constraint for RTT$'$.

Suppose TT has a solution; it means that there exists an edge 4-coloring $(M_1, \ldots, M_h)$ of G with $|M_k| \leq h_k$ for $k = 1, \ldots, h$. We can extend each M_k to a perfect matching in G': if an edge $[c_i, t_j]$ is in M_k, then we also introduce $[\overline{c}_j, t_i]$ into M_k; then one can introduce $|T| + |C| - 2h_k$ edges of the form $[\overline{c}_j, t_j]$ or $[c_i, \overline{t}_i]$ into M_k in order to obtain a perfect matching M'_k of G' (which contains M_k).

Repeating this for $k = 1, \ldots, h$, we get an edge h-coloring $(M'_1, \ldots, M'_h)$ of G' which satisfies the requirement that in $\mathcal{M}$ at least $|T| + |C| - 2h_k$ edges (not specified in advance in the data!) receive color k $(k = 1, \ldots, h)$.

Conversely, assume that for RTT$'$ there is a solution; it is represented by an edge h-coloring $(M'_1, \ldots, M'_h)$ of G' where

$$|M'_k \cap \mathcal{M}| \geq |T| + |C| - 2h_k \quad (k = 1, \ldots, h).$$

Remove the edges of $M'_k \cap \mathcal{M}$ from M'_k; we get a partial matching M^*_k in G'. Furthermore we observe that $[c_i, \overline{t}_i]$ (resp. $[\overline{c}_j, t_j]$) is in M'_k iff both c_i and $\overline{t}_i$ (resp. $\overline{c}_i$ and t_j) are not adjacent to any edge of M^*_k. If $[c_i, \overline{t}_j]$ (resp. $[\overline{c}_j, t_j]$) is not in M'_k then both c_i and $\overline{t}_i$ (resp. $\overline{c}_j$ and t_j) are adjacent to some edge of M'_k. Hence M^*_k contains the same number of edges $[c_i, t_j]$ with $i \leq m$, $j \leq n$ as of edges $[\overline{c}_j, \overline{t}_i]$ with $i \leq m$, $j \leq n$. Since $|M^*_k| = |M'_k| - |M'_k \cap \mathcal{M}| \leq 2h_k$, we will have $|M^*_k|/2 \leq h_k$ edges of M^*_k with endpoints in $C = \{c_1, \ldots, c_m\}$ and in $T = \{t_1, \ldots, t_n\}$. Let M_k be the set of these edges.

Repeating this for $k = 1, \ldots, h$, we will get an edge h-coloring $(M_1, \ldots, M_h)$ of G with $|M_k| \leq h_k$ for $k = 1, \ldots, h$; it defines a solution of TT. □

This construction is illustrated in Figure 8.

Fig. 9 gives a 4-coloring $M'_1, \ldots, M'_4$ corresponding to the example in Fig. 8. One sees that by considering the submatrix R of R' (first m rows and first n columns) we get a 4-coloring $(M_1, \ldots, M_4)$ of G with $|M_k| \leq h_k$ $(k = 1, \ldots, 4)$.

The timetable shown in Fig. 10 may be viewed as the matrix $\mathcal{A}$ of the associated $RP(m, m, p)$. In each one of the first two columns, we have at least one row $\overline{c}_j$ containing the associated colour t_j; in each one of the last two columns of $\mathcal{A}$ we have at least two rows $\overline{c}_j$ (or c_i) containing the associated colour t_j (or $\overline{t}_i$): in column 4 for instance, row c_2 contains colour $\overline{t}_2$, row $\overline{c}_1$ contains colour t_1 and row $\overline{c}_2$ contains colour t_2.

Since this timetabling problem is NP-complete, we can observe that it is also difficult to decide the existence of an edge h-coloring in a regular bipartite multigraph with $\Delta(G) \leq h$ and with the constraint that in a given multimatching $\mathcal{M}$ there are at least p_k edges (not specified in advance) which must get color k $(k = 1, \ldots, h)$.

In terms of image reconstruction problem RP(m, n, p), we have the following formulation.

A multimatching $\mathcal{M}$ corresponds to a collection of disjoint pairs $[\overline{c}_j, t_j]$ or $[c_i, \overline{t}_i]$ where $\overline{c}_j$ (resp. c_i) is a row and t_j (resp. $\overline{t}_i$) is a color. By calling $1, 2, \ldots, m$

	t_1	t_2	t_3
c_1	2		1
c_2		1	2

$H = \{1, \ldots, 4\}$

$\mathcal{H} = (h_1, h_2, h_3, h_4) = (2, 2, 1, 1)$

Is there a timetable with $\leq h_k$ lectures at period k $(k = 1, \ldots, 4)$?

Regularisation

	t_1	t_2	t_3	$\bar{t}_1$	$\bar{t}_2$
c_1	2		1	1	
c_2		1	2		1
$\bar{c}_1$	2			2	
$\bar{c}_2$		3			1
$\bar{c}_3$			1	1	2

R'

	t_1	t_2	t_3	$\bar{t}_1$	$\bar{t}_2$
c_1				1	
c_2					1
$\bar{c}_1$	2				
$\bar{c}_2$		3			
$\bar{c}_3$			1		

$\mathcal{M}$

G' regular $\Delta(G) = 4$

solution: edge coloring (M'_1, M'_2, M'_3, M'_4) of G' with

$$|\mathcal{M} \cap M'_1| \geq |T| + |C| - 2h_1 = 1$$
$$|\mathcal{M} \cap M'_2| \geq |T| + |C| - 2h_2 = 1$$
$$|\mathcal{M} \cap M'_3| \geq |T| + |C| - 2h_3 = 2$$
$$|\mathcal{M} \cap M'_4| \geq |T| + |C| - 2h_4 = 2$$

Fig. 8. Construction of RTT for classroom availabilities

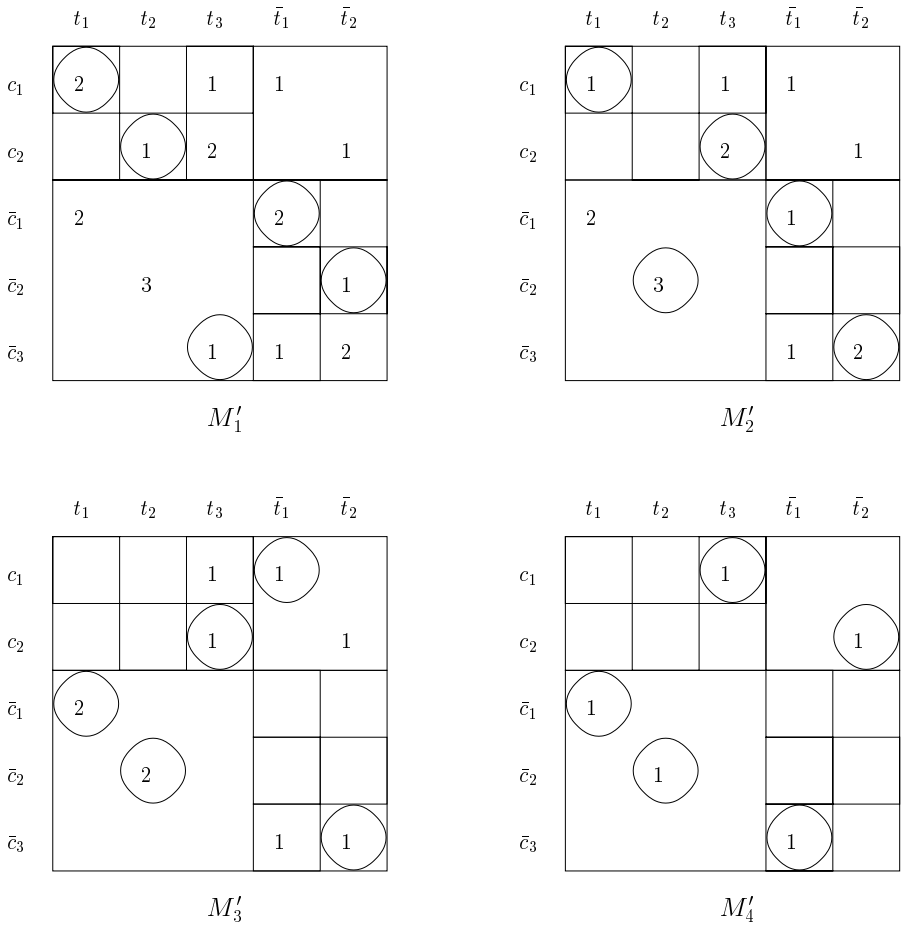

Fig. 9. The 4-coloring $(M'_1, \ldots, M'_4)$ of G'

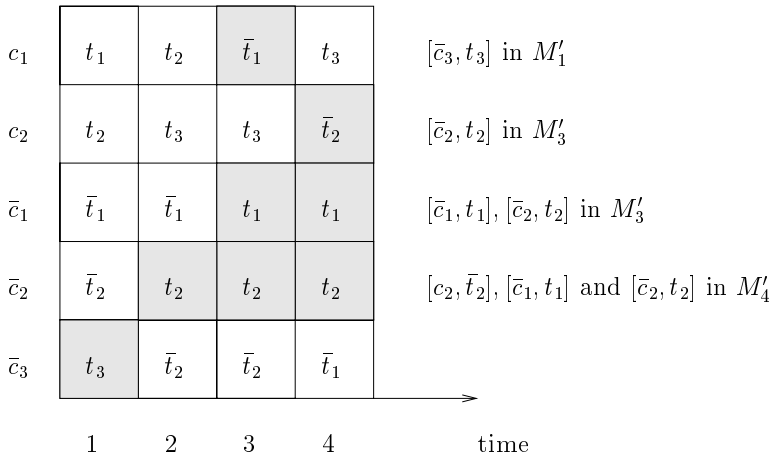

Fig. 10. The timetable associated to the coloring of Fig. 9

the rows of the array and $1, 2, \ldots, p$ the colours, we have the following formulation: in a problem $RP(m, m, p)$ where each colour occurs once in each column (so that $p = m$), we have to take into account the additional requirement: given integers $p_1, \ldots, p_n$ in each column k there are at least p_k rows i which contain precisely colour i (see figure 10 for an illustration).

5 Double lectures

We shall finally examine the case where all lectures do not have the same length. More precisely we shall assume that in addition the two one-period lectures given by a teacher t_j to a class c_i, there are also *double lectures* which are also given by a single teacher to a single class but they have a length of two periods; they consist of two normal lectures which have to be scheduled on two consecutive periods (in general, when we have lunch breaks or even breaks between a day and the next one, we have to avoid scheduling double lectures on the last period before a break; we shall not examine this here).

These double lectures may be viewed as creating some type of dynamic unavailability constraints: if a double lecture of t_j to c_i is scheduled to start at period k, then t_j and c_i become unavailable for other lectures at period $k + 1$.

We shall assume for the moment that there are no other unavailability constraints, i.e. given sets C, T, H of classes, teachers and periods, we have $C_i = T_j = H$ for all i, j.

In addition we are given a requirement matrix $R = (r_{ij})$; to keep notations simple we shall suppose that $r_{ij} \in \{0, 1, 2\}$ and whenever $r_{ij} = 2$, this means that t_j has to give a double lecture to c_i.

Let $TT(C, T, H, R, M_2)$ be the corresponding timetabling problem where M_2 defines the set of double lectures, i.e. the set of entries (i, j) of R with $r_{ij} = 2$.

We shall first transform the problem into a regular form in a similar way to the reductions described in the previous sections.

Notice that the following reduction is valid also if M_k is a set of k-tuple lectures.

Property 5.1. *$TT(C, T, H, R, M_k)$ characterized by a requirement matrix R with $r_{ij} \leq k$ for all i, j can be transformed into a problem $TT'(C', T', H, R', M'_k)$ satisfying the following:*

1. *TT' has a solution if and only if TT has one*
2. *$|C'| = |T'| = |T| + |C|$*
3. *$\sum_j r'_{ij} = |H|$ for all classes c'_i*
   $\sum_i r'_{ij} = |H|$ for all teachers t'_j
4. *$|M'_k| = 2|M_k|$*

Proof. We transform R into R' exactly as in Property 4.1. The multiple edges between $\overline{c}_j, t_j$ and between $c_i, \overline{t}_i$ correspond to single lectures; so 2. 3. and 4. are satisfied. To show that 1. holds, we observe that from any solution to TT$'$ we can trivially deduce a solution of TT by considering only the first m rows and the first n columns of R'.

Conversely if TT has a solution, we can take the symmetric solution for the submatrix of R' located in the last n rows and the last m columns:

> If t_j gives a lecture to c_i at period k, then $\overline{t}_i$ gives a lecture to $\overline{c}_j$ at the same period; moreover, if c_i (resp. t_j) has no lecture at period k, then c_i meets $\overline{t}_i$ (resp. t_j meets $\overline{c}_j$) at this period.
> This will give us a solution for TT$'$ which will respect the requirements of M'_k if those of M_k are respected. Hence 1. holds.

□

We shall restrict the problem to the special case of double lectures in the following complexity result:

Property 5.2 ([9]). *Consider problem $TT(C, T, H, R, M_2)$ with $h = |H| \geq 4$ periods, where TT is reduced as above. It is NP-complete to decide whether TT has a solution in h periods.*

There is however a solvable case that can be stated as follows:

Property 5.3. *Consider a reduced problem $TT(C, T, H, R, M_{h-1})$ with $h = |H|$ periods. Assume M_{h-1} contains only $(h-1)$-tuple lectures. Then TT has a solution in at most $2h-2$ periods. Moreover, there is a polynomial algorithm for constructing a solution in h periods if there is such a solution.*

From this we have immediately:

Corollary 5.1 ([9]). *Consider a problem $TT(C, T, H, R, M_2)$ with $h = |H| = 3$ periods. Then there exists a polynomial algorithm to decide whether TT has a solution in 3 periods. If there is no such solution, then one can get one in 4 periods.*

Proof (Property 5.3). Let G be the multigraph associated to problem TT(C, T, H, R, M_{h-1}). We can assume that TT has been reduced so that G is regular; if a teacher t_j (resp. a class c_i) has a multiple lecture (i.e. a $(h-1)$-tuple lecture), then it has to be scheduled either in the first $h-1$ periods or in the last $h-1$ periods.

In both cases, at periods $2, 3, \ldots, h-1$ the multiple lecture will be running.

There will be a timetable in exactly h periods if and only if G contains a perfect $(h-2)$-matching $\mathcal{M}$ (i.e. a partial subgraph having exactly $h-2$ edges at each node of G) that uses exactly $h-2$ edges of each multiple edge.

This can be seen easily by coloring with colours $2, 3, \ldots, h-1$ the edges of $\mathcal{M}$; the remaining edges of G form a 2-regular graph which can be colored with colours 1 and h. This gives a timetable in h periods where each multiple lecture is scheduled in h consecutive periods. $\mathcal{M}$ can be constructed in polynomial time by using network flow techniques (see [3] or [4]).

If there is no such 2-matching $\mathcal{M}$, we can construct a perfect $(h-2)$-matching as before; these edges are the lectures scheduled in periods $2, 3, \ldots, h-1$. The remaining graph has degree 2 and corresponds to periods 1 and h. All lectures have been scheduled, but some multiple edges do not have consecutive colors.

In fact, each $(h-1)$-tuple lecture has been scheduled within an interval of h periods. There is exactly one period l where no part of a given multiple lecture $c_i - t_j$ is scheduled. If $l = 1$ or h, we are done. If $2 \leq l \leq h-1$, we reschedule the $l-1$ first periods of $c_i - t_j$ in periods $h+1, h+2, \ldots, h+l-1$. This can be done for all such multiple lectures independently, because from our assumption no teacher and no class in involved in more than one multiple lecture. Since $l \leq h-1$, we will in all cases get a timetable in at most $2h-2$ periods.

□

One could interpret the timetable problem with multiple lectures in terms of image reconstruction; then it would simply mean that for some rows i and some colors s with $a(i,s) > 1$, color s has to occur in $a(i,s)$ consecutive entries of row i. The problem is difficult when there are at least 4 columns and when the multiple occurrences in a row are limited to pairs of consecutive occurrences, according to Property 5.2.

6 Extensions and conclusions

We have restricted our attention to some variations of the basic timetabling problem; this has allowed us to exploit the similarity with some elementary image reconstruction problems, like $RP(m, n, p)$.

We have not mentioned in an explicit way the analogies with other types of scheduling problems. like open shop scheduling (see [4]). In particular from the timetabling problem $TT(C, T, H, R, M_2)$ one may derive the fact that the open shop problem with processing times of value 0, 1 or 2 is difficult (when the total processing time is at least 4).

Our purpose was to examine the simple variations of the class teacher model, to reduce them to some "canonical" forms and to derive some complexity properties.

The knowledge of these properties will be useful when real cases have to be solved for instance by decomposing them into smaller problems which can be solved in polynomial time.

One could as well start from other basic timetabling models like the ones taking the classroom assignment into account, as in [5] or the models where group lectures (involving several classes at a time) are present, like in [1]. For these problems, the image reconstruction $RP(m, n, p)$ may not be the most natural neighbor problem to consider. It will be an interesting research area to explore this field with the objective of exploiting analogies either for the design of heuristics or for complexity studies.

Lots of other basic timetabling problems could have been discussed here, but a cat cannot run after several rats simultaneously.

References

[1] A.S. Asratian and D. de Werra. A generalized class-teacher model for some timetabling problems. *(to appear in European Journal of Operational Research)*, 2002.

[2] A.S. Asratian and R.R. Kamalian. Interval edge coloring of multigraphs. *Applied Mathematics Yerevan University (in Russian)*, 5:21–34, 1987.

[3] Claude Berge. Graphs and hypergraphs. *North-Holland, Amsterdam*, 1973.

[4] J. Blazewicz, K. Ecker, E. Pesch, G. Schmidt, and J. Weglarz. *Scheduling Computer and Manufacturing Processes*. Springer,Berlin, 1996.

[5] M.W. Carter and C.A. Tovey. When is the classroom assignment problem hard? *Operations Research*, 40 (Supp. 1):S28–S39, 1996.

[6] M.C. Costa, D. de Werra, and C. Picouleau. On some special cases of an image reconstruction problem. *(submitted for publication)*, 2002.

[7] S. Even, A. Itai, and A. Shamir. On the complexity of timetable and multicommodity flow problems. *SIAM J. Comput.*, 5:691–703, 1976.

[8] T. Gabow, O. Nishizcki, D. Kariv, O. Leven, and O. Terada. Algorithms for edge-coloring graphs. *unpublished manuscript, University of Colorado (Boulder)*, 1983.

[9] D.P. Williamson, L.A. Hall, J.A. Hoogeven, C.A.J. Hurkens, J.K. Lenstra, S.V. Sevastianov, and D.B. Shmoys. Short shop schedules. *Operations Research*, 45:288–294, 1997.

Integer and Constraint Programming Approaches for Round Robin Tournament Scheduling

Michael A. Trick

Graduate School of Industrial Administration, Carnegie Mellon, Pittsburgh, PA USA, 15213 trick@cmu.edu

Abstract. Real sports scheduling problems are difficult to solve due to the variety of different constraints that might be imposed. Over the last decade, through the work of a number of researchers, it has become easier to solve round robin tournament problems. These tournaments can then become building blocks for more complicated schedules. For example, we have worked extensively with Major League Baseball on creating "what-if" schedules for various league formats. Success in providing those schedules has depended on breaking the schedule into easily solvable pieces. Integer programming and constraint programming methods each have their places in this approach, depending on the constraints and objective function.

1 Introduction

There has been a lot of recent work on combinatorial optimization methods for creating sports schedules. Much of this work has revolved around creating single round-robin schedules, where every team plays every other team once, and double round-robin schedules, where every team plays every other team twice (generally once at its home venue and once at each opposing venue).

Round-robin scheduling is interesting in its own right. Some leagues have a schedule that is a single or double round-robin schedule. Examples of this include many U.S. college basketball leagues and many European football (soccer) leagues. For such leagues, the scheduling problem is exactly a constrained round-robin scheduling problem, where the constraints are generated by team requirements, league rules, media needs, and so on.

For other leagues, the schedule is not a round-robin schedule but it can be divided into sections that are round-robins among subsets of teams. This division is clearly a heuristic technique, but there are good reasons, both algorithmically and operationally, to make these divisions. By creating these sections, schedulers are able to use results about round-robin schedules to optimize the pieces. Operationally, such schedules are often appealing to the leagues since they offer an understandable structure and often are perceived as being fairer than unstructured schedules. For instance, by scheduling four consecutive round-robins,

each team will see each other in each quarter of the schedule; an unstructured schedule would not necessarily require that.

In the next section, we justify looking at round-robin schedules by looking at a large practical scheduling problem, that of Major League Baseball (MLB). Fully defining the MLB schedule is a daunting task, requiring the collation of more than 100 pages of team requirements and requests, along with an extensive set of league practices. Rather than fully define the problem, we outline the major constraints. We then explore integer and constraint programming approaches for round robin tournaments , and show that each method has advantages and disadvantages relative to the other. It appears that while we have a learned a lot about round-robin schedules, there are still many critical problems that we cannot solve: we conclude by listing some of these issues.

2 Major League Baseball

In the United States, Major League Baseball (MLB) is one of the "Big Four" professional sports (the others being the National Football League, the National Basketball Association, and the National Hockey League). In the summer months, MLB games dominate the television schedule; many Americans are able to watch a dozen or more games a week. For many Americans, a summer is not complete without going to a game with family or friends.

Baseball is also big business. In 2001, Major League Baseball estimated its total revenue in excess of $3.5 billion. As part of that value, individual teams earn up to $100 million in ticket sales and up to a further $50 million for local television coverage. MLB has a multi-billion dollar national television contract to show games on multiple national television channels. Individual players can earn up to $25 million per year and the average player in 2001 earned more than $2 million.

One important aspect of MLB is the schedule it plays. Fans care about the schedule since it determines when they can see their team play in the local stadium. Television networks care about the schedule since they need to show games on a consistent basis and a night without a "good" matchup will translate into lower viewership and advertising revenues. Players care about the schedule since they are the ones who have to do the travel and play the games.

Creating the MLB schedule is a difficult task. There are 30 teams playing 162 games each (2430 games total) over a 180 day time period, making this one of the largest sports scheduling problems in existence. Each team has dozens of requests and requirements, many of them contradicting the needs of other teams.

Complicating this situation is the need to create several hypothetical schedules. Every year, MLB considers a number of alternative structures to its league format. Should there be more teams? Should there be fewer? Currently, teams are divided into two leagues, and each league is divided into three divisions. Should teams play more games within their division? Should the number of divisions be changed? For each of these alternatives, it would be useful to have some insight into the effect on the schedule. Does the amount of travel increase

or decrease? Will team requirements be easier to meet or harder? Are some alternatives easier to schedule, creating many choices for the league to choose from?

Since 1996, my partner (Doug Bureman, a former Senior Vice President of a MLB team) and I have worked with MLB to create schedules for hypothetical situations (we are MLB's "backup schedulers"). One key to our ability to create schedules (and the problem size means we are often working at the very limits of computational feasibility) is our breaking up of a schedule into manageable pieces. Normally, this means breaking the schedule in a series of round robin schedules either among all the teams or among subsets of teams. These schedules have to meet a number of restrictions. Here we outline some of the most critical requirements on the schedules. Cain [3] provides a description of the issues MLB faced in the mid-1970s.

The current situation in MLB is as follows. The 30 teams of MLB are divided into two leagues: the National League with 16 teams, and the American League with 14 teams. Each team plays 162 games over the course of the season, which runs from April through September. For most of the year, the Leagues play independently. Each team plays only 12 to 15 games against teams in the other league. Each league is divided into 3 divisions: the East, the Central, and the West. The National League has 5 teams in the East, 6 in the Central, and 5 in the West (5-6-5), while the American league is 5-5-4.

No matter what the format, there are certain requirements on any Major League Baseball schedule. Each team must play 162 games, half at home and half away. These games are divided into series: 2 to 4 games played consecutively against an opponent. Each week is divided into two slots: the during-week slot and the weekend-slot. With a limited number of exceptions, each team will play one series per slot. Since the schedule is 25 and a half weeks long, there are 51 slots in a schedule.

It may be that the number of series does not exactly equal 51. If there are more series than slots, it is possible to create a *squeeze week*, where two series are put into one slot, generally a during-week slot.

The following are the most important constraints on a schedule:

1. No team can have five or more consecutive home slots or consecutive away slots.
2. Every team must have 13 home weekend-slots.
3. No team can have more than three consecutive home weekend-slots or consecutive away weekend-slots.
4. Four cities have a team in each of the two leagues: such cities should have only one team at home at any time.
5. No team begins or ends the season with more than 2 consecutive home or 2 consecutive away slots.
6. No pair of teams have consecutive series against each other.
7. No pair of teams can have consecutive series against each other at the same venue in a six slot sequence.

In addition, the league has preferences on the slots:

1. Strong dislike for four consecutive away or consecutive home slots.
2. Strong dislike for three consecutive away or consecutive home weekend-slots.
3. Weak dislike for singleton home or away series.
4. Weak dislike for pairs of teams that play twice in a three series sequence (must be once at home and once away).

The league would also like to keep each team's travel low, due primarily to wear-and-tear on the players. The travel is not just the miles flown, but also includes penalties for time zone changes. Teams are assumed to begin and end the season at their home venue and otherwise travel directly from one opponent to the next. No distance is charged, of course, if a team has consecutive home slots.

In addition to the travel costs, there is a value to having particular games in certain slots. For instance, many teams have natural rivals: teams wish to see such rivals on summer weekends. The league may want to have more intra-divisional games at the end of the season. Television preferences may push towards certain games. All of this comes together in a complicated objective function.

Once we have a slot schedule, we then assign games to the days consistent with the slot schedule. The requirements for this assignment are

1. Teams play at most one game per day
2. Week-day slots are scheduled from Monday through Thursday
3. Week-end slots are scheduled on Thursday through the following Monday (note that slots overlap).
4. Every team plays on every Friday, Saturday, and Sunday
5. No team plays more than 20 consecutive days.
6. No team gets two consecutive days off.

During a squeeze week, if there are two series in a during-week slot, then the first series will be played on Monday and Tuesday and the second will be played Wednesday and Thursday.

MLB's scheduling problem seems too difficult to solve in one piece: the problem is too big and the constraints are too complex. Instead, a reasonable approach is to divide the schedule into smaller sections and solve the sections in sequence. For instance, for the schedule we designed for 2002, we designed a schedule that began with a double round-robin tournament among all the teams, followed by a single round-robin tournament among all the teams, finishing with double round-robin tournament within the divisions. This gave a schedule where teams played 5 series against every team in their division, and 3 against those outside their division. The smaller sections are much easier to solve.

If we break the schedule into smaller pieces, there are a number of additional constraints that the pieces must satisfy. These might include:

- Limitations on number of home versus away series. Normally, we try to keep these close in all phases of the schedule.
- Limitations on weekend home and away counts.

– Limitations on home/away patterns beginning and ending a section of the schedule, to allow for smooth transitions between sections.
– Limitations on permissible opponents. For instance, television issues might require a particular matchup. Alternatively, results from other sections of the schedule might preclude certain matchups at certain times (so if a section ends with Team A playing Team B, then the next section cannot begin with that same series).
– Objective values to be satisfied, including travel costs, bad home/away patterns, and so on. A bad travel section for a team can be offset by good travel in another section.

Our goal is therefore to find techniques for solving the sections that can robustly handle the variety of constraints that are generated through this process.

3 Round Robin Scheduling

We begin with the most fundamental problem in sports scheduling: designing a round robin schedule. In a round robin schedule, there are an even number n teams each of whom plays each other team once over the course of the competition. We will work only with *compact* schedules: the number of slots for games equals $n-1$, so every team plays one game in every slot. We refer to this as a Single Round Robin (SRR) Tournament.

A related problem is the Bipartite Single Round Robin (BSRR) Tournament problem. Here, the teams are divided into two groups X and Y, each with $n/2$ teams. There are $n/2$ slots during which all teams in X need to play all teams in Y, but teams within X (and within Y) do not play each other.

Henz, Müller, and Thiel [12] (who we will refer to as HMT) have recently examined SRR carefully in the constraint programming context. We expand on their work by examining integer programming formulations and further exploring the strengths and weaknesses of the different approaches.

HMT point out that the SRR can be formulated with two major types of constraints. First, the games in every slot correspond to a one-factor (or matching) of the teams. Second, for any team i, its opponents across all of the slots must be exactly the set of teams except for i. We will call the first the *one-factor constraint* and the second the *all-different constraint*. Different formulations have different ways of encoding and enforcing these constraints.

3.1 Integer Programming Formulation

Our basic integer program for SRR begins with binary variables x_{ijt} which is 1 if teams i and j play each other in slot t, and is 0 otherwise. Since the order of i and j does not matter, we could either define this only for $i > j$ or set $x_{ijt} = x_{jit}$ for all i, j, t.

This leads to the formulation (in the OPL language[25]):

```
int n=...;
range Teams [0..n-1];
range Slots [1..n-1];
range Binary 0..1;
var Binary plays[Teams,Teams,Slots];

solve {
   //No team plays itself
   forall (i in Teams, t in Slots) plays[i,i,t] = 0;
   //Symmetry in plays variables
   forall (ordered i,j in Teams, t in Slots)
      plays[i,j,t] = plays[j,i,t];

   //Every team plays one game per slot
   //One-factor constraint
   forall (i in Teams, t in Slots)
      sum (j in Teams) plays[i,j,t] = 1;

   //Every team plays every other team
   //All-different constraint
   forall (i,j in Teams: i<>j) sum(t in Slots) plays[i,j,t] = 1;

};
```

We call the above the Base-IP Formulation. We can strengthen this formulation by a stronger modeling of the one-factor constraint. The polyhedral structure of the one-factor polytope is perhaps the most well-studied polytope in combinatorial optimization. Edmonds [9] showed that the polytope is defined by adding *odd-set* constraints. In this context, the odd set constraints are as follows. For a particular slot t, let S be a set of teams, $|S|$ odd. Then,

$$sum_{i \in S, j \notin S} x_{ijt} \geq 1$$

is valid for the one-factor constraint. If we add all of these constraints, then the one-factor constraint is precisely defined in the polyhedral sense (all extreme points of the polyhedron are integer).

We call the formulation with all of the odd-set constraints the Strong-IP formulation. Of course, there are too many odd-set constraints to simply add them all to the integer program. We can solve Strong-IP by using a constraint generation method. In this method, we begin with a limited set of odd-set constraints and solve the linear relaxation of the instance. We then identify violated odd-set constraints (this can be done with a method by Padberg and Rao [16] using cut-trees) and add them to the formulation. We repeat until either we have added "enough" constraints or until all odd-set constraints are satisfied. At that point, we then continue our normal branch and bound approach to integer programs.

Strong-IP is not needed for the BSRR problem. In this case, the odd set constraints are redundant, so need not be added.

3.2 Constraint Programming Formulation

HMT extensively analyze constraint programming formulations for SRR. Their basic variables are `opponent[i,t]` which gives the opponent i plays in slot t. Their formulation can be represented very simply:

```
int n=...;
range Teams [0..n-1];
range Slots [1..n-1];
var Teams opponent[Teams,Slots];

solve {
   //No team plays itself
   forall (i in Teams, t in Slots) opponent[i,t] <>i;

   //Every team plays one game per slot
   //One-factor constraint
   forall (t in Slots)
      one-factor(all (i in Teams) opponent[i,t]);

   //Every team plays every other team
   //All-different constraint
   forall (i in Teams)
      all-different(all (t in Slots) opponent[i,t]);

};
```

The key issue is to define how the one-factor and all-different constraints are implemented. There are a variety of propagation algorithms available for each. HMT argues convincingly that the all-different constraint should use arc-consistent propagation by the method of Régin [17]. Briefly, arc-consistency means that the domains of the variables are such that for any value in a domain, setting the variable to that value allows settings for all the other variables so that the constraint is satisfied. For more details on the fundamentals of constraint programming, see [14].

For the one-factor constraint, the main emphasis of HMT, they examine three different approaches. The first is the simplest. It uses the constraints:

```
forall (i in Teams)
   opponent[opponent[i,t]] = i;
```

This set of constraints is sufficient to define the one-factor constraint, but its propagation properties are not particularly strong. In particular, the only domain reduction that is done is when i is in the domain for j for a particular time period t, but j is not in the domain of i for that time period. In that case, i can be removed for the domain of j.

The propagation properties of this constraint can be improved by adding the redundant constraint:

```
all-different(all (i in Teams) opponent[i,t]);
```

HMT give an example where adding the all-different constraint leads to improved domain reduction. We call this combination the *all-different* one-factor approach.

The combination of these constraints do not create an arc-consistent propagation for the one-factor constraint. HMT provide an arc-consistent propagation method using results from non-bipartite matchings. In general, this approach is much better than the all-different approach. The exceptions mimic when the Strong-IP does not improve on Basic-IP: if the underlying graph is bipartite, then the all-different approach is arc-consistent.

To prove this, let D_i be the feasible opponents for i. We say the D_i are bipartite if we can divide the teams into X and Y such that

1. $|X| = |Y| = n/2$
2. $i \in X \rightarrow D_i \subseteq Y$
3. $i \in Y \rightarrow D_i \subseteq X$

Theorem 1. *If the D_i are bipartite, then arc-consistency for the constraints*

```
forall (i in Teams)
  opponent[opponent[i]] = i;
all-different(opponent[i]);
```

implies arc-consistency for `one-factor(opponent)`

Proof. Suppose the D_i are consistent for the constraints

```
forall (i in Teams)
  opponent[opponent[i]] = i;
all-different(opponent);
```

Let $j \in D_i$. We will show there is a one-factor that has j as the opponent for i. Without loss of generality, we will assume $j \in Y$, so $i \in X$. By consistency of the all-different constraint, there is a setting of opponent values so that `j=opponent[i]`, and `all-different[opponent]`. Create a new setting of the opponent values `opponent'` such that `opponent'[i] = opponent[i]` for $i \in X$ and `opponent'[i] = opponent[opponent[i]]` for $i \in Y$. By the consistency requirement, `opponent'[i]` $\in D_i$ for all i. `opponent'` is therefore a one-factor that has `j=opponent[i]` as required. □

Therefore, for either BSRR or for cases where the home/away pattern for a time slot is fixed (creating a bipartition between teams that need to be home versus those that need to be away), one-factor propagation can be replaced by all-different propagation.

3.3 Computational Tests

If there were no further requirements on the schedules, creating a SRR schedule would be straightforward. Kirkman (1847) gave a method for creating such a schedule (outlined in [1]) which also shows there is an ordering of the decision variables such that a constraint program would not need to backtrack in assigning variables (provided arc-consistent approaches to the all-different constraint is used).

Most league schedules have a number of additional constraints, however. A few of the most common are:

- Fixed games. A set of games are fixed to occur in certain slots.
- Prohibited games. A set of games are fixed not to occur in certain slots.
- Home/Away restrictions. Each team has a home venue, and each game must be assigned to a venue. There are additional constraints on such things as the permitted number of consecutive home or away games.

It might seem that adding constraints would make the problem easier, since it reduces the possible search space. In fact, adding fixed games (or, equivalently, prohibiting games) can make the relatively easy problem of finding a schedule become an NP-complete problem. This has been shown by Colbourn [5] for the bipartite SRR case, where a schedule is equivalent to a Latin Square, and by Easton for the general SRR case. So it is clear that there may be very difficult instances of these restricted problems.

There may also be an objective function to be optimized. For instance, there may be an estimate c_{ijt} for the number of people who would attend a game between i and j during time slot t. Can we maximize the total number of people who attend games during the tournament?

All the testing in this paper was done using ILOG's OPL Studio version 3.5 ([13]) running under Windows XP on a 1.8Gz Pentium IV processor computer with 512Mb of memory.

Tightly Constrained Round Robin Tournaments For our first test, we use a data set from HMT, called "Tightly Constrained Round Robin Tournaments". For these instances, there are random forbidden opponents, with a sufficient number to lead to instances with very few or no feasible schedules.

For this test, we compare two codes: Basic-IP and the all-different constraint program. Our codes were implemented within the OPL system, version 3.5 and used default branching strategies for the integer program and default search strategies for the constraint program.

In addition, we repeat the computational results of HMT for their arc-consistent one-factor method. To offset different machine capabilities, we divided their computation times by 4.5 to represent a rough approximation of the difference between their 400Mz machine and our 1.8Gz machine.

For each instance, we give the number of failures (F) in the search tree (for the constraint programs) or number of nodes (N) in the search tree (for the

integer program) as well as the computation time in seconds. In the following the instances that end in "yes" are feasible, though generally with a small number of solutions; those that end in "no" are infeasible. For feasible problems, the timing for HMT includes work needed to find all solutions, while those for the IP and the all-different CP only find the first solution. This suggests that the "no" instances, where the codes perform the same task, is the fairer comparison.

Problem	n	all-different		Basic-IP		HMT	
		F	T	N	T	F	T
s_6_yes	6	5	0.00	0	0.01	4	0.01
s_8_yes	8	17	0.01	4	0.04	10	0.04
s_10_yes	10	4	0.02	1	0.09	1	0.02
s_12_yes	12	376	0.41	57	0.46	179	1.39
s_14_yes	14	862	1.24	276	5.54	527	4.53
s_6_no	6	3	0.00	0	0.01	4	0.00
s_8_no	8	11	0.01	10	0.04	6	0.01
s_10_no	10	23	0.02	4	0.10	6	0.03
s_12_no	12	24	0.07	0	0.14	25	0.17
s_14_no	14	135	0.23	50	1.02	69	0.56
s_16_no	16	79	0.30	0	0.39	86	1.19
s_18_no	18	43	0.32	0	0.42	30	0.50
s_20_no	20	696.30	5.47	0	0.78	254	5.11

Table 1. Benchmarks on Tightly Constrained SRR

Basic-IP is competitive with the constraint programming approaches, and can do markedly better in proving infeasibilty (as in `s_20_no`).

This table might lead to the conclusion that these tightly constrained SRR problems are relatively easy to solve in this size range. That is not the case. The instance `s_16_no` is actually just one of a series of instances, corresponding to varying numbers of prohibited games. The instance begins with 1192 prohibited games. The timing above corresponds to prohibiting all but the final 7 of the prohibited games. We can create new instances by varying the number of prohibited games. The instances remain infeasible through prohibiting all but the final 54 games, and which point the instance becomes feasible.

The computational effort for these instances varies tremendously for the Basic-IP and the all-different approaches, as illustrated in Figure 1. Basic-IP generally does poorly for a broad range of prohibitions; the computation time for the feasible instance that is created by prohibiting all but the final 80 games is more than 9000 seconds. The all-different constraint program never does that poorly but can still take more than 300 seconds for various instances.

Clearly many of these instances are difficult for our codes. For the final version of this paper, we plan to compare both the Strong-IP and HMT's one-factor approach on these instances. We expect both codes to exhibit similar behavior, though not as extreme as the current codes.

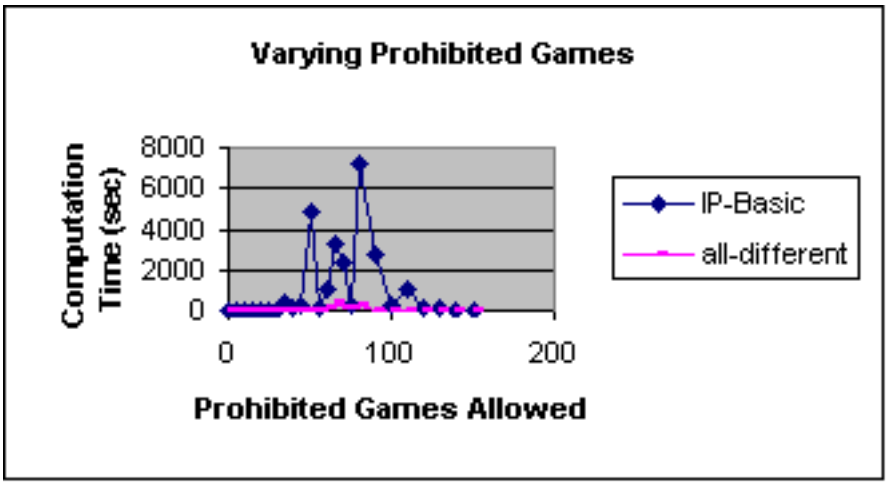

Fig. 1. Effect of Varying Number of Prohibited Games

Divisional Schedules There is a natural set of restrictions that are extremely difficult for the constraint-based formulations but are solved much more quickly by the integer programs. Many leagues are divided into two or more divisions. In such cases, some leagues like to begin by playing games between divisions and finish the schedule with games between divisional opponents. For two equally sized divisions, this approach works fine if n is divisible by 4, but does not work well for cases where $n = 2 \mod 4$. In that case, divisions have an odd number of teams, so no compact round robin schedule is possible with only divisional play at the end.

We can create difficult instances by fixing a large number of games. Suppose there are n teams, with $n = 2 \mod 4$, numbered 0 up to $n-1$. Divide the teams into two groups $X = [0..n/2-1]$ and $Y = [n/2..n-1]$. For the first $n/2-1$ slots, play X and Y as a bipartite tournament, leaving one game between X and Y unplayed for each team. Then, in slot $n/2$ fix two of the remaining games between X and Y. For six teams, the schedule might be:

	Team					
Slot	0	1	2	3	4	5
1	3	4	5	0	1	2
2	4	5	3	2	0	1
3	5	3		1		0
4						
5						

With these fixtures, the schedule is infeasible. In the above example, teams 2 and 4 need to play in slot 3, and then 0, 1, and 2 need to play a round robin among themselves in the remaining two slots, which is impossible.

The size 6 example is easy for any approach; things get more interesting for larger problems, as shown in Table 2. In this table, the dashes mean that no proof of infeasibility was found in half an hour of computation time (1800 seconds).

For the Strong-IP, the only needed constraints are for the odd sets associated with each division in each time period. That is sufficient for the linear relaxation to be infeasible for every n, which gives the near-constant computation times. For the one-factor method of HMT, there is no immediate proof of infeasibility for $n \geq 10$ by domain reduction, so at least some branching is needed (and we believe the amount of work will be significant).

Size	all-different		Basic-IP		Strong-IP	
	F	T	N	T	N	T
10	116	0.20	393	0.19	0	.20
14	—	—	—	—	0	.34
18	—	—	—	—	0	.32
22	—	—	—	—	0	.38

Table 2. Divisional Schedules (30 minute time limit)

Maximum Value Schedules As a final test for SRR, we randomly generated values for each game in each slot and tried to find the maximum value schedule. For an instance of size n, we independently generated a value uniformly among the integers $1 \ldots n^2$ for each game (i, j, t). There were no other restrictions on the schedule. We also generated bipartite versions of these problems (denoted b in the tables below).

For these sorts of optimizations, the search strategy is critical for constraint programming. The strategy used was to set the variables slot by slot, beginning with the first slot. The opponents for each team are ordered in decreasing order by value, and high value opponents are tried first.

The results are shown in Table 3. Not surprisingly, the integer programming based approach does much better at this test. In order for constraint programming to be competitive in this test, some sort of cost-based domain reduction would have to be done. Note that even the bipartite problems are difficult for constraint programming despite the arc-consistent propagation we do through the all-different constraint.

Size	all-different		Basic-IP	
	F	T	N	T
8	84962	5.33	0	.03
10	—	—	66	.29
12	—	—	402	3.59
14	—	—	7263	133.03
8b	1458	0.04	0	0.02
10b	3832	0.36	0	0.04
12b	4800172	216.75	0	0.09
14b	—	—	0	0.10

Table 3. Maximum Value Schedules (30 minute timelimit)

3.4 Home/Away Pattern Restrictions

The final set of constraints we would like to consider is extremely important in practice. For some leagues, every team has a home venue and every game is played at the home venue of one of the two teams competing. In this situation, there are often constraints on the home and away patterns of these teams. These constraints might include

- Restrictions that a particular team be home (or away) in a particular slot.
- Limitations on the number of consecutive home (or away) games a team may play.
- Requirements on the number of home games that must appear in some subset of the slots. For instance, a team might want to be at home at least half of the weekend games, or half of the games during the summer.
- Restrictions on pairs of slots. For instance, a if a team begins with an away game, the final game of the tournament might be required to be a home game.

There has been much work on scheduling with home/away patterns. Much of this work has concentrated on multiple phase approaches, where first the home/away pattern is fixed, and then the games are chosen consistent with this pattern (see, for example, [6, 7, 23, 20, 15, 11]. Alternatively, some work has reversed the process where first the games are chosen and then the home/away pattern chosen ([19, 24]). While these approaches are often very successful, there are cases where they do not work very well. For instance, depending on the restrictions on home/away patterns, there can be a huge number of feasible patterns, and a correspondingly large number of basic match schedules (see [23]). Enumerating and searching through all of them can be a computationally prohibitive task.

Is it possible to have one model that contains both game assignment and home/away pattern decisions? Conceptually, the models are straightforward to formulate. We consider a *double round robin* (DRR) tournament where every team plays every other team twice, once at home and once away.

For the integer program, we reinterpret the x_{ijt} variables to mean that i plays at j during slot t. We also create auxilliary variables h_{it} which is 1 if i is home in slot t and 0 otherwise. Clearly $h_{it} = \sum_j x_{jit}$.

For the constraint program, there are a number of possible formulations. For this test, in order to maximize the effect of the `all-different constraint`, we used the variables `plays[i,j]` to be the slot number in which i plays at j. This gives the basic formulation of

```
forall (i in Teams)
    all-different(all (j in Teams) plays[i,j],
                  all (j in Teams) plays[j,i]);
```

(For notational convenience, we actually created a n by 2n array of variables with `plays[i,j]` for `j <=n` giving the slot where i is at home to j and for `j>n` giving the slot where i is away to $j - n$, but will continue the exposition with the original variables).

Our home/away variables are given by

```
forall (i, j in Teams)
   home[i,plays[j,i]] = 1;
   home[i,plays[i,j]] = 0;
```

For both the integer and constraint programming approaches, some schedule requirements are easy to formulate with these variables. Fixing teams to be at home (or away) at a particular time is simply a matter of fixing the h (or `home`) variable to take on the appropriate value. Fixing the number of home games in a subset of slots is simply a linear constraint on the sum of the home variables. Putting an upper bound on the number of consecutive home games can be done as follows: if no more than k consecutive home games are permitted, then for every i and t, add a constraint

```
sum(t1 in Slots: t>=t & t<=t+k) home[i,t1] <= k
```

A similar restriction on consecutive away games can be done by using `1-home[i,t1]`.

OPL has a stronger way of handling these constraints: the `sequence` constraint allows for the explicit bounding of the number of times a value can appear in a subsequence of an array. We add to the constraint program the redundant constraints that half the teams must be at home in every slot (without them, the constraint program works very poorly).

Our first test is to simply determine whether our programs can find schedules in the absence of additional constraints. It was shown in HMT that constraint programs can find unrestricted schedules quickly for more than 20 teams (and more than 40 teams with their one-factor improvements). How does adding home and away requirements affect that?

In the following table, we give the time to find one schedule with n teams and an upper bound of k consecutive home or away games. For $k = 1$, there is no feasible schedule, so the time given is the time to prove infeasibility. With just one exception, the constraint programming approach did much better, though

		Integer Program		Constraint Program	
n	k	N	T	F	T
8	1	4	1.04	40	0.05
8	2	7	1.41	6	0.05
8	3	4	1.04	21	0.04
8	4	0	0.56	4	0.02
10	1	6	8.82	40	0.01
10	2	4	5.92	199	0.24
10	3	1	2.87	462	0.44
10	4	6	3.72	1141	0.98
12	1	6	24.84	220	0.54
12	2	4	17.29	2	0.84
12	3	4	15.11	—	—
12	4	17	32.42	0	0.12
14	1	2	59.71	312	1.21
14	2	9	70.10	11	0.20
14	3	11	82.34	3	0.18
14	4	20	169.42	2	0.19
16	1	2	163.52	420	2.48
16	2	35	604.86	184	0.74
16	3	—	—	197	.32
16	4	124	1557.02		
18	1	2	669.14	544	4.82
18	2	28	892.64	227	1.16
18	3	—	—	9	.042
18	4	—	—	1	.047

Table 4. Length Constrained H/A Schedules (30 minute limit)

the integer program was able to generate solutions in a reasonable amount of time.

The entry for the constraint program for $n = 12$, $k = 3$ is not a misprint: despite the ease at which the constraint program solved the other instances, the search went poorly in this case, and no solution was found within one-half hour. This suggests that either an improved search procedure is needed (we simply instantiated the `play` variable before the `home` variable team-by-team) or a stronger propagation algorithm is needed to ensure consistency in computation time. Still, it is clear that constraint programming is by far superior at this stage.

To move closer to to the types of schedules needed in practice, we add a constraint that there cannot be any length-one home stands or road trips. This is done by adding constraints of the form

```
home[i,t] <= home[i,t-1]+home[i,t+1];
(1-home[i,t]) <= (1-home[i,t-1])+(1-home[i,t+1]);
```

with the obvious changes for the beginning and end of the schedule. This makes the $k = 2$ instances infeasible.

The results are shown in Figure 5. Clearly this approach to limiting the home/away pattern is not consistent with the rest of the constraint programming model: constraint programming is unable to find any feasible solutions with half an hour. The integer programs are slow, but do find solutions. Given the success the multiple-phase approaches have with instances like this, it is clear that a smarter search rule (mimicking the multiple phase approach) or a better propagation rule should have significant effect on the constraint program.

		Integer Program		Constraint Program	
n	k	N	T	F	T
8	2	1427	21.42	2166	0.99
8	3	513	60.12	—	—
8	4	750	83.86	—	—
10	2	115	111.09	42768	26.18
10	3	1354	921.6	—	—
10	4	2214	1290.38	—	—

Table 5. Length Constrained H/A Schedules, No Singles (30 minute limit)

Finally, we added values for matchups on particular days, generating random values in the range $1 \ldots n^2$ for each pairing in each slot. Unfortunately, neither code at this stage can solve even the $n = 8$, $k = 3$ instance with a no-singleton constraint within 30 minutes. The results in the table are without the no-singleton constraint.

One final set of requirements we have not yet included has to do with travel requirements on teams. For leagues like MLB where travel a concern, it is important to minimize the travel distances of each team. Unfortunately, the direct

		Integer Program		Constraint Program	
n	k	N	T	F	T
8	3	1516	22.73	—	—
8	4	77	0.92	—	—
10	3			—	—
10	4	15268	594.70	—	—

Table 6. Length Constrained H/A Schedules, No Singles (30 minute limit)

formulation of this does not lead to solvable models. More complicated models involving different variables seems to be needed [10].

We can, within the models given, preclude terrible travel by including constraints that require trips from, say, the east coast to the west coast to include at least two west coast teams before returning. Such constraints are similar to (for both formulation and computation) the constraints that preclude length-one homestands and roadtrips.

4 Conclusions

We have shown that round-robin schedules with constraints of practical interest can be modeled by both constraint and integer programming techniques. The constraint programs were often faster except when there was an objective function, or in certain infeasible cases where the propagation was not strong enough to recognize infeasibiilty.

Returning to the Major League Baseball example, once the problem has been divided into smaller pieces, it is clear that IP/CP approaches are reasonable methods to solve the sections. Computation times are low enough to allow for multiple iterations of each section. In these iterations, constraints and objectives can be modified to push the process towards a good overall schedule.

In the course of this study, a number of gaps in current knowledge have been identified, and these make interesting future research directions.

- Is it worth adding constraints via the Strong-IP formulation? Henz, Müller, and Thiel [12] show that stronger propagation is generally a good idea for constraint programs. Is it also true that stronger relaxations are good for these integer programs?
- How can costs be better handled for the constraint programs? Handling costs is an active issue in the constraint programming community, and round-robin scheduling makes a good test-bed for these approaches.
- Clearly it would be good to include the strong propagation of HMT to the home/away models. Is there stronger propagation available combining the opponents with the home/away structures? Are there additional constraints that can be added to the integer programs? Is there a better way of handling home/away models which does not require a multi-phase approach?
- Can the integer programming and constraint programming approaches be usefully combined for these problems?

Round-robin scheduling makes an interesting test-bed for exploring algorithmic issues in combinatorial optimization. Success in this scheduling also provides the building blocks for scheduling of real-world sports leagues.

References

1. Anderson, I. 1997. *Combinatorial Designs and Tournaments*, Oxford University Press.
2. Ball, B.C. and D.B. Webster. 1977. "Optimal scheduling for even-numbered team athletic conferences", AIIE Transactions 9, 161-169.
3. Cain, W.O., Jr. 1977. "A computer assisted heuristic approach used to schedule the major league baseball clubs", in Optimal Strategies in Sports, S.P. Ladany and R.E. Machol (eds.), North Holland, Amsterdam, 32-41.
4. Campbell, R.T. and D.-S. Chen. 1976. "A minimum distance basketball scheduling Problem", in Management Science in Sports, R.E. Machol, S.P. Ladany, and D.G. Morrison (eds.), North-Holland, Amsterdam, 15-25.
5. Colbourn, C.J. 1983. "Embedding partial Steiner triple Systems is NP-complete", Journal of Combinatorial Theory, Series A, 35, 100-105.
6. de Werra, D. 1980. "Geography, games, and graphs", Discrete Applied Mathematics 2, 327-337.
7. de Werra, D. 1988. "Some models of graphs for scheduling sports competitions", Discrete Applied Mathematics 21, 47-65.
8. Easton, K.K. 2002. *Using Integer Programming and Constraint Programming to Solve Sports Scheduling Problems*, doctoral dissertation (in preparation), Georgia Institute of Technology.
9. Edmonds, J. 1965. "Maximum matching and a polyhedron with (0,1) vertices", *Journal of Research of the National Bureau of Standards Section B*, 69B, 125-130.
10. Easton, K.K., G.L. Nemhauser, M.A. Trick. 2002. Solving the Traveling Tournament Problem: A Combinted Integer Programming and Constraint Programming Approach, PATAT IV, Gent, Belguim.
11. Henz, M. 2001. "Scheduling a Major College Basketball Conference: Revisted", Operations Research, 49, 163-168.
12. 2002. Henz, M., T. Müller, and S. Thiel, "Global Constraints for Round Robin Tournament Scheduling", to appear, European Journal of Operational Research.
13. ILOG. 2000. "ILOG OPL Studio", User's Manual and Program Guide.
14. Marriott, K. and P.J. Stuckey. 1998 *Programming with Constraints: An Introduction*, MIT Press.
15. Nemhauser, G.L. and M.A. Trick. 1998. "Scheduling a Major College Basketball Conference", Operations Research, 46, 1-8.
16. Padberg, M.W. and M.R. Rao. 1982. "Odd minimum cut-sets and b-matchings", Mathematics of Operations Research, 7, 67-80.
17. Régin, J.-C., 1994. "A filtering algorithm for constraints of difference in CSPs", *Proceedings of the AAA 12th National Conference on Artificial Intelligence*, 362–367.
18. Régin, J.-C., 1999. "The symmetric alldiff constraint", in T. Dean (ed) *Proceedins of the International Joint Conference on Artificial Intelligence*, 1, 420-425.
19. Régin, J.-C. 1999. "Minimization of the Number of Breaks in Sports Scheduling Problems using Constraint Programming", DIMACS Workshop on Constraint Programming and Large Scale Discrete Optimization.

20. Russell, R.A. and J.M Leung. 1994. "Devising a cost effective schedule for a baseball league", Operations Research 42, 614-625.
21. Schaerf, A. 1999. "Scheduling Sport Tournaments using Constraint Logic Programming", Constraints 4, 43-65.
22. Schreuder, J.A.M. 1980. "Constructing timetables for sport competitions", Mathematical Programming Study, 13, 58-67.
23. Schreuder, J.A.M. 1992. "Combinatorial aspects of construction of competition Dutch Professional Football Leagues", Discrete Applied Mathematics 35, 301-312.
24. Trick, M.A. 2001. "A schedule-then-break approach to sports timetabling", in E. Burke and W. Erben (eds) *Practice and Theory of Automated Timetabling III*, LNCS 2079, Springer.
25. Van Hentenryck, P. 1999. *The OPL Optimization Programming Language*, MIT Press.

Theory and practice of the shift design problem

WOLFGANG SLANY
Institut für Informationssysteme
Technische Universität Wien
Favoritenstr. 9–11, A-1040 Wien
AUSTRIA
wsi@dbai.tuwien.ac.at http://www.dbai.tuwien.ac.at/staff/slany/

Abstract: Generating high-quality schedules for a rotating workforce is a critical task in all situations where a certain staffing level must be guaranteed, such as in industrial plants, hospitals, or airline companies. Shift scheduling comprises several problems, one of primordial importance being the shift design problem which is concerned with finding optimal starting times and lengths of shifts. This problem presents an interesting challenge both from practical as well as theoretical points of view. We describe our results in both areas, stressing in particular how our quest for complexity results lead to improved practical algorithms which are now part of a successful commercial package.

In the shift design problem we are given a collection of shift templates and workforce requirements for a certain cycle time (usually one week). We look for an optimal selection of the shift templates together with an optimal assignment of workers to these shifts such that the overall deviation from the requirements is small, with a number of additional constraints:

- The shift templates are potential shifts under legal, ergonomic, and operating constraints.
- Over- and underhead can be differently weighted.
- The number of selected shifts should be small.
- Workers should come to work 4–5 times per week on average.
- Some other constraints varying from case to case, e.g. there should be little variation in the selection of shifts between weekdays.

Example: Shift templates = all possible shifts (typical temporal resolution: 15 minutes).

Example: Workforce requirements: How many persons should ideally be present at each time of the week.

Characteristics of the solution of Table 3:

- Number of shifts used: 5 (out of the 384)
- Underhead: each day from 10h–11h 2 workers
- Overhead: Wednesday 9h–10h 2 workers
- Average number of times workers have to commute per week: 4 (assuming 21 workers will work for 40h/week)

Another solution, found with a very fast min cost max flow algorithm: 2 worker-hours less, but the solution features a total of 18 shifts to achieve this better fitting of the requirements.

It is possible to model the shift design problem as a network flow problem, namely as the cyclic multi-commodity capacitated fixed-charge min cost max flow problem (following ideas from [1]). Basic idea:

- A matrix A with consecutive ones property (in the columns, corresponding to shift templates).
- A vector b of positive integers (corresponds to workforce requirements).

$$\text{E.g.,} \quad A = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \end{bmatrix},$$

$b = (2, 3, 5, 4, 2, 1)$

Abbreviation	Shift Type	Possible Start Times	Possible Durations
M	Morning	06:00 – 08:00	7h – 9h
D	Day	09:00 – 11:00	7h – 9h
A	Afternoon	13:00 – 15:00	7h – 9h
N	Night	22:00 – 24:00	7h – 9h

Table 1: Typical set of shift templates. With a resolution of 15 minutes, this example corresponds to 384 possible shift templates.

Start time	End time	Mon	Tue	Wen	Thu	Fri	Sat	Sun
06:00	08:00	2	2	2	6	2	0	0
08:00	09:00	5	5	5	9	5	3	3
09:00	10:00	7	7	7	11	7	5	5
10:00	11:00	9	9	9	15	9	7	7
11:00	14:00	7	7	7	13	7	5	5
14:00	16:00	10	9	7	9	10	5	5
16:00	17:00	7	6	4	6	7	2	2
17:00	22:00	5	4	2	2	5	0	0
22:00	06:00	5	5	5	5	5	5	5

Table 2: Sample workforce requirement table for one week.

- We look for a (positive) vector x such that
  1. $|Ax-b|_1$ is minimum (= minimum over-/underhead).
  2. Among all x minimizing $|Ax-b|_1$, the one that has a minimum number of non zero entries (= minimum number of shifts).
- We can also accommodate the other constraints and objectives, as we will hint at later.

Let T denote the following matrix:

$$T = \begin{bmatrix} 1 & -1 & 0 & 0 & 0 & & 0 \\ 0 & 1 & -1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 1 & -1 & 0 & & 0 \\ & \vdots & & \ddots & & & \vdots \\ 0 & 0 & 0 & & 1 & -1 & 0 \\ 0 & 0 & 0 & \cdots & 0 & 1 & -1 \\ 0 & 0 & 0 & & 0 & 0 & 1 \end{bmatrix}$$

As T is regular the two sets of feasible x vectors for $Ax = b$ and for $TAx = Tb$ are equal. However, TA can (almost) be interpreted as a flow matrix, Tb specifying the finite capacities on source (s) and sink (t) edges, all other edges having infinite capacity.

$$TA = \begin{array}{c} \begin{matrix} -1 & -1 & -1 & & & & & \end{matrix} \\ \begin{bmatrix} 0 & 0 & 0 & -1 & -1 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & -1 & -1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \end{bmatrix} \end{array},$$

$Tb = -2 \quad (-1, -2, 1, 2, 1, 1)$

$x^T = (0\ 1\ 1\ 1\ 0\ 0\ 1\ 1)$

Thus, shifts 2, 3, 4, 7, and 8 each are assigned one worker to perfectly satisfy this shift design problem.

- To minimize possible over- and underhead, we introduce slack variables (edges of infinite capacity and cost according to the relative weights of over- versus underhead).

  This part (min Σ cost $\times$ flow) can be solved by any min cost max flow algorithm (in polynomial time).
- To minimize the average number of times workers have to come in, assign uniform costs

Shift	Start time	End time	Mon	Tue	Wen	Thu	Fri	Sat	Sun
M1	06:00	14:00	2	2	2	6	2		
M2	08:00	16:00	3	3	3	3	3	3	3
D1	09:00	17:00	2	2	2	4	2	2	2
A1	14:00	22:00	5	4	2	2	5		
N1	22:00	06:00	5	5	5	5	5	5	5

Table 3: A typical solution for the problem from Table 2

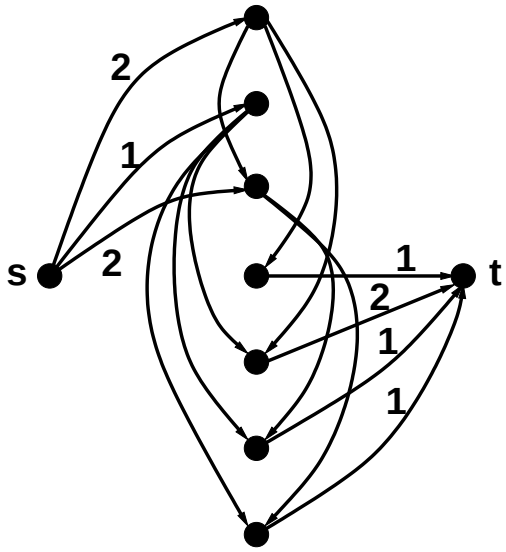

Figure 1: Flow graph corresponding to the sample shift problem.

to all edges corresponding to shifts. Again solved through min cost max flow algorithm.

- To minimize number of selected shifts assign fixed charges to shift-edges. Closely related to the NP-complete Minimum Edge Cost Flow problem (MECF = [ND32] in Garey & Johnson). Remember also the second solution of the sample problem used many more shifts than the first one.

1 Theoretical results

Theorem 1 ([2]): There is a constant $c < 1$ such that approximating the shift design problem with polynomial bounds on numbers appearing in the requirements within $c \ln n$ is NP-hard.

The proof is based on a reduction from Set-Cover.

Theorem 2 ([2]): The shift design problem admits a $O(\sqrt{n \log M})$ ratio approximation algorithm (where M is the largest number in the requirements).

This means that we find a vector minimizing the overhead with $O(\sqrt{n \cdot \log M}) \cdot opt$ non-zero entries where *opt* is the minimum possible number of non-zero entries in a maximum flow.

The proofs of both results can be found in [2].

Similar results exist for the Minimum Edge Cost Flow problem and variants thereof (see [2]).

This problem is one of the more fundamental flow variants with many applications. A sample of these applications include optimization of synchronous networks, source-location, transportation, scheduling (for example, trucks or manpower), routing, and designing networks (for example, communication networks with fixed cost per link used, e.g., leased communication lines), see [2] for references.

2 Practical algorithms

The Operating Hours Assistant is sold by Ximes corp[1]. In the program, a min cost max flow algorithm is used to get reference values for all constraints besides the minimization of the number of shifts. To additionally minimize the number of shifts, a Tabu search heuristic is employed that is delivering good results for most problems appearing in real life.

A screenshot of a German language version is given in Figure 2.

Acknowledgments

This research was partially supported by the Austrian Science Fund Project N Z29-INF and by FFF project No. **801160/5979**.

[1] http://www.ximes.com/

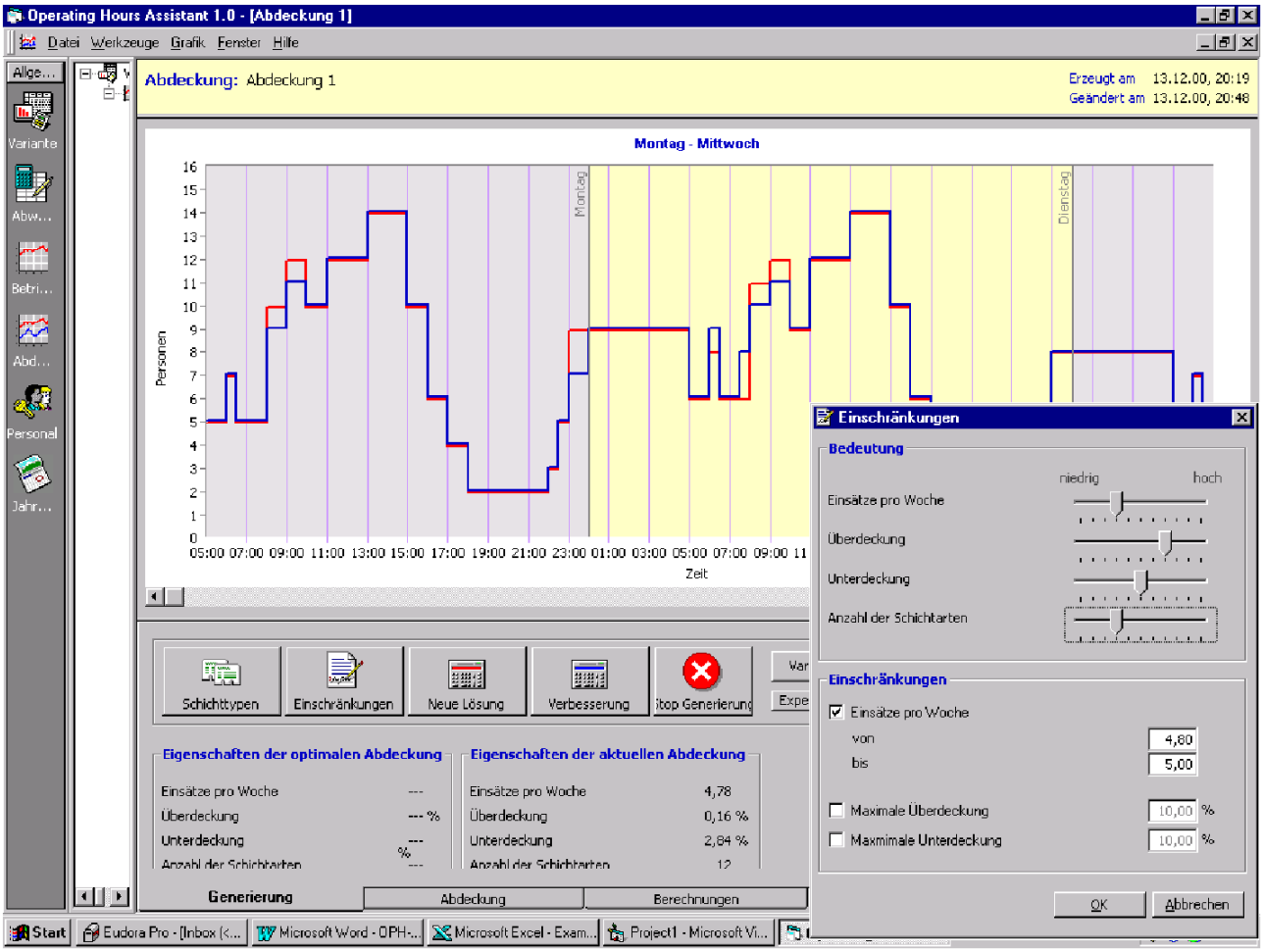

Figure 2: Operating Hours Assistant screenshot.

References

[1] J.J. Bartholdi, J.B. Orlin, and H.D. Ratliff. Cyclic scheduling via integer programs with circular ones. *Operations Research*, 28:110–118, 1980.

[2] Guy Kortsarz and Wolfgang Slany. The minimum shift scheduling problem. Unpublished manuscript, 2000.

University Course Timetabling

University Course Timetabling with Soft Constraints

Hana Rudová[1] and Keith Murray[2]

[1] Faculty of Informatics, Masaryk University
Botanická 68a, Brno 602 00, Czech Republic
hanka@fi.muni.cz
[2] Space Management and Academic Scheduling, Purdue University
1128 ENAD, West Lafayette, IN 47907, USA
kmurray@purdue.edu

Abstract. An extension of constraint logic programming that allows for weighted partial satisfaction of soft constraints is described and applied to the development of an automated timetabling system for Purdue University. The soft constraint solver implemented in the proposed solution approach allows constraint propagation for hard constraints together with preference propagation for soft constraints. Repair search methods are proposed to improve upon initially generated (partial) assignments of the problem variables. The model and search methods applied to the solution of the large lecture room component are presented and discussed along with the computational results.

1 Introduction

This paper describes the design approach and solution techniques being developed for an automated timetabling system at Purdue University. The initial problem considered here is the design of an intelligent system to assist with construction of the large lecture component of the university's master class schedule. The design anticipates expanding the scope of the problem to accommodate a demand-driven approach to timetabling all classes at the University. In *demand-driven timetabling*, student course selections are utilized to construct a timetable that attempts to maximize the number of satisfied course requests. In the initial problem we consider the course demands of almost 29,000 students enrolled in approximately 750 classes, each taught several times a week.

A solution to the timetabling problem is being developed using *constraint logic programming* [21, 13]. CLP is a respected technology for solving hard problems which include many complicated (non-linear) constraints [11]. Its main advantages over other frameworks are the declarative nature of problem descriptions via logical constraints, and a constraint propagation technique for reducing the search space.

Timetabling problems [7, 20] are often over-constrained, which is the case with our problem since it is not possible to satisfy all requests of students for enrollment to specific courses. Preferential requirements for time and room assignment may also lead to the problem being over-constrained. *Soft constraints* [8, 4]

can be applied to define all requirements declaratively rather than encapsulating many of them into the control part of the problem solution. In our problem solution, we have applied a weighted CSP [8] approach which considers weights/costs for each constraint and minimizes the weighted sum of unsatisfied constraints.

Soft constraints are often applied to solve timetabling problems with the help of constraint satisfaction. They are mostly applied via the standard constraint satisfaction method, which offers no special support for more effective resolution of the soft constraints. Golz et al. [10] applies the typical solution — the given unary soft constraints, with priorities, are integrated into the solution search through value and variable ordering heuristics. An optimization constraint was applied for solving medium-sized problems [12]. Abdennadher et al. [1] describe the solution of a department-sized problem, including soft constraint solver, for a cost-based approach implemented with Constraint Handling Rules [9].

Our work includes development of a new solver for soft constraints. The solver was implemented as an extension of the CLP(FD) library [5] of SICStus Prolog. This approach is of particular importance for the construction of demand-driven schedules where complete satisfaction of all constraints is not feasible. Soft constraints have also been applied to accommodate the preferences of instructors with respect to time and room assignments for their classes. We will also describe a new repair search technique which allows us to find a solution even when the problem is over-constrained.

The following sections of this paper will give a more complete description of our timetabling problem. Section 3 explains the soft constraint solver that was implemented, together with the search procedures used on this problem. This is followed by a description of how the problem has been modeled, including the representation of soft and hard constraints. In addition, the methods applied to the search for a feasible solution are discussed. Computational results are discussed in Sec. 5. Comparison with other approaches to solving demand-driven timetabling problems is presented in Sec. 6. The final section reviews the results of our work and looks to future extensions of the problem solution and soft constraint solver improvements.

2 Problem Description

At Purdue University, the timetabling process currently consists of constructing a master class schedule prior to student registration. The timetable for large lecture classes is constructed by a central scheduling office in order to balance the requirements of many departments offering large classes that serve students from across the university. Smaller classes, usually focused on students in a single discipline, are timetabled by "schedule deputies" in the individual departments. This process has been tailored to the political realities of a decentralized university, where faculty can be quite put off by the idea of having a central office tell them when to teach, or even by providing such an office with much information about when they are available to teach.

A natural decomposition of the university timetabling problem has therefore resulted, consisting of a central large lecture timetabling problem and 74 disciplinary problems. The large lecture problem consists of approximately 750 classes having a high density of interaction that must fit into 41 lecture rooms with capacities up to 474 students. In this problem, the course demands of almost 29,000 students out of a total enrollment of 38,000 must be considered. The departmental problems range from only a few classes up to almost 700, with an average size of slightly more than 100 organized classes. The largest departmental problems are simplified by having many sections of the same course that are offered at multiple times.

The timetable maps classes (students, instructors) to meeting locations and times. A major objective in developing an automated system is to minimize the number of potential student course conflicts which occur during this process. This requirement substantially influences the automated timetable generation process since there are many specific course requirements in most programs of study offered by the University.

To minimize the potential for time conflicts, Purdue has historically subscribed to a set of standard meeting patterns. With few exceptions, 1 hour x 3 day per week classes meet on Monday, Wednesday, and Friday at the half hour. 1.5 hour x 2 day per week classes meet on Tuesday and Thursday during set time blocks. 2 or 3 hours x 1 day per week classes must also fit within specific blocks, etc. Generally, all meetings of a class should be taught in the same location. Conforming to this set of standard meeting patterns will be seen to have a strong influence on both the problem definition and the solution process, since the meeting patterns defined for each class introduce hard restrictions on the acceptability of any generated solution.

The other major constraints on the problem solution are instructor availability and a limited number of rooms available with sufficient capacity, specific equipment, and suitable location. Some of these constraints must be satisfied; others can be introduced within an optimization process in order to avoid an over-constrained problem.

This paper describes the construction of the timetable for the large lecture problem. The nature of this problem is the same as the timetabling problem at the university level. There is a demand for specific class combinations to meet individual student's needs. Meeting patterns direct possible time and location placement. Classroom allocation must respect instructional requirements and the preferences of faculty. All instructors may have specific time requirements and preferences for each class.

Another aspect of the timetabling problem that must be considered here is the need to perform an initial student sectioning. Most of the classes in the large lecture problem (about 75 %) correspond to single-section courses. Here we have exact information about all students who wish to attend a specific class. The remaining courses are divided into multiple sections. In this case, it is necessary to divide the students enrolled to each course into sections that will constitute the classes. Without this initial sectioning it is not easy to measure the desirability

or undesirability of having classes overlap in the timetable. Our current approach sections students in lexicographic order before joint enrollments between classes are computed. This gives us the worst case possibility. The university currently processes a precise student schedule after the master class schedule is created, however, which should introduce some improvements. Possible directions for improving this solution will be discussed in the final section.

3 Solver for Soft Constraints

Constraint propagation algorithms for the soft constraints are implemented as a part of the preference constraint solver [17]. This constraint solver is built on top of the CLP(FD) solver of SICStus Prolog [5] and implemented with the help of attributed variables. An advantage of this implementation is the ability to include both hard constraints from the CLP(FD) library and soft constraints from a new *preference solver*.

3.1 Preference Variables and Preference Propagation

The preference solver handles preferences for each value in the domain of the variable which will be called the *preference variable*. Each preference corresponds to a natural number indicating the degree to which any soft constraint dependent on the domain value is violated. An increase of the preference during the computation with soft constraints is called a *preference propagation*. Let us note the difference with constraint propagation, which removes values from the domain of the domain variable during the computation of hard constraints. Removal of domain values may also occur with preference variables. This corresponds to violation of a hard constraint. We may also set the degree of acceptable violation for any preference variable. If the preference associated with a value in the domain of the preference variable should exceed this limit, it is removed from the domain. This possibility is of particular interest for time (or classroom) variables since all classes should be relatively equal in importance.

Zero preference means complete satisfaction of the constraint for the corresponding value in the domain of the variable. Any higher preference expresses a degree of violation that would result from the assignment of this value to variable. All values which are not present in the domain of the preference variable have the infinite preference `sup`. Preferences for each value in the domain of the variable may be initialized with a natural number. This allows us to handle initial preferences of values in the domain of the variable.

For each preference variable, the preference solver maintains an additional domain variable (*cost variable*) having the current best preference of the preference variable as its lower bound. The initial upper bound is set to infinity. Any preference propagation results in an increase of the current best preference, with the lower bound of the cost variable being increased accordingly. The sum of all cost variables gives us the total cost of the solution (*cost function*).

Example 1. The unary soft constraint `pref(PA, [7-5, 8-0, 10-0], CostPA)` creates the preference variable `PA` with initial domain containing values `7`, `8`, and `10` and preferences `5`, `0`, and `0`, resp. It means that the value `7` is discouraged wrt. other values. Preferences for remaining values are assumed as infinite, indicating complete unsatisfaction. The domain variable `CostPA` is created with the domain `0..sup`.

3.2 Soft Disjunctive Constraints

Let us describe two basic binary soft constraints we have implemented.

```
soft_different( PA, PB, Cost )
soft_disjunctive( PStart1, Duration1, PStart2, Duration2, Cost )
```

The `soft_different` constraint expresses that the two preference variables `PA`, `PB` should have different values. The constant `Cost` gives us the cost for violation of this constraint. The `soft_disjunctive` constraint asks for the non-overlapping of the two tasks specified by the preference variables `PStart1`, `PStart2` and the constant durations `Duration1`, `Duration2`. Again the `Cost` is the weight of this constraint.

Algorithms for both constraints are based on the partial forward checking algorithm and inconsistency counts [8]. Let us take a look at the `soft_different` constraint. Once the first preference variable is instantiated to some value `X`, the inconsistency count for the second variable and the value `X` should be increased by `Cost`, i.e., the preference propagation is processed for this variable and value. The `soft_disjunctive` would process preference propagation for all values in the interval `X..(X+Duration-1)`.

Inconsistency counts are maintained for all of the unassigned variables and the values remaining in their domains. These can be applied during labeling and optimization, e.g., values with the smallest inconsistency count can be selected first (an optimistic approach). The change in inconsistency counts for each preference variable is also reflected by the corresponding cost variable, e.g., the smallest inconsistency count for the preference variable corresponds to the lower bound of its cost variable.

3.3 Search

The aim of our problem solution is to be able to search for the complete assignment of the preference variables giving the best possible satisfaction of all soft constraints. Since the evaluation of the assignment is given by the sum of the corresponding cost variables, we may apply a standard branch & bound algorithm.

Unfortunately it may not be easy to find any complete assignment. Mistakes in the assignment of some variable(s) may lead to a time consuming exploration of the search space with no complete solution. A complete assignment may not even exist due to conflicts among the hard constraints. Our approach is to find

some initial partial assignment of the variables and subsequently repair it such that all, or at least most, of the variables have been assigned a value.

Let us describe the *initial search* first. For each preference variable, we maintain a count of how many times a value has been assigned to the variable. A limit is set on the number of attempts that are made to assign a value. Currently this limit corresponds to the size of the domain of each variable. If the limit is exceeded, the preference variable is left unassigned and we continue in the search. Whereas any assignment of a value to this variable will lead to a time consuming trashing[1], allowing several values to remain in its domain will not initiate a constraint propagation that would detect inconsistency with the already assigned variables. As a result of this search, we obtain a partial assignment of variables and the sets of values which were tried unsuccessfully for the remaining variables.

Any *repair search* can use the result of the former (initial or repair) search. First, we apply heuristics with the successfully assigned variables to give us a value for each variable to be tried *first*. Second, we apply heuristics with the remaining variables to give us values to be tried *last*.

This change in the value ordering can be included in the problem with the help of preferences. We can encourage certain values by increasing the initial preferences or discourage them by decreasing their initial preferences. This type of redefinition allows us to run a repair search using the same procedure as the initial search.

After construction of a partial assignment, the user can decide on a desirable continuation of the search. The following possibilities are of particular interest:

1. processing the repair search directed by the proposed value ordering heuristics;
2. changing variable ordering such that unassigned variables are tried first, then processing the repair search directed by the proposed value ordering heuristics;
3. defining other values to be tried first or last based on user input, and process the repair search directed by the updated value ordering heuristics;
4. relaxing some hard constraints based on user input, and processing the initial search or the repair search directed by the updated value ordering heuristics.

The first two possibilities are aimed at automated generation of a better assignment. The second approach seeks to find an assignment of variables which may be more difficult to locate in the search space using the original variable ordering. This step may lead to significant changes in the generated solution. The third possibility allows the user to direct the search into parts of the search space where a complete assignment of variables might more easily be found. The last step can be useful if the user discovers a conflict among some hard constraints. The repair searches can reuse most of the results from the former search and permute only some parts of the last partial assignment. Let us note that there is often an advantage to a user directed search in timetabling problems, since

[1] A large number of the variable assignments having to be undone repeatedly.

the user may be able to detect inconsistencies or propose a suitable assignment based on the partially generated timetable.

The user can also change the problem definition (add/delete/change classes or add/delete constraints) and apply the repair search while reusing results of the former solution for classes with no changes.

4 Problem Solving

We would like to describe a model for the timetabling problem which consists of variables for the time and room assignments of each class and of both hard and soft constraints, applying an approach described in the previous section. We also explore the control portion of the solution, which consists of the application of the proposed initial and repair searches.

4.1 Time and Classroom Variables

The domain of the time variables is represented by the natural numbers `0..104`, corresponding to `5` days of `21` half-hours. The domain of the classroom variables is represented by the natural numbers `1..Number_Of_Classrooms`.

Each class consists of between one and five meetings per week (typically two or three). All meetings have the same duration and are typically taught at the same time of day. Valid combinations of the number of meetings and the duration are called *meeting patterns*. Each meeting pattern (e.g., 1 hour x 3 meetings) has a defined set of days on which the meetings may be scheduled (e.g., Monday, Wednesday, Friday for 1 hour x 3 meetings). Interestingly, the start time of the first meeting of a class differs from the start times of the following meetings by a constant factor for most combinations (see Table 1). Excepting the MF (Monday and Friday) combination for 2 meetings per week

Table 1. Maximal sets of the possible combinations of days for class with given number of meetings per week (e.g., TTh means that course can have its meetings on Tuesday and Thursday).

Number of meetings	Possible combination of days
1	M or T or W or Th or F
2	MW or TTh or WF or MF
3	MWF
4	TWThF or MWThF or MTThF or MTWF or MTWTh
5	MTWThF

and the 4 meeting patterns (includes less than 1% of classes), one time variable is sufficient to contain the complete information about the start time of classes. This is a preference variable indicating the start time of the first meeting (`T1`).

It will be referred to as the *time preference variable*. The starting times of all remaining meetings (`T2,..., Tn`) are domain variables only, and may be related to the time preference variable by the simple constraint

$$\texttt{Ti} \#= \texttt{T1} + \texttt{Constant} * (\texttt{i} - 1) \ . \tag{1}$$

Preferences associated with each value in the domain of the time preference variable allow us to express the degree to which any time assignment for a class is preferred or discouraged. The remaining domain variables may be referenced in the hard constraints (e.g., `serialized`), but they do not require the more expensive processing by the preference solver.

Since all class meetings should be taught in the same room, we suffice with only one common classroom variable for all meetings (called the *classroom preference variable*). As a preference variable, it associates a preference with each classroom expressing how desirable or undesirable it is for a given class.

4.2 Hard Constraints

Let us summarize the requirements which are implemented in the system using hard constraints:

1. meeting pattern specification;
2. prohibited or required times for classes;
3. class requires room with sufficient seating capacity;
4. class requires or prohibits some building(s) or room(s);
5. class requires or prohibits classroom of a specified generic type (computer, computer projection, audio recording, document camera, ...);
6. classes taught by the same instructor do not overlap;
7. sections of the same course do not overlap;
8. additional constraints over selected sets of classes: classes must be taught at the same times, on the same days, in the same classrooms, ...

Meeting pattern constraints relate the domain variables for all class meetings as was described in Eqn. 1. In addition, the domain of the time preference variable is reduced such that all invalid values are removed.

Example 2. A 1.5 hour x 2 meetings class is represented by the two variables `T1`, `T2`. The first of these is the time preference variable with the initial domain (`0..104`) reduced to the values `21,24,...,39` because the TTh combination is valid only. The second domain variable is related with `T1` by the constraint `T2 #= T1 + (21*2)*1`. The constant separating start times here is 21 periods x 2 days.

A 2 hours x 2 meetings class has MW, TTh, and WF as valid meeting day combinations. It is represented by the two variables `T1`, `T2` related by the same constraint as before. The reduced domain of `T1` corresponds to the values `0,4,8,12,16, 21,25,29,33,37, 42,46,50,54,58`.

Requirements 2–5 are implemented by domain reduction in the corresponding domains of the time and classroom preference variables. Requirements 6 and 7 are included with help of the constraint `serialized` which constrains tasks, each with a start time and duration, so that no tasks ever overlap. Built-in constraints of CLP(FD) library of SICStus Prolog are used to implement various requirements over selected sets of classes as mentioned in the item 8.

Additional hard constraints must be posted to assure that each class is assigned to just one suitable classroom. This requirement could be implemented via the `disjoint2` constraint, which ensures non-overlapping of a set of rectangles. In our case, the rectangle is defined by the start time variable (`Time`) and the duration (`Duration`) of each meeting, and by the classroom variable (`Classroom`) for the corresponding class:

```
disjoint2( [ rectangle(Time, Duration, Classroom, 1) | _ ] ) .
```

The number `1` represents the requirement of *one* classroom for each meeting.

A different type of propagation among the time variables is achieved via the `cumulative` constraint. It ensures that a resource can run several tasks in parallel, provided that the discrete resource capacity is not exceeded. If there are `N` tasks, each starting at a certain time (`StartI`), having a certain duration (`DurationI`) and consuming a certain amount of resource (`ResourceI`), then the sum of resource usage of all the tasks must not exceed resource limit (`ResourceLimit`) at any time:

```
cumulative([Start1,...,StartN], [Duration1,...,DurationN],
           [Resource1,...,ResourceN], ResourceLimit) .
```

The `cumulative` constraint helps to assign a classroom of sufficient size to each meeting while allowing smaller classes to be assigned to larger classrooms.

Example 3. Let us imagine a small example with 2 rooms for 40 students, 3 rooms for 20 students, and 1 room for 10 students. The set of `cumulative` constraints follows

```
cumulative(Time_meetings_with_size_40, Dur_40, ListOf1, 2),
cumulative(Time_meetings_with_size_20_40, Dur_20_40, ListOf1, 5),
cumulative(Time_all_meetings, Dur_all, ListOf1, 6).
```

The first constraint ensures that the largest classes are accommodated into the largest rooms, the second constraint allows medium-sized classes to be placed into rooms for 20 students, and also into rooms for 40 students if they are not already asked for by the first constraint. The third constraint allows movement of small classes between all rooms, subject to the condition that they are not occupied by any larger classes at the same time.

More precisely, we can post one `cumulative(Starts, Durations, ListOf1, Limit)` constraint for each possible size of classroom denoted by `Size`. The constant `ListOf1` denotes a list of 1 representing a unit resource requirement (one classroom) by each course. `Durations` represents the durations of classes with

the start time `Starts`. Variables `Starts` and `Limit` should satisfy the following properties

$$\texttt{Starts} = \{\texttt{Start} \mid \texttt{meeting(Start,Duration,Capacity)} \wedge \texttt{Capacity} \geq \texttt{Size}\}$$
$$\texttt{Limit} = \text{card}\{\texttt{Id} \mid \texttt{classroom(Id,Capacity)} \wedge \texttt{Capacity} \geq \texttt{Size}\}$$

Actually, it is sufficient to post this constraint only for some specific sizes of classrooms. Classrooms of similar size are grouped together to achieve better efficiency.

Another possibility for taking `cumulative` constraints into account consists of splitting classrooms into *groups* by size. Each class would be included in the group of corresponding size only. Such division can be useful if we do not want to put smaller classes into larger classrooms of other group at any time (e.g., smaller classes must be in the classrooms with a capacity smaller than 400 students).

4.3 Soft Constraints

Three types of soft constraints are currently handled by the system which will be discussed in this section:

1. unary constraints on time variables — faculty time preferences;
2. unary constraints on classroom variables — faculty preferences on the classroom selection for classes;
3. binary constraints for each joint enrollment between two classes.

Instructors may specify preferences for the days, hours, or parts of days they wish to encourage or discourage. This specification is transformed into a list of integer preferences corresponding to the possible start time of each class. We have seen that the initial selection of start times for each class is determined by its meeting pattern. The domain size of this set of start times can differ greatly among meeting patterns (it ranges from 5 to 50 possible values). This causes the relative effect of any given preference to vary greatly among the meeting patterns. To compensate for this effect, the number of preference points associated with instructor time preferences differ based on the meeting pattern.

Each class is associated with a time preference variable with preferences initialized either as specified by the instructor or to a set of default preferences. These default preferences are very important — their exclusion would lead to the construction of timetables which discriminate against classes for which no preferences have been provided. Many such classes would be placed in undesirable times, which no human timetabler would want to do.

Instructors may also specify positive or negative preferences towards the room selection for each class. It is possible to prefer or discourage particular classrooms, buildings, or properties of the room (e.g., "I prefer classrooms with a computer." or "I discourage classrooms without windows."). Each value in the domain of the classroom preference variable has either the specified preference or the neutral preference specification.

Any two classes potentially have a number of students who are enrolled to both at the same time. We seek to control their degree of overlap in the timetable

by a generalization of the soft disjunctive constraint (see `soft_disjunctive` in Sec. 3.2). Such binary soft constraints include two time preference variables for corresponding classes, with the cost given by the number of students enrolled in both. Since each class may have several meetings, such a generalized disjunction needs to propagate preferences to the all values of the uninstantiated preference time variable which could be affected by the overlap of any of the meetings.

Preferences associated with the time preference variables are influenced by both the first and third soft constraints. Constraints of the first type initialize preferences. Constraints of the third type propagate (increase) them during the computation of the same cost function. The sum of the cost variables (see Sec. 3.1) for the time preference variables gives this cost function, i.e., the solution cost wrt. time assignment. As a consequence we need to balance the number of student joint enrollments from the third constraint with the number of preference points assigned by the first constraint, (e.g., the summarized preference points of the instructor for one class corresponds to overlapping for 20 students). The relationship between preference points and joint enrollments is specified as part of the input data.

4.4 Labeling

Separated Time & Classroom Variable Labeling. In our timetabling problem, we have time and classroom preference variables and two types of cost functions based on the sums of the corresponding cost variables. These cost functions are independent of each other and introduce two different criteria in our problem. Since minimizing student conflicts and accommodating the time preferences of classes were judged to be a more critical aspect of the problem than meeting preferences for classrooms, we have explored the following approach: labeling of time preference variables is processed first, followed by the labeling of room preference variables. Potential improvements of this search are mentioned in Sec. 7.

Initial & Repair Searches. The overall process of searching for a solution consists of two main parts — the initial search and the repair search. Results of a prior search are used only by the time preference variables. Information about an unassigned classroom variable reflects back upon the corresponding time preference variable as we describe in the next paragraph.

In the initial search, preferences associated with the time preference variables are set as they are defined by the problem. Values of the unassigned time preference variables that were unsuccessfully tried during the prior search are discouraged during the next search. There are other time preference variables for which a particular value is discouraged. These are the variables with a corresponding unassigned classroom preference variable in the former search. Since we have processed the assignment of the time variables first, such a non-assignment is the result of no classroom being available for the corresponding time. Successfully assigned values for all remaining time preference variables are encouraged

in an automated repair search as it was proposed in item 1 in Sec. 1. In addition, variable ordering can be changed for unassigned time variables (item 2). The user can also introduce his own preferences (item 3) or relax the hard constraints (item 4).

Value & Variable Ordering. Different heuristics were used for time and classroom variable labeling. We have employed the first-fail heuristics to determine the ordering of classes for time assignment. It selects the most highly constrained time preference variables with respect to both hard and soft constraints. First, we select among the variables having the smallest domain. Ties are broken based on the greatest number of soft constraints related to this variable. If not selected early, the domain of such a variable may become too small to select a sufficiently preferred value, or it may even become empty and cause a backtracking. Early propagation of soft constraints is also encouraged, so as not to discover mistakes too late. A specific class time assignment was selected among the most preferred values, i.e., we have chosen an optimistic approach for the value selection. This approach lead us to a good first solution from the point of view of cost function over time variables.

The first-fail approach was also used to choose a class to be placed into a classroom. The best results of the cost function over classroom variables were achieved by the following value ordering heuristics. First, the most preferred classroom was selected. Ties were broken by the selection of the smallest available classroom so as not to waste available resources. A branch and bound search was applied to improve the cost function over room variables.

During the repair searches, we have updated our variable ordering heuristics as it is pointed in Section 3.3 about the proposed search procedure, i.e., each variable unassigned in the last search was assigned prior to successfully assigned variables in the consequent search (see item 2).

5 Computational Results

Our data set from fall semester 2001 includes 747 classes to be placed into 41 classrooms. The classes included represent 81,328 course requirements for 28,994 students. The results presented here were computed by SICStus Prolog 3.9.0 on a PC with AMD Athlon/850 MHz processor and with 256 MB of memory.

Table 2 shows the computational results for the initial search and the subsequent automated repair searches. *Unassigned classes* refers to the number of classes with either time or classroom variables that were not assigned during labeling. *Satisfied time* gives the percentage of how many encouraged times for classes were selected. *Unsatisfied time* refers to the percentage of the discouraged times for classes which were selected. *Student conflicts* estimates the percentage of unsatisfied requirements for courses by students. *Preferred classrooms* measures the percentage of classes for which encouraged classrooms were selected. The current data set does not include any preferential requirements which discouraged specific classrooms.

Table 2. Results of the initial search and the automated repair searches

Run	initial	repair I.	repair II.	repair III.	repair IV.
Unassigned classes	19	15	10	5	3
Satisfied time (%)	83.6	81.1	79.4	79.9	79.7
Unsatisfied time (%)	4.1	4.3	4.0	4.0	4.0
Student conflicts (%)	1.6	1.7	1.9	1.9	1.9
Preferred classrooms (%)	77.6	49.7	49.0	48.3	49.0

The initial search procedure took about 2-3 minutes for the time labeling, one step of the branch and bound search for classroom variables took 1-2 seconds. The time labeling takes a longer time due to the preference propagation and more complex constraints (e.g., `cumulative`) posted on time variables. The length of the overall run depends on the number of repair steps, computation of the best solution took about 15 minutes (one initial step followed by four repair searches).

Originally we started to solve the problem using a built-in backtracking search algorithm with a variety of variable and value ordering heuristics. This attempt did not lead to any solution after 10 hours of computations however. Too many failed computations were repeated exploring parts of search tree with no solution.

We can see that most of preferential requirements of instructors were satisfied during assignment of time variables. The unsatisfied time percentage mostly illustrates that a number of classes must be taught at unpopular times due to the limited number of rooms available. Let us also note that these results include default preferences for classes with no preferred times. The automated repair search (see item 1 in Sec. 1) during time assignment was able to improve on the initial solution substantially. The solution continued to improve through four repair searches. Additional steps did not improve the quality of the solution further.

One of the more important lessons learned here is that the `disjoint2` and `cumulative` constraints must be used together in a redundant manner to find an acceptable solution (for description of both constraints see Sec. 4.2). The `cumulative` constraints were able to introduce additional constraint propagation for the time variables by informing them of the available room resources. Neither `cumulative` nor `disjoint2` constraints alone were able to find an acceptable solution. Results using only the `cumulative` constraints left about 20 unassigned classes. Using only the `disjoint2` constraint resulted in 50 unassigned classes.

6 Related Work

Purdue University. Currently the timetable for Purdue University is constructed by a manual process. An earlier approach examined for automating construction

of the Purdue University timetable modeled the room-time assignment problem as a multiple choice quadratic vertex packing and utilized a tabu search algorithm [14]. This approach was further developed into a prototype system used to create a schedule for large lecture classes in spring 1994, but was never adopted by university schedulers due to inadequacies in the way it handled instructor time preferences and student conflicts.

Constraint Programming. There have not been many attempts [18, 3] to apply constraint programming to the solution of demand-driven timetabling problems where it is not possible to satisfy all requests of students. Such over-constrained problems require enhancements to the classical constraint satisfaction approach. Once these are developed, we can apply all of the advantages of constraint programming, including a declarative description of the problem together with strong propagation techniques.

We have previously constructed a demand-driven schedule for the Faculty of Informatics at Masaryk University [18] having 270 classes and about 1250 students. Conflicts of students between classes were controlled using a similar cost function as in our current approach. Constraint logic programming allowed implementation of a variety of constraints available in ILOG Scheduler [15]. Soft constraints were implemented with the help of special variable and value ordering heuristics defined by the preferences of particular variables in the constraints. The timetable constructed was able to satisfy 94 % of the demands of students and more than 90 % of the preferential requirements of teachers. Unfortunately, it was not easy to extend this implementation to larger problems due to the bound consistency algorithms in ILOG Scheduler. Since these algorithms only propagate changes over the bounds of the domain variables, both constraint and preference propagations were much weaker than there are now.

A solution of the section assignment sub-problem is included in Banks, van Veek, and Meisels [3] via iterative addition of constraints into a CSP representation. Inconsistent constraints are not included in the final CSP representation, which allows solution of an over-constrained problem. This implementation, including its own constraint satisfaction solver, was verified using random timetabling problems based on problems from high schools in Edmonton, Alberta, Canada. The largest data set included requirements of 2,000 students and 200 courses. They were able to satisfy 98 % of student demand on more than half of the experiments. The solution presented is influenced by a special set of times that must be assigned to each course. It conforms well to high schools, but is rather different from the situation in university course timetabling. University class meeting patterns are not as strict, which results in a problem with a variety of additional requirements and preferences.

Other Approaches. The comprehensive university timetabling system described by Carter [6] is characterized by problem decomposition with respect to both type and size of final sub-problems. They were able to solve the problem for 20,000 students and 3,000 course sections. The system was used for 15 years at the University of Waterloo.

Aubin&Ferland [2] propose an iterative heuristic method to solve the problem which alternately assigns times and students to course sections until no further improvements to the solution can be found. The system was tested on data from a High School in Montreal including demands of 3,300 students and 1,000 courses.

Robert&Hertz [16] decompose the problem into a series of easier sub-problems corresponding to time, section, and classroom assignments and solve them via tabu search methods. The method presented is able to generate an initial solution which can be incrementally improved after problem redefinition (negotiation on initial constraints with teachers and students). The initial solution for about 500 students and 340 course sections satisfied approximately 10 % of the student requirements and 80 % of the preferential requirements of teachers.

The local search heuristic procedure of Sampson, Freeland, and Elliot [19] solves a problem having a smaller solution space with 89 course sections and 230 students. They were able to meet 94 % of the student scheduling requirements at the Graduate School of Business Administration at the University of Virginia.

7 Conclusion

We have proposed and implemented a solution to a large scale university timetabling problem. We have constructed a demand-driven schedule which is able to reflect diverse requirements of students during course enrollment. Our solution is able to satisfy the course requests of 98 % of students. About 80 % of preferential requirements on time variables were also met with only a small number of classes taught at discouraged times (about 4 %). The automated search was able to find suitable times and classrooms for 744 classes. The remaining 3 classes should be possible to assign with user input.

Our proposal included a new solver for soft constraints which is of particular interest for timetabling problems where the costs in the problem are directly related to the present values for time and room assignments of classes. We have proposed a special repair search algorithm which allows us to find a solution to the problem (even when it is over-constrained). We have also discussed a special set of `cumulative` constraints which, together with the `disjoint2` constraint, processes stronger constraint propagation.

Our future research will include an extension of the problem solution together with improvements to the preference solver that has been described. Also, our approach must be validated using data sets from other semesters.

Currently we are working on the inclusion of new variable ordering heuristics which will process an interleaved labeling for both time and classroom variables. We intend to process an earlier assignment of classroom variables for which late labeling proved to be difficult or impossible. Other heuristics will be aimed at improving the solution with the help of a pattern matching mechanism based on the sets of meeting patterns in the problem.

We would like to do an extensive study of the proposed repair search algorithm and its possible application to a general problem solution. We also intend

to explore how it may be extended to the solution of the minimal perturbation problem, which is very important for all timetablers. Once timetables are published they require many changes based on additional input. These changes should be incorporated into the problem solution with minimal impact on any previously generated solution.

Purdue University currently relies on a completely manual process for constructing its timetable. A detailed comparison of results between the approach described in this paper and the manual process for the full large lecture problem is one of the next steps in our work.

A new approach for making initial student section assignments for courses with multiple sections will also be examined in future work. This is required to construct a joint enrollment matrix that is more representative of the probable cost of having two classes overlap. An approach such as the homogeneous sectioning used by Carter [6] is attractive since it tends to result in fewer classes having joint enrollments with others. This simplifies the task of finding non-conflicting assignments and appears to be an accurate representation when a final sectioning process will take place after construction of the timetable.

We feel that the solution methods used for the large lecture problem should be directly applicable to construction of the 74 academic unit timetables. Some solution refinements may be necessary to simplify time assignments for introductory courses with large numbers of sections. Additional system architecture work will also be necessary to allow unit timetablers to use local preference files, and to work cooperatively if there is a high degree of interrelationship between classes offered by the units.

Acknowledgements

This work is partially supported by the Grant Agency of Czech Republic under the contract 201/01/0942 and by Purdue University.

We would like to thank our students who are assisting with the solution of this problem, and Purdue staff who have helped in many ways.

References

[1] Slim Abdennadher and Michael Marte. University course timetabling using constraint handling rules. *Journal of Applied Artificial Intelligence*, 14(4):311–326, 2000.

[2] Jean Aubin and Jacques A. Ferland. A large scale timetabling problem. *Computers & Operations Research*, 16(1):67–77, 1989.

[3] Don Banks, Peter van Beek, and Amnon Meisels. A heuristic incremental modeling approach to course timetabling. In *Proceedings of Artificial Intelligence'98, Canada*, 1998.

[4] Stefano Bistarelli, Ugo Montanari, and Francesca Rossi. Semiring-based constraint solving and optimization. *Journal of ACM*, 44(2):201–236, March 1997.

[5] Mats Carlsson, Greger Ottosson, and Björn Carlson. An open-ended finite domain constraint solver. In *Programming Languages: Implementations, Logics, and Programming*. Springer-Verlag LNCS 1292, 1997.

[6] Michael W. Carter. A comprihensive course timetabling and student scheduling system at the University of Waterloo. In Edmund Burke and Wilhelm Erben, editors, *Practice and Theory of Automated Timetabling III*, pages 64–82. Springer-Verlag LNCS 2079, 2001.

[7] Michael W. Carter and Gilbert Laporte. Resent developments in practical course timetabling. In Edmund Burke and Michael Carter, editors, *Practice and Theory of Automated Timetabling II*, pages 3–19. Springer-Verlag LNCS 1408, 1998.

[8] Eugene C. Freuder and Richard J. Wallace. Partial constraint satisfaction. *Artificial Intelligence*, 58:21–70, 1992.

[9] Thom Frühwirth. Constraint handling rules. In Andreas Podelski, editor, *Constraint Programming: Basics and Trends*, LNCS 910. Springer, 1995. (Châtillon-sur-Seine Spring School, France, May 1994).

[10] Hans-Joachim Goltz, Georg Küchler, and Dirk Matzke. Constraint-based timetabling for universities. In *Proceedings INAP'98, 11th International Conference on Applications of Prolog*, pages 75–80, 1998.

[11] Christelle Guéret, Narendra Jussien, Patrice Boizumault, and Christian Prins. Building university timetables using constraint logic programming. In Edmund Burke and Peter Ross, editors, *Practice and Theory of Automated Timetabling*, pages 130–145. Springer-Verlag LNCS 1153, 1996.

[12] Martin Henz and Jörg Würtz. Using Oz for college timetabling. In Edmund Burke and Peter Ross, editors, *Practice and Theory of Automated Timetabling*, pages 162–177. Springer-Verlag LNCS 1153, 1996.

[13] Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19:503–581, 1994.

[14] Edward Mooney. *Tabu Search Heuristics for Resource Scheduling*. PhD thesis, Purdue University, 1991.

[15] Claude Le Pape. Implementation of resource constraints in ILOG SCHEDULE: a library for the development of constraint-based scheduling systems. *Intelligent Systems Engineering*, 3(2):55–66, 1994.

[16] Vincent Robert and Alain Hertz. How to decompose constrained course scheduling problems into easier assignment type subproblems. In Edmund Burke and Peter Ross, editors, *Practice and Theory of Automated Timetabling*, pages 364–373. Springer-Verlag LNCS 1153, 1996.

[17] Hana Rudová. Soft scheduling. In *Proceedings of the 2001 ERCIM Workshop on Constraints*. Charles University, Faculty of Mathematics and Physics, 2001. See http://arXiv.org/html/cs.PL/0110012.

[18] Hana Rudová and Luděk Matyska. Constraint-based timetabling with student schedules. In Edmund Burke and Wilhelm Erben, editors, *PATAT 2000 — Proceedings of the 3rd international conference on the Practice And Theory of Automated Timetabling*, pages 109–123, 2000.

[19] Scott E. Sampson, James R. Freeland, and Elliot N. Weiss. Class scheduling to maximize participant satisfaction. *Interfaces*, 25(3):30–41, 1995.

[20] Andrea Schaerf. A survey of automated timetabling. Technical Report CS-R9567, CWI, Amsterdam, NL, 1995.

[21] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.

A GA evolving instructions for a timetable builder

Christian Blum[1], Sebastião Correia[2], Marco Dorigo[1], Ben Paechter[3], Olivia Rossi-Doria[3], and Marko Snoek[4]

[1] IRIDIA, Université Libre de Bruxelles
Brussels, Belgium
{cblum,mdorigo}@ulb.ac.be
[2] Chronopost International
Paris, France
s-correia@chronopost.fr
[3] School of Computing, Napier University
Edinburgh, Scotland
{o.rossi-doria,b.paechter}@napier.ac.uk
[4] Department of Computer Science, University of Twente
Twente, The Netherlands
snoek@cs.utwente.nl

Abstract. In this work we present a Genetic Algorithm for tackling timetabling problems. Our approach uses an indirect solution representation, which denotes a number of instructions for a timetable builder on how to sequentially build a solution. These instructions are composed by a set of predefined heuristics. The ongoing work presented in this abstract was started by the authors at the EvoNet summer school 2001.

1 Introduction

Timetabling problems are variants of the general resource allocation problem where resources have to be allocated to certain tasks. A university timetabling problem – as considered in this work – consists in assigning each class from a set of classes to a suitable room and a timeslot. Numerous metaheuristic algorithms have been proposed to tackle this highly constrained problem. Among them are Simulated Annealing (SA) approaches (e.g., [1]), Tabu Search (TS) methods (e.g., [7]), and quite a few approaches from the area of Evolutionary Computation (EC) (e.g., [3, 2]).
The Genetic Algorithm (GA) presented in this work searches the space of heuristic rules to be applied during a step–by–step construction of a timetable. The idea of evolving instructions – composed by a set of heuristic rules – for a construction mechanism to build solutions is inspired by the research on evolutionary algorithms for scheduling problems (e.g., [4, 5]). For scheduling problems, it resulted in very successful algorithms, successful in terms of solution quality, but also in robustness and flexibility.

2 Problem definition

The timetabling problem considered here is a reduction of a typical university timetabling problem. A problem instance generator produces problem instances with different characteristics for different values of given parameters. The problem consists of a set of events E to be scheduled in 45 timeslots (5 days of 9 hours each), a set of rooms R in which events can take place, a set of students S who attend the events, and a set of room features F required by events. Each student attends a number of events and each room has a size. A feasible timetable is one in which all events have been assigned a timeslot and a room so that the following hard constraints are satisfied:

- no student attends more than one event at the same time;
- the room is big enough for all the attending students and satisfies all the features required by the event;
- only one event is in each room at any timeslot.

In addition, a candidate timetable is penalized equally for each occurrence of the following soft constraint violations:

- a student has a class in the last slot of the day;
- a student has more than two classes in a row;
- a student has a single class on a day.

3 Our approach

We chose to use the usual framework of a genetic algorithm for our approach. This means, our algorithm works on a population of individuals, applying crossover (e.g., one–point or uniform crossover) and a simple mutation operator to them. As a selection scheme we chose tournament selection. In the following we outline the most important features of this algorithm, namely the specification of individuals, the timetable builder which takes an individual as input and produces a solution from it, and the specification of the fitness function.

3.1 Individuals

In order to reduce the complexity of the timetabling process, we chose a sequential approach. In this approach events are assigned consecutively to a room and then to a timeslot. This requires $|E|$ constructions steps, each one consisting of (i) choosing an unprocessed event $e \in E$, (ii) assigning a room $r \in R$ to e, (iii) assigning the pair $< e, r >$ to a timeslot. Individuals consist of a three row matrix. For each one of the three steps (i)–(iii) there are a number of priority rules available. The number of columns in the matrix is equal to the number of events $|E|$. Position j in the first row specifies the priority rule to choose for performing step (i) in the jth construction step mentioned above (the choice of an unprocessed event). Consequently, position j in the second row specifies the priority rule for performing step (ii) in the jth construction step, and position j in the third row specifies the priority rule for performing step (iii) in the jth construction step.

3.2 Heuristic priority rules

A number of heuristic priority rules for steps (i)–(iii) have been devised. For instance, for choosing a class to be processed next, priority can be given to classes with a high number of students attending. The reason for this is, that one would expect these classes to be problematic in the sense that they are likely to produce clashes when there are parallel classes (classes in the same timeslot). The following *class choice priority rules* have been devised. Among the unprocessed classes, choose the class

1) that has the highest number of students attending,
2) that requires a room which is most required according to the set of unprocessed classes.

For the *assignment of a room* to an already chosen event e, the following rules were devised. Choose among the rooms of correct room–type the

1) smallest possible room,
2) the room with the lowest utilization rate in the partial timetable (where the partial timetable is the current timetable that is under construction).

Using any of these rules will assure that every event will take place in a suitable room. So, these room assignment rules already take care of one of the two possible types of hard constraints. For *assigning an event–room pair to a timeslot*, we implemented the following priority rules. Choose among the timeslots which would cause the lowest number of clashes the one

1) which has the most parallel classes,
2) which has the most students attending a class in parallel.

In applying any of the rules above we use the following policy. If there are ties, we always take the event, room, or timeslot with the lowest label. This is done to make the process deterministic.

3.3 The timetable builder

Given an individual, the timetable builder can incrementally construct a timetable in the following, deterministic way. For construction steps $j = 1, ..., |E|$ do:

- Choose an unprocessed class $e \in E$ using the rule specified in the j-th position of the first row of the chromosome matrix.
- Assign class e to a room r using the rule specified in the j-th position of the second row.
- Assign the pair $< e, r >$ to a timeslot using the rule specified in the j-th position of the third row.

3.4 The fitness function

The evaluation of a timetable consists of two stages. First of all, the feasibility of a timetable is checked. A timetable is feasible if and only if there are no violated hard constraints. Secondly, after feasibility has been checked successfully, a preference for the timetable is calculated by counting the number of soft constraint violations. This number is called the score of the timetable. For use in an evolutionary algorithm, this

score is inadequate, because we also have to allow unfeasible timetables. Therefore we chose the common approach of evaluating the fitness by summing up the weighted number of hard and soft constraints. These weights have to be carefully chosen.

4 Conclusions and outlook to the future

In this paper we have presented a Genetic Algorithm that evolves a policy for a timetable builder to construct good timetables. The availability of powerful heuristic priority rules is of crucial importance to our approach. The extension of the set of heuristic rules, for instance by more smartly using the features of the partial timetable, seems to be an obvious direction for future work. Also, the sequential construction mechanism, which is fixed in its order, may not be optimal. Different orderings (a permutation of steps (i)–(iii) in Sec. 3.1) could be tested. We also plan to use the local search method developed in [6] in a Lamarckian learning approach to support our algorithm in finding good regions in the search space. We will test these different extensions to the algorithm in the near future.

Acknowledgments: This work was partly supported by the "Metaheuristics Network", a Research Training Network funded by the Improving Human Potential program of the CEC, grant HPRN-CT-1999-00106. The information provided is the sole responsibility of the authors and does not reflect the Community's opinion. The Community is not responsible for any use that might be made of data appearing in this publication.

References

1. D. Abramson. Constructing school timetables using simulated annealing: Sequential and parallel algorithms. *Management Science*, 37:98–113, 1991.
2. D. Abramson and J. Abela. A parallel genetic algorithm for solving the school timetabling problem. Technical report, 1991.
3. A. Colorni, M. Dorigo, and V. Maniezzo. Genetic algorithms and highly constrained problems. In *Parallel Problem Solving from Nature - Proceedings of the 1st Workshop*, volume PPSN 1, pages 55–59. Springer-Verlag, 1991.
4. U. Dorndorf and E. Pesch. Evolution based learning in a job shop scheduling environment. *Computers and Operations Research*, 22(1):25–40, 1995.
5. E. Hart, P.M. Ross, and J. Nelson. Solving a real-world problem using an evolving heuristically driven schedule builder. *Evolutionary Computing*, 6(1):61–80, 1998.
6. O. Rossi-Doria, C. Blum, J. Knowles, B. Paechter, M. Sampels, and K. Socha. A local search for the timetabling problem. To appear in the proceedings of PATAT'02, 2002.
7. A. Schaerf. Tabu search techniques for large high-school timetabling problems. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI'96)*, pages 363–368. AAAI Press/MIT Press, 1996.

Multi-Neighbourhood Local Search for Course Timetabling

Luca Di Gaspero[1] and Andrea Schaerf[2]

[1] Dipartimento di Matematica e Informatica
Università di Udine
via delle Scienze 206, I-33100, Udine, Italy
email: digasper@dimi.uniud.it
[2] Dipartimento di Ingegneria Elettrica, Gestionale e Meccanica
Università di Udine
via delle Scienze 208, I-33100, Udine, Italy
email: schaerf@uniud.it

1 Multi-Neighbourhood Local Search

One of the most critical features for applying local search to timetabling problems is the definition of the neighbourhood structure. In fact, for most common timetabling problems, there is more than one neighbourhood structure that is sufficiently natural and intuitive to deserve investigation.

The reason for considering combination of diverse neighbourhood is related to the diversification of search needed to escape from local minima. In fact a solution that is a local minimum for a given definition, is not necessarily a local minimum for another one, and thus an algorithm that uses both has more chances to move toward better solutions.

There are actually many ways to combine different neighbourhoods. In this work, we investigate the following three.

Neighbourhood Union: the neighbourhood is the union of many basic neighbourhoods. A local search algorithm selects, at each iteration, a move belonging to any of the basic components.

Neighbourhood Composition: the neighbourhood is composed by chains of atomic moves of a fixed length. Each atomic move belongs to a given basic neighbourhood.

Token-ring search: Given a set of algorithms based on different neighbourhood functions, the token-ring search makes circularly a run of each algorithm, always starting from the best solution found by the previous one.

All these three notions are not new, and they have been proposed in the literature in similar forms (under various names). For example, the effectiveness of token-ring search for two neighbourhoods has been stressed by several authors (see, e.g.,[3]). In particular, when one of the two algorithms is used exclusively for diversifying the search region, this idea falls under the name of *iterated local search* [5]. As an example, we employed the alternation of tabu search and hill

climbing using two different neighbourhoods for the solution of the high-school timetabling problem [7].

The alternation of simple search and move chains is also the basis of the so-called *Variable Neighbourhood Search* strategy proposed by Hansen and Mladenović [4], which has been used in many applications (see, e.g., [1]).

Our contribution consists in the attempt to systematize the different ideas, and to perform a comprehensive experimental analysis on a real application.

The case study we consider is the COURSE TIMETABLING problem. An application built around the algorithms presented here is actually used to make the working timetable at our university. However, the version of the problem we present in this paper is simplified, by eliminating very specific constraints, so as to reduce it to a more general form.

2 Local Search Kickers

As noticed by several authors (see, e.g., [5]), local search can benefit from alternating regular runs with some perturbations that allow the search to escape from the attraction area of a local minimum. In our settings, we define a form of perturbation, that we call *kick*, in terms of neighbourhood compositions. A *kicker* is an algorithm that makes just one single move, and uses a neighbourhood composition of a relatively long length. A kicker can perform either a random kick, i.e. a random sequence of moves, or a best kick, which means an exhaustive exploration of the composite neighbourhood searching for the best sequence.

Random kicks roughly correspond to the notion of random walk used in [9]. The notion of best kicks is based on the idea of ejection chains (see, e.g., [6]), and generalizes it to generic chains of moves (from different neighbourhoods). In our experiments with kickers as part of a token-ring search, called Run & Kick, the use of best kicks turned out to be very effective.

Notice that the cardinality of a composition is the product of the cardinalities of all the base neighbourhoods, therefore even if the base neighbourhoods have some few thousand members, the computation of the best kick for a composition of length 3 or more is normally intractable. In order to reduce this complexity, we introduce the problem-dependent notion of *synergic moves*. For every pair of neighbourhood functions, the user defines a relation, typically based on equal values of some move features, that specifies whether two member moves are synergic or not.

A move sequence is evaluated only if all pairs of adjacent moves are synergic. In order to build chains of synergic moves, the kicker makes use of a backtracking algorithm that builds it starting from the current state. The algorithm backtracks if a given component move has no feasible synergic move in the subsequent neighbourhood.

The intuition behind the idea of synergic moves is that a combination of moves that are not all focussed on the same features of the current state have little chance to produce improvements. In that case, in fact, the improvements would have been found by one of the runners that make one step at the time.

Conversely, a good sequence of "coordinated" moves can be easily overlooked by a runner based on a basic neighbourhood function.

Different definitions of synergy are possible for a given problem. In general, there is a trade-off between the time necessary to explore the neighbourhood and the probability to find good moves.

3 Course Timetabling

There are various formulations of the Course Timetabling (CTT) problem (see, e.g., [8]), which differ from each other mostly for the hard and soft constraints (or objectives) they consider. For the sake of generality, we consider in this work a quite basic version of the problem. The constraint types considered are the following:

1. **Lectures (hard):** The number of lectures of each course is fixed.
2. **Room Occupancy (hard):** Two distinct lectures cannot take place in the same room in the same period.
3. **Conflicts (hard):** Lectures of courses in the same *curriculum* must be all scheduled at different times.
4. **Availabilities (hard):** Teachers might be not available for some periods.
5. **Room Capacity (soft):** The number of students that attend a course must be less or equal than the number of seats of all the rooms that host its lectures.
6. **Minimum working days (soft):** The lectures of each course must be spread into a minimum number of days.
7. **Curriculum compactness (soft):** The daily schedule of a curriculum should be as much compact as possible, avoiding gaps between courses.

In the CTT problem, we are dealing with the assignment of a lecture to two kinds of resources: the time periods and the rooms. Therefore, one can very intuitively define two basic neighbourhoods which deal separately with each one of these components.

The first neighbourhood, Time, is defined by simply changing the period assigned to a lecture of a given course to a new one which satisfies the availabilities. The Room neighbourhood, instead, is defined by changing the room assigned to a lecture in a given period.

Given these two basic neighbourhoods we define the neighbourhood union Time$\oplus$Room, whose moves are either a Time or a Room, and the neighbourhood composition Time$\otimes$Room which involves both the resources at once.

We define 8 algorithms, obtained equipping hill-climbing and tabu search with the four neighbourhoods: Time, Room, Time$\oplus$Room, and Time$\otimes$Room.

Moreover, we define two kickers both based on the composition of Time and Room. The kickers differ to each other on the definition of the synergic moves: one is more strict, requiring always same course and same period, the other is more relaxed and allows also combination of moves on different courses.

Our software tool EasyLocal++ [2] generates automatically the code for exploration of complex neighbourhoods starting from the code for the basic ones. This is very important, from the practical point of view, so that the test for multi-neighbourhood techniques is inexpensive not only in terms of design efforts, but also it terms of human programming resources.

4 Outline of the Experimental Results

Up to our knowledge no benchmark instance for the CTT problem is made available in the scientific community. For this reason we decided to test our algorithms with three real-world instances from the School of Engineering of our university. Real data have been simplified so as to adapt to the problem version of this work, but the overall structure of the instances is not affected by the simplification. These instances will be made available through the web.

The main features of these instances are reported in Table 1. All of them have to be scheduled in 5 days of 4 periods each, for a total of $p = 20$ periods.

Instance no.	Courses	Lectures	Rooms	Conflicts	Occupancy
1	46	207	12	4.63%	86.25%
2	52	223	12	4.75%	92.91%
3	56	252	13	4.61%	96.92%

Table 1. Features of the instances used in the experiments

The columns "Conflicts" and "Occupancy" show the density of the conflict matrix, and the percentage of occupancy of the rooms ($\sum_i l_i/r \cdot p$), respectively. They are the main indicators of the instance constrainedness.

In order to obtain a fair comparison among all algorithms, we fix an upper bound on the overall computational time (600 secs per instance) of each algorithm during multiple trials, and we record the best value found up to that time. This way, each algorithm can take advantage of a multi-start strategy proportionally with its speed, thus having increased chances to reach a good local minimum.

We run the HC and TS Multi-Neighbourhood algorithms with the best parameter settings found in a preliminary test phase. Namely, the tabu tenure is fixed to the range $20 \div 30$ and the number of idle iterations allowed is 1 million for HC and 1000 for TS.

From the results, it turns out that the HC algorithms are superior to the TS ones for two out of three instances. Concerning the comparison of neighbourhood operators, the best results are obtained by Time⊕Room for both HC and TS.

We take into account 3 types of kicks. The first two are best kicks with the strict and the more relaxed definition of move synergy. To the aim of maintaining the computation time under a certain level we experiment these kickers only with step $h = 2$ and $h = 3$. The third is a random kick of length $h = 10$ or $h = 20$. In preliminary experiments, we have found that shorter random walks are almost

always undone by the local search algorithms in token-ring alternation with the kicker. On the contrary, longer walks perturb too much the solution leading to a waste of computation time.

Our results show that kickers implementing the best kick strategy of length 2 increase the ability of the local search algorithms independently of the search technique employed. Unfortunately, the same conclusion does not hold for the best kicks of length 3. In fact, the time limit granted to the algorithms makes possible only to perform a single best kick of this length at early stages in the search. Therefore, for instances of our size the improvement in the search made by these kicks is hidden because of their high computational cost.

Furthermore, it is possible to see that for TS the random kick strategy obtains moderate improvements in joint action with union and composition neighbourhoods, favouring a diversification of the search. Conversely, the behaviour of the HC algorithms with this kind of kicks is not uniform, and it deserves further investigation.

References

1. Matthijs den Besten and Thomas Stützle. Neighborhoods revisited: An experimental investigation into the effectiveness of variable neighborhood descent for scheduling. In J. Pinho de Sousa, editor, *Proc. of the 4th Metaheuristics International Conference (MIC-01)*, pages 545–550, 2001.
2. Luca Di Gaspero and Andrea Schaerf. EasyLocal++: An object-oriented framework for flexible design of local search algorithms. Technical Report UDMI/13/2000/RR, Dipartimento di Matematica e Informatica, Università di Udine, 2000. Available at `http://www.diegm.uniud.it/schaerf/projects/local++`.
3. Fred Glover and Manuel Laguna. *Tabu search*. Kluwer Academic Publishers, 1997.
4. P. Hansen and N. Mladenović. An introduction to variable neighbourhood search. In S. Voß, S. Martello, I.H. Osman, and C. Roucairol, editors, *Meta-Heuristics: Advances and Trends in Local Search Paradigms for Optimization*, pages 433–458. Kluwer Academic Publishers, 1999.
5. Helena Ramalhino Lourenço, Olivier Martin, and Thomas Stützle. Applying iterated local search to the permutation flow shop problem. In F. Glover and G. Kochenberger, editors, *Handbook of Metaheuristics*. Kluwer, 2001. to appear.
6. E. Pesch and F. Glover. TSP ejection chains. *Discrete Applied Mathematics*, 76:175–181, 1997.
7. Andrea Schaerf. Local search techniques for large high-school timetabling problems. *IEEE Transactions on Systems, Man, and Cybernetics*, 29(4):368–377, 1999.
8. Andrea Schaerf. A survey of automated timetabling. *Artificial Intelligence Review*, 13(2):87–127, 1999.
9. Bart Selman, Henry A. Kautz, and Bram Cohen. Noise strategies for improving local search. In *Proc. of the 12th Nat. Conf. on Artificial Intelligence (AAAI-94)*, pages 337–343, 1994.

Addressing the Availability-Based Laboratory/Tutorial Timetabling Problem with Heuristics and Metaheurstics

David Corne[1], Jeffrey Kingston[2]

[1] School of Computer Science, Cybernetics and Electronic Engineering, University of Reading, UK, Email: d.w.corne@reading.ac.uk
[2] Department of Computer Science, University of Syndey, Australia, Email: jeff@cs.usyd.edu.au

1 EXTENDED ABSTRACT

The availability-based laboratory/tutorial timetabling problem (ALTP) is a distinct and very difficult problem faced regularly in many university departments worldwide. As ever, the problem occurs in a great variety of forms, but we consider here a particular form of the problem which we believe will become more common in future, and which also happens to be the form of the ALTP which is regularly faced by one of the co-authors.

In this extended abstract, we describe the problem, describe some approaches to solving it, and describe some experiments. The full paper, if this is abstract is accepted, will naturally expand on all aspects of this abstract but especially in terms of results; the full paper will also include complexity analyses of variants of the ALTP.

The ALTP

We first sketch out the scenario in which the ALTP arises. Imagine a thriving university running a highly modular degree scheme, whereby a number of courses are being delivered, different students tend to have different timetables and course options, and several of the courses being taken require laboratory and/or tutorial sessions to be set up during the week. In particular, a course will typically involve 50 or maybe 100 students; the students will usually be together in lectures, but need to be partitioned into small groups for laboratory or tutorial sessions. So, the precise makeup of each such group, together with a time and a place for each group, needs to be determined.

It is easiest to get a feel for the ALTP by considering increasingly complicated versions of it. We will do this, but before that we will initialise by noting certain assumptions which underlay most ALTPs. The main assumption is that the basic lecture timetable is already in place. In other words, the only events for which we need to find times are the lab and tutorial groups (which first need to be specified!). Another assumption, which is commonly true, is that there is a designated room for each laboratory or tutorial session. That is, for example, all

of the Systems Architecture laboratoray sessions will take place in the Systems Architecture Lab. Finally, we assume that each student involved has provided us with a list of available times, and an order of preference on those times.

The last of these assumptions may seem strange, and deserves some attention. In the good old days in the UK, the great majority of university students had student grants which (just about) covered their tuition fees and living expenses, so that the student need not worry about work, and hence could get on with the business of being a student, and also get on with studying. In more recent times, there have been two key factors which have caused a massive shift in student demographics. First, the grant system is now gone, and second: policies of various kinds have led to a massive increase in mature students and part-time students. The end result is that a very significant proportion of the student body have availabilities and preferences regarding their 'free' time which the University needs to take seriously.

In each of the ALTP variants we describe next, we therefore assume that data includes a preference-ordered list of available timeslots for each student. In terms of data input (and also in practice) this list already and conveniently takes into account each students timetabled as well as personal commitments. Hence, there is no need for software addressing the ALTP to consider pre-arranged timetable data and student registration data.

In all ALTPs described next, we therefore assume that there are s students, for each of which we have an array s_i of timeslots, associated with an array c_i of preferences. Eg, if $s_i[3] = 12$ and $c_i[3] = 3$, then student s_i is available at timeslot 12 with a preference level of 3.

ALTP-1 The simplest form of ALTP, which we call ALTP-1, involves just one course which requires lab or tutorial sessions. In ALTP-1, a course with n students needs to be partitioned into g groups each of a maximum size m (typically $m << n$), and a timeslot must be found for each of the resulting groups. Clearly, the timeslot assigned to g_i must be available to all of the students placed in group g_i. Above this, we would expect to maximise the degree to which preferred timeslots were used.

ALTP-1 is therefore a straightforward partitioning problem.

The next form of the problem is ALTP-k, in which there are now k (greater than 1) courses which involve labs or tutorials. We need a further array, t_i, associated with each student i, to identify which subset of these k courses the student takes. For example, $t_i[3]$=0 indicates that student i does not take course 3, and hence need not be involved in any lab sessions for that course.

The problem is now to find a partition of students for each course, and times for each group from each resulting partition. Further, we now must ensure that the same student is not assigned to the same timeslot in partitions from different courses. That is, if A. Student is assigned to the Monday 11:00am lab slot for the Communications course, A. Student is not available at that time for labs of other courses.

This is the most common form of the problem.

A further form of the problem, which we do not consider here, can be called ALTP-k-n, which is as ALTP-k, with the additional complication that students on at least one of the courses need to be assigned more than one lab session per week for that course (ie: m lab sessions per week, where $m > 1$). Solutions to this problem might simply produce one partition for such a course, but assign two suitable times for each group in that partition. It may be necessary in some cases, however, to produce up to m distinct partitions for such a course and assign a suitable time to each group in each partition.

Heuristics and Metaheuristics for the ALTP

In this paper we will concentrate on the ALTP-k problem, for which we have several archived real-world examples. Heuristics, Metaheursitics, and combinations will be prepared. and where possible the results compared with hand-derived solutions. Our aim is to develop a flexible, fast and robust way of solving this problem which optimises well over the preference constraints while (of course) meeting all of the hard constraints.

The heuristics experimented with will include:

Placement-1 : A straightforward constructive technique which assigns students one by one to a group for each course they need to take, maintaining integrity of the developing solution as it occurs. Students are pre-ordered according to difficulty, which is measured in terms of available times and number of courses they take.

Placement-2 : First derive a partition for each course via a simple heuristic which tries to maximise the degree to which partitions between courses are similar. That is, a partition for course 1 is first derived arbitrarily, then a partition for course 2 is initialised to be the same as for course 1, and then repaired, and so forth. Then, times are assigned to each group of each partition via a simple greedy heuristic.

I-P1 : A randomised iterated version of Placement 1; ie, running placement-1 several times with different student orderings.

I-P2 : A randomised iterated version of Placement 2; ie, running placement-2 several times with different student orderings and course orderings.

The metaheuristics will be hill-climbing, simulated annealing, and an evolutionary algorithm, each using two different approaches. In the first approach, the chromosome represents an ordering of students for decoding by the placement-1 heuristic. In the second approach, there will be two stages: in the first stage the metaheuristic aims to optimise the set of partitions so as to minimise the potential conflicts bewteen groups in differnt partitions. In the second stage the metaheuristic attempts to assign times to the groups in such a way as to satisfy tehhard constraints and optimise over the preference constraints.

Preliminary Discussion

Most of the above methods have been tested on one or more different real versions of ALTP-k. We have yet to perform a systematic set of experiments comparing

all of the techniques over a good range of problems. Although the placement heuristics work quite well and quickly in terms of providing a feasible solution, they are terrible at producing preference-optimised results. Evolutionary algorithms along the lines discussed above have also been tried on different ALTP variants, and have produced good results, though quite slowly.

The full paper, should we be fortunate enough to have this abstract accepted, will answer several burning questions concerned with which method works best on a range of real problems, what the speed/quality tradeoffs are, and so forth.

The full paper will also include a URL via which source code will be available for any who wish to either continue research on solving the ALTP, or actually need to solve the ALTP.

School Timetabling

An Average Case Approximation Bound for Course Scheduling by Greedy Bipartite Matching

Gary Lewandowski[1], Prakash Ojha[2], Jennifer Rizzo[1], and Abigail Walker[1]

[1] Xavier University, Mathematics and Computer Science Department, Cincinnati OH 45207-4441, USA {lewandow,rizzo,awalker}@cs.xu.edu
[2] Hope College, Computer Science Department, Holland MI 49423 ojha@cs.hope.edu

1 The Problem we are studying

1.1 Which scheduling problem?

We are studying the combined problem of generating a timetable and student schedules from initial data that includes student course requests, the number of sections allowed of each course, the maximum number of students allowed into each course, and the maximum number of times available in the timetable.

Our approach extends to handle issues of time constraints for instructors, time requests for students, and limitations on room availability but we do not consider these in this paper.

The goal, given the initial input, is to develop a timetable indicating the time at which each section of a course will be offered as well as a schedule for each student indicating the times at which the courses requested are scheduled. The timetable and schedules should be constructed to minimize the number of conflicts experienced by the students.

1.2 Metric for counting conflicts

Initial input to this problem is a set of course requests from each student. Upon completion each student has an actual schedule listing the particular sections into which the student has been placed. The difference between the number of courses requested and the number of timeslots used by the actual sections is the number of conficts experienced by the student.

This metric recognizes two ways of handling conflicts: a scheduling algorithm could either schedule the student into sections that are scheduled simultaneously or simply not schedule the student into sections that will have conflicts (i.e. if two sections are at the same time, the student will get one of those courses but not both).

2 Our focus

Our interest in this study is approximation bounds for the problem. In particular, we want to know if we can prove any approximation bound at all on the quality generated by a heuristic solution.

While many techniques have been used to approach school timetable problems, little work has appeared discussing the quality of solution. T-coloring graphs is related, and bounds have been proven using randomization, by Vitanyi [3], and semi-definite programming, by Frieze and Jerrum [2]. However, the definition of a conflict in coloring is different from a conflict in course scheduling so these results do not immediately apply. Berry, Condon and Halsey [1] were able to adapt Vitanyi's result to show that in the case of single-section courses, randomly scheduling each course yields a $1/(1-1/e)$ approximation algorithm, but it has proven difficult to adapt Frieze and Jerrum's result.

Our study examines bipartite matching as the basis for a greedy approximation algorithm. The heuristic to build the timetable and student schedules is very simple. Given a randomly ordered list of student course choices, each student's choices are processed as follows: construct a bipartite graph consisting of the courses desired by the student in one component, and the T timeslots in the other component. Edges are placed between course i and time j if the course has an unfilled section scheduled at time j, or if the course has a section that has not yet been scheduled. After building this bipartite graph, bipartite matching creates a match between courses and timeslots. The matching provides a schedule for the student and may also cause a section to be scheduled if the timeslot with which a course is matched has not already been assigned a section of the course.

3 Theoretical Results

The quality of approximation achievable using the bipartite matching heuristic varies with the number of sections available for each course. Our analysis makes a simplifying assumption that students choose their courses randomly. While this does not appear realistic, our empirical tests indicate this assumption appears to be a cause for underestimating the quality rather than overestimating.

Our first result shows that when each course has a single section, this heuristic achieves an approximation ratio $< 1/(1-1/e)$, i.e, we can prove the performance is better than random assignment of times to sections, but cannot prove it is much better.

Our second result analyzes the case of each course having two sections. We are able to show that the approximation ratio ρ is between $1/(1-1/e^2)$ and 1.255. That is, average use of greedy bipartite matching yields results with at most 25.5% more conflicts than optimal. Empirical results show that in practice we see an average of about 18.5%.

Analyzing the algorithm beyond the case of two sections per course proves untractable and we turn instead to simulations and empirical studies.

4 Empirical Evidence

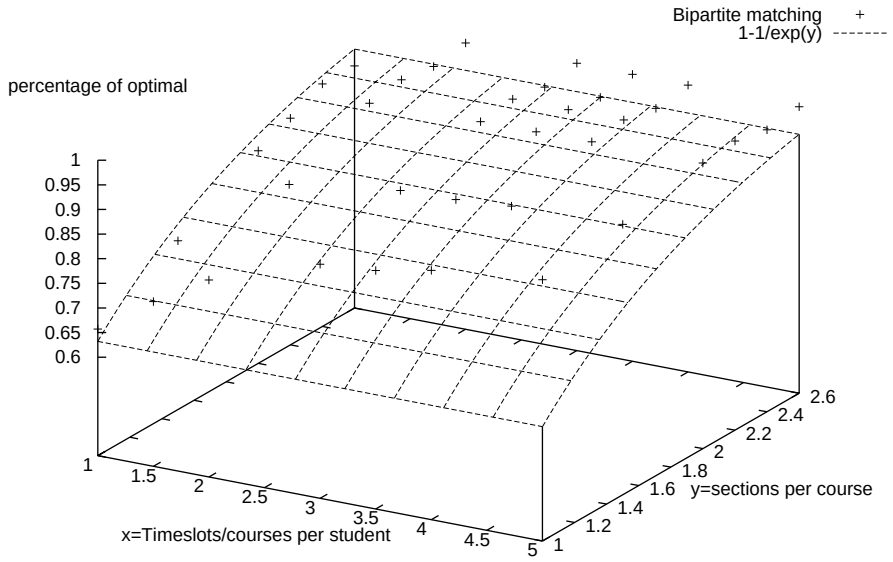

Fig. 1. Varying T/r as well as section ratios

We simulated the algorithm on a wide variety of cases, varying the number of timeslots, number of courses requested by the students, number of section, and number of student. Figure 1 shows our main results, plotted as 1/approximationRatio. We see that

- as the *average* number of sections, s, increases, the quality of the solution increases correspondingly, following the curve $1/(1-1/e^{\Theta(s)}$.
- if the number of timeslots T does not equal the number of courses chosen by each student, r, then the approximation bound follows a curve which is roughly $1/(1-1/e^{\Theta(s*T/r)}$.

5 How this compares to other aproximation techniques

Having analytically discovered the greedy bipartite matching algorithm should give promising results, we next studied it in comparison with simulated annealing.

We report here early results and are working on a more tuned version of simulated annealing for a better comparison. Our initial results indicate that the matching heuristic is competitive in quality with simulated anealing. In Table 1 we display the results of our first study. The data sets are named as numberOfSections-totalSeatsDesired-totalAvailableSeats, e.g. 2-653-900. The seats desired is simply the total number of course requests by students. The total available seats is the number of seats allowed per section times the total number of sections in the timetable. We report the highest, lowest, and the median number of conflicts over five runs of each algorithm. We also include the approximation ratio.

Table 1 indicates that simulated annealing does much better if only one section is available per course, but the two methods are relatively comparable when there are two sections of each course.

Data Set	Simulated Annealing				Bipartite Matching			
	h	l	m	approx ratio	h	l	m	approx ratio
1-40-100	40	40	40	1.000	39	34	36	1.111
1-70-100	70	70	70	1.000	70	57	59	1.186
1-650-900	650	650	650	1.000	512	480	504	1.290
1-2800-3750	2767	2608	2713	1.032	1935	1891	1926	1.454
2-38-100	38	38	38	1.000	38	36	38	1.000
2-36-200	36	36	36	1.000	36	35	36	1.000
2-72-200	72	72	72	1.000	72	70	72	1.000
2-653-900	639	635	637	1.025	622	605	617	1.058
2-2556-3000	2397	2384	2395	1.067	2378	2358	2364	1.081
2-6922-15000	6278	6244	6258	1.106	6186	6173	6181	1.120
2-6952-25000	6425	6396	6413	1.084	6309	6256	6282	1.107
2-10382-15000	9320	9257	9262	1.121	9207	9185	9195	1.129
2-10422-25000	9475	9404	9439	1.104	9329	9278	9307	1.120
2-13849-15000	12402	12211	12243	1.131	12230	12117	12154	1.139
2-13880-25000	12509	12159	12443	1.115	12336	12301	12322	1.126
2-20363-25000	18208	18157	18190	1.119	18128	18103	18110	1.124

Table 1. Simulated Annealing and Bipartite Matching results

References

1. C. Berry, A. Condon, B. Halsey. Best Schedule, manuscript, 1995.
2. A. Frieze and M. Jerrum. Improved approximation algorithms for MAX k-CUT and MAX BISECTION. Algorithmica, 18(1):67–81, May 1997.
3. P.M.B. Vitányi. How well can a graph be n-colored?, *Discrete Mathematics*, 34, 1981, 69-80.

Medical Employee Timetabling

Personnel Scheduling in Laboratories using IPS

Philip Franses[1] and Gerhard Post[1,2]

[1] ORTEC Consultants BV, Osloweg 131, 9723 BK Groningen, The Netherlands
[2] Department of Mathematical Sciences, University Twente,
P.O.Box 217, 7500 AE Enschede, The Netherlands

Abstract. This paper describes the software package IPS, which was developed for personnel-planning in laboratories related to health care. We describe the characteristics of these laboratories, and review how IPS tries to assist the planner. In particular we give a description of the task-generator in IPS, and explain the assignment algorithm that is behind it. Its main characteristic is the introduction of profiles, which easily allows the user to steer the algorithm.

1 Introduction

The planning of personnel is both at the heart of the human side of an organisation and open to increased efficiency through technical aids. The theory of automated time tabling must be such that it does not eclipse the practice of planning the work times of people.

The typical user of IPS in laboratories, chemists, X-ray departments, operation departments has to deal with the attendance of the workers, their holidays and which task or work-place they shall occupy during their work. The concern of the worker is that there is a good rhythm to the tasks he does and he does not work more of the disliked tasks or shifts than his colleagues.

IPS is designed particularly for the situation in which a number of tasks are to be carried out by skilled employees during a period of at least half a day[3]. Previous attempts to do this planning by hand were time-consuming and achieved in an ad-hoc manner. The idea of the generator was to provide a tool to arrive at an optimal match between the tasks that have to be done and the available employees. It should be flexible in its approach, to cater for different and changing organisations.

The generator comes into action at the end of a long process of planning, in which the information, relevant to the organisation, is input and maintained. This process can be summarised as taking place in a number of stages.

The first stage is the long term planning in which attendance, night shifts and roster-free days are planned in advance; then there is the middle-term planning in which holidays are planned and shifts swapped; and finally the short-term planning in which the roster focuses in on the tasks that have to be done.

[3] That a task lasts at least half a day is not very essential. Essential is that any two tasks either are disjoint in time, or overlap in time.

Cyclical rosters achieve the long term planning. These rosters provide a regular cycle of day shifts, evening shifts, night shifts and roster-free days. This ensures a variation in the work and a fair sharing of shifts such as night shifts.

The roster is in the middle term adjusted with the practice of holidays, absence and shift swapping.

Finally we come to the allocation of who does which jobs. These tasks are jobs that can be done by a number of specialist individuals and the puzzle is to match the jobs to the qualified workers. This is where the task generator of IPS is used.

2 The Task Generator

2.1 Tasks

The situation in laboratories as we consider is that an employee is assigned to a task in the morning and in the afternoon. The task generator assigns tasks to employees that are qualified for it, and are available. These are facts that are known in advance. In a run of the algorithm we assign the tasks of (typically) a week to a group of employees that usually consists of 50 to 100 persons. Such a run should take at most a few minutes. In IPS three types of tasks are distinguished:

half-day task: This kind of task is required to circulate as much as possible among all people that are qualified to do it. Hence in the morning and the afternoon different people should do this task.

day task: This task should be assigned to the same person during the morning and the afternoon, but the next day a different task for this person is required.

week task: This task should be assigned for the whole week to one person.

It will depend on the particular situation which of the types 'day task' and 'week task' correspond to their working pattern: some prefer to give the employees the same task during the day, while others do this for the whole week. The first type, the 'half-day task', usually is connected with the boring tasks. As an example one can think of the task of answering the telephone.

Apart from the requirements of the type of task, the most important aspect we have to cope with is the 'history'. By this we mean how often a person has worked on each of the tasks over the previous weeks. The tasks for the week we want to plan usually are aimed to provide variation from the previous weeks, but can also aim to maintain workers in the same tasks.

In practice there might occur extra requirements with respect to the schedule. We mention two important ones, though the mechanism of 'profiles' (see below) is general enough to incorporated several other requirements as well.

1. *Priorities.* Some tasks are more important than other tasks. This means that these tasks should be assigned first to cater for cases where not all tasks can be assigned.

2. *Supervisors.* There are tasks that require the presence of extra qualified employees. The task is accomplished by a team, of which one acts as a supervisor without the task 'supervisor' being specified separately.

2.2 The algorithm

We have chosen an approach that makes a schedule optimal for certain criteria, and improve it by local search. To incorporate the extra requirements we work with 'profiles'. A profile consists of a number of employees, tasks, type of tasks and the next profile. This way we get a chain of profiles, which we treat consecutively. At first the top profile is active. We select all tasks and employees in it, and give to the combination of task and employee a score depending on skill (qualification), history, and the relation to the history, as specified in the profile. The tasks and employees in the lower profiles are also incorporated, but the available employees get a very low score, equal for all of them. This expresses the fact that, at the moment, we are not interested where these employees are scheduled, but only that it is possible to schedule them. The actual scheduling of these lower profiles will be done at a later stage.

The first step is finding the optimal match for each half day. This is done by the (weighted) Hungarian method. The resulting matching is the starting point for the second step: improving the matching. The way to improve it depends on the type of tasks. In case we are handling half-day tasks or day tasks, we change the scores for other days immediately.

The procedure for week tasks is more intricate and interesting. After all days matchings are computed, a new score is considered, composed of two parts: the first is the percentage the employee is available for tasks in the week to plan. The second part is the percentage that the task is assigned to the employee. Consequently the maximal score is 200. Next we determine the combination task-employee with the highest score. We check if it is possible, without loosing maximal matching, to assign this combination during the whole week. If so, we assign the combination permanently (we fix it), and we proceed with the next best task-employee combination.

In practical situations this procedure works quite well: usually the tasks for several persons are already the same or 'almost' the same during the week. Employees that work the most during the week we are trying to plan, are assigned the week tasks first, due to the score we introduced. This seems quite reasonable, since these are employee are the most difficult to fit in at the end.

When all task-employee combinations are tried, we come the the final step. We repeat the procedure above, but now without requirement of assignment for the whole week. So we calculate the scores in the same way as above, and find the best task-employee combination. We assign this combination, and lower all other scores of this employee by 200. This forces the algorithm to consider first other employees, i.e. those for which no tasks have been found sofar. Once all employees have been considered, we continue with the tasks that are left over.

3 Appraisal

The system IPS was developed 2 years ago, originally with a priority-based algorithm for the task scheduling. It turned out to be hard to steer this algorithm to a result that is considered a 'good' planning. From several sides the question came to incorporate the possibility of requirements as described in the sections 2.1 and 2.2.

The new algorithm has been rigorously tested (and is now in daily use over several departments) at Canisius Wilhelmina Ziekenhuis Nijmegen Administration and X-ray departments and in the Medisch Laboratorium Noord, Groningen in two Laboratory Departments. Critical points on which the algorithm were judged was the sharing of unwanted shifts such as telephone duty and the continuity in the allocation of standard tasks, workers doing where possible the same task over a week. Use was made of the profiles to generate a planning over the various task-sorts.

The success of the algorithm lies in the fact that the generator improved the success of task-planning, while providing an enormous saving in the time previously needed to make the planning. The planner is able to trust that the algorithm provides a solution that both planner and workers can happily accept. Following on the success of the above mentioned test-sites, the generator is also in increasing use in the growing number (now over thirty) of hospital departments where IPS is used.

Subproblem-centric algorithms for the nurse scheduling problem

Atsuko Ikegami and Akira Niwa

Seikei University, Tokyo 180-8633, Japan
{atsuko,niwa}@is.seikei.ac.jp
http://www.is.seikei.ac.jp/indexe.html

Maintenance of high level nursing care is a primary factor in the provision of quality health care in hospitals. A head/chief nurse must assign nurses to each shift according to numbers and skill levels required. However, limitations to individual workloads make this task laborious and time consuming. To clarify the complexities of the scheduling problem, we administered a questionnaire to head/chief nurses of 40 units at Tokyo Women's Medical College Hospital in 1994, before extending our survey to a total of 315 units of 23 hospitals in 1997. Based on the information we collected, we have been able to design a nurse scheduling model. The survey clarified the characteristics of nurse scheduling in Japan as follows:
(1) The scheduling period is monthly (sometimes 4 weeks).
(2) Nurses are grouped for scheduling according to skill level and the patients they are responsible for.
(3) Most nurses work rotating shifts.
(4) These nurses work in rapid shift rotation.
(5) The skill level of the team is considered carefully due to a lack of nurses, particularly skilled nurses, to schedule (mainly due to high turnover caused by early resignations).
(6) Scheduling must account for nurses' requests.

These characteristics create constraints involving more than one group, such that scheduling cannot be developed for each group autonomously. In addition, balancing workloads for shorter scheduling periods is difficult, as nurses' preferences must be considered. Constraints were also found to differ among nurses as well as among days in the scheduling period. Thus we determined that existing algorithms for nurse scheduling are not applicable for use in most Japanese hospitals because they apply mainly to nurses working a fixed shift [1] or a slow shift rotation [2], or to shorter scheduling periods [3].

Our nurse scheduling model accounts for the two types of constraints: a set of 'shift constraints' and a set of 'nurse constraints'. Shift constraints maintain the desired level of service in terms of numbers and skill sets for each shift, where each of the shift constraints sets the lower bound and/or upper bound for the total number of nurses working in each shift as well as within each group. Nurse constraints consider the workload of each nurse in relation to: (1) the number of days off, day shifts and night shift within a given range; (2) preferences for days off, attending seminars etc.; and (3) arrangement of shifts as certain arrangements are deemed detrimental to nurses' health. The objective

in Japan, rather than to minimize costs, is to make a schedule which satisfies all of these constraints.

We first defined a subproblem for a specific nurse to minimize the degree of violation for the shift constraints, using the specific nurse's constraints and the schedules of other nurses as specified by a trial solution. Starting from an appropriate trial solution, our approach solves this subproblem for each nurse and accepts the best solution from the results as the next trial solution. It continues this process until the objective value is zero.

Based on this approach the algorithm for a 2-shift system creates a schedule in two phases: Phase I finds a schedule that satisfies all the constraints associated with nightshifts and the dayshift lower bounds, and Phase II completes the schedule by satisfying the remaining constraints. The algorithm repeats the same process according to our approach in each phase. However, for a 3-shift system, we cannot construct such a phased algorithm because of the many constraints associated with the arrangement of shifts. Therefore our algorithm for a 3-shift problem finds a monthly schedule for a specific nurse by combining schedules for shorter periods (e.g., 7 days) in the subproblem. We use a branch-and-bound algorithm and other heuristics to solve the subproblem. Here we will describe only the algorithm for a 2-shift system.

In Phase I, we prepare a set of individual nightshift schedules for each nurse that satisfy personal requirements as well as general requirements applicable to each nurse. The nightshift schedule contains additional information about whether the nurse is available for dayshifts on the remaining days. A modified objective function then ignores the dayshift upper bounds. All feasible schedules prepared by this process are compared at each stage of computation for each nurse in terms of the value of the objective function, given that the other nurses' schedules are specified by the current trial solution. Finally, the best solution is chosen from the best solutions found for each nurse. This solution in turn will be used as the next trial solution to define the new optimization problem to be solved at the next stage. For the first trial solution we use a 'dummy' solution that assigns no nurse to any shift on any day. We also isolate the specific schedule that produced the current trial solution from future comparison processes to avoid creation of a loop. The iteration is continued until we have a schedule for which the value of the objective function is zero. The same process is carried out in Phase II, except that we prepare a set of completed schedules for each nurse based on the result from Phase I. Each phase of our algorithm is equivalent to a tabu search. The neighborhood around the current solution is a set of schedules that can be obtained by changing a feasible pattern for a single nurse.

A modified algorithm examines only those schedules in Phase I that have little difference from the current solution, where the difference is defined to be the number of days whose night shifts are staffed in one solution while not in the other. Our algorithms produced feasible schedules for two sets of actual data provided by Tokyo Women's Medical College Hospital. Compared to our original algorithm, processing time was reduced by more than 80% for the same sets of

data when applying our modified algorithm, enabling head/chief nurses to create a schedule for a 2-shift system within only a few minutes.

References

1. Warner, D.M.: Scheduling nursing personnel according to nursing preference: a mathematical programming approach. Operations Research **24** (1976) 842–856
2. Randhawa, S.U., Sitompul D.: A heuristic-based computerized nurse scheduling system. Computer & Operations Research **20** (1993) 837–844
3. Dowsland, K.A., Thompson, J.M.: Solving a nurse scheduling problem with knapsacks, networks and tabu search. Journal of the Operational Research Society **51** (2000) 825–833

Scheduling Agents - Distributed Employee Timetabling (DETP)

Amnon Meisels and Eliezer Kaplansky
Department of Computer Science
Ben-Gurion University of the Negev
Beer-Sheva, 84-105, Israel
e-mail: {am,kaplan_e}@cs.bgu.ac.il

Abstract

Employee timetabling problems (ETPs) involve an organization with a set of tasks that need to be fulfilled by a set of employees, with their own qualifications, constraints and preferences. The organization enforces overall regulations and attempts to achieve some global objectives such as lowering the overall cost, or an equitable division of work among certain employees. This leads naturally to the formulation of ETPs as *constraints networks (CNs)*.

Many real world ETPs are composed of organizational parts that need to timetable their staff in an independent way, while adhering to some global constraints. Later the departments timetables are combined to yield a coherent, consistent solution. This last phase involve negotiations with the various agents and requests for changes in their own solutions.

An example for such real-life problem is the construction of a weekly timetable of nurses in several wards of a large hospital. Each ward needs a weekly timetable for its nurses, that satisfies the shift and task requirements, attempting to satisfy also the nurses' personal preferences. The generation of a weekly schedule for the nurses in a ward is typically performed by each ward independently.

Based on the departments timetables, a transportation plan for all the nurses must be constructed. Since the number of vehicles sent and the distances they cover are limited by hospital saving policies, such a transportation plan generates additional constraints between the weekly schedules of the wards.

Since the separate timetables for wards are generated locally, the constraints among these timetables, imposed by the limits on transportation vehicles, have to be negotiated among the scheduling agents of the wards.

The generic scenario of scheduling agents is that of distributed search. Agents solve their local assignment problem and interact via messages to

make their local schedules compatible with the global constraints of transportation. This scenario falls exactly into the domain of *distributed constraint networks (DCNs)* and distributed search algorithms [3]. The agents that generate the weekly schedules of the wards are termed *Scheduling Agents (SAs)*.

The above real world *Distributed Employee Timetabling Problem* (DETP) is an instance of distributed constraints network. Its DCN has a typical structure, in which the local timetabling problem is quite complex and the global problem is very large. One can think of a DETP as a network of scheduling agents.

The present paper investigates solution methods for networks of SAs. In our model, we proposed to assign one of the agents as the *Central Agent* (CA). All agents communicate by sending and receiving messages. SAs send their local timetables to the central agent. The CA checks the local timetables for conflicts with global constraints and decides which agents have to be requested to change their local timetables. Based on this model, two concurrent algorithms for solving DETPs are proposed and compared.

There are two well known families of algorithms for solving timetabling problems - exhaustive (or complete) search algorithms and stochastic (or local) search algorithms. Exhaustive search can go on systematically to find all possible timetables, while stochastic search, typically, produces one solution, reaching a minimum. It has been shown in the past [1] that for real-world ETPs, stochastic search methods outperform complete search in efficiency terms. Several search algorithms for distributed constraints networks have been proposed in the last decade [2, 3], however, existing algorithms for solving DCNs are all exhaustive. Nevertheless we have found that both methods are useful in solving DETPs.

One approach, proposed here, uses exhaustive search for the SAs. In this approach, the SAs perform their exhaustive, systematic, algorithm continuously. Each new local solution they find is sent as a message to the CA. This method is based on the fact that any global solution will always incorporate one local timetable from each SA (i.e. for each ward of the hospital in our example). In order to produce the overall solution, the CA has to search among the set of local solutions, while its domain enlarged dynamically, for one subset that satisfied all the global (inter-agent) constraints.

The advantage of the exhaustive search approach is in it's high degree of concurrency. In addition it is guaranteed to find a global solution if such a solution exists. However, for real world problems, complete search can be very slow [1] .

In the second approach proposed in this paper, each SA uses local search to generate its local timetable. Each SA can find a local timetable relatively fast, but then it waits for a request, from the central agent, to improve their solution. This request is result of the CA attempt to resolve some additional global constraint violations. Upon receiving such

a message, an SA resumes its local search (i.e. hill climbing) in order to find an improved solution.

The advantage of the local search agents approach is that it is likely to converge quickly to some global solution, by using the strategy of improving local solutions.

The main drawback of this approach is that the CA may find itself performing cycles in searching for a globally improved solution, since local solutions improve only locally.

The paper compares the behavior of the above two approaches on realistic instances of DETPs. The local ETPs are real world instances of wards in a large hospital. The constraints among timetables of the wards are simple limit constraints on the number of nurses that have to transported to each location, in each of the daily shifts. The induced inter-agent constraint on SAs can be represented very simply by the CA and are also simple to calculate, as partial limits on the number of nurses in a certain group in each of the shifts.

References

[1] Amnon Meisels and Andrea Schaerf. Solving employee timetabling problems by generalized local search. *Proc. Italian AI, Bologna*, 1:281–331, May, 1999.

[2] G. Solotorevsky, E. Gudes, and A. Meisels. Modeling and solving distributed constraint satisfaction problems (dcsps), 1996. Submitted to CP96, February 1996.

[3] M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara. Distributed constraint satisfaction problem: Formalization and algorithms. *IEEE Trans. on Data and Kn. Eng.*, 10:673–685, 1998.

Relaxation of Coverage Constraints in Hospital Personnel Rostering

Patrick De Causmaecker and Greet Vanden Berghe

KaHo St.-Lieven, Information Technology,
Gebr. Desmetstraat 1, 9000 Gent, Belgium
{patdc,greetvb}@kahosl.be

Abstract. Automated personnel scheduling deals with a large number of constraints of a different nature, some of which need to be satisfied at all costs. In hospitals in particular, it is unacceptable not to fully support patient care needs. In addition to personnel coverage constraints, nurse rostering problems deal with time related constraints for all the personnel members arranging work load, free time, and personal requests.
Real-world nurse rostering problems are usually over-constrained but schedulers in hospitals manage to produce solutions anyway. In practice, coverage constraints, which are generally defined as hard constraints, are often relaxed by the head nurse or personnel manager.
The work presented in this paper builds upon a previously developed nurse rostering system that is used in several Belgian hospitals. Nearly all the data in the model are modifiable by the users. This enables a large number of very diverse teams to cover the hospital needs, while the system offers a general approach for calculating solutions. Drawbacks are the possibilities for users to define parts of their problems in such a way that they cause difficulties for the algorithms to find good quality or even feasible solutions.

We introduce a set of specific algorithms for handling and even relaxing coverage constraints, some of which were not supposed to be violated in the original personnel rostering model [4, 6, 8]. The motivation is that such practices are common in real scheduling environments. Relaxations enable the generation of better quality schedules without enlarging the search space nor the computation time.

Keywords: timetabling, nurse rostering, personnel scheduling

1 Introduction

Automating the personnel scheduling process of large organisations increases the quality of the personal schedules [2–4, 14]. Compared to manual scheduling approaches, timetabling heuristics reduce the computation time considerably [6, 8, 15].

The relaxation of coverage constraints is part of the development of a nurse rostering package developed for practical problems. *Plane* is designed to meet the

requirements of personnel rostering in Belgian hospitals. Cyclical scheduling is very unusual in these practical health care environments because the round the clock work is irregular, part time contracts are common, nurses want the freedom to choose days off and holiday periods freely, the personnel requirements fluctuate more than in most other personnel scheduling problems (such as employee scheduling in banks, shops, crew scheduling in public transport, etc).
It is the main objective of Plane to construct a schedule that respects as many soft constraints on personal schedules as possible (see Burke et al. [6]), while maintaining the personnel coverage at any time. In real rostering environments, feasible schedules are very rare and are often less attractive than under- or overstaffed schedules with lower soft constraint violations. In this paper, we present a set of flexible algorithms for increasing the quality of the solutions by relaxing appropriate hard constraints. Some of the algorithms are interactive, others cooperate with the meta-heuristics that generate the solutions. The methodology can be adapted to different practical contexts of hospitals and it can be extended towards other timetabling and scheduling approaches.

Section 2 schematically introduces the sort of problems tackled in this research. The constraints and their contribution to the quality of the schedule are explained briefly. The scheduling options and planning algorithms contributing to better satisfying the constraints on personal schedules, provided that some hard constraints can be relaxed, are explained in Section 3. Section 4 presents test results of the relaxations for real world problems. We draw conclusions on the approaches in Section 5.

2 Problem Description

Different personnel scheduling approaches can have widely varying objectives. The personnel rostering problem in this paper concerns finding a good quality schedule that satisfies the coverage constraints. The quality of a schedule is expressed by its value of the cost function, counting the violations of time related constraints.
We briefly explain the terminology used in this nurse rostering approach:

- **shift type** A shift type is a personnel task with a fixed start and end time. Unlike in other organisations, hospitals require more shift types than the regular early, late, and night shift. The number of different shift types is T. In the representation of the problem, we define a assignment unit t for each shift type, whether or not they represent overlapping shift types. For more details about the representation we refer to [6].
- **skill category** Personnel members in a ward belong to skill categories. The number of different skill categories in the model is Q. The categories are based upon the level of qualification, the experience, and the responsibility of the nurses. Examples of skill categories are: head nurse, regular nurse, junior nurse, caretaker, etc. When personnel members are allowed to replace people belonging to other skill categories, they are said to have alternative skills.

This allows for a more complex replacement model than the hierarchical skill categories.

- **work regulation** Hospital personnel have work regulations or contracts with their employer. Most healthcare organisations allow several job descriptions such as part time work, night nurses, weekend workers, etc. Each of these regulations is subject to a different set of constraints.
- **schedule** A two-dimensional structure S in which the rows p (1 ... P) represent the timetables for all the personnel members. The columns denote the assignment units and there is one assignment unit per shift type and per day t (1 ... T), where $T = S * D$ and D is the number of days in the planning period.

We define the coverage constraints as the required number, minimum or preferred, of people of each skill category at any time (or shift) in the planning period. They belong to the category of hard constraints so they must always be satisfied, i.e. at least the minimum number of personnel and at most the preferred number should be scheduled at any time.
All the time related constraints on personal schedules are soft constraints. They will preferably be satisfied but violations can be accepted to a certain extent. Examples of such constraints are overtime, undertime, maximum number of assignments per week, minimum number of consecutive working days and free days, personal requests for days or shifts off, personal requests for working a specific shift on a particular day, cyclical patterns, etc. An extended list of such constraints can be found at the web page [19]. In our approach, planners can set the cost parameters of the soft constraints and even modify the constraint definition. The modular cost function sums all the violations of soft constraints multiplied by this cost parameter. For details about the evaluation function, we refer to [6].
In Fig. 1, we briefly introduce a few examples of soft constraints which, in our approach, play an important role in the relaxation of hard constraints. The set contains mainly constraints for which the point in time is relevant, such as special personal requests for specific days. Although these constraints are soft, it might be unavoidable to violate them in the feasible part of the search space. Their dominant influence on the satisfaction of coverage constraints, and vice versa, makes it worthwhile pre-evaluating them. Other constraints for evaluating maximum values or consecutiveness cannot be easily pre-evaluated. The constraints on overtime and undertime are exceptional. They only determine one of the post planning options (see 'add hours' in Fig. 2).
When comparing a set of nurse rostering approaches in literature, we can distinguish two main objectives: the coverage constraints and the time related constraints on personal schedules. Some models allow the scheduling algorithms to make coverage decisions by incorporating them as soft constraints [9, 15–17, 21]. Other approaches consider a fixed number of staff, which is supposed to be sufficient for satisfying the coverage constraints [2, 4, 6, 8, 11, 13, 14, 21]. The approaches which require a strict application of the time related constraints are rare and solve smaller size problems than those tackled in this paper. Berrada

- **day off** This constraint is a personal request for a day off. Personnel members can have a list of such requests.
- **shift off** Unlike the previous constraint, this one only forbids the assignment of a particular shift type on a certain day.
- **assignment** Nurses can list their personal requests for performing certain shifts on particular days. For each of the previous constraints, a high or low priority can be set. It determines the penalty for violating the constraint.
- **pattern** A pattern is a very complex cyclical constraint which is built with a combination of different pattern types. Each pattern has a predefined length, equal to a number of days. Per pattern day, one of the following restrictions hold:
  - PAT–1 no free day (at least one shift type must be assigned)
  - PAT–2 assignment of a particular shift type
  - PAT–3 assignment of a shift type of a certain duration (a small deviation of that duration, generally 15 minutes, is allowed)
  - PAT–4 no restriction on that day
  - PAT–5 free day
  - PAT–6 day off
  - PAT–7 forbidden shift types

  Although both PAT-5 and PAT-6 do not allow assignments on the corresponding days, there is a difference in interpretation. PAT-6 represents a day off which can influence the evaluation of some other constraints (e.g. when it is a compensation day, holiday, refresher course, etc) by contributing to the working hours.
- **overtime** According to their work agreement or contract, personnel members have a different maximum working time per planning period. The constraint is cumulative in that access hours in the previous planning period are added to the prestation in the current period for the evaluation.
- **undertime** Similarly, a minimum number of working hours can be defined per work agreement. Schedules with fewer hours assigned are penalised. However, holiday periods and illness leave will induce a recalculation of the minimum number of hours a person should work. Note that undertime is not related to undercoverage, which is a shortage in personnel.

Fig. 1. Set of time related soft constraints that can easily be pre-evaluated

et al. [3], for example, define less time related constraints than we do, but some of them are hard (e.g. weekend working patterns and the number of weekly working days). Miller et al. [16] define the personnel requirements as a minimum and preferred number of people per day (and not per shift type, as it is in our approach). They divide the time related constraints into two categories: a feasibility set (maximum number of assignments, minimum/maximum number of consecutive working days) and non-binding constraints (examples are: number of working weekends, working complete weekends, consecutive free days, in addition to stricter formulations of the constraints in the feasibility set). The total number of constraints is very low compared to the set used in this research.

In the literature, there is a range of different ways to tackle coverage constraints. Isken and Hancock [12] allow variable starting times for the personnel members instead of working strictly defined shifts. Over- and understaffing can occur in solutions, but they are penalised. The method proposed by Ozkarahan [18] aims at minimising over- and understaffing. Although there is a defined range of feasible personnel attendance, solutions generated by Miller et al. [16] do not necessarily satisfy all the coverage constraints. Warner [20] allows to schedule more nurses than strictly required in order to compensate for unwanted personal schedules.
The goal function in Warner and Prawda's work [21] aims at minimising the difference between a given lower limit for the number of nurses, and their actual number. The difference must not be under zero. Ahmad et al. [1] only search solutions in the feasible domain, which includes some time related constraints in addition to personnel coverage. Scheduling too many personnel members is not penalised. Meyer auf'm Hofe [15] defines minimum and standard staffing levels which are treated as fuzzy constraints, there is a considerably larger penalty for understaffing than for overstaffing.

The observation of the behaviour of manual planners revealed that certain circumstances can permit the relaxation of some hard constraints. Undertime is often considered as a worse violation than overtime, whereas overcoverage is generally less wanted than understaffing, etc. It is not possible to incorporate this knowledge in the problem description because it reflects implicit dissatisfaction rather than countable violations. Moreover, it differs strongly from one hospital to another. Therefore, it is important to provide interactive scheduling relaxation tools.

3 Algorithms for Relaxing the Hard Constraints

3.1 Overview of the algorithms

The relaxation algorithms are pre- and post-planning heuristics, which can be combined with previously described meta-heuristic approaches. Examples of such meta-heuristics are variable neighbourhood search [7], tabu search [8], and memetic algorithms [4], in which several hybridisations can be selected.

The relaxation heuristics are presented as separate planning options, in order to isolate the meta-heuristics from typical shortcomings of the model when applied in practice. Fig. 2 schematically demonstrates the order in which the relaxation heuristics (presented in bold) appear in the total planning process.

Certain circumstances require a reduced search space, as is schematically presented in the box between the consistency check and the initialisation. It is the case when a previously generated schedule can or must be partly re-used. The large box 'per skill category' denotes that the nurse rostering problem in our approach can be divided into sub problems which are related to the skill categories. Planners define a hierarchy for the skill categories which determines their planning order. The large solution space is normally split into smaller regions which are not necessarily completely disjunctive (see Section 2). The meta-heuristics are not specified in this figure because any search algorithm maintaining the coverage is allowed. Boxes presented on the same level denote exclusive options, e.g. 'accept' and 'repair' when the consistency check discovers infeasibilities. The 'post planning' options start after finishing the meta-heuristics but they can require re-iterating the meta-heuristics. Additional assignments made during these options will be 'marked' in order to highlight them for later possible search actions.

3.2 Consistency check on available people

When it happens that on one or more days of the planning period the hard constraints are too strong for there to exist any feasible solution, hospital planners can opt to relax them manually. In most cases, the hard constraints are so strong that it is obvious, after a preliminary check, that some of the soft constraints cannot be satisfied. The planning meta-heuristics are not developed for handling infeasible problems and therefore, the outcome of the consistency check will influence the initialisation of the meta-heuristics.
In order not to expect too much insight from the hospital planners who set up the data, a simple consistency check is performed before the planning starts. In this pre-planning process, the number of available people of a certain skill category is compared with the number of requested people for that skill category at any time during the planning period.
Apart from an obvious check on the hard constraints, users in practice demanded an extra check on some 'precedence' soft constraints, namely on patterns, personal requests for days and shifts off, and requested shift assignments (see Fig. 1). The consistency check respects the planning order of the skill categories. For every personnel member, a list of available time slots (shift types or time intervals) is constructed. Time slots that are not available when satisfying these soft constraints are presented in Fig. 3. The personal requests for assignments and the requested shifts in patterns should be satisfied. Fig. 4 presents a list of these necessary assignments.
The term 'requirements' will be used instead of the more specific options minimum or preferred personnel requirements. We explain in Section 3.4 how the

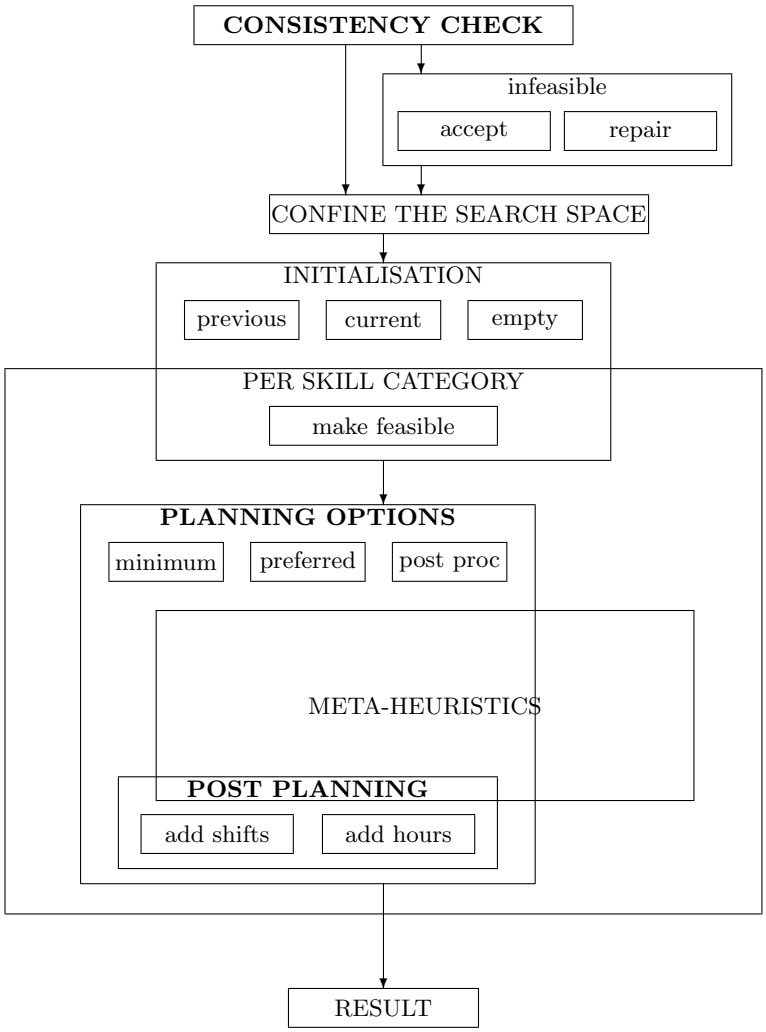

Fig. 2. Overview of the relaxation algorithms which are applied in combination with the meta-heuristics

- the days off in personal schedules
- the assignment units corresponding to shifts off in personal schedules
- the assignment units corresponding to other than requested assignments on the same day in personal schedules
- in case a pattern is defined in a person's work regulation:
  - the assignment units which do not correspond to the specified shift type in case of PAT-2
  - the assignment units corresponding with shift types having a different than the specified duration in case of PAT-3
  - the entire day corresponding to PAT-5 and PAT-6
  - the assignment units which correspond to the specified shift types in case of PAT-7

Fig. 3. Set SE: Schedule positions which should remain without assignments, according to the list of soft constraints presented in Fig. 1

- the assignment units corresponding to personal requests for the assignment of shifts
- in case a pattern is defined in a person's work regulation:
  - one of the assignment units corresponding to the day in case of PAT-1
  - the assignment units corresponding to the specified shift type in case of PAT-2
  - one of the assignment units corresponding with shift types having the specified duration in case of PAT-3

Fig. 4. Set SA: Schedule positions which should have assignments, according to the list of soft constraints presented in Fig. 1

hard coverage constraints for the scheduling algorithm are derived from the personnel requirements and the planning options.

The entire consistency check procedure is presented schematically in Fig. 5. The variable *occupied* has been introduced in the algorithm to keep track of the areas in the schedule which are not free for assignments. All the free days in patterns (type PAT-5), for example, render the corresponding days in the schedule occupied. When estimating the number of assignments in a schedule, the algorithm starts with calculating all the personal requests for particular shifts. Patterns, in which a type PAT-2 for the shift type corresponding with the day, are also added to the assignable schedule positions. In all these cases, we set $schedule_{\{p,t\}}$ equal to *occupied*, for all the assignment units t of the day in person p's schedule on which the imaginary assignment was made. If, for a certain assignment

$\forall q\ (1 \leq q \leq Q) : \forall p, t\ (1 \leq p \leq P, 1 \leq t \leq T) :$

$IF\ \{p,t\} \in SE(seeFig.\ 3)$
$schedule_{\{p,t\}} = `occupied'$
$IF\ \{p,t\} \in SA(seeFig.\ 4)$
$schedule_{\{p,t\}} = `assignable'$
stop = false
WHILE ($assignable_{q,t} < required_{q,t} \wedge$!stop)
search, in the following order, an unoccupied and not yet assignable schedule position for a new assignment
- p with skill category q, PAT-1 type for the day, and not yet assignable for t
- p with skill category q and not yet assignable for t
- p with alternative skill category q, PAT-1 type for the day, and not yet assignable for t
- p with alternative skill category q and not yet assignable for t

IF (found)
$schedule_{\{p,(t/S)*S\}} = schedule_{\{p,(t/S)*S+S\}} = `occupied'$
$schedule_{\{p,t\}} = `assignable'$
ELSE
stop = true
$inconsistent_{q,t} = assignable_{q,t} - required_{q,t}$

RESULT

IF ($\sum_{q,t} |inconsistent_{q,t}| > 0$)
inconsistent = true
ELSE
inconsistent = false

WHERE $assignable_{q,t} = \sum |\{p\ \xi\ 1 \leq p \leq P \wedge p_q \wedge schedule_{\{p,t\}} = assigned\}|$ and p_q reflects that p can carry out work for skill class q

Fig. 5. Procedure for Consistency Check

unit t, the assignments exceed the requirements for qualification q, the value $inconsistent_{q,t}$ is set equal to the excess.
In the other case, the algorithm searches positions in the schedule where extra assignments can be made. For all the assignment units on which the assignments are less than the requirements, we try to add shifts to unoccupied schedule positions with a pattern type PAT-1, or PAT-2 if it corresponds to the duration of the shift type. All the assignments in this step will make the schedule positions on the corresponding days occupied. When there is still an excess of requirements, the algorithm continues by assigning the shifts to personal schedules which are still unoccupied on the particular day. If there are not enough such locations, the value $inconsistent_{q,t}$ equals the number of assignments minus the number of requirements.
Blocking the entire day on which an assignment is made is often more strict than necessary. However, in real-world situations, it rarely occurs that a personnel member is assigned to more than one shift per day in healthcare environments. We found it better to mark such problems as inconsistent. Manual planners are free to accept or ignore the diagnosis. The more flexibility the algorithm allows for doing the imaginary assignments, the more accurate the diagnosis will be. In our model, alternative skill categories are also taken into account. Assignments for alternative skill categories can often help to satisfy the pattern constraint (PAT-1, PAT-2, and PAT-3) when the problem is inconsistent for the main skill category. The scenarios are equivalent for all the possible personnel requirement formulations, minimum or preferred requirements.
Since the system does not handle infeasible problems, the algorithm always reduces the personnel requirements if an unavoidable violation is detected. At the start of a planning, the algorithm informs the planner about inconsistencies in the coverage constraints. The values are 0 when the requirements cause no inconsistency. A value larger than 0 indicates that the requirements should be increased by that value, a negative number requires a corresponding decrease. Either the user can accept the remedy by relaxing the personnel requirements or he can deliberately opt for unavoidable violations of the checked soft constraints. This option determines whether the search algorithms will respect the original or the relaxed coverage constraints. Both options are mentioned in the overview of Fig. 2.

3.3 Initialisation

The initialisation of the scheduling algorithm consists of constructing a feasible initial solution. It suffices for a schedule to satisfy all the hard constraints (or the relaxed hard constraints proposed by the consistency check) to be called feasible. We have split the initialisation in 2 phases. The first phase loads the input (after an input option has been selected). The other phase makes the schedule feasible.

Input options for the initialisation For practical planning problems three possible strategies are introduced.

Current schedule The option starts from the currently available schedule, which can either be the result of a previous attempt to generate a solution, or the planning which existed before certain extra restrictions occurred.
Schedule of the previous planning period This option is useful when the schedule in the previous planning period is of a very high quality and when the constraints for the current and the previous planning period are similar. It is not recommended to use the previous schedule as an input when the number of personnel is not the same in both periods, when the planning periods includes bank holidays, or when the pattern constraint has a different period from the planning period, for some personnel members.
Empty schedule The simplest input option starts the initialisation from an empty schedule.

Although the two first initial schedule constructors may seem very attractive, our experiments show that it is not too difficult for the meta-heuristic algorithms to produce schedules of comparable quality starting from a random initial schedule. Indeed it is often the case that with the two latter initialisations, the solution represents a local minimum in the search space and the algorithm has problems escaping from it.

Create a feasible solution In order to satisfy the hard constraints in the initial schedule, an algorithm is applied which adds and/or removes shifts until the personnel requirements (according to the planning options of Section 3.4) are met. The process is mainly random driven, although it takes some of the soft constraints into account. Users of the system suggested to initially force satisfaction of the personal requests for days and shifts off and satisfaction of the patterns. These are precisely the soft constraints which play a role in determining the consistency of the data (Section 3.2). It seems contradictory to the previously explained concept of treating the cost function (which sums violations of all the constraints [6]) as the only evaluation means. However, even though the planners can freely set cost parameters, a schedule in which less personal constraints are violated will often be preferred to better quality schedules (with respect to the cost function) which contain more violations of these particular constraints.
The initialisation is always executed per skill category. The group of people belonging to the skill category is extended with the people who have that skill as an alternative qualification. The initialisation algorithm makes a first attempt, only in the case when the *Empty schedule* option is selected, to satisfy the pattern constraint. When a pattern requires a shift, the algorithm assigns a randomly chosen shift to the corresponding people, provided the assignment never violates the hard constraint on the number of people required. In order to be general, we use the simplified notation *requirements* and do not specify whether it is minimum or preferred requirements.
In case the shift type is specified (building block PAT-2 of the pattern), the shift will be assigned to the person if there is a shortage in the schedule for the shift on the particular pattern day. For the 3rd building block, PAT-3, the

algorithm randomly chooses a shift type of the specified duration (allowing the small deviation of the preset time), for which the personnel requirements are not yet met.
The algorithm afterwards moves to an iterative phase which stops when the personnel requirements are fulfilled for every shift type or when a maximum number of attempts to assign randomly is reached. The maximum number of attempts is function of the number of people authorised to carry out jobs for the skill category. In case the number of scheduled personnel on a particular day is too low to meet the requirements, the system will assign the lacking shift type randomly to a personal schedule, provided that person is skilled and available. The people are divided into groups with equal eagerness for the assignment. In the case where all people in a group already have an assignment for the shift (either for the currently scheduled skill category or for another), the assignment fails. The algorithm stops if a random assignment is possible in a group, that is if a person of the group has no assignment. If no such person exists, the algorithm moves to the next group. The groups of appropriate people considered for adding assignments are listed below in order.

ADD–1 All the people having a personal request for the shift corresponding to the time to be scheduled and no assignment yet.

ADD–2 All the people working according to a predefined pattern, for which a PAT-2 type corresponds to the day and the detail to the shift to be scheduled, with an empty schedule for that assignment unit.

ADD–3 All the people working according to a predefined pattern, for which a PAT-3 type corresponds to the day and the detail to the duration of the shift to be scheduled (+/- the deviation), and an empty schedule for the corresponding assignment unit.

ADD–4 All the people who belong to the scheduled skill category (main skill) and do not have a personal request (with a high importance) for a day off or shift off at the time to be scheduled, and have an empty schedule at that time.

ADD–5 All the people who have the skill category as alternative and do not have a personal request (with a high importance) for a day off or shift off at the time to be scheduled, and have an empty schedule at that time.

ADD–6 All the people who belong to the skill category, have an assignment for another skill category, which has not been scheduled in this run (a skill category which is lower in the planning order hierarchy).

ADD–7 All the people who have the skill category as alternative, have an assignment for another skill category, which has not been scheduled in this run (a skill category which is lower in the planning order hierarchy) and which is different from the main skill category of the particular people.

ADD–8 All the people who are authorised for the skill category and have an empty schedule for the shift.

An analogous procedure has been developed for removing shifts when a schedule exceeds the requirements for certain shifts. The hierarchy of the groups to which the removal of a shift is applied is listed below:

- REM–1 All the people who have the skill category as alternative and have a marked (see Section 3.4) assignment for the skill category being scheduled.
- REM–2 All the people who belong to the skill category and have a marked assignment for it.
- REM–3 All the people who have the skill category as alternative and have an assignment for the skill category being scheduled.
- REM–4 All the people who belong to the skill category and have an assignment for it.

In Section 3.4, we explain how the personnel requirements for the initialisation depend on the planning options.

3.4 Coverage Options

In practice, the number of required personnel on a certain day is not completely strict. Experienced planners know very well when it is reasonable to plan more or less personnel than required. However, there are no clear rules for decisions like this and in our model the user can optionally choose among different coverage strategies. The approach allows a feasible range between minimum and preferred personnel requirements. Any solution with fewer assignments than the minimum requirements, or with more than the maximum requirements, violates the hard constraints.
The system provides planning options to set the coverage constraints, which will be hard constraints for the rostering algorithms. It is also possible to allow a few post-planning algorithms, which can change the coverage after a solution has been generated.

Minimum - Preferred requirements The hospital scheduler can choose to plan the minimum personnel requirements or the preferred requirements as hard constraints. During the entire meta-heuristic planning process, the number of planned shifts (be it minimum or preferred) will not change.

Plan towards preferred requirements Instead of strictly setting the hard constraints, this option allows to work with a range in which the hard constraints are considered satisfied. In Fig. 6, the procedure for this option is schematically presented. The algorithm first takes the minimum requirements as hard constraints. After a result has been calculated by the scheduling meta-heuristics, the system tries to add shifts to the schedule wherever this does not involve an extra violation of soft constraints. For every day on which there is a difference between what is actually scheduled and the preferred requirements, the

$\forall q\ (1 \leq q \leq Q)$:

initialise a solution $schedule_q$ satisfying the minimum personnel requirements (whether consistent or not) for q
improve the solution $schedule_q$ by applying meta-heuristics until a stop criterion is met
$quality(schedule_q) = \sum_{p\ \xi\ p_q=q} quality(schedule_{\{p,q\}})$

$\forall q\ (1 \leq q \leq Q)$:

stop = false
WHILE $(assigned_q < preferred_q \wedge !stop)$
select *best* (the least worsening with respect to the quality) assignment position $schedule_{\{p,t\}}$ (or 'twins' $schedule_{\{p,t\}} + schedule_{\{p,t+S\}}$) for which the schedule is currently unassigned and for which extra assignments would not exceed the preferred personnel coverage
IF $(best \leq quality(schedule_q) \vee best_p \leq threshold_{shifts})$
$schedule_{\{p,t\}} = schedule_{\{p,t+S\}} = assigned$
$assigned_{q,t} = assigned_{q,t} + 1$
ELSE
stop = true

WITH

$assigned_q = \sum_{t,p\ \xi\ p_q=q} schedule_{\{p,t\}} = assigned$
$assigned_{q,t} = \sum_{p\ \xi\ p_q=q} schedule_{\{p,t\}} = assigned$

Fig. 6. Procedure for adding shift types after planning

system adds a shift to the personal schedule that improves the overall quality of the schedule most. Adding a pair of identical shifts on consecutive days in a personal schedule (called 'twins') is often less harmful than adding an isolated shift. In the competition for the best candidates to assign extra shifts to, 'twin assignments' are also considered (provided none of the assigned shifts causes an excess with respect to the preferred personnel requirements of the schedule). These 'twin' assignments are often more beneficial for the quality of the schedule because they influence many soft constraints (such as constraints on complete weekends, identical shifts in weekends, a minimum number of consecutive shifts or days, etc see [19]). Since the complexity of finding optimal 'twins' to add to the schedule is exponential, we reduced the search to the selection of the best set of equal shift types on two consecutive days for a personal schedule.
The system even allows for a more flexible approach by providing a threshold value for the individual cost function value: $threshold_{cost}$. In this case, the algorithm will add extra shifts, whenever the personal cost function value does not exceed that threshold.

Every additional assignment to the schedule made after the planning algorithm has stopped, will be marked. The location of such marked shifts in the schedule is the result of a post planning algorithm, while the other shift types have been assigned by search algorithms looking at the entire search space. It is recommended during some planning activities to remember which shift removals will harm the schedule less. We explained the importance of marking in Section 3.3.

Add hours This option attempts to add shifts to personal schedules with undertime. It is a pure post-planning option which does not necessarily respect the (hard) preferred personnel coverage constraints. Once a schedule has been calculated, an algorithm searches, for every personal schedule, the best point in time to add extra shifts. The constraint holds that a shift cannot be added unless such shift already occurs in the personnel requirements for that day and skill category. By applying this option, the coverage can exceed the level of the preferred requirements unless when the level is 0.
By default, nothing will happen if it would generate overtime. As explained in the previous option, there is a possibility for setting a parameter $threshold_{hours}$ for the cost function value of the personal schedule. When a personal schedule has reached this number, the algorithm will not add extra shifts. Just like in the previous section the extra shifts are marked. Fig. 7 schematically demonstrates the procedure.

$\forall q\ (1 \leq q \leq Q)$:

identical to Fig. 6

$\forall q\ (1 \leq q \leq Q)$:

$\forall p\ (1 \leq p \leq P)$: FOR each personnel member p belonging to q
stop = false
WHILE ($undertime_p > 0 \wedge !stop$)
select *best* (the least worsening with respect to the quality) assignment position $schedule_{\{p,t\}}$ (or twin $schedule_{\{p,t\}} + schedule_{\{p,t+S\}}$) for which the preferred personnel requirements (at t and t+S) are not 0 and for which ($best_p < threshold_{hours}$)
IF (found)
$schedule_{\{p,t\}} = schedule_{\{p,t+S\}} = assigned$
ELSE
stop = true

WITH $undertime_p$ = the difference between the number of hours p should work and the number of hours that is scheduled for p

Fig. 7. Procedure for adding hours after planning

Since it might be better in terms of the cost function to add a pair of shift types on consecutive days, we also consider 'twins' when searching for the best time to assign extra shifts. As explained in Section 3.4, we restrict multiple assignments to 'twins' in order to keep the search space small.

4 Test Results

We have tested the relaxation algorithms on a few real-world problems. All the experiments make use of the hybrid tabu search approach introduced in [8]. It consists of an algorithm that moves single shifts while maintaining the personnel coverage, combined with greedier algorithms concentrating on weekend constraints and improving the worst personal schedule.

In Problem I, there is a cyclic pattern for every personnel member. Some of the people have loose patterns, for others the pattern constraints are rather strict. We have tested different paths in the planning procedure (see Fig. 2). The results are presented in Table 1. Problem II is a completely different real world problem, with fewer strict patterns but more people on leave. The same tests have been carried out and the results are shown in Table 2. For each experiment, the planning option is presented in the first column. We distinguish, according to Section 3.4, 'minimum' and 'preferred' coverage, 'shifts' for planning towards the preferred requirements, and 'hours' for adding hours. The columns '$< M$' and '$> P$' refer to the shortage or excess in coverage. Violations of the hard coverage constraints are presented in bold. When the planning option 'minimum' is selected, we call excesses of personnel assignments (compared to the minimum requirements) also violations of the coverage. The same holds for a shortage of assignments when 'preferred' is selected. We have carried out experiments with two different cost functions. In the first one, undertime does not contribute to the value of the cost function (results are in the upper part of the tables). The second cost function has a penalty of 1 per hour undertime (lower part of the tables).
All the examples have inconsistent requirements with respect to the hard constraints and precedence soft constraints of Fig. 1. Most experiments in which the 'repair' option is selected after inconsistencies were discovered, produce better quality results (with respect to the cost function) than the schedules generated with the original hard constraints. This is especially remarkable for the experiments with Problem II, in which many people take days off. However, the solutions for the original requirements (accept option), do not have violations of hard constraints. Exceptions to this finding are the tests with the option 'preferred' in Table 1. In these cases, 'repair' leads to higher values of the cost function because many other than the soft constraints checked by the consistency procedure are violated due to the increased workload. The option of planning towards the preferred requirements (shifts in the first column) does not violate hard constraints (unless combined with 'repair'). Moreover, it approaches the satisfaction of the preferred personnel coverage better than the option 'minimum' and it produces

Options	**Threshold**	**Consistent**	**Quality**	**Coverage**			
				<M	>M	<P	**>P**
no penalty for undertime							
minimum		accept	790	0	0	19-	0
minimum		repair	625	0	**7+**	16-	**4+**
shifts		accept	608	0	6+	13-	0
shifts	100	accept	608	0	6+	13-	0
shifts	200	accept	608	0	6+	13-	0
shifts		repair	550	0	9+	14-	4+
hours		accept	849	0	20+	0	1+
hours		repair	709	0	13+	13-	7+
preferred		accept	938	0	19+	0	0
preferred		repair	947	0	23+	0	**4+**
penalty 1 per hour undertime							
minimum		accept	1457	0	0	19-	0
minimum		repair	1249	0	**7+**	16-	**4+**
shifts		accept	1209	0	6+	13-	0
shifts	200	accept	1209	0	12+	7-	0
shifts		repair	1138	0	16+	7-	**4+**
shifts	200	repair	1138	0	16+	7-	**4+**
hours		accept	1298	0	24+	0	**5+**
hours	20	accept	1298	0	24+	0	**5+**
hours	50	accept	1305	0	25+	0	**6+**
hours	100	accept	1348	0	19+	0	**7+**
hours	200	accept	1348	0	20+	0	0
hours		repair	1178	0	18+	11-	**10+**
hours	200	repair	1249	0	7+	16-	**4+**
preferred		accept	938	0	19+	0	0
preferred		repair	1291	0	23+	0	**4+**

Table 1. Test Results for Problem I

better quality schedules in all cases. It is the default option for most users in practice. It is very difficult to evaluate the results for the option 'hours' without knowing the subjective wishes of the personnel and the hospital. The schedules obtained with this option all have many violations of soft constraints. Satisfying the constraint on undertime better induces violations of other soft constraints. We have carried out a few tests with a threshold value for the cost function value per person. For the 'shift' option, only one result was influenced by increasing the threshold (Problem I, threshold 200: more shifts are assigned but the value of the cost function remains the same). Higher thresholds have not been considered because they would deteriorate the quality too much. Setting a threshold value for adding hours has a bigger effect than adding shifts towards the preferred requirements. The results in both test sets suffer from undertime,

Options	**Threshold**	**Consistent**	**Quality**	**Coverage**			
				<M	>M	<P	**>P**
no penalty for undertime							
minimum		accept	311	0	0	18-	0
minimum		repair	85	**4-**	**1+**	21-	0
shifts		accept	261	0	2+	16-	0
shifts	100	accept	261	0	2+	16-	0
shifts		repair	85	**4-**	1+	13-	0
hours		accept	824	0	18+	0	0
hours		repair	85	**4-**	3+	21-	**2+**
preferred		accept	836	0	19+	0	0
preferred		repair	373	**4-**	15+	**7-**	0
penalty 1 per hour undertime							
minimum		accept	379	**4-**	1+	21-	0
minimum		repair	199	0	0	18-	0
shifts		accept	310	0	4+	14-	0
shifts		repair	168	**4-**	5+	17-	0
hours		accept	878	0	18+	0	0
hours		repair	180	**4-**	4+	21-	**3+**
preferred		accept	923	0	18+	0	0
preferred		repair	388	**5-**	15+	**8-**	0

Table 2. Test Results for Problem II

as can be noticed when comparing the results with and without a penalty for undertime.

Relaxing the hard constraints does not necessarily lead to better quality schedules with respect to the evaluation algorithm although in most cases it does. It is clear that none of the relaxation algorithms outperforms all the others. They all aim at different goals, and their result strongly depends on the particular problem.

5 Conclusions

The nurse rostering system developed for application in Belgian hospitals is very flexible. It provides many possibilities for setting constraints and requirements. However, the main drawback of allowing the users to define and set their problem is the danger of creating over-constrained and unsolvable problems. Experienced hospital planners can deal with infeasibilities in practice. They know which constraints to relax in difficult circumstances.
By observing the reaction of hospital planners on the results of the scheduling algorithms for nurse rostering, we developed algorithms for addressing over-constrained and even infeasible problems, without drastically changing the objective of giving high priority to coverage constraints. The consistency check algorithm not only considers the hard constraints, but pre-evaluates a few of the

most touchy soft constraints, such as personal preferences for days off and for assignments. This interactive tool allows the users to adapt their requirements to the given recommendations for creating feasible problems.
The other relaxation algorithms are pre- or post-scheduling algorithms which do not necessarily interfere with the meta-heuristic approaches. Depending on the goal chosen by the user (to aim at meeting the minimum or the preferred coverage), these algorithms adapt the overall objective before or after generating the schedule. Adding more hours to some personal schedules is a typical example of a real-world habit which is not revealed when looking at the formulation of the requirements. We developed interactive ways of adapting the schedule to the soft constraint on 'undertime' while ignoring the hard constraint on coverage to a controlled extent.
Experiments have pointed out that relaxing the hard constraints lead to a much higher overall satisfaction for the personnel members. Many infeasible, but not unwanted, solutions are within reach of the modified goals. The interactive software of the proposed model provides enough feedback to guide the planners and it allows them to specify their own solution strategy. All the relaxation algorithms described in this paper are applied in practice. They have considerably increased the applicability of our nurse rostering model to various kinds of hospital settings with varying requirements and scheduling habits.
Although the effect of the relaxation procedures is not necessarily clearly visible in quality in terms of the cost function value, they have dramatically contributed to modelling specific wishes and rostering customs.

References

1. J. Ahmad, M. Yamamoto, A. Ohuchi: Evolutionary Algorithms for Nurse Scheduling Problem, Proceedings of the 2000 Congress on Evolutionary Computation, CEC00, San Diego, ISBN 0-7803-6375-2, 2000, 196-203
2. U. Aickelin, K. Dowsland: Exploiting problem structure in a genetic algorithm approach to a nurse rostering problem, Journal of Scheduling, Volume 3 Issue 3, 2000, 139-153
3. I. Berrada, J. Ferland, P. Michelon: A Multi-Objective Approach to Nurse Scheduling with both Hard and Soft Constraints, Socio-Economic Planning Science 30, 1996, 183-193
4. E.K. Burke, P. Cowling, P. De Causmaecker, G. Vanden Berghe: A Memetic Approach to the Nurse Rostering Problem, Applied Intelligence special issue on Simulated Evolution and Learning, Vol. 15, Number 3, Springer, 2001, 199-214
5. E.K. Burke, P. De Causmaecker, S. Petrovic, G. Vanden Berghe: Floating Personnel Requirements in a Shift Based Timetable, working paper KaHo Sint-Lieven, 2001
6. E.K. Burke, P. De Causmaecker, S. Petrovic, G. Vanden Berghe: Fitness Evaluation for Nurse Scheduling Problems, Proceedings of Congress on Evolutionary Computation, CEC2001, Seoul, IEEE Press, 2001, 1139-1146
7. E.K. Burke, P. De Causmaecker, S. Petrovic, G. Vanden Berghe: Variable Neighbourhood Search for Nurse Rostering Problems, Proceedings of 4th Metaheuristics International Conference, MIC2001, Porto, 2001, 755-760

8. E.K. Burke, P. De Causmaecker, G. Vanden Berghe: A Hybrid Tabu Search Algorithm for the Nurse Rostering Problem, B. McKay et al. (Eds.): Simulated Evolution and Learning, 1998, Lecture Notes in Artificial Intelligence, Vol. 1585, Springer, 1999, 187-194
9. J.-G. Chen, T. Yeung: Hybrid Expert System Approach to Nurse Scheduling, Computers in Nursing, 1993, 183-192
10. M. Chiarandini, A. Schaerf, F. Tiozzo: Solving Employee Timetabling Problems with Flexible Workload using Tabu Search, E.K. Burke, W. Erben (Eds.): Proceedings of the 3rd international conference on the Practice and Theory of Automated Timetabling, ISBN 3-00-003866-3, 2000, 298-302
11. K. Dowsland: Nurse scheduling with Tabu Search and Strategic Oscillation. European Journal of Operations Research 106, 1998, 393-407
12. M. Isken, W. Hancock: A Heuristic Approach to Nurse Scheduling in Hospital Units with Non-Stationary, Urgent Demand, and a Fixed Staff Size, Journal of the Society for Health Systems, Vol. 2, No. 2, 1990, 24-41
13. H. Kawanaka, K. Yamamoto, T. Yoshikawa, T. Shinogi, S. Tsuruoka: Genetic Algorithm with the Constraints for Nurse Scheduling Problem, Proceedings of Congress on Evolutionary Computation, Seoul, IEEE Press, 2001, 1123-1130
14. A. Meisels, E. Gudes, G. Solotorevski: Employee Timetabling, Constraint Networks and Knowledge-Based Rules: A Mixed Approach, E.K. Burke, P. Ross (Eds.): Practice and Theory of Automated Timetabling, First International Conference Edinburgh, Springer, 1995, 93-105
15. H. Meyer auf'm Hofe: Solving Rostering Tasks as Constraint Optimization, E.K. Burke, W. Erben (Eds.): Practice and Theory of Automated Timetabling, Third International Conference, Konstanz, Springer, 2000, 191-212
16. M.L. Miller, W. Pierskalla, G. Rath: Nurse Scheduling Using Mathematical Programming. Operations Research 24, 1976, 857-870
17. M. Okada: An approach to the Generalised Nurse Scheduling Problem - Generation of a Declarative Program to represent Institution-Specific Knowledge. Computers and Biomedical Research 25, 1992, 417-434
18. I. Ozkarahan: A Disaggregation Model of a Flexible Nurse Scheduling Support System, Socio-Economical Planning Science, Vol. 25, No. 1, 1991, 9-26
19. G. Vanden Berghe: Soft constraints in the Nurse Rostering Problem, http://www.cs.nott.ac.uk/ gvb/constraints.ps, 2001
20. M. Warner: Scheduling Nursing Personnel According to Nursing Preference: A Mathematical Programming Approach. Operations Research 24, 1976, 842-856
21. M. Warner, J. Prawda: A Mathematical Programming Model for Scheduling Nursing Personnel in a Hospital, Management Science 19, 1972, 411-422

Scheduling Doctors for Clinical Training Unit Rounds Using Tabu Optimization

Christine A. White[1] and George M. White[2]

[1] Dept. of Internal Medicine
[2] School of Information Technology and Engineering,
University of Ottawa,
Ottawa K1N 6N5, Canada
white@site.uottawa.ca

Abstract

Hospitals must be staffed 24 hours a day, 7 days a week by teams of doctors having certain combinations of skills. The construction of schedules for these doctors and the medical students who work with them is known to be a difficult NP-complete problem known as *personnel scheduling, employee timetabling, labour scheduling* or *rostering* .

At the Ottawa Hospital Clinical teaching Unit, the chief resident is responsible for casting and posting the call schedule that assigns medical staff to their overnight duty rosters. A poll of former chief residents reported that the single most unpleasant duty they faced was the creation of these schedules by hand and reaction of the staff when they were posted. It was suggested that a scheduling program might be able to provide better quality solutions faster than can be done manually.

Schedules that are are intended to be used in a real setting, have to to be solved in their entirety, without omission of anything relevent. The common practice of eliminating awkward side constraints is not permissible. Because of the difficulty of finding solutions that are acceptable to those concerned, a number of heuristic techniques have been brought to bear on these problems and results have been reported for several similar problem instances in both medical and non-medical settings. Carter and Lapierre [CL96] have listed the constraints that govern the scheduling of physicians in an emergency ward. Cowling *et al.* [CKS01] have employed a *hyperheuristic* approach to personnel scheduling by means of a mechanism to guide the specific heuristic optimization method used depending on the properties of the solution space being examined at a given time. Chan and Weil [CW01] have generated solutions to problems similar to ours using a constraint logic (CLP) approach implemented in CHIP. Harald Meyer auf'm Hofe [aH01] investigated the incorporation of fuzzy constraints and branch & bound procedures into a constraint logic approach in his generation

of solutions for nurse rostering in German hospitals. Similar approaches using constraint logic have described by Abdennadher and Schlenker [AS99a][AS99b] . Cheng *et al.* [CLW97] used what they termed *redundant modeling* within a CLP formulation to reduce search time. A genetic algorithm approach was used by Kragelund [Kra97] and the evolutionary algorithm has been studied by Jan *et al* [JYO00] .

There are many possible schedules for the rounds of medical personnel in a hospital that do not violate any "hard" constraints. *i.e.* a person cannot be scheduled to do two different tasks simultaneously. These are called *feasible* schedules. However, feasible schedules can have very pronounced differences that can lead to one being considered much better than another. Some of these schedules may be considered quite bad. A schedule that required someone to work two consecutive rounds or to work every weekend would not be well regarded. Some of the constraints are imposed by conservation laws, others by union rules and others by some cherished traditions that flourish within the unit. It seems that no two hospitals have the same set of constraints [CL96][GK00] . Basic circadian rhythms must also be considered. Costa [Cos96] and Knauth [Kna93][Kna96] summarize some useful guidelines for casting duty rosters.

In the end, the "best" schedule is the one that pleases most of the people involved most of the time.

For our purposes, medical personnel are experienced doctors, newer doctors and medical students, classified by *rank* , seniors, juniors and students, and by *unit* , team A, team B, GM-consult, Ambulatory, Float, 2nd year, and 3rd year (for seniors), team A and team B (for juniors and students). At any time of night, medical duties are supported by a senior and two teams consisting of a junior (if available) and a student (if available). The most desirable situation occurs when the unit is staffed by a senior, two juniors and two students; unfortunately this occurs rarely because of the continuing shortage of medical staff. Usually, the round is supported by a mandatory senior and either two juniors and a student, a junior and two students or two juniors with an additional Float senior. If a student is on duty without a junior from the same team, the senior's unit should be from the same team as the junior to provide additional support. Each of these possible combinations is assigned a *penalty* ranging from 0, for the best case, a senior, two juniors and two students to 300, for the worst case, a senior, no juniors and two students, accompanied by an additional Float senior. These penalties can be adjusted at run time by the operator to represent the perceived seriousness of personnel deficiencies at various times. These may vary depending on the mix of available staff.

The medical staff may take vacations at random times and are permitted to work a maximum of 7 calls in any 28 day scheduling block. This maximum number of calls may be reduced depending on the number of vacation days taken, and the rank of the person involved.

Calls should be spaced out evenly over the period and weekends should be assigned fairly to all personnel respecting a pattern that groups a Friday and a Sunday call in one weekend and a Saturday call in another. Staff must not work

two consecutive shifts and the number of patterns of two shifts in three days should be minimized. The penalty for consecutive shifts is set to 500 currently but can be adjusted at run time. Some other rules are in effect dealing with long weekends and certain other personnel matters.

Penalties can be divided into two broad classes. The composition of staff on duty during a given call is governed by rank and team considerations that are referred to as *horizontal* constraints. The penalties that are incurred because the weekends are not as they should be or because a member has to work two shifts in three days are due to *vertical* constraints. These two sets conflict directly with each other and are the chief obstacle to casting perfect schedules. Depending on the number and the mix of the medical staff on duty during a block the operator may chose to favour the vertical constraints over the horizontal constraints or vice versa. A constraint can be effectively eliminated by setting its penalty to zero.

An automatic scheduler has been developed that uses tabu search [GL97] to perform an heuristic optimization of the scheduling space in an attempt to find the least objectionable schedule. An initial call schedule is generated by bin packing that considers vacation and rank factors. Then a multiphase tabu search is performed that heuristically minimizes a penalty function by considering the seniors, juniors and students in sequence. The first of two basic moves that define the neighbourhood is a "swap", exchanging the places of two seniors, juniors or students as appropriate. As there are 5 persons on duty during a call, there are 5 neighbourhoods generated, one for the seniors, one for the juniors on team A, one for the juniors on team B, one for the students on team A and one for the students on team B. This is complicated by the requirement that when a junior is not available, the place must be taken by a student from the missing junior's team. This implies that a second junior from the same team should not be placed on the call.

A tabu minimization is performed using each of the five neighbourhoods in turn. When this is completed, the schedule is examined to discover whether the solution could be improved by *removing* one of the students. Curiously, in this environment, more is not always better. In spite of the chronic understaffing of the hospital, if there are too many students on duty and too few seniors and juniors to supervise their work, the situation is deemed worse than if there fewer students. Therefore, during this part of the algorithm, surplus students are removed from the schedule if this would reduce the total schedule penalty.

At this point, the neighbourhood is again redefined. The juniors and students from team A and the juniors and students of team B are joined to produce a larger neighbourhood that is created by redefining the *move* to be a *rotation* across two lines of the schedule. This is done for the A team followed by the B team.

Finally the original moves and neighbourhoods are restored and the tabu optimization is recycled through the five neighbourhoods until no further improvement can be found. An example of a call schedule follows.

```
                                Call Schedule

Tue: Narayan      Seidler     Holden*    Firoz*     ------
Wed: Schneider_A  Trottier*   Amhalal    ------     Dickie*
Thu: Al_Qassabi   Aggarwal*   Charlebois ------     Davidson*
Fri: Davis        Seidler     Kay        Matar*     Pinto*
Sat: Schneider_A  Stewart     Amhalal    Lund*      Davidson*
Sun: Davis        Seidler     Kay        Aggarwal*  Holden*
Mon: Al_Qassabi   Lund*       Davidson*  ------     ------
Tue: Ling_B       Abdelgader  Holden*    Matar*     ------
Wed: White        Firoz*      Kay        ------     Pinto*
Thu: Al_Qassabi   Aggarwal*   Amhalal    ------     Holden*
Fri: Ling_B       Abdelgader  Dickie*    Firoz*     ------
Sat: Al_Qassabi   Seidler     Kay        Aggarwal*  Davidson*
Sun: Ling_B       Abdelgader  Dickie*    Lund*      ------
Mon: White        Stewart     Pinto*     Trottier*  ------
Tue: Schneider_A  Firoz*      Amhalal    ------     Dickie*
Wed: Davis        Matar*      Charlebois ------     Davidson*
Thu: Narayan      Seidler     Pinto*     Trottier*  ------
Fri: Schneider_A  Lund*       Charlebois ------     Holden*
Sat: Davis        Trottier*   Dickie*    ------     ------
Sun: Schneider_A  Aggarwal*   Charlebois ------     Pinto*
Mon: White        Abdelgader  Davidson*  Matar*     ------
Tue: Narayan      Stewart     Amhalal    Aggarwal*  Dickie*
Wed: Ling_B       Seidler     Holden*    Matar*     ------
Thu: Al_Qassabi   Lund*       Dickie*    ------     ------
Fri: Narayan      Stewart     Amhalal    Trottier*  Davidson*
Sat: Ling_B       Abdelgader  Charlebois Firoz*     Pinto*
Sun: Narayan      Stewart     Amhalal    Matar*     Holden*
Mon: Davis        Trottier*   Kay        ------     Pinto*

Final penalty is: 4976
```

In this instance, there are 5 seniors, 6 juniors and 9 students. The block extends over 28 days.

Many experiments have been done to evaluate the effectiveness of certain features. Our most important conclusions are:

a) The initial solution has a great influence on the quality of the final solution.
b) Modifying the neighbourhood definition by modifying the *move* leads to improved solutions.
c) Unifying two disjoint neighbourhoods (a form of *ejection chain*) by redefining the basic move improves the solution quality.
d) A fixed tabu tenure of 40 results in good behaviour of the algorithm.

We would like to compare our algorithm with others using similar constraints and data but we have been unable to find a sufficiently equivalent problem.

A working implementation has been developed in Java using the abstract window toolkit (AWT) to build the user interface. Specially written "tabu classes" have been developed to implement evaluation functions, moves and tabu lists. The program runs on an IBM ThinkPad, model A21e, under Windows 2000. The call schedule shown above took about 1 minute to cast. The program is now being used at the Ottawa Hospital.

References

[aH01] Harald Meyer auf'm Hofe. Solving rostering tasks as constraint optimization. *Lecture Notes in Computer Science*, 2079:191–212, 2001.

[AS99a] Slim Abdennadher and Hans Schlenker. Interdip - an interactive constraint based nurse scheduler. In *Proc. of Int. Conf. on Practical Applications of Constraint Technology and Logic Programming - PACLP-99*, 1999.

[AS99b] Slim Abdennadher and Hans Schlenker. Nurse scheduling using constraint logic programming. In *Proc. of 11th Ann. Conf. on Innovative Applications of Artifical Intellence*, 1999.

[CKS01] Peter Cowling, Graham Kendall, and Eric Soubeiga. A hyperheuristic approach to scheduling a sales summit. *Lecture Notes in Computer Science*, 2079:176–190, 2001.

[CL96] Michael W. Carter and Sophie D. Lapierre. Scheduling energency room physicians. In *INFORMS Fall 1996 Meeting - Publication 99-23*, 1996.

[CLW97] B.M.W. Cheng, J.H.M. Lee, and J.C.K. Wu. A nurse rostering system using constraint programming and redundant modeling. *IEEE Trans. Inf. Technology in Biomedicine*, 1(1):44–54, 1997.

[Cos96] G. Costa. The impact of shift and night work on health. *Applied Ergonomics*, 27:9–16, 1996.

[CW01] Peter Chan and Georges Weil. Cyclical staff scheduling using constraint logic programming. *Lecture Notes in Computer Science*, 2079:159–175, 2001.

[GK00] H.N. Gabow and T. Kohno. A network-flow-based scheduler: design. performance history and experimental analysis. In *ALENEX - Algorithm Engineering and Experiments*, Jan. 7-8, 2000.

[GL97] F. Glover and M. Laguna. *Tabu search*. Kluwer Academic Publishers, 1997.

[JYO00] Ahmad Jan, Masahito Yamamoto, and Asuma Ohuchi. Evutionary algorithms for nurse scheduling problem. In *2000 Congress on Evolutionary Computing*, 2000.

[Kna93] P. Knauth. The design of shift systems. *Ergonomics*, 36:15–28, 1993.

[Kna96] P. Knauth. Design better shift systems. *Applied Ergonomics*, 27(1):39–44, 1996.

[Kra97] L.V. Kragelund. Solving a timetable problem using hybrid genetic algorithms. *Software Practice and Experience*, 27(10):1121–1134, Oct. 1997.