Timetabling for Schools:

an Exercise in Program and Data Structuring.

C.A.R. Hoare and H.C. Johnston.

~ 1978

Summary.

The purpose of this paper is to illustrate some recent theories of program and data structuring, with the aid of a realistic large example problem, - that of constructing an acceptable timetable for a school.

1. Specification of the Problem.

A school may be described in terms of the items, which constitute the school, - teachers, classes, rooms, and equipment. The set of items may be defined for a particular school by enumeration:

type Item = {Jones,Smith,...,IV,VA,VB,...,physlab,gym,...,projector,..};

The number of items will be of order 250.

Some of the items will have more than one unit; for example, there may be two physics laboratories or four projectors which can be used simultaneously. The number units of an item defines the number of possible simultaneous users of an item,and will be given by a mapping:

lives: Item → 1..NL

where NL will be typically 8. Most items will have only one life.

The school engages in a number of activities, which are to appear in the timetable: for example "form VA Latin", or "form IV physics". These activities may be defined by enumeration; there will usually be less than 500 of them.

The school timetable is constructed over a week, consisting of a number of periods, usually between 30 and 48:

type Period = 1..NP;

Each activity will in general be required to occur several times during the week; this is defined by a mapping:

times: Activity → 1..NT;

where NT will usually be not more than 10.

Each activity involves participation of a set of items; for example, "IV form physics" will require a meeting of form IV with a teacher, (say Jones); it also requires a physlab, and perhaps also a projector. The requirement for each activity is given by a mapping:

requirement: Activity → Item set;

We also shall use the inverse of this mapping:

users: Item → Activity set

where a ∈ users(i) ≡ i ∈ requirement(a)

A timetable may be specified by giving for each activity the set of periods in which that activity is to take place, that is, by the mapping:

timetable: Activity → Period set

This mapping must satisfy the following constraints:

(1) Each activity must take place exactly the right number of times:

∀a:Activity size(timetable(a)) = times(a) (C1)

(2) Each Item ∀i,p must be used exactly the right number of times in each period: busy(i,p) = lives(i),

where busy(i,p) = size $\{ a \in users(i) \mid p \in timetable(a) \}$ (C2)

The requirement that an item may not be underused is not in practice restrictive. If an item i is going to have free periods, their number can be computed by the formula:

$$NP \times lives(i) - \sum_{a \in users(i)} times(a)$$

Then for each unit of an item a further artificial activity can be inserted to represent its free periods. A "free" activity is thus an activity which has only one item in its requirement set.

The number of such artificial "free" activities may be up to 250, bringing the total number of activities up to 750.

2. Additional Constraints.

In practice, this relatively simple characterisation of the timetabling problem is complicated by a number of additional constraints, described below.

2.1 Preassignments.

Certain activities are specified to take place at particular times; for example it may be necessary that one of the cooking classes takes place just before lunch, when its products are to be eaten; or perhaps a teacher is unavailable at a certain time, so that his "free" activity must be scheduled at that time. Such constraints may be expressed by a mapping:

$timetable_o$: Activity → Period set

such that $timetable_o(a)$ consists of those periods in which that activity a must occur in the completed timetable, i.e.

∀a $timetable_o(a) \subseteq timetable(a)$ (C3)

2.2 Forbidden assignments.

Sometimes an activity must be prevented from occurring at certain times; for example, swimming should not occur immediately after lunch; or practical classes should not occur in the first period of the day. Such constraints may be expressed by a mapping:

$$possible_o : Activity \rightarrow Period\ \underline{set}$$

such that $possible_o(a)$ gives the set of periods in which the activity a may occur in the completed timetable, i.e.

$$\forall a \qquad timetable(a) \subset possible_o(a) \qquad (C4)$$

2.3 Multiperiod activities.

Certain activities of a school may be such that they can only effectively be carried out in a number of consecutive periods, for example, practical classes, or games. Such constraints may be specified by a mapping:

length:Activity→1..NL, where NL will usually be 3 or 4,

which gives for each activity the length of the multiple period which must be assigned to it. We also need a mapping:

$$starts : 1..NL \rightarrow \mathbb{P} Period\ \underline{set}$$

which gives for each length the set of periods in which a multiple period of that length may start (for example, not the periods at the end of a day, or before a lunchbreak). We can now define a function

$$trim : 1..NL \times \mathbb{P} Period\ \underline{set} \rightarrow Period\ \underline{set}$$

such that trim(a,ps) is the result of removing from ps any isolated periods, which cannot form part of a multiple period of the required length. The required constraint may now be expressed

$$\forall a \qquad trim(a;timetable(a)) = timetable(a) \qquad (C5)$$

2.4 Spread.

Another constraint is imposed by the desire not to have two occurrences of the same activity on the same day. Activities to which this constraint is to be applied belong to the set:

spread: Activity set.

This constraint, of course, cannot be satisfied by an activity which occurs too often; but we may artificially split such an activity into two or more activities with identical requirements. The days in the week may be defined:

type Day = 1..ND;

where ND is usually between 5 and 7.

We are also given mappings:

periods in:Day→Period set

day of:Period→Day

which give the set of periods in a given day, and the day in which any given period occurs. Obviously,

$$p \in \text{periods in}(d) \equiv d = \text{day of}(p) \quad \text{(P1)}$$

$$a \in \text{spread} \supset \text{times}(a) \leq ND \times \text{length}(a) \quad \text{(P2)}$$

We also assume that the periods in a day form a continuous range, so that

$$\text{ps-periods in}(d) = \text{trim}(a, \text{ps-periods in}(d)) \quad \text{(P3)}$$

Another realistic assumption is that all multiple periods must be spread:

$$\text{length}(a) > 1 \supset a \in \text{spread} \quad \text{(P4)}$$

The constraint may be expressed:

$$\forall a \quad a \in \text{spread} \supset \text{possdays}(\text{timetable}(a)) = \text{times}(a) \div \text{length}(a) \quad \text{(C6)}$$

where $\text{possdays}(ps) = \text{size}\{d \mid ps \wedge \text{periods in}(d) \neq \text{empty}\}$

2.5 Ties.

In some cases, it is desired to prevent certain related but not identical activities from occurring on the same day, for example, physics theory and physics practical, or two non-academic subjects. We may express this by a mapping:

tie: Activity→Activity set

which for a given activity specifies the set of other activities which must not occur on the same day, i.e.

$$\forall a,a',p,p'. \quad a' \in \text{tie}(a) \;\&\; p \in \text{timetable}(a) \;\&\; p' \in \text{timetable}(a') \supset \text{day of}(p) \neq \text{day of}(p') \quad \text{(C7)}$$

For most activities, tie(a) will be empty.

The conjunction of conditions (C1) to (C7) will be known as (C).

2.6 Conclusion.

The constraints given above are in practice never wholly observed by a human timetabler; and there are grounds for belief that it is often logically impossible to construct a timetable that observes all the constraints initially specified by a school. Thus in spite of the rigour with which the problem has been stated, in practice the computer cannot be expected to solve the problem as posed, but only to "do as much as it can" within the constraints, or alternatively "break as few constraints as possible". But the latter approach is not very promising, since the importance of the constraints varies from case to case, and cannot reasonably be specified in advance. "Is it more important that Mr. Jones gets his nap after lunch, or form IV should not have current affairs and swimming on the same day?" - no schoolmaster is willing to answer many hundreds of such questions in advance; but will answer a few such questions when he knows that completion of the timetable depends upon it.

actually, we preserve the relation timetable$_o$ ⊆ possible$_o$ & try to get them to equal each other, either by cancellation from possible or insertion in timetable (see constraint C3, C4).

3. The Timetabling Method.

The method of timetabling is to make successive decisions about the assignment of activities to periods, until all activities have been successfully assigned. Each decision can be made in two ways

either (a) to assign an activity to a period

or (b) not to assign an activity to a period.

Choice (b) is known as a cancellation. If a decision turns out not to lead to a successful timetable, the alternative decision is taken. If neither decision is successful, one of the previous decisions must have been mistaken, and further backtracking must occur. If neither way of taking the very first decision is successful, then the problem was incorrectly posed, and there is no solution.

This method requires that we keep a record of decisions made previously. This may be done by two mappings:

T,P:activity→period set

where T(a) is the set of periods which have been assigned to a (initially empty) timetable$_o$

and P(a) is the set of periods which have not been cancelled for a (initially the full set) initially possible

Obviously, if our decisions are non-contradictory, the following will always hold:

$\forall a.\quad T(a) \subseteq P(a)$

When all decisions have been taken, T = P, and it is possible to check whether T satisfies the constraints C1 to C7. C3 and C4 can be guaranteed by setting the value of T as timetable$_o$, and the value of P as possible$_o$.

Thus the basic solution method is as follows:-

Ich würde sagen, dieses Programm gibt ohne weitere Tests die Funktion

$(\forall a)_{\in A}\ f(a) = P$ (set of periods)

timetable program:

```
begin
      success label print T;
      progress recursive procedure
        if T ≠ P then
          begin new a:Activity,p:Period;
                select (a,p) such that p ∈ P(a)-T(a);
            {T(a):+p;
             call progress;
             T(a):-p} note Q1 • Q2 means either (Q1;Q2)
          • {P(a):-p;                          or (Q2;Q1)
             call progress;
             P(a):+p}
          end
      else if T satisfies C then go to success;
end
```

sinke Rückspeite

Trivialste prozess

Problem Konstruktion einer Funktion $f : \mathfrak{A} \to 2^P$ wobei

$\mathfrak{A}$: endl. Menge von activities

P: endl. Menge von periods

so dass ein $Test_1$ (restrictions) auf f erfüllt

und ein $Test_2$ (completeness, sufficiency)

Konstruktion von einer Aufzählung $\langle \alpha_j, \beta_j \rangle$ aller paare $\langle a, p \rangle$
ausgehend kann man f etwa wie folgt konstruieren

Sei $f \restriction j$ die Funktion, die durch Teilfolge $\langle \alpha_{j_1}, \beta_{j_1} \rangle \ldots$

$\ldots \langle \alpha_{j_k}, \beta_{j_k} \rangle$ von $\langle \alpha_1, \beta_1 \rangle \ldots \langle \alpha_j, \beta_j \rangle$ eindeutig def
ist

ohne f(

if $Test1 (f \restriction j \cup \langle \alpha_{j+1}, f \restriction j (\alpha_{j+1}) \cup \{ \beta_j \} \rangle)$ then
direkt inkorrekt

$f \restriction j{+}1$:= Fkt def durch $\langle \alpha_{j_1}, \beta_{j_1} \rangle \ldots \langle \alpha_{j_k}, \beta_{j_k} \rangle \langle \alpha_{j+1}, \beta_{j+1} \rangle$

Wenn Algorithmus irgendwo stoppt bevor $f \restriction j$ hinreichend
zurückgehen zum letzten Paar dieses Paar vervollständigen
und an diesem Punkt nochmals starten.

~~else if T satisfies C then goto success;~~

input and check the data;

T:=timetable$_o$; P:=possible$_o$;

call progress;

print failure message

end

There can be little doubt of the "correctness" of the program; it will always produce a solution if there is one and print a failure message if there is not. But the time taken to do so in practice will be many orders of magnitude larger than can be accepted; and the main part of the programming problem still remains.

3.1 Consistency.

One of the most obvious inefficiencies of the program is that it waits until the very end before testing whether T and P are satisfactory; whereas in many cases it can be seen well in advance that there could never be any satisfactory timetable based on decisions taken so far.

i.e. $\neg\exists$ timetable. C&$\forall$a.T(a)Ctimetable(a)CP(a)

When this is detected, there is no point in making further calls on "progress", since failure is inevitable. We thus change our program thus:

replace: call progress

by: if consistent then call progress

where consistent:Boolean (initially true) is a marker set to false if inconsistency is detected. Unfortunately, it is too much to hope for a 100% test of possibility of success, since this would require a guarantee of the existence of a complex object like a timetable before it had been constructed. So all that can be done is to find a set of necessary conditions for success; for falsity of a necessary condition will then be a clear indication that failure is inevitable. A condition N is shown to be necessary if it can be proved that:

C&$\forall$a (T(a)Ctimetable(a)CP(a))$\supset$N.

The following necessary conditions can readily be proved:

$\forall$a size(T(a))$\leq$times(a)$\leq$size (~~trim~~P(a)) N1

$\forall$i,p. busy (i,p)$\leq$lives(i)$\leq$possbusy(i,p) N2

where busy(i,p) = size $\{a \in$ users(i) $\mid p \in T(a)\}$ I1

possbusy(i,p) = size $\{a \mid$ users(i) $\mid p \in P(a)\}$ I2

$\forall$a a$\in$spread $\supset$ times(a) $\leq$ size(possdays(a))xlength(a)

where possdays(a) = $\{d \mid ps \wedge$periods in (d)$\neq$empty$\}$ N3

$\forall$a,d a$\in$spread$\supset$max (T(a)$\wedge$periods in (d))-min(T(a)$\wedge$ periods in (d))$<$length(a) N4

$\forall$a,a',p,p' a'$\in$tie(a)& p$\in$T(a)&p'$\in$T(a)$\supset$day of (p)$\neq$day of (p') N5

In writing the program to check consistency, much time can be saved if it can be assumed that consistency held before the most recent decision, and it is known what that decision was. It also pays to store the values of busy and possbusy rather than recomputing them each time.

busy,possbusy:ItemxPeriod → 1..NP

where initially busy is uniformly zero, and possbusy is uniformly equal to NP.

```
after  T(a):+p  do
  begin  consistent :&size(T(a))≤ times(a); note N1
         for i∈ requirement(a)do{busy(i,p):+1;noteI1;consistent:&busy(i,p)≤lives(i)};
                                                                    noteN2;
         ps:= periods in (day of (p));
         if a∈spread then consistent:&(max(T(a)∧ps)-min(T(a)∧ps)<length(a));
         note N4;
         for a'∈tie(a)do consistent:&T(a')∧ps = empty;
         note N5;
  end;
after  T(a):-p do
         for i∈requirement (a) do{busy(i,p):-1;consistent:=true; note I1}
after  P(a):-p do
  begin  consistent:&times(a)≤ size(P(a));note N1;
         for i∈requirement(a)do{possbusy(i,p):-1;noteI2;consistent:&lives(i)≤
                                                          possbusy(i,p)};
         note N2;
         if a∈spread then consistent:&times(a)≤ size(possdays(a))xlength(a);noteN3
  end
after  P(a):+p do
         for i∈requirement(a)dopossbusy(i,p):+1; note I2.
```

Since we assume consistency of T and P before these decisions, it is necessary to check the consistency of the data immediately after input. This may be done on the assignment of $timetable_o$ to T and $possible_o$ to P, using the above fragments of code. Alternatively, it is a trivial matter to write a more efficient program for initial checking.

3.2 Forcing.

A decision is said to be forced if the alternative decision would lead inevitably to failure. An assignment of a to p is forced if

$$\forall timetable\,[C\,\&\,\forall a(T(a)\,C\,timetable(a)\subset P(a))\supset p\in timetable(a)] \qquad (1)$$

and a cancellation is forced if

$$\forall timetable\,[C\,\&\,\forall a(T(a)\,C\,timetable(a)\subset P(a))\supset p\in timetable(a)] \qquad (2)$$

Now it would again be too much to hope that all forced decisions will be detectable, since if they could, all unforced decisions could be taken either way, and the whole timetable could be constructed without backtracking. The best that can be expected is to find a powerful set of sufficient conditions for forcing.

Of course, a forced decision will not lead to a successful timetable if one of the previous decisions was incorrect, or if the problem was originally insoluble. Thus it will frequently be necessary to backtrack over forced decisions; and so we must keep a record what they were. This can be done in a stack declared locally in the procedure progress:

stack: (ActivityxPeriod) sequence initially empty.

When unmaking a forced decision, there is obviously no point in making the alternative decision.

It can be readily proved that the following conditions are sufficient for forced assignment of a to p, where in all cases $p \in P(a)-T(a)$

size (P(a)) = times(a) FA1

$\exists$ i $\in$ requirement (a). lives(i)=possbusy(i,p) FA2

a $\in$ spread&times(a)=size(possdays(a))xlength(a)
&size(P(a)$\wedge$periods in (day of (p))=length(a) FA3

length(a)>1&$\exists$d(p $\in \bigcap$ {ps $\in$ right length (a,d) | (T(a)$\subset$ps$\subset$P(a)}) FA4

where ps $\in$ right length (a,d) means that ps is a multiple period of length (a) in day d.

The following conditions are sufficient for cancellation of a from p, where in all cases, $p \in P(a)-T(a)$;

size (T(a))=times(a) FC1

$\exists$ i $\in$ requirement (a)(busy(i,p)=lives(i)) FC2

a $\in$ spread&length(a)=1&$\exists$u$\in$T(a)&size(T(a)$\wedge$periods in (day of (p)))=1 FC3

length(a)>1&$\exists$d(p $\tilde{\in} \bigvee$ {ps $\in$ right length (a,d) | T(a)$\subset$ps$\subset$P(a)}) FC4

$\exists$a', a' $\in$ tie(a)&T(a')$\wedge$periods in (day of (p))$\neq$empty FC5

$\exists$a' a' $\in$ tie(a)&size(possdays(a')xlength(a)=times(a)&day of (p)
$\in$ possdays(a')) FC6

The forced decisions are to be taken as soon as the corresponding sufficient condition becomes true; which can only happen when an assignment is made to one of the variables which it contains. Thus we design the following mutually recursive procedures:

```
after  T(a):+p do recursive
       if p ∈̃ P(a) then consistent:=false else if consistent then
begin  if size (T(a))=times(a) then
       for p ∈ P(a)-T(a) do {stack:^(a,p); P(a):-p}; note FC1;
       if  a ∈ spread&length(a)=1 then for p' ∈ P(a)∧periods in (day of (p))
             -T(a) do {stack:^(a,p');P(a):-p'}; note FC3;
       if  length (a)>1 then call multilength;
```

```
        for  a' ∈ tie(a) do
           let p^s=periods in (day of (p));
           for p' ∈ P(a) ∧ ps do
                    {stack:^(a,p');P(a):-p'}                              FC5
end
after  busy(i,p):+1 do recursive
           if lives(i)=busy(i,p)then
             for a ∈ users(i)do
              if p ∈ P(a)-T(a)then{stack:^(a,p);P(a):-p}; note FC2;
after  P(a):-p do
if   p ∈ T(a) then consistent:=false else if consistent then
begin if size (P(a))=times(a) then
             for p'  P(a)-T(a)do{stack:^(a,p');T(a);+p} ; note FA1;
      if  a ∈ spread&times(a)=size(possdays(a))xlength(a)
                    &size(P(a) ∧ periods in (day of(p))=length(a)
             then for p' ∈ P(a) ∧ periods in (day of(p))-T(a)do
      if length(a)>1 then {stack; (a,p);T(a):+p'^};  note FA3;
      call multilength;
      if size (possdays(a))xlength(a)=times(a) then
            for a' ∈ tie(a) do for d ∈ possdays(a)do
             for p' ∈ P(a') ∧ periods in (d)do{stack:^(a,p');P(a'):-p'};
      note  FC6;
end
multilength procedure
begin let d = day of(p);intersec:=periods in (d); union:=empty;
      for ps ∈ right length (a,d)do {intersec: ∧ps;union:∨ps};
      for p' ∈ periods in(d)do
            if p' ∈ intersec-T(a)then {stack:^(a,p');T(a);+p'}
       else if p' ∈ P(a)-union then {stack:^(a,p');P(a):-p}
end;  note  FA4 and FC4;
after  possbusy(i,p):-1do recursive
       if possbusy(i,p)=lives(i) then
          for a' ∈ users(i) do
             if  p ∈ P(a')-T(a') then {stack:(a',p)T(a');+P}               FA2
before  T(a):-p do unstack;
before  P(a):+p do unstack;
where   unstack procedure
        while stack ≠ empty do
        begin (a',p') from stack;if p ∈̃ P(a') then P(a');+p'
                                              else  P(a');-p';T(a'):-p'
end ⟶
```

4. Tight Sets.

It is fairly obvious that the number of calls of progress will depend on the typical number of unforced decisions which have to be remade after backtracking. We put

D = total number of decisions to be taken.

F = typical number of forced decisions following upon each unforced one.

p = probability that an unforced decision will be correct.

B = typical depth of recursions required before an incorrect decision has been corrected by backtracking.

Then the total number of recursions required for a successful timetable will be proportional to

$$\frac{D}{F} \times p \times 2^{B}.$$

Thus the most important parameter to minimise is B; in fact to reduce B by 1, it is worthwhile to double the time taken by the main procedure. Reduction of B can be achieved by

(1) strengthening our tests of consistency, so that inconsistency can be detected early by explicit tests rather than backtracking;

(2) always selecting for the next decision an assignment which is most likely to reveal any latent inconsistency.

A factor to be maximised is F, the number of forced decisions. This can be achieved by means that will also help in reducing B.

(1) A strengthened forcing condition will require more forced decisions for each unforced one.

(2) To select always for next decision that activity which is likely to lead to the longest chain of forced decisions will both increase the typical value of F, and give an early indication of latent inconsistency.

The next task is to maximise p, the probability of making an initial correct decision in the unforced case, i.e. where p is known to be less than 1. Unfortunately, there does not seem to be any ready way of even estimating this probability, which must therefore be assumed to be around a half. Since the size of improvement obtainable by increasing p is bounded by a factor of two, it is not worthwhile to spend much time "guessing" the right decision. One might as well make a wrong decision, and hope that it will be soon detected.