3.1. Construct a timetable.

The method of constructing a timetable is to make successive decisions about the assignment of activities to periods. Each decision can be made in one of two ways

either (a) to assign an activity ~~to~~ at a period
or (b) not to assign it

Choice (b) is known as a cancellation of the activity at the period

We construct a procedure "progress" which jumps to print out the timetable if and when it is successfully completed, and exits normally if there is no way of completing a timetable on the basis of previously taken decisions. Each activation of progress takes one decision, and then enters itself recursively. If normal exit from the recursive call takes place, the decision must be reversed, and another recursive call is made. If this also exits normally, this proves that no decision at this stage can lead to a successful timetable, and thus the given activation should also exit normally, so that further backtracking may take place.

Thus the basic structure of the procedure is

```
progress recursive procedure
    begin if complete then go to printout;
          choose appropriate (a,p);
          try assignment (a,p);
          try cancellation (a,p)
    end
```

In order to detect completion of the timetable, we keep a count of all decisions (both assignments and cancellations) taken so far, and compare it with the number of decisions which need to be taken.

count:Integer initially 0

to be decided:Integer ~~initially~~ constant ~~$\sum_a$ size (possible$_0$(a) - timetable$_0$ (a))~~ size (Period) x size (Activity)

In constructing a backtracking algorithm to tackle a problem of any size it is most important to avoid as far as possible the pursuit of decision sequences which can be readily detected to be inconsistent, in the sense that they can never lead to success. Thus before taking each decision, the consistency of the previous decision should be rigorously checked, and if inconsistency is detected, an immediate exit from the current activation of "progress" should be made. We thus derive the following program:

construct the timetable:

begin progress **recursive procedure**

begin check consistency;

if inconsistent then go to impossible; stet

~~if count=countmax then go to printout;~~ to be decided

select suitable (a,p);

try assignment(a,p);

try cancellation (a,p);

impossible: ~~end~~

end;

progress;

print failure message; stop;

printout: ~~end~~

end

3.2. Assignment and cancellation.

This method requires that we keep a record of decisions (as they are) 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 set equal to timetable$_0$)~~ initially empty

and P(a) is the set of periods which have **not** been cancelled for a ~~(initially set equal to possible$_0$)~~ initially the full set.

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

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

space

This is ⊆

If $p \in P(a)-T(a)$, this means that no decision has yet been made about assigning or cancelling a ~~from~~ at p. An assignment may then be made by

$$T(a):+p$$

and a cancellation may be made by

$$P(a):-p.$$

We can now write the programs for assignment and cancellation:

Thus each decision either increases the number of periods in T or decreases the number of periods in P, until eventually they are equal, ~~at which stage~~ to each other, and therefore to the final timetable. The printout will output the value of T try assignment(a,p):

begin T(a):+p;count:+1;progress;count:-1;T(a):-p end

space

try cancellation(a,p):

begin P(a):-p;count:+1;progress;count:-1;P(a):+p end

3.3. Check Consistency.

It can be seen that the efficiency of the program is critically dependent on the success of the consistency check in ensuring that only those decisions are pursued which lead to a successful timetable. If an absolute test of consistency were available, it would never be necessary to backtrack over more than one decision, since the futility of a decision which failed to lead to a successful timetable would be immediately detected.

Unfortunately, it is too much to hope for a 100% test of consistency, 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 the existence of a timetable based on current decisions; for falsity of a ~~mx~~ 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:

this means, close up the space

C & $\forall a$ $(T(a) \subseteq timetable(a) \subseteq P(a)) \supset$ N.

The following necessary conditions can readily be proved:

$\forall a$ size(T(a)) $\leq$ times(a) $\leq$ size (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) & p $\in$ T(a)} I1

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

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

~~where possdays(a) = {d | P(a) $\wedge$ periods in (d) $\neq$ empty}~~ I3 N3

insert N3, N4, N5, N6 here. (from ~~next~~ inserted page)

~~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.~~

~~We therefore introduce variables~~

~~olda: Activity; oldp: Period; action: {assig, cancel}.~~

~~after T(a):+p do {olda:=a; oldp:=p; action:=assig}~~

~~after P(a):-p do {olda:=a; oldp:=p; action:=cancel~~

The following necessary conditions can be ~~easily~~ derived from the corresponding constraints, together with the observation that T approaches the final timetable "from below" and P approaches it "from above." Thus ~~derivation is so~~ the necessity of the conditions is sufficiently obvious to require no proof.

$$\forall a: \quad size(T(a)) \leq times(a) \leq size(P(a)) \qquad N1$$

$$\forall a,i: \quad busy(i,p) \leq lives(i) \leq possbusy(i,p) \qquad N2$$

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

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

$$\forall a,d: \quad a \in spread \supset size(T(a) \wedge periodsin(d)) \leq 1 \qquad N3$$

$$\forall a,d: \quad a \in spread \vee length(a) > 1 \supset$$

$$possdays(T(a)) \times length(a) \leq times(a) \leq possdays(P(a)) \times length(a) \qquad N4$$

~~$\forall d$~~

$$\forall a,d. \quad length(a) > 1 \supset times(a) \leq size\{d \mid posstuples(a,d) \neq empty\} \times length(a) \qquad N5$$

where $posstuples(a,d) = \{ps \mid T(a) \subset ps \subset P(a) \wedge periodsin(d)$
$\;\&\; ps = tuple(length(a), first(ps))$
$\;\&\; first(ps) \in starts(a)\}$

The conjunction of these five conditions is known as N.

The correctness of the program depends on the fact that ~~th~~ when the ~~twisted~~ jump to printout is made, the consistency check which guarantees that ~~truth~~ T satisfies the truth of N will also guarantee the truth of the constraints C. When count = size (Period) × size (Activity), every decision will have been taken, and T will necessarily equal P. Substitution of T for P in ~~the~~ N will imply the truth of C1 to C5. C6 we ignore; and C7 and C8 can be guaranteed by making all preassignments ~~before any other assign~~ first.

The construction of a program to check N and jump to impossible if false may be ~~eas~~ easily derived from the definition of N, and may be postponed to a later stage.

$length(a) > 1 \supset countuples\ (length(a), P(a) \times length(a) \geq times(a)$ N4

$a \in spread \supset$ define posttuples(a,d) = {ps ∈ period set | ∃p ∈ [illegible] length(a) [illegible]

$\forall d(counttuples(length(a), T(a) \wedge periods\ in(d)) \leq 1$

$\&\ size\ \{d \mid counttuples(length(a), P(a)) \geq 1\} \times length(a) \geq times(a))$ N5

p' ∈ dayof(p)   p ∈̃ T(a)

$\forall a,a',p,p'\quad a' \in tie(a) \,\&\, \cancel{p \in T(a)} \,\&\, p' \in T(a) \supset \cancel{dayof(p) \neq dayof(p')}$ N6

The conjunction of these ~~six~~ five conditions is known as N.

~~The program to check the truth of N may be fairly easily derived from the definition of N, and its detailed coding may therefore be postponed.~~

3.4. Select suitable (a,p).

Another factor which is critical to the efficiency of timetable construction is the judicious selection of the next decision to take. An obviously sensible strategy is to take first those decisions which are forced, in the sense that they can only be taken one way; because if such a decision leads to inconsistency, backtracking is comparatively cheap.

(1) A condition FA is said to be sufficient for forcing assignment of ~~p to a~~ a at p if it can be proved that

$$FA(a,p) \supset \neg N^{P}_{P'}$$

where $P' = (P \frown a{:}P(a)-p)$, ie, to cancel p from P(a) leads to immediate inconsistency.

(2) A condition FC is said to be sufficient for cancelling ~~p for~~ a at p ~~a~~ if it can be proved that

$$FC(a,p) \supset \neg N^{T}_{T'}$$

where $T' = (T \frown a{:}\ T(a)+p)$, ie, to assign ~~a to p~~ p to T(a) leads to immediate inconsistency.

(3) An easy way to ensure that all preassignments are carried out first is to treat them as forced decisions.

Since a single decision may give rise to several further forced decisions, it is necessary to store these decisions for future execution in two sets

forced assign: Activity ⇒ period set initially timetable

~~forced assign, forced cancel: (Activity x Period) set initially empty~~

forced cancel: Activity ⇒ period set initially forbidden.

Decisions are placed in these sets as soon as a sufficient condition for forcing is detected; thus

$(a,p) \in forced\ assign \supseteq FA(a,p)\ \vee\ p \in timetable(a)$

$(a,p) \in forced\ cancel \supseteq FC(a,p)\ \vee\ p \in forbidden(a)$

We can now split selection of the next decision into two parts:

select suitable(a,p);

{select forced(a,p); select unforced(a,p)}

where

select forced(a,p):

if forced assign ≠ empty then

{(a,p) from forced assign; try assignment(a,p); go to impossible}

else if forced cancel ≠ empty then

{(a,p) from forced cancel; try cancellation(a,p); go to impossible}

Sufficient conditions for forcing can be readily derived from the corresponding consistency check.

~~The detection of forced decisions should be made during the check of consistency.~~ ~~Sufficient conditions for forcing can readily be derived from the corresponding consistency check.~~

~~Each of the following is a sufficient condition for assignment of a at p where $p \in P(a)-T(a)$:~~

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

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

~~day determined(a) & size(P(a) $\wedge$ periods in(day of(p))) = length(a)~~ FA3

day of(p) $\in$ Tday(a)

~~where day determinated (a) = df a $\in$ spread & times(a) = size(possdays(a)) x length(a)~~ ~~FA4~~

where for all a $\in$ spread

Tday (a) = if times(a) = size (possdays(a)) x length(a) then possdays(a) else {d | T(a) $\wedge$ periodsin(d) ≠ empty} I3.

~~length(a) > 1 & a $\in$ spread & p $\in \wedge$ posstuples(a,p)~~ FA4

~~where posstuples(a,p) =~~

~~{ps:Period set | T(a) $\subset$ ps $\subset$ P(a) $\wedge$ periods in(day of(p))~~

~~& ps = tuple(length(a),first(ps))~~

~~& first(ps) $\in$ starts(a)}~~

copy FA1 to FA4 here

~~Each of the following is a sufficient condition for forced cancellation of p from a, where $p \in P(a)-T(a)$~~

a at p

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

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

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

~~length(a) > 1 & a $\in$ spread & p $\tilde{\in} \vee$ posstuples(a,p)~~ FC4

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

~~$\exists$a'(a' $\in$ tie(a) & day determined(a') & day of (p) $\in$ possdays(a'))~~ Tday (a') FC5

copy FC1 to FC5 here

~~As before, the program to detect these conditions can be easily derived, and its detailed coding may therefore be postponed.~~

(10a)

The following are sufficient conditions for cancellation of a at p, where $p \in P(a)-T(a)$:

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

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

$daydetermined(a) \;\&\; dayof(p) \not\in possdays(T(a))$ FC4

where $daydetermined(a) \equiv$
$(a \in spread \lor length(a) > 1) \;\&\; times(a) = possdays(P(a)) \times length(a)$.

FC4

$\exists p' \; a \in spread \;\&\; p' \in T(a) \;\&\; p \in periodsin(dayof(p'))$ FC3

$length(a) > 1 \;\&\; p \not\in \bigcup possTuples(a, dayof(p))$ FC5

The following are sufficient conditions for assignment of a at p, where $p \in P(a)-T(a)$:

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

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

(no corresponding condition) FA3

$daydetermined(a)$

$\&\; size(P(a) \land periodsin(dayof(p))) = length(a)$ FA4

$length(a) > 1 \;\&\; p \in \bigcap possTuples(a, dayof(p))$ FA5

close up

When an inconsistency has been detected, there may still be some forced decisions in the set. These should be removed before taking the next unforced decision. Therefore, between ~~be~~ tryassignment and trycancellation there should be

forced assign := forced cancel := empty.

The construction of a program to detect sufficient conditions for forcing can ~~again~~ be easily derived; and will again be postponed until after more difficult decisions have been taken.

3.5, select unforced (a,p);

In selecting an unforced decision, it is a good idea to select a decision which is most likely to lead to ~~revel~~ early revelation of a latent inconsistency if there is one. The most suitable candidates will be those decisions which are ~~most~~ likely to lead to the longest chains of consequential forced decisions; ~~These decisions will be found in areas which the human timetabler for these are~~ Such decisions will occur in those areas ~~in~~ which ~~there is~~ offer least freedom of ~~decision~~ choice in making decisions, — the areas which the human timetabler would regard as "sticky", and on which he also would concentrate his early attention.

~~More precisely, for each (a,p) we can estimate roughly how many forces would result from can assignment and from cancellation; we choose the decision for which one of these is the greatest; and then, to give best chance of success, we take the choose first the other decision.~~

$\cancel{\text{times}(a) - \text{size}(T(a)) = 1} \vee \text{size}(P(a)) - \text{times}(a) = 1$ ST1

$\exists i \in \text{requirement}(a)\ \&\ \cancel{\text{lives}(i) - \text{busy}(i,p) = 1}$
$\text{possbusy}(i,p) - \text{lives}(i) = 1$ ST2

ST3

(no corresponding condition)

~~ST4~~

$\text{daydetermined}(a)\ \&\ \text{size}\left(P(a) \wedge \text{periodsin}(\text{dayof}(p))\right) - \text{length}(a) = 1$ ST4

ST5

$\text{length}(a) > 1\ \&\ T(a) \wedge \text{periods in}(\text{day of}(p)) \neq \text{empty}.$

ST5 ensures that a partially assigned multiple period is always regarded as stiff.

We therefore introduce a set:

Activity → Period set

stiff: ~~(Activity x Period) set~~

into which decisions are placed when they are recognised to be stiff. The main criterion of stiffness is that the decision ~~was~~ is likely to result in further forced decisions and this can conveniently be detected ~~in~~ during (consistency check, at the same time as) the search for forced decisions. ~~are derived from~~

The most obvious criteria are for stiffness of (a, p), where $p \in P(a) - T(a)$ are:

copy from

$\text{size}(P(a)) - \text{times}(a) = 1$ ST1

$\exists i\ (i \in \text{requirement}(a)\ \&\ \text{possbusy}(i,p) - \text{lives}(i) = 1)$ ST2

$\exists d.\ p \in d\ \&\ d \in \text{Tday}(a)$ ~~daydetermined(a,d)~~ $\&\ \text{size}(P(a) \cap \text{periodsin}(d)) - \text{length}(a) = 1$ ST3

$\text{times}(a) - \text{size}(T(a)) = 1$ ST4

$\exists i\ \ i \in \text{requirement}(a)\ \&\ \text{lives}(i) - \text{busy}(i,p) = 1$ ST5

The conjunction of these conditions will be known as ST.

~~These conditions may most readily be detected during the check on consistency; again, the detailed programming coding may be postponed.~~

In using the stiff set to select an unforced decision, it is necessary to recall that many of the decisions which it contains may already have been taken as forced decisions. Such decisions must be removed from the stiff set as they are encountered.

select unforced (a,p):

<u>if</u> stiff $\neq$ empty <u>then</u> ~~(a,p) := one of (stiff)~~ select (a,p) such that $p \in$ stiff(a);

~~<u>repeat</u> <u>begin</u> (a,p) := one of (stiff); stiff :− (a,p) <u>end</u>~~

~~<u>until</u> $p \in P(a) - T(a) \vee$ stiff = empty;~~

~~if~~ <u>while</u> $p \tilde{\in} P(a) - T(a)$ & stiff $\neq$ empty <u>do</u>

~~select (a,p) such that~~ p

~~{(a,p) : stiff(a) (a,p); (a,p) := one of (stiff)}~~

{stiff(a) :− p; select (a,p) such that $p \in$ stiff(a)}

<u>if</u> stiff = empty <u>then</u> select unstiff (a,p)

~~Att~~

Since decisions will be removed from stiff after they have been forced, it is a good idea to put them back whenever a forced decision is changed. This may result in decisions being ~~regarded~~ put ~~as~~ into stiff even when they do not satisfy ST; but anyway it seems a good idea to ~~recognise~~ regard as stiff those forced decisions which are already known to lead to inconsistency.

It remains to decide how to select non-stiff decisions (4)

The easiest way of doing this is ~~« stiff »~~ to allocate to each activity a score, indicating its "a priori" difficulty, and to ~~sort~~ maintain a sorted ~~list~~ sequence from which the activity with highest score will always be ~~selecting~~ selected. A scoring formula will ~~dive~~ give highest scores to ~~multiple p~~ multiperiod activities; ~~and to tied activities;~~ ~~the score~~ ~~of an activ~~ a medium score to activities that have to be spread, a low score to activities with only a few units required, and a zero score to free periods. The exact weighting coefficients are somewhat arbitrary, and may need to be adjusted in the light of experience.

~~sorted stiffness: 1 .. size(Activity) → Activity sequence~~

In addition to a static allocation of priority to activities, it is worth while to keep a record of activities and periods which ~~during the co~~ become ~~[illegible]~~ stiff during the course of timetabling. It might be worth while to maintain ~~several~~ two grades of stiffness, ~~from the most stiff (N) to the least stiff (1), where N might be 3 or 4.~~

~~stiff: 0..N → (Activity × Period) set~~

stiff, verystiff: (Activity × Period) set

From within

~~This requires a sequence~~ Thus we introduce a sequence.

sorted: Activity sequence,
which holds all activities in order of their a priori difficulty

Thus we can code.

select undiff(a, p) q:

~~if sorted = empty,~~
~~if sorted ≠ empty then { a := head (sorted); p := any(Period)~~
~~if P(a) − T(a) = empty then {sorted := tail(sorted)~~
~~while p ∉ P(a) − T(a) ∧ sorted ≠ empty do~~
~~{ if sorted = empty then go to success~~
~~sorted := tail (sorted)~~
~~if~~

repeat **begin**
  if sorted = empty **then** **go to** ~~success;~~ printout;
  a := head(sorted);
  if P(a) − T(a) = empty **then** { sorted := tail (sorted);
        else p := any(P(a) − T(a))

until p ∈ P(a) − T(a);

3.5. Premature Termination

~~A very rough~~

The program as designed so far will always find a timetable if there is a one, and report failure if there is not. However, a very cursory calculation shows ~~that~~ the backtracking may take a wholly impractical amount of time. ~~It is~~ ~~would be better~~ When the difficulty of making further progress is too great, it would be better to stop and print out the contents of T(a) as it is so far. On the other hand, it would be a shame to stop at a point when there ~~was~~ is ~~every~~ a good chance of ~~proceeding to a successful conclusion,~~ making good progress towards a successful conclusion. This suggests that a measure be taken of the inherent difficulty of the current situation, and when some predetermined limit is reached, the timetable will be printed out.

A good measure of the difficulty of the situation is ~~a count of~~ indicated by ~~[illegible]~~ the amount of backtracking has taken place in the recent past. ~~If the number of decisions back~~ This can ~~count~~ ~~can~~ be estimated by comparing the current value of the count with the highest value it has ever achieved. Thus we need a variable

countmax: Integer initially 0

~~and after~~

after count:+1 *do* *if* count > countmax *then* countmax := count

after count:−1 *do* *if* countmax − count < limit *do*

{*print* T ; stop}

~~4. Detailed Coding Programming of Consistency Check.~~

4. Details of Consistency Check

The time has now come to carry out ~~the combined tasks~~ of the tasks of detailed ~~coding~~ programming which have been postponed in the previous sections with the claim that ~~they were~~ trivial. We must therefore program ~~The task of this coding~~ the consistency check which is to guarantee ~~the truth of consistency~~ (N) and also to take the necessary steps to ensure the truth of conditions FA, FC, ~~and ST~~, and S ~~and also I1, I2, and I3. All these tasks are ~~ to be carried out during the check of consistency which follows each decision made~~ In view of the very large number (perhaps 100,000) of times the check will be made ~~decisions involved~~ in a complete timetable, it is most important that this ~~coding~~ program be highly efficient.

The best way to ensure efficiency is to take cognisance of the fact that only ~~one~~ one decision has been taken since the last time the conditions were checked, and so it is necessary only to examine those data which ~~have been changed since the last time~~ might have been affected by this one change.

The nature of the change can be discovered by examining the current values of a and p; ~~and~~ ~~iff~~ if. $p \in T(a)$ the decision was an assignment; if $p \mathrel{\tilde{\in}} P(a)$ it was a cancellation.

~~In the program which follows~~

Since the derivation of the program follows so closely from the invariants which it is attempting to establish ~~restore~~, it is not worth explaining ~~it in detail~~ or annotating it in detail; all that is necessary is to reference the ~~approp~~ relevant invariant at the appropriate place where it becomes true again.

check consistency(a: Activity, p: Period): 14

begin if $p \in T(a)$ then note the most recent decision was $T(a) :+ p$;

begin if $p \tilde{\in} P(a)$ then goto impossible;

case times(a) − size(T(a)) of

{< 0: goto impossible; N1

= 0: for $p' \in P(a) - T(a)$ do forcedcancel(a) :+ p'} FC1

for $i \in$ requirement(a) do

case lives(i) − busy(i,p) of

{< 0: goto impossible; N2

= 0: for $a' \in$ users(i) do if $p \in P(a) - T(a)$ then forcedcancel(a) :+ p; FC2

if a ∈ spread then if size(T(a)∧periods in(day of(p))) ≥ 1 then goto impossible N3

for p ∈ periods in(day of(p)) do if p ≠ p then forcedcancel(a) :+ p FC3 (and N3)

if (a ∈ spread ∨ length(a) > 1) then

case times(a) ÷ length(a) − possdays(T(a)) of N4

{< 0: goto impossible;

= 0: for d ∈ possdays(T(a)) do for p' ∈ P(a) ∧ periods in(d) do forcedcancel(a) :+ p'} FC

else note the most recent decision was $P(a) :- p$;

begin case size(P(a)) − times(a) of

{< 0: goto impossible; N1

= 0: for $p' \in P(a) - T(a)$ do forcedassign(a) :+ p'; FA1

= 1: for $p' \in P(a) - T(a)$ do stiff(a) :+ p'}; ST1

for $i \in$ requirement(a) do

case possbusy(i,p) − lives(i) of

{< 0: goto impossible; N2

= 0: for $a' \in$ users(i) do if ~~a' ≠ a &~~ $p \tilde{\in} T(a)$ then forcedassign(a') :+ p; FA2

= 1: for $a' \in$ users(i) do if $a' \neq a$ & $p \tilde{\in} T(a)$ then stiff(a') :+ p} ST2

if (a ∈ spread ∨ length(a) > 1) then

case possdays(P(a)) − times(a) ÷ length(a) of N4

{< 0: goto impossible; daydetermined(a,P)

= 0: for d ∈ possdays(P(a)) do note daydetermined

case size(P(a)∧periods in(d)) − length(a) of

{< 0: goto impossible; comment never happens.

= 0: for p' ∈ P(a) ∧ periodsin(d) − T(a) do forcedassign(a) :+ p'; FA4

stiff(a) :+ p' ST4

else if p ∈ spread & size(P(a) ∧ periods in(day of(p))) = 1 then stiff :+ first(P(a) ∧ periodsin(dayof(p)))

} ST3

end check consistency;

if length(a) > 1 & T(a) & periodsin(dayof(p)) ≠ empty then

for p' ∈ P(a) − T(a) do stiff(a) :+ p' ST4

Check consistency:

begin

if length(a) > 1 **then**

begin intersection := periods in (day of (p)) ; union := empty;

for p′ ∈ starts(length(a)) ∧ day of (p) **do**

{ps := tuple(p′, length(a));

if T(a) ⊂ ps ⊆ P(a) **then** {intersection : ∧ ps;
union : ∨ ps}

}

~~**for**~~ **if** p′ ∈ intersection − T(a) **do** forcedassign :+ (a, p′) FA4

for p′ ∈ P(a) − union **do** forcedcancel :+ (a, p′) FC5

~~(**for** p′ ∈ P(a) ∧ periodsin(day of (p)) − T(a) **do** stiff :+ (a, p′)~~ ST

5. Data Representation.

Now that all the general decisions have been taken, and the program itself written in outline, the time has come to design representations of the data. This design must be made in the light of a knowledge of the most frequent operations on the data, and its likely size, and it seems that at this stage we have most of the necessary information. But it must also be made in the light of a knowledge of the ~~storage~~ characteristics of the computer on which the program will be run. We will therefore assume a 24-bit word length, and attempt to accommodate all the data in about ten thousand words.

One of the characteristics of this problem is the wide variability of some of the data sizes. For example, most activities require only one, two, or three items, but some activities may require twenty or more. Thus it will

5.1 Permanent Data.

The storage of data that does not vary during the execution of the main part of the program in general presents little difficulty. However, it does seem worth while to use packed representations to economise on storage. Where several ~~mappon~~ packed mappings ~~have~~ the same domain, it is usually worth while to combine them into a single mapping onto a Cartesian Product. The (permanent) mappings which have activities as their domain can be represented

$$A: \text{Activity} \rightarrow (\text{times}: [0..31]; \text{length} [0..3]; \text{spread}: \text{Boolean};$$
$$\text{requirements}(\text{first}: [0..255]; \text{if first} = 0 \text{ then long: Poolpointer}$$
$$\text{else short}: [1..4] \rightarrow [0..255]))$$

Where (1) we will add 1 to length (and times) before using them

(2) S is selected so that the short case of users fills the remainder of a computer word. (e.g. S = 4 on a 48-bit word length)

(3) In the short case the zero item-number will stand for the end of the sequence.

(4) In the long case, the poolpointer will index the pool at the point where the sequence of users actually begins.

(5) Each element of A requires two words, making 1500 words in all

(6) Most activities will have less than five items in their requirement, so little space in the pool will be needed.

often pay to allocate enough storage for the common case, and use a pointer to a common pool of storage in the exceptionally long case. We thus declare // type Poolpointer = 1..2047, and Pool: Poolpointer → Word where Word is a normal computer word.

(new para)

(new para)

The mappings which have Item as their domain can similarly be represented:

L: Item → (users: Poolpointer;
lives: [1..31]) ; ~~case ld of~~
~~word: (busy, possbusy: 1..48 → [0..15]), long: (Poolpointer))~~

(1) users points to the first of a (contiguous) sequence of 12-bit items in the pool, which list the activities which require this item. It is terminated by a zero; and will be typically about five words long.— 1250 in all.

~~(2) busy and possbusy occupy some 16 words for each item. Add one word for lives and users, and the total size of the table will be 4250 words.~~
~~(3) An item with more than fifteen users will present special problems.~~

Since there are less than 48 periods in the week, the obvious way of storing a Period set is ~~as~~ ~~be~~ a bitmap representation, using two words. Thus, the arrays T and P can readily be represented, as arrays of these word pairs:

T, P: 1..750 → twowords.

and will occupy 1500 locations each

The mapping starts will occupy about eight words, periodsin about 10 words, dayoff, 48 words;

The mappings forcedassign, forcedcancel, and stiff could be stored in the same way as simple contiguous stacks. The two forced sets together are very unlikely to exceed a hundred or two, ~~and although~~ and if they ever do, it would be extremely likely that one of them would generate a contradiction. Thus we would be quite justified in backtracking immediately in case either of these overflow. The stiff list may also get quite long; but if it overflows, a "scavenging" routine can eliminate those entries which ~~are also~~ represent decisions which are already taken or forced; and in the last resort, entries can simply be discarded from the list. ~~Thus.~~

~~forcedassign, forcedcancel, stiff: 1..100 →~~

~~(a: 1..1023, p: 1..48)~~

~~FT~~ fatop, fctop, ~~stifftop: integer~~

The two forced ~~lists~~ sets can be combined into a single ~~list,~~ stack by including a marker with each element to indicate from which set ~~each~~ it comes from. Thus we may ~~allocate~~ use ~~a single array, with to hold bot two~~ one ends of a single array to hold ~~both~~ the forced stack and the ~~stiff stack~~ other end to hold the stiff stack. This ~~takes advantage of~~ caters for the fact that the stiff stack will be largest when the forced stack is empty.

S: 1..300 → (a: 1..1023; p: 1..48,
d: {assign, cancel})

forcedtop, stifftop: Integer
forced top := 0;
stifftop := 301;

Thus the total number of data words required is:

Pool	2050
A	1500
L	4250
T	1500
P	1500
others:	400
	11 200

Which is just within our target of 12 000 words.

The next problem arises from consideration of the depth of the recursion. A complete time-table may require $750 \times 40 = 30\,000$ decisions, each of them involving a recursion. In most automatic implementations of recursion this would require an excessively large amount of storage. A clue to the solution arises from the realisation that the only reason for maintaining the stack is in the event of a backtrack. But we have deliberately set a limit (quite low) (perhaps 100) on the amount of backtracking we shall perform. Thus after a hundred recursions, the inactive end of the stack may be overwritten.

This means that the recursion in the program will have to be implemented by explicit stack manipulation on a ~~stack~~ stack which is treated as a "cyclic buffer". Each element of the stack needs to record the activity and period which is affected, and also whether the decision was forced or not.

stack: 1..limit → (a: Activity; p: Period; forced: Boolean)

4. Tight sets.

As mentioned above, the feasibility of the timetabling method depends critically on very early detection of inconsistency; for if an inconsistent decision is detected only after n subsequent assignments, it may take 2^n backtracking operations before the error is corrected. Furthermore, very powerful detection methods for forced decisions are vitally important, since latent inconsistencies can often be detected after a chain of forced decisions, which can comparatively cheaply be backtracked.

We are therefore interested in strengthening the conditions for consistency and forcing; and welcome a suggestion made in [2], namely the search for tight sets. We first note that the following is a necessary condition of consistency:

$$\forall i, \forall as: \text{Activity } \underline{\text{set}} \left(as \subset users(i) \supset size(\bigcup_{a \in as} P(a)) x lives(i) \geq \sum_{a \in as} times(a) \right)$$ ~~N6~~ N7

Proof. The activities in as will require a total of $\sum_{a \in as} times(a)$ unit periods of item i; and these must be taken during the periods of $\bigcup_{a \in as} P(a)$. But if there are too few such periods, this will be impossible.

If equality in ~~N6~~ N7 obtains, the set as is said to be tight in i.

$$as \in tight\ in(i) =_{df} as \subset user(i)\ \&\ size(\bigcup_{a \in as} P(a)) x lives(i) = \sum_{a \in as} times(a)$$

Note that the empty set and the full set (users(i)) are both trivially tight.

Now it is clear that if as is tight in i, all the period-units of item i will be occupied during $\bigcup_{a \in as} P(a)$ in satisfying the needs of the activities in as; and none can be spared during these periods for any activity outside as. We can thus derive a sufficient condition for forced cancellation of ~~a from p~~ a at p, where $p \in P(a)$:

$$\exists as, i \quad as \in tight\ in(i)\ \&\ a \in users(i) - as\ \&\ p \in \bigcup_{a' \in as} P(a') \quad \ldots \qquad FC7$$

Note that FC7 can be true only if as $\neq$ empty and as $\neq$ users(i); thus no cancellations are forced by these trivially tight sets.

4.1. Example.

In order to develop a deeper understanding of the nature of a tight set, we shall give an example of a tight set search. We shall initially confine attention to an item with only one life, and suppose that each of its user activities is to occur only one time. For the sake of illustration, we assume size (Period) = 10. Now each value of P(a) may be regarded as a Boolean vector of length 10, with 1 corresponding to each $p \in P(a)$, and 0 for each $p \tilde{\in} P(a)$:

eg P(a) = 1 0 1 1 0 1 0 0 0 0

signifies that periods 1, 3, 4 and 6 are members of P(a). Now the requirement that an item be kept fully busy during 10 periods implies that it must have exactly 10 users (since our simplification states that each user uses the item exactly once). Consequently the rows for each of the users may be extracted to form a square Boolean matrix.

Periods / users	p1	p2	p3	p4	p5	p6	p7	p8	p9	p10
a 1	1	0	1	1	0	1	0	0	0	0
a 2	1	0	1	1	0	1	0	0	0	0
a 3	0	1	0	1	1	0	0	0	0	0
a 4	1	0	0	1	0	1	0	0	0	0
a 5	1	0	1	0	0	1	0	0	0	0
a 6	1	1	1	0	1	1	0	0	0	0
a 7	1	1	1	1	0	1	1	1	0	0
a 8	1	1	1	1	0	1	1	1	0	0
a 9	1	1	0	0	0	1	1	0	1	0
a10	0	0	0	1	1	1	0	1	1	1

The first fact to note is that a1 to a6 form a tight set, with $\bigcup_{a} P(a)$ comprising the first six periods (columns). This may most clearly be recognised by noticing that there is a solid 6 x 4 rectangle of zeroes on its right of the 6x6 square on the major diagonal. The rule FC7 now permits cancellation of all 1's in the corresponding bottom left hand rectangle, leaving the following:

$$
\begin{array}{|cccccc|c|}
\hline
1&0&1&1&0&1& \\
1&0&1&1&0&1& \\
0&1&0&1&1&0& 0\\
1&0&1&1&0&1& \\
1&0&1&1&0&1& \\
1&1&1&0&1&1& \\
\hline
\multicolumn{6}{|c|}{0} & \begin{array}{cc|cc} 1&1&0&0\\ 1&1&0&0\\ \hline 1&0&1&0\\ 0&1&1&1 \end{array} \\
\hline
\end{array}
$$

Now it is clear that the complement of a tight set in the matrix is also tight - after the cancellation has taken place.

But a1 to a6 is not the only tight set in this matrix. For example in the bottom right hand square the first two activities form a tight subset, and the bottom left corner of that square should be blanked out.

$$
\begin{array}{c|cc|cc|}
 & p7 & p8 & p9 & p10; \\
\hline
a7 & 1 & 1 & 0 & 0 \\
a8 & 1 & 1 & 0 & 0 \\
\hline
a9 & 0 & 0 & 1 & 0 \\
a10 & 0 & 0 & 1 & 1 \\
\hline
\end{array}
$$

Now a9 has only one possible time when it can be assigned; this is in fact a special case of a tight set, and justifies yet another cancellation:

$$
\begin{array}{c|cc|}
 & p9 & p10 \\
\hline
a9 & 1 & 0 \\
a10 & 0 & 1 \\
\hline
\end{array}
$$

Of course, it cannot in general be expected that the rows or the columns of a tight set will be contiguous, as they were in the cases described above. However, if there is a tight set, its rows and columns could be made contiguous by suitable interchange. For example, in the top left square of the original matrix, activities a1, a2, a4 and a 5 form a tight set, with a union containing p1, p3, p4,p6. By interchange of rows and columns we obtain:

	p1	p3	p4	p6	p2	p5
a1	1	1	1	1	0	0
a2	1	1	1	1	0	0
a4	1	0	1	1	0	0
a5	1	1	0	1	0	0
a3	0	0	1	0	1	1
a6	1	1	0	1	1	1

Thus four ones in the bottom left-hand corner of this square should be cancelled.

This reasoning applies also if some activities are intended to occur more than once. An activity which is intended to occur n times may be regarded as equivalent to n identical rows, each of which is to occur once (thus a4 and a5 in our example might be a single activity; also a7 and a8).

If an item has multiple lives, say n lives, the Boolean matrix will be n times as long as it is wide; and the "squares" along the diagonal will be rectangles, also n times as long as wide. Apart from this, the reasoning given above applies also to this case.

Since searching for tight sets is an extremely laborious business, we do not wish to do it too often. In particular, we can avoid doing it in the case of forced decisions; instead we wait until all forced decisions have been taken and the next ~~xx~~ unforced decision is due; consequently the tight set search should occur just before "select unforced (a,p)", and if one or more forced decisions have been uncovered by the tight set search, one of these should be selected.

<u>before</u> select unforced(a,p) <u>do</u> {tight set search; select forced(a,p)}

4.2. Scanning for tight subsets.

The search for tight subsets of a given set ~~ka~~ ts can be made only by considering each subset individually. The scan of all subsets can conveniently be done by a recursive procedure "scan". This procedure is designed to exit normally if there are no tight subsets, and to jump to a "success" label when it finds one, having noted the resulting forced cancellations. The procedure ~~in fact~~ takes three parameters:

as: a subset of ts which has already been ~~chosen~~ searched.

$$ps = \bigcup_{a \in as} P(a)$$

$$n = \sum_{a \in as} \text{times}(a)$$

The procedure will in fact only examine supersets of the set as which has already been chosen; and it does this by adding in turn each member of (ts - as) to the set as and entering itself recursively. But "scan" must also remember not to accept ts itself as a tight subset. We thus obtain the procedure:

```
scan (as,ps,n)  recursive procedure

compare size(ps)xlives(i)with n

 if < then go to impossible
 if  =λ as ≠ ts, then { for  a ∈ ts-as do
                        for  p ∈ P(a) ∧ ps do  forced cancel:+(a,p);
           if forcedcancel ≠ empty then go to success }
           else
 for  a ∈ ts-as  do  scan (as+a,ps ∨ P(a),n+times(a))
```

In practice it will be highly advantageous to write this little procedure in machine code, since it is effectively the "inner loop" of the entire timetabling process.

4.3 Reduction of Inefficiency.

In view of the extremely time-consuning nature of the "scan" procedure, it is necessary to take firm steps to reduce the frequency with which it is called, and the size of the sets on which it is to operate. Suppose the set of users of an item i have suffered no cancellation since they were last scanned for tight subsets. Then there is no point in making a further scan. If one or more user activities have suffered cancellation since the last scan, then any tight subsets within users(i) must contain at least one of those cancelled activities. This suggests that we keep an account of all activities cancelled since the last scan, together with all items which need rescanning.

```
            changed: Activity set   initially empty
            needscan:  Item    set  initially empty.

whenever   P(a):-p  do   {changed:+a; needscan:+requirement(a)}
```

Now we can code:

```
            tight set search:

          begin    for i ∈ needscan do
                         begin   ts:= users(i);
                                    scan ts;
                                    needscan:-i
           end;
           changed:=empty;
           success: end
where scan ts: for  a ∈ ts ∧ changed do {scan(unitset(a),P(a),times(a));
                                                                 ts:-a }
```

Further efficiency can be gained if we remark that:

(1) cancellation can never cause a tight set to become nontight
(2) any new tight set must be wholly contained in some previously existing tight set.

Point (2) may be established by visualising the diagonal form of tight sets. Since it is much more efficient to scan two separate sets than their union, it will pay us to record any tight sets as we discover them, and confine future searches to these individual tight sets. For this purpose we introduce a variable

tss: Item→Activity ~~set~~ sequence initially tss(i) = unitsequence(users(i))

which maps each item onto a sequence of sets which are ~~whose elements~~ are tight in that item.

Whenever a tight set is discovered, both it and its ~~relative~~ complement with respect to the set being searched must be added to the appropriate tight set sequence

<u>in</u> scan <u>before</u> <u>go</u> <u>to</u> success <u>do</u>

tss(i):⌢as; tss(i):⌢(ts-as)

Whenever backtracking occurs, any tight sets discovered as a result of a changed decision must be removed. Thus the item for which a tight set has been discovered must be recorded in a variable local to progress

tightset found: <u>Boolean</u> initially <u>false</u>

item with tightset:Item

in scan <u>before</u> <u>go</u> <u>to</u> success <u>do</u>

tightset found:=true; item with tight sets:=i

<u>before</u> impossible <u>do</u>

<u>if</u> tightset found <u>then</u> remove two elements from tss(item with tight set)

Now the tight set search for item i involves scanning through all sets in tss(i) and selecting those which contain at least one changed activity. However, once a tight set containing a given activity has been scanned, there is no point in scanning a subsequent tight set containing that activity. We therefore keep in a variable "changed" that subset of the users(i) which have been changed but not yet dealt with.

Now we can code a more efficient version of

```
    tightset search:
    begin  for  i ∈ need scan do
        begin  new  ichanged initially  changed ∧ users(i);
            for  ts in tss(i) while ichanged ≠ empty do
                for  a ∈ ts ∧ ichanged do
                {scan ts;  ichanged:-a} ;
            needscan:-i
    end;
    changed:=empty;
  success:end
```