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:

(a) to assign an activity to 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 previous 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 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
```

(both assignments and cancellations)

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

```
count: Integer initially 0
```

```
size (Period) × one (Activity)
```

to be decided: Integer initially \( \sum \) size (possible (a) - timetable (a))

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{align*}
\text{begin progress recursive procedure} \\
\quad \text{check consistency;} \\
\quad \text{if inconsistent then go to impossible;} \\
\quad \text{if count \(C\) then go to printout;} \\
\quad \text{select suitable \((a,p)\);} \\
\quad \text{try assignment\((a,p)\);} \\
\quad \text{try cancellation \((a,p)\);} \\
\text{impossible:} \\
\text{end progress;} \\
\text{print failure message \& stop;} \\
\text{printout:} \quad \text{end}
\end{align*}
\]

3.2. Assignment and cancellation.

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

\[
T_P: \text{Activity} \rightarrow \text{Period set}
\]

where \(T(a)\) is the set of periods which have been assigned to a \((\text{initially set equal to timetable})\)  \(\text{initially empty}\) and \(P(a)\) is the set of periods which have not been cancelled for a \((\text{initially set equal to possible})\)  \(\text{initially the full set}\).

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

\[
\forall a. \ T(a) \subseteq P(a)
\]

If \(p \in P(a) - T(a)\), this means that no decision has yet been made about assigning or cancelling a from \(p\). 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 they are equal, at which point each other, and therefore to the final timetable. The program will output the value of \(T\)
try assignment(a,p):
begin T(a):=p;count:+1;progress;count:-1;T(a):=pend
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:

\[ C \subseteq \neg a \subseteq (T(a) \cap \text{timetable}(a) \cap P(a)) \subseteq N. \]

The following necessary conditions can readily be proved:

- \( \forall a \; \text{size}(T(a)) \leq \text{times}(a) \leq \text{size}(P(a)) \) (N1)
- \( \forall i,p \; \text{busy}(i,p) \leq \text{lives}(i) \leq \text{possbusy}(i,p) \) (N2)
  
  where \( \text{busy}(i,p) = \text{size}\{a \in \text{users}(i) \mid p \in T(a)\} \) (I1)
  and \( \text{possbusy}(i,p) = \text{size}\{a \in \text{users}(i) \mid p \in P(a)\} \) (I2)
- \( \forall a \; \text{spread}(a) = \text{size}(\text{possdays}(a)) \times \text{length}(a) \) (N3)
  
  where \( \text{possdays}(a) = \{d \mid \text{periods in } (d) \neq \text{empty}\} \)

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:

- \( \text{olda}: \text{Activity} \); \( \text{oldp}: \text{Period} \); \( \text{action}: \{\text{any, cancel}\} \)

afters T(a):=p do \( \text{olda}:=a; \text{oldp}:=p; \text{action}:=\text{any} \)

afters P(a):=p do \( \text{olda}:=a; \text{oldp}:=p; \text{action}:=\text{cancel} \)
The following necessary conditions can be 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 condition is sufficiently obvious to require no proof.

\[ \forall a:\ \text{size}(T(a)) \leq \text{twins}(a) \leq \text{size}(P(a)) \] \hspace{1cm} N1

\[ \forall a, i:\ \text{busy}(i, p) \leq \text{twins}(i) \leq \text{possbusy}(i, p) \] \hspace{1cm} N2

where \( \text{busy}(i, p) = \text{size} \{ a | a \in \text{users}(i) \land p \in T(a) \} \)

and \( \text{possbusy}(i, p) = \text{size} \{ a | a \in \text{users}(i) \land p \in P(a) \} \)

\[ \forall a, d:\ \text{a is spread} \supset \text{size}(T(a) \cap \text{periods}(d)) \leq 1 \] \hspace{1cm} N3

\[ \forall a, d:\ \text{a is spread} \lor \text{length}(a) > 1 \supset \text{possdays}(T(a)) \times \text{length}(a) \leq \text{twins}(a) \leq \text{possdays}(P(a)) \times \text{length}(a) \] \hspace{1cm} N4

\[ \forall d:\ \text{length}(a) > 1 \supset \text{twins}(a) \leq \text{size} \{ d \mid \text{possstaple}(a, d) \neq \text{empty} \} \times \text{length}(a) \] \hspace{1cm} N5

where \( \text{possstaple}(a, d) = \{ ps \mid T(a) \cap ps \subseteq P(a) \land \text{periods}(d) \} \)

\& \( ps = \text{tuple}(\text{length}(a), \text{first}(ps)) \)

\& \( \text{first}(ps) \in \text{starts}(a) \)

The conjunction of these five conditions is known as N.
The correctness of the program depends on the fact that when the indicated jump to printout is made, the consistency check which guarantees that truth of the constraint C. When count = size (Period) \times 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 as easily derived from the definition of N, and may be postponed to a later stage.
3.4. Select suitable \( \langle \alpha, \rho \rangle \).

The conjunction of these six conditions is known as N5. The conjunction of these six conditions is known as N5.

Another factor which is critical to the efficiency of timetable construction is the judicious selection of the next decision to take. An obvious 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.
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 cancelation(a, p); go to impossible}

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

Each of the following is a sufficient condition for assignment of p to a, where p ∈ P(a) - T(a):

size (P(a)) = times(a) \( \Xi i \) ( requirement(a) & lives(i) = possbusy(i, p)) FA1

\( \Xi i \) ( requirement(a) & lives(i) = possbusy(i, p)) FA2

day(p) & tie(a) \( \Xi i \) ( requirement(a) & lives(i) = possbusy(i, p)) FA3

day determined(a) & size(P(a) \ A periods in(day of (p))) = length(a) FA4

length(a) > 1 & a ∈ spread & p ∈ \ A possstuples(a, p)

where possstuples(a, p) =

\{ ps: period set | T(a) ∈ ps ∈ P(a) \ A periods in (day of (p)) & ps = tuple(length(a), first(ps)) & first(ps) ∈ starts(a) \}

Each of the following is a sufficient condition for cancellation of a at p, where p ∈ P(a) - T(a):

size(T(a)) = times(a)
\( \Xi i \) ( requirement(a) & lives(i) = busy(i, p)) FC1

a ∈ spread & size(T(a) \ A periods in(day of (p))) = length(a) FC2

length(a) > 1 & a ∈ spread & p ∈ \ A possstuples(a, p) FC3

\( a', a ∈ tie(a) \ A periods in (day of (p)) ≠ empty). FC4

\( a', a ∈ tie(a) \ A periods in (day of (p)) ≠ empty) FC5

As before, the detailed checks and decision procedures can be easily derived, and the detailed coding may be postponed.
The following are sufficient conditions for cancellation of \( a \) at \( p \),
where \( p \in P(a) - T(a) \):

\[ \text{twice}(a) \leq \text{size}(T(a)) \]

\[ \text{If } i \in \text{requirement}(a) \& \text{ status}(i) = \text{busy}(i,p) \]
\[ \text{and determined}(a) \& \text{dayof}(p) = \text{posdays}(T(a)) \]
\[ (a \in \text{spread} \& \text{length}(a) > 1) \& \text{twice}(a) = \text{posdays}(P(a)) \times \text{length}(a). \]

\[ (a \in \text{spread} \& p \in T(a) \& p \in \text{posdays}(\text{dayof}(p)) \]
\[ \text{length}(a) > 1 \& p \notin \bigcup \text{posstuples}(a, \text{dayof}(p)) \]

When an inconstancy 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 try assignment; and try cancellation 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.
35, 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 the 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 decision-maker will tend to assign to areas in which there is least freedom of choice in making decisions, the areas which the human decision-maker would regard as "sticky," and which he would concentrate his early attention.

More precisely, for each \((a, p)\) we can estimate roughly how many forces would result from an 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 choice first of the other decision.
\[ \text{times}(a) \times \text{size}(T(a)) = 1 \lor \text{size}(P(a)) - \text{twins}(a) = 1 \]

If \( e \) is requirement \( (a) \) and \( \text{lines}(i) \times \text{busy}(i,p) = 1 \),
\[
\text{posbusy}(i,p) - \text{lines}(i) = 1
\]
(no corresponding condition)

day determined \( (a) \) and \( \text{size}(P(a) \times \text{periods in (day of (p)))} \)
\[
- \text{length}(a) = 1
\]

\( \text{length}(a) > 1 \) and \( T(a) \times \text{periods in (day of (p))} \neq \text{empty} \).

ST5 ensures that a partially assigned multiple period is always regarded as stiff.
We therefore introduce a set:

\[
\text{Activity} \rightarrow \text{Period set}
\]

stiff: \((\text{Activity} \times \text{Period})\) set

into which decisions are placed when they are recognized to be stiff. The main criterion of stiffness is that the decision now is likely to result in further forced decisions, and this can be detected in during the search for forced decisions.

The most obvious criterion for stiffness of \((a, p)\), where \(p\) & \(P(a) - T(a)\) are:

\[
\text{size}(P(a)) - \text{times}(a) = 1
\]  
\[
\text{Ei } (i \in \text{requirement}(a) \& \text{posbusy}(i, p) - \text{lines}(i) = 1)
\]

\[
d \in T_{day}(a)
\]

\[
\text{Ed. pod} \& \text{daydetermined}(a, d) \& \text{size}(P(a) \times \text{period in}(d)) - \text{length}(a) = 1
\]

\[
\text{times}(a) - \text{size}(T(a)) = 1
\]

\[
\text{Ei } (i \in \text{requirement}(a) \& \text{lines}(i) - \text{busy}(i, p) = 1
\]

The conjunction of these conditions will be known as ST.

These ST conditions may most readily be detected during the check on consistency; again, the detailed planning 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. Such decisions must be removed from the stiff set as they are encountered.

```
select unforced (a, p):

if stiff ≠ empty then (a, p) := one of (stiff) x such that p ∈ stiff (a);
repeat begin
(a, p) := one of (stiff) x stiff := (a, p) end
until p ∈ P(a) - T(a) ∨ stiff = empty
```

```
if stiff = empty then select unstuff (a, p)
```

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 into stiff even when they do not satisfy $S^f$ but anyway it seems a good idea to recognize as stiff those forced decisions which are already known to lead to inconsistencies.
At some stage, it is necessary to decide how to select non-stiff decisions.

The easiest way of doing this is to allocate to each activity a score indicating its "a priori" difficulty, and to select from a sorted list of activities which activity with highest score will always be selected. A scoring formula will also give highest scores to multiple-period activities and to least activities, the score of an activity a medium score to activities that have to be spread, and a zero score to the free periods. The exact weighting coefficients are somewhat arbitrary, and may need to be adjusted in the light of experience.

In addition to a static allocation of priority, it is worth while to keep a record of activities and periods which during the case have become stiff, and that stiff during the course of time-stabilizing. It might be worth while to maintain several grades of stiffness, from the most stiff (N) to the least stiff (1).

\( \text{Stiff: } C \rightarrow (\text{Activity} \times \text{Period}) \) set

From within
\begin{align*}
\text{while} \ (P(x) - T(x)) = \text{empty} \ \text{do} \\
\end{align*}
3.5. Premature Termination

A very rough

The program as designed so far will always find a nonterminal if there is one, and report failure if there is not. However, a very cursory calculation shows that the backtrack may take a wholly impractical amount of time. Therefore, it 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 of a good chance of proceeding to 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 nonterminal will be printed out.

A good measure of the difficulty of the situation is to keep a count of the amount of backtracking that has taken place, or the number of decision levels. This can be estimated by...
comparing the current value of the count with
the highest value it has ever achieved. Thus,
we need a variable

\[
\text{countmax: Integer initially 0}
\]

\[
\text{and after}
\]
\[
\text{after count t do if count > countmax then countmax := count}
\]
\[
\text{after count t do if countmax - count < limit do}
\]
\[
\text{print T; stop}\
\]
4a. Detailed Coding Programming of Consistency Check
4b. Details of Consistency Check

The time has now come to carry out the combined checks of the tasks of detailed coding (which have been postponed until the previous sections) and the consistency check which is to ensure consistency of the models (N) and also to take the necessary steps to ensure the truth of conditions FA, FC, and S,
and also T1, T2, T3.

All these tasks are necessary to be carried out during the check of consistency which follows each decision made. In view of the very large number of 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 if 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 change).
The nature of the change can be discovered by examining the current values of \( a \) and \( p \); and if \( p \in T(a) \) the decision was an assignment; if \( p \in P(a) \) 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, it is not worth explaining it in detail or amodating it in detail; all that is necessary is to reference the relevant invariant at the appropriate place where it becomes true again.
check consistency(a: Activity, p: Period);
begin if p ∈ T(a) then note the most recent decision was T(a)=p;
begin if p ∈ P(a) then goto impossible;
   case times(a) = size(T(a)) <
   { ≤ 0: goto impossible;
   = 0: for p ∈ T(a) do forcedcancel(a)=p’;}
   for i ∈ requirement(a) do
      case links(i) = busy(i,p) of
      { ≤ 0: goto impossible;
      = 0: for a’ ∈ uses(i) do if p ∈ T(a) then forcedcancel(a)=p;
      = 1: for p ∈ P(a) do if i ∈ P(a) then forcedcancel(a)=p;}
end begin case size(P(a)) = times(a) of
   { ≤ 0: goto impossible;
   = 0: for p ∈ P(a) do forcedassign(a)=p’;
   = 1: for p ∈ P(a) do if i ∈ P(a) then forcedcancel(a)=p’;}
end if a ∈ spread & length(a) > 1 then
   case possday(?a) = times(a) - length(a) of
   { ≤ 0: goto impossible;
   = 0: for d ∈ possdays(P(a)) do note daydetermined;
   = 1: daydetermined(a,p) }
begin
if $\text{length}(a) > 1$ then
    begin
        intersection := $\text{day of}(p)$;
        union := empty;
        for $p' \in \text{starts}(\text{length}(a)) \land \text{day of}(p)$ do
            $\{ps := \text{tuple}(p', \text{length}(a))\}$
            if $T(a) \subseteq ps \subseteq P(a)$ then
                begin
                    intersection := $\cup ps$;
                    union := $\cup ps$;
                end
            end
    end
end
for $p' \in \text{intersection-}T(a)$ do
    forcedassign := $+(a, p')$
for $p' \in P(a) - \text{union}$ do
    forcedcancel := $+(a, p')$
for $p' \in P(a)$ s.t. $\text{permutations}(\text{day of}(a)) = T(a)$ do
    shift := $+(a, p')$
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. The 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 program is the wide variability of some of the data sizes. For example, most activities require one, two, or three<br>things, but some activities may require twenty or more. Thus it will be

5.1 Permanent Data.

The storage of data that does not vary during the execution of the main part of the program is called permanent. However, it does seem worthwhile to use packed representations to economize on storage. Where several mappings are used, it is usually worth while to combine them into a single mapping onto a Cartesian Product. //The mappings which share activities on their domain can be represented

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

Where:
1. we will add 1 to length (before naming 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. The second item-number will stand for the end of the sequence.
4. In the long case, the pool-pointer will indicate 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 few items in their requirement, so little space in the pool will be used.

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 then declare //type Pool-pointer = 1..2047; and Pool: Pool-pointer-Adress.
The mapping, which have Item as their domain, can similarly be represented:

\[
L : \text{Item} \rightarrow (\text{users} : \text{Poolpointer})
\]

\[
| \text{dirs} : [1..3] | \text{pool} \in \text{Pool} | \text{size} : [1..48] \rightarrow [0..15] | \text{long} : (\text{Poolpointer}) |
\]

1) users point to the first of a 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 possibly occupy some 16 words for each item. Add one word for users 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 to have a bitmap representation, using two words. Thus, the arrays T and P can readily be represented, as arrays of these word pairs:

$$T, P : 1..750 \rightarrow \text{twowords}.$$ and will occupy 1500 locations each.

The mapping sheets will occupy about eight words, periodin.
about 10 words, dayof, 48 words.

The mappings forced assign, forced cancel, and stuff could be stored in the same way as simple contiguous stacks. The two forced sets 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 backing out immediately in case either of these overflows. The stuff list may also get quite long; but if it overflows, a "rearranging" routine can eliminate these entries which are also the terms which are already taken or forced; and in the last resort, entries can simply be discarded from the list.

```
iferrancy, forced cancel, stuff: 1..100 ->

(d: 1..1023, p: 1..48)
```

if stop, if top, if stop, integer
The two forced lists can be combined into a single list, by including a marker with each element to indicate from which set each element comes from. Thus we may allocate use a single array, with one element at two ends of the array to hold both the forced stack and the stiff stack. Other end to hold the stiff stack. This takes advantage of the fact that the stiff stack will be largest when the forced stack is empty.

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

forcedtop, stiftop: \text{Integer}
forscedtop := 0;
stifftop := 301;
Thus the total number of data words required is:

<table>
<thead>
<tr>
<th>Pool</th>
<th>2050</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1500</td>
</tr>
<tr>
<td>L</td>
<td>4250</td>
</tr>
<tr>
<td>T</td>
<td>1500</td>
</tr>
<tr>
<td>P</td>
<td>1500</td>
</tr>
<tr>
<td>Others</td>
<td>400</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>11200</strong></td>
</tr>
</tbody>
</table>

Which is just within our target of 12000 words.

The real problem arises from consideration of the depth of the recursion. A complete Trie-table may require $750 \times 40 = 30000$ decisions, each of these involving a recursion. In most automatic implementations of this would require an excessively large amount of storage. A clue to the solution arises from the realization that the only reason for maintaining the stack is the need for backtracking. In most cases, we have deliberately used a limit 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 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.
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 \( m \) \( n \) 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:

\[
\forall i, \forall a \in as: \text{Activity set } \left( \sum_{a \in as} \text{size} \left( \bigcup_{a \in as} P(a) \right) \times \text{lives}(i) \geq \sum_{a \in as} \text{times}(a) \right) \geq \sum_{a \in as} \text{times}(a)
\]

Proof. The activities in \( as \) will require a total of \( \sum_{a \in as} \text{times}(a) \) 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 \( \geq \) obtains, the set \( as \) is said to be tight in \( i \).

\[
\text{as is tight in } (i) \iff \sum_{a \in as} \text{size} \left( \bigcup_{a \in as} P(a) \right) \times \text{lives}(i) = \sum_{a \in as} \text{times}(a)
\]

Note that the empty set and the full set \( \text{users}(i) \) 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 \( p \), where \( p \in P(a) \):

\[
\exists as, i : \text{as is tight in } (i) \land \sum_{a \in as} \text{users}(i) - as \cap p \cup P(a') \land FC7
\]

Note that FC7 can be true only if \( as \) is not empty and \( as \) is \( \text{users}(i) \); 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 \( p \notin P(a) \):

\[
\text{eg } P(a) = 1 \ 0 \ 1 \ 1 \ 0 \ 1 \ 0 \ 0 \ 0 \ 0
\]

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.

<table>
<thead>
<tr>
<th>Users</th>
<th>p1</th>
<th>p2</th>
<th>p3</th>
<th>p4</th>
<th>p5</th>
<th>p6</th>
<th>p7</th>
<th>p8</th>
<th>p9</th>
<th>p10</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>a2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>a3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>a4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>a5</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>a6</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>a7</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>a8</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>a9</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>a10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
The first fact to note is that all a's form a tight set, with U \( P(a) \) 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:

\[
\begin{array}{cccc}
1 & 0 & 0 & 1 \\
1 & 0 & 1 & 0 \\
0 & 1 & 0 & 1 \\
1 & 0 & 1 & 0 \\
1 & 0 & 1 & 0 \\
1 & 1 & 0 & 1
\end{array}
\begin{array}{c}
0
\end{array}
\]

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

But a_1 to a_6 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.

\[
p7p8p9p10;
\]

\[
\begin{array}{c}
a7 \ 1 & 1 & 0 & 0 \\
a8 \ 1 & 1 & 0 & 0 \\
a9 \ 0 & 0 & 1 & 0 \\
a10 \ 0 & 0 & 1 & 1 \\
\end{array}
\]

Now a_9 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:

\[
p9p10
\]

\[
\begin{array}{c}
a9 \ 1 & 0 \\
a10 \ 0 & 1 \\
\end{array}
\]

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 a5 form a tight set, with a union containing p1, p3, p4, p6. By interchange of rows and columns we obtain:

<table>
<thead>
<tr>
<th></th>
<th>p1</th>
<th>p2</th>
<th>p3</th>
<th>p4</th>
<th>p5</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>a2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>a3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>a4</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>a5</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>a6</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

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 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.

before select unforced(a,p) do { tight set search; select forced(a,p) }
4.2. Scanning for tight subsets.

The search for tight subsets of a given set $\mathbf{ts}$ 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:

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

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

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)x(times(i)) with n

if < then go to impossible

if = \emptyset then {for a \in ts-as do
    for p \in \text{p}(a) \land ps \land \text{forced cancel}(a, p);}
    if forced cancel then go to success
else
    for a \in ts-as do scan (ts, ps+a, ps' \land \text{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-consuming 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
needsca:n: Item set initially empty.

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

Now we can code:

tight set search:

begin for $i$ $\in$ needscan do
            begin $ts:=$ users($i$);
                scan $ts$;
                needsca:n: $-$i
            end;
            changed: $=$empty;
            success: end
where scan $ts$: for $a$ $\in$ $ts$ \& 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

\[ \text{tss}: \text{Item} \rightarrow \text{Activity set sequence} \, \text{initially} \, \text{tss}(i) = \text{unitsequence(users}(i)) \]

which maps each item onto a sequence 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

\[
\text{in scan before go to success do}
\text{tss}(i) := \text{as; tss}(i) := (\text{ts-as})
\]

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

\[ \text{tightset found: Boolean initially false} \]

\[ \text{item with tightset: Item} \]

\[
\text{in scan before go to success do}
\text{tightset found:=true; item with tight sets:=i}
\]

before impossible do

\[
\text{if tightset found then remove two elements from tss(item with tight set)}
\]

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 };
      end;
    end;
  changed:=empty;
end;

success:end