Noname manuscript No.
(will be inserted by the editor)

Modelling History in Nurse Rostering

Jeffrey H. Kingston

Received: date / Accepted: date

Abstract One strand of current research into automated timetabling is the development of standard data formats—formats that researchers can use to exchange data and verify each others' solutions. Nurse rostering is lagging in this respect: each step forward seems to bring a new format. Standardization requires clarity in the underlying model, and this paper is a contribution to clarity in the area of *history*: how solutions to previous instances affect the current instance. The paper addresses several issues, including avoiding double counting of penalties, constraining consecutive busy times, and completeness. The work is implemented within the XESTT model of nurse rostering.

Keywords Nurse rostering · Data models · XESTT

1 Introduction

One strand of current research into automated timetabling is the development of standard data formats—formats that researchers can use to exchange data and verify each others' solutions. Nurse rostering is lagging in this respect: each step forward seems to bring a new format.

Standardization requires clarity in the underlying model. This is largely present in nurse rostering: concepts such as shifts, nurses, and constraints are clear, and consistent through the literature. There is one area, however, which is less clear, although good progress has been made recently: *history*, or how solutions to previous instances affect the current instance.

History has been discussed for over 15 years [1], but this paper mainly draws on two strong recent works. The Second International Nurse Rostering Competition [2,3], referred to here as 'the competition', brought history to a

J. Kingston (`http://jeffreykingston.id.au`)
School of Information Technologies, The University of Sydney, Australia
E-mail: jeff@it.usyd.edu.au

wide audience in a concrete form. The other work [13] adjusts the usual integer programming formulation of nurse rostering to define history precisely.

The main contribution of this paper is as follows. All previous work known to the author has treated each constraint, or each class of similar constraints (counters, patterns, etc.), as a separate problem for history, needing a separate analysis. Because there are many constraints, there is no criterion for deciding when the work is complete. This is noticeable in [13], for example, where no claim of completeness is made, even though the work is thorough and there is little doubt that it is, for all practical purposes, complete.

This paper, in contrast, defines and implements history in the framework of the XESTT nurse rostering data format [9,10]. All relevant constraints are formulated using just two kinds of constraints, the *cluster busy times constraint* and the *limit active intervals constraint*. Although the universality of these constraints is only a hypothesis, good evidence for it exists. By applying history to these two constraints, this paper does all the analysis at once; there is less analysis to do, and good evidence that it is complete.

Section 2 describes XESTT and the cluster busy times and limit active intervals constraints. Section 3 defines the history problem and applies the well-known method of counter start and counter remainder values to cluster busy times constraints. Later sections address other issues related to history: avoiding double counting of costs (Section 4), understanding heuristically derived constraints (Section 5), optimizing sequence constraints (Section 6), and using preassigned events to represent history (Section 7). An appendix presents the formulas that form the basis of the author's implementation (Section 9).

2 XESTT

This section introduces the XESTT nurse rostering data format and its cluster busy times and limit active intervals constraints, in enough detail for the purposes of this paper. For more detail, see [9]; and for the complete story, see the specification pages of [7].

XESTT is an extension of the well-known XHSTT high school timetabling data format [7]. Its name reflects this origin, with 'ES' for 'employee scheduling' replacing 'HS' for 'high school'.

An XESTT instance of the nurse rostering problem contains four main kinds of entities: times, resources, events, and constraints.

A *time* represents an indivisible interval of time in which events may occur.

A *resource* represents something that attends an event. In nurse rostering, all resources are nurses.

An *event* represents an indivisible piece of work, and contains a starting time, an integer duration (number of times), and any number of resources, which are considered to be busy attending the event from the starting time for the duration. The duration is a fixed constant, but the starting time and the resources may either be preassigned or left open for the solver to assign.

Each shift is represented by an event with its own unique, preassigned starting time, and duration 1. This is artificial, but it works well in practice, because nurse rostering constraints tend to limit the number of shifts worked, whereas in high school timetabling, where XESTT originated, they limit the number of busy times. Giving each shift duration 1 unifies the two models. For example, the requirement that a nurse work at most one shift per day is expressed by requiring the nurse to be busy for at most one time on each day.

A shift's resources represent the demands for nurses for that shift. They are usually left open for the solver to assign, although XESTT does allow any subset of them to be preassigned. Constraints, given separately, may be used to specify that some or all of the nurses should have particular skills.

For definiteness, examples will assume that there are three shifts, and hence three times, per day. Time names consist of a week number, a short weekday name, and an index within the day. For example, the three times of the Monday of the first week are `1Mon1`, `1Mon2`, and `1Mon3`. Their associated events may be named `E-1Mon1`, `E-1Mon2`, and `E-1Mon3`.

Finally come the *constraints*, hard and soft, which are rules that solutions should obey. Violations of hard constraints are usually interpreted to mean that a solution is infeasible. In XESTT they contribute a penalty to a hard cost total. Soft constraint violations contribute a penalty to a soft cost total.

Nurse rostering constraints fall into two classes. *Cover constraints* are the demands of shifts for certain numbers of nurses, often with nominated skills. XESTT offers *assign resource constraints*, *prefer resources constraints*, and *limit resources constraints* for them. Cover constraints are always independent of history, so nothing further is said of them here.

Resource constraints are constraints on nurses' timetables: limits on their total workload, on consecutive night shifts, and so on. It turns out that all the problems created by history arise from its interaction with resource constraints.

The key insight that this paper takes from XESTT is that many resource constraints, very likely all that arise in practice, have a common structure, which is embodied in the XESTT *cluster busy times* and *limit active intervals* constraints. The rest of this section is devoted to these important constraints.

One of these constraints may apply independently to many resources (for example, to all nurses who share a given contract). However, for simplicity of presentation, it is assumed here that one constraint applies to one resource.

A cluster busy times constraint contains any number of *time groups*, which are arbitrary sets of times. Associated with each time group is a *polarity*, whose value may be either `positive` or `negative`. A time group is said to be *positive* or *negative* depending on its associated polarity. When presenting constraints, an asterisk will be used to indicate negative polarity. For example,

```
{1Mon1, 1Mon2, 1Mon3}
{1Tue1, 1Tue2, 1Tue3}*
{1Wed1, 1Wed2, 1Wed3}
```

represents three time groups, the second of which is negative.

A time is *busy* for a resource when the resource attends an event at that time, and *free* for the resource otherwise. A time group is busy for a resource when it contains at least one busy time for the resource, and free for the resource otherwise. Within a cluster busy times constraint, a time group is *active* when either it is busy for the constraint's resource and positive, or free for the constraint's resource and negative. Otherwise it is *inactive*.

A cluster busy times constraint also contains a Boolean `required` attribute, saying whether the constraint is hard or soft, a non-negative integer `weight` attribute, and non-negative integer `minimum` and `maximum` attributes called its *limits*. The constraint is violated when the number of its time groups which are active is above the maximum limit or below the minimum limit. The cost of a violation is the amount by which the number of active time groups exceeds the maximum or falls short of the minimum (or occasionally the square of that number), multiplied by the weight.

Here are a few examples of how typical constraints may be formulated as cluster busy times constraints. To say that a resource may work at most one shift on day `1Mon`, define a cluster busy times constraint with time groups

```
{1Mon1}
{1Mon2}
{1Mon3}
```

and maximum limit 1. To impose this constraint each day, add one constraint for each day. To require a resource to work between 16 and 20 days in four weeks, define a cluster busy times constraint with time groups

```
{1Mon1, 1Mon2, 1Mon3}
{1Tue1, 1Tue2, 1Tue3}
...
{4Sun1, 4Sun2, 4Sun3}
```

and limits 16 and 20. To limit the number of busy weekends, use time groups

```
{1Sat1, 1Sat2, 1Sat3, 1Sun1, 1Sun2, 1Sun3}
{2Sat1, 2Sat2, 2Sat3, 2Sun1, 2Sun2, 1Sun3}
{3Sat1, 3Sat2, 3Sat3, 3Sun1, 3Sun2, 3Sun3}
```

and so on.

Negative time groups help with unwanted patterns. For example, to say that free days must occur in sequences of at least 2 is to say that the pattern 'busy day, then free day, then busy day' is unwanted. This is done for the first three days by a constraint with time groups

```
{1Mon1, 1Mon2, 1Mon3}
{1Tue1, 1Tue2, 1Tue3}*
{1Wed1, 1Wed2, 1Wed3}
```

and maximum limit 2, then replicated for each day the pattern could begin.

More examples may be found in [9], including constraints prohibiting or penalizing arbitrary unwanted patterns, and all the constraints from the two competitions [5,6,2,3]. This suggests that every resource constraint needed in practice can be formulated using the cluster busy times constraint. Of course, it is easy to define complex constraints that cannot be handled, such as the RosterBooster [4] constraints containing Boolean conditions, but the consensus of the literature seems to be that such constraints are not needed in practice.

The limit active intervals constraint contains the same attributes as the cluster busy times constraint, but its limits apply to the lengths of sequences of consecutive active time groups, rather than to their total number. We defer further discussion to Section 6.

One small gap in the present work concerns constraints which limit total workload (measured in minutes, say) rather than busy shifts. XESTT allows shifts to be assigned a workload, and it offers a *limit workload constraint* which limits total workload over a set of times. Incorporating history into this constraint is future work; it is not likely to present any difficulties.

There is a close affinity between the cluster busy times constraint and the integer programming formulations of the resource constraints [11,12], as is only natural. XESTT has two advantages over integer programming for present purposes. First, if our two key constraints can be extended to handle history, then, assuming that they really can support all resource constraints, this will completely solve the history problem. It is not clear how such a claim of completeness could be made for integer programming formulations without restricting them to something like the cluster busy times constraint, in which case the difference is merely one of notation. Second, the limit active intervals constraint is introduced just because the cases it handles are not handled efficiently by the cluster busy times constraint, as will be seen. Integer programming does not lend itself to this optimization.

3 Global and local instances

For any timetabling problem, a basic decision is the choice of the interval of time for which requirements are to be gathered and a solution sought. In a dynamic environment like a hospital ward, one can only hope to fix the timetable for a week, or a few weeks, ahead. So instances must be defined over short intervals of time. But some constraints span longer intervals, for example limits on monthly or even yearly workloads, while others routinely cross the boundaries of short intervals, for example limits on consecutive night shifts.

A *global instance* I is an instance covering a long interval of time called the *cycle* (a term from high school timetabling), which could be as long as one year. It is not practicable to solve I, or even to specify it, all at once, but it is a useful conceptual aid because constraints are defined naturally within I.

Divide the cycle into a sequence of contiguous time intervals $w_1, \ldots, w_n$, called the *weeks*. Of course, they do not have to be seven days long. For each week w_i, the aim is to derive a *local instance* I_i which represents the part of I concerned with w_i. This will be called the *projection* of I into I_i. It is assumed that solutions $S_1, \ldots, S_{i-1}$ are available for all previous weeks $w_1, \ldots, w_{i-1}$, as is usually the case, because I_i is only created shortly before w_i begins. Although I must contain everything relevant to I_i when the projection is made, it will usually be incomplete, missing the cover requirements for $w_{i+1}, \ldots, w_n$.

All nurses of I are projected into each I_i. Times and events are projected by copying those lying within each w_i into the corresponding I_i. Many constraints can be projected in the same way. All cover constraints can be, as can some resource constraints. For example, a global constraint requiring a nurse to work at most one shift on each day of the cycle projects to a local constraint requiring the nurse to work at most one shift on each day of w_i; a global constraint requiring a nurse to work at most 5 shifts per week projects to a local constraint requiring the nurse to work at most 5 shifts in w_i.

So the only hard cases for projection are constraints which, even taking advantage of opportunities of division into smaller parts, depend on a nurse's timetable across more than one week. These can be handled by the method of *counter start values* and *counter remainder values*, credited by [13] to [1]. This method will be applied now to cluster busy times constraints.

Consider projecting global constraint C into local instance I_i for week w_i, producing local constraint C_i. Since projection of cover constraints is trivial, it may be assumed that C is a resource constraint; and then, by Section 2, C may be assumed to be a cluster busy times constraint (or a limit active intervals constraint, which will be considered in Section 6).

Although the constraint may cross week boundaries, it will be assumed that none of its individual time groups do so. For example, if the constraint concerns weekends and so contains time groups such as

```
{1Sat1, 1Sat2, 1Sat3, 1Sun1, 1Sun2, 1Sun3}
{2Sat1, 2Sat2, 2Sat3, 2Sun1, 2Sun2, 1Sun3}
{3Sat1, 3Sat2, 3Sat3, 3Sun1, 3Sun2, 3Sun3}
```

then the week boundary cannot fall between Saturday and Sunday. The method presented here could be extended to cover such cases, but that would only add complexity that never seems to be needed in practice.

Many constraints monitor only one or two weeks, and do not give rise to a C_i in every week. But, for uniformity, a special case will not be made of them. The following constructions work for them, but produce trivial constraints in the irrelevant weeks which would be omitted in practice.

Consider the time groups of C. Those within w_i can be represented within I_i, and are copied into C_i. The rest must be omitted, because the times they refer to do not exist in I_i.

What matters about the omitted time groups is whether they are active or not. This can be determined for each time group before w_i, by seeing whether any of its times is a busy time in the solution for its week, and consulting its polarity. So let x_i be the number of C's time groups preceding w_i which are known to be active. This is the *counter start value* of [1,13].

On the other hand, the activity of the time groups following w_i is not known, because no timetable exists for those weeks. Let c_i be the number of C's time groups following w_i. This is the *counter remainder value* of [1,13].

There are cases where the activity of some time groups following w_i is known in advance—when the resource is going on holiday, for example. In practice it may well be worth taking account of such information. However, to limit the complexity of what follows, that will not be attempted here.

If C's minimum limit is L and its maximum limit is U, then C_i's limits should be $L - x_i - c_i$ and $U - x_i$ [13]. The x_i omitted active time groups are compensated for by subtracting x_i from both limits. The c_i omitted time groups of unknown activity could turn out to be active too, so the minimum limit must be reduced by c_i to ensure that it is not violated in that case. But they could also be inactive, so the maximum limit cannot be reduced as well.

The XESTT cluster busy times constraint optionally accepts x_i and c_i values for its resources. It increases the number of active time groups by x_i when comparing with a maximum limit, and by $x_i + c_i$ when comparing with a minimum limit. This has several advantages: it is more concise when one constraint handles many resources identically except for their x_i and c_i values; it makes x_i and c_i available for another use (Section 4); and it gives the desired result when the cost function is quadratic. Full details appear in Section 9.

For any global solution containing local solutions $S_1, \ldots, S_{i-1}$, if C is not violated then neither is C_i. Conversely, for any global solution containing $S_1, \ldots, S_{n-1}$, if C_n is not violated then neither is C. So the I_i taken together are the same as I: they have the same times, resources, events, and constraints.

4 Avoiding double counting of constraint penalties

The argument just given, showing that the projections I_i taken together are the same as the global instance I, has a flaw. The I_i enforce the same constraints, but some violations may be penalized more than once.

For example, suppose I has four weeks, and a constraint C with weight 1 limits the total number of busy times to at most 20. Suppose C's resource is busy for 7 times in each of the 4 weeks. The penalty in the global instance is $(4 * 7) - 20 = 8$; but C_3 attracts a penalty of $(3 * 7) - 20 = 1$, and C_4 attracts a penalty of $(4 * 7) - 20 = 8$, making a total penalty of 9. The problem is that C_i in fact constrains $I_1, \ldots, I_i$, so it includes C_{i-1} within itself.

This *double counting issue* was addressed by the competition. Its solution is not ideal: it varies from one constraint to another, and involves long case analyses [2]. Our other main source [13] does not mention the issue.

There are at least four ways to handle the double counting issue.

Ignore it. When solving I_i, any double counting depends on the solutions to $I_1, \ldots, I_{i-1}$, which are fixed and equal for every solution to I_i. So a solver cannot be led astray by double counting, and ignoring it is a real option.

Make I_i cumulative. Redefine I_i to be a *cumulative instance*, defined over $w_1, \ldots, w_i$ rather than just w_i (Section 7). Then double counting is natural, since I_i includes I_{i-1}. The problem here is that users do not want cumulative instances, because they include data that are not currently relevant, such as the coverage constraints from two months ago.

Redefine projection. Project each constraint C onto just one I_i, which must be the I_i containing C's last time group, since $c_i = 0$ there. The competition does this for two constraints. It is simple, but weak: in the example, one would want to include C_3 in I_3, but this approach omits it.

Adjust costs. Adjust the cost of C_i by subtracting any cost already reported by C_{i-1}. The competition mostly does this, although it expresses the idea in its own way. It is this paper's preferred method.

One way to implement this approach is to give each cluster busy times constraint a cost adjustment to subtract from its cost before reporting it. Instead, XESTT *calculates* a cost adjustment, based on x_i and c_i. The details appear in Section 9. It is intricate to implement, but it limits the user's task to supplying x_i and c_i, which is easy to do, and it applies to sequence constraints (Section 6), for which a simple subtraction of a cost is not sufficient.

However the cost adjustment is determined, it is a constant and needs to be calculated only once. It is subtracted from the cost before reporting it, each time the cost changes, taking a negligible amount of running time.

5 Heuristic constraints

Consider again the example from the previous section, in which C constrains its resource to work between 16 and 20 shifts over four weeks. Notice that C_1, to take the extreme example, cannot constrain the resource at all during w_1, because whatever happens then it is always possible to assign a workload in later weeks that satisfies C in the end. This slackness gradually reduces until, by the time C_n is reached, it has vanished altogether.

It is natural to want to add constraints in early weeks which guide solvers to good global outcomes. In the example, one would want between $4i$ and $5i$ shifts worked by the end of w_i. Such constraints will be called *heuristic constraints*, because they are heuristic in nature, not derived by projection. For example, they probably should not be hard constraints, even if the constraints whose satisfaction they are trying to promote are hard.

In the example, c_i is 7 multiplied by the number of weeks following w_i. But if some other constraint limits a resource to at most 5 shifts per week, c_i can be reduced to 5 times the number of following weeks, making for a tighter constraint. Still, even these cases are classified here as heuristic constraints, in view of the non-trivial, opportunistic analyses required to derive them.

This paper mentions heuristic constraints for completeness, but there seems to be little to say about them in general.

The cost adjustment method could be applied to heuristic constraints, since the basic idea of a later constraint subsuming an earlier one is still present. However, the adjusted costs can be negative. For example, if C_i limits its resource to a total workload (including history) of between $4i$ and $5i$ shifts, then it is easy to find cases where C_1 is violated but C_2 is not.

6 Optimizing sequence constraints

This paper does not classify resource constraints. Still, there are some that limit the number of consecutive occurrences of something, rather than the total number of that thing: the number of consecutive days worked, consecutive weekends worked, and so on. These will be called *sequence constraints*.

Cluster busy times constraints implement sequence constraints in the same way that integer programming formulations do, using *time windows*. Suppose there is a constraint limiting a resource to at most 5 consecutive busy days. This is formulated using one cluster busy times constraint for each day of the cycle except for the last 5 days. The first constraint has time groups

```
{1Mon1, 1Mon2, 1Mon3}
{1Tue1, 1Tue2, 1Tue3}
{1Wed1, 1Wed2, 1Wed3}
{1Thu1, 1Thu2, 1Thu3}
{1Fri1, 1Fri2, 1Fri3}
{1Sat1, 1Sat2, 1Sat3}
```

and maximum limit 5. The second constraint follows the same pattern, only starting on `1Tue`, and so on. The days of one constraint make one time window.

There is inefficiency here, because these constraints monitor much the same time groups. Time groups not near the ends of the cycle appear in 6 constraints. Sequence constraints with minimum limits can be even worse. Requiring free days to come in sequences of at least 3, for example, requires time windows of length 3 to prohibit patterns of the form ‘busy day, then free day, then busy day’, overlapping with time windows of length 4 to prohibit patterns of the form ‘busy day, then free day, then free day, then busy day’. It is lucky that sequential minimum limits tend to be small, otherwise time window formulations would be swamped by myriads of interrelated constraints.

It would be simpler to limit the number of consecutive active time groups directly. This does not seem to suit integer programming, but it is done by the XESTT *limit active intervals* constraint, which is like the cluster busy times constraint except that its limits apply to the lengths of maximal sequences of active time groups, not to their number. (The author learned of this design from Gerhard Post.) For example, to limit the number of consecutive busy days to at most 5, use a limit active intervals constraint with one time group for each day, in chronological order, and maximum limit 5.

The limit active intervals constraint would have little value if it could only be projected by expanding it into cluster busy times constraints first, so the counter start values and counter remainder values method needs to be adapted to apply to it directly. When projecting limit active intervals constraint C onto C_i, what is needed is not the total number of active time groups from before w_i, but rather the number of active time groups immediately adjacent to w_i on the left, called x_i as before. The definition of c_i is as before. Full details of the cost calculations appear in Section 9.

7 An alternative approach to history

This section describes an alternative approach to history, based on including preassigned events representing the solutions to all previous instances. This approach can be used with XESTT, since it supports preassignments.

The idea is to add to I_i all times and events from all previous instances, with their resource demands preassigned, based on previous solutions. Constraints are projected onto I_i as before, except that time groups from weeks $w_1, \ldots, w_i$ are included, and x_i is not used. However, c_i is used as before.

It is simplest to interpret this I_i as a cumulative instance, incorporating previous instances within itself (Section 3). So double counting is normal.

Two objections are raised when this is proposed. First, a large amount of historical data must be retained. In this era of big data, this cannot carry much weight. Some data must be kept anyway: each resource's total workload and total weekends worked, perhaps, as in the competition. This data might as well be complete as partial. However, as remarked in Section 3, users do not want cumulative instances, and this is the real point of this objection.

The second objection is that this increases the size of instances, causing problems for solvers. If this means that the search space for (say) a simulated annealing solver is increased, that is easily fixed by making the solver recognize the preassigned areas and exclude them from its search. If it means that the underlying solve platform has to evaluate many more constraints, that is not true of an incremental platform, which only evaluates changes. The author's KHE platform [8] would evaluate the parts of the constraints that monitor preassignments just once, as the preassignments are loaded. In effect, it calculates the x_i values for itself, just once, then carries on as before.

The preassignment approach is merely a different and conceptually simpler way to give x_i values to the constraints of I_i. The x_i values which stand in for omitted time groups are replaced by evaluations of the time groups themselves.

8 Conclusion

The ideas in this paper have been implemented by the author. The XESTT format accepts x_i and c_i values for its cluster busy times and limit active intervals constraints. The NRConv program [9,10], which converts instances and solutions in several formats into XESTT, adds history when converting weekly instances from the competition. History can also be added when constructing new instances. The KHE solve platform [8] reads XESTT and implements history in full, including cost adjustment to avoid double counting. The HSEval web service [7] evaluates solutions. It is based on KHE, so history is included. Altogether this is a comprehensive implementation of history which is arguably complete, except that knowledge of a resource's future timetable is not utilized.

Successful standards require consensus, which this author is not well placed to create. But a standard is imperative, it will come, and it will include history. This paper shows what such a standard could be.

9 Appendix: Cost calculations in detail

This section presents detailed cost formulas. It is the basis of the implementation of history in the author's KHE platform [8].

9.1 General formulas

Let I be a global instance with projections $I_1, \dots, I_n$, where $n \geq 1$, and let C be a constraint with projections $C_1, \dots, C_n$. Let the instance currently being solved be I_i, so that solutions $S_1, \dots, S_i$ are available, with $S_1, \dots, S_{i-1}$ fixed.

Let $c(C_i)$, the *cumulative cost* of C_i, be the minimum, over all solutions S of I containing $S_1, \dots, S_i$, of $c(C)$, the cost of C in S. In other words, $c(C_i)$ is the largest cost assignable to C_i without risk of exceeding $c(C)$.

As i grows, the set of solutions S containing $S_1, \dots, S_i$ shrinks, so $c(C_i)$ increases:

$$c(C_1) \leq c(C_2) \leq \cdots \leq c(C_n) = c(C)$$

When cost adjustment is used to avoid double counting, the cost actually contributed to the solution cost, called the *reported cost*, is $c(C_1)$ when $i = 1$, and $c(C_i) - c(C_{i-1})$ when $i > 1$. The total reported cost is then

$$c(C_1) + (c(C_2) - c(C_1)) + \cdots + (c(C_n) - c(C_{n-1})) = c(C_n) = c(C)$$

as required. All reported costs are non-negative, since $c(C_i) \geq c(C_{i-1})$.

It is usual to calculate cost in two stages. First, information specific to each constraint is used to calculate the *deviation*, also called the *degree of violation*. This is usually the amount by which some quantity exceeds a maximum limit or falls short of a minimum limit. Then a non-negative, non-decreasing *cost function* $f(x)$ is applied to the deviation to produce the cost. Examples are $f(x) = wx$ and $f(x) = wx^2$, where w is a non-negative constant *weight*.

When cost is calculated in this way, one can define $d(C_i)$, the *cumulative deviation* of C_i, as the minimum, over all solutions S of I containing $S_i, \dots, S_i$, of $d(C)$, the deviation of C in S. Then $c(C_i) = f(d(C_i))$ and $c(C) = f(d(C))$.

9.2 Cluster busy times constraints

Suppose now that C is a cluster busy times constraint with minimum limit L and maximum limit U, where $0 \leq L \leq U$.

Suppose these quantities are available to C_i for calculating costs with:

a_i	The number of C's time groups in $C_1, \dots, C_{i-1}$
b_i	The number of C's time groups in C_i
c_i	The number of C's time groups in $C_{i+1}, \dots, C_n$
x_i	The number of C's time groups which are active in $S_1, \dots, S_{i-1}$
y_i	The number of C's time groups which are active in S_i

with $0 \leq x_i \leq a_i$, $0 \leq y_i \leq b_i$, and $0 \leq c_i$. There is no z_i because solutions are available only for the past and present, not for the future.

The cluster busy times constraint has an `AllowZero` option, which when true causes zero active time groups to produce cost 0, whatever the limits. As an aid to expressing this in algebra, introduce the low-precedence operator

$$a :: b$$

Its value is 0 when `AllowZero` is true and $a = 0$, and b otherwise.

The number of active time groups of C in any solution S of I containing $S_1, \ldots, S_i$ is at least $x_i + y_i$ and at most $x_i + y_i + c_i$, so

$$d(C_i) = x_i + y_i :: max(0, L - x_i - y_i - c_i, x_i + y_i - U)$$

$L \leq U$ implies that $L - x_i - y_i - c_i$ and $x_i + y_i - U$ cannot both be positive.

If cost adjustment is desired, C_i also needs to know whether there is a C_{i-1} and what its cumulative cost is if so, so that it can subtract it away. There is a C_{i-1} when $a_i > 0$, and, by the previous formula, its cumulative deviation is

$$d(C_{i-1}) = x_{i-1} + y_{i-1} :: max(0, L - x_{i-1} - y_{i-1} - c_{i-1}, x_{i-1} + y_{i-1} - U)$$

But $x_{i-1} + y_{i-1} = x_i$ and $c_{i-1} = b_i + c_i$, so

$$d(C_{i-1}) = x_i :: max(0, L - x_i - b_i - c_i, x_i - U)$$

and in this form $d(C_{i-1})$, and hence $c(C_{i-1})$, is easy for C_i to calculate. It should do this just once, since the value is constant. When $a_i = 0$, there is no C_{i-1}, but we define $d(C_{i-1})$ to be 0 then, since it is faster to always subtract something than to test whether a subtraction is required and then do it if so.

It would be convenient if $a_i = 0$ implied $d(C_{i-1}) = 0$, since then $a_i = 0$ would not be a special case and a_i itself would not be needed. But although $a_i = 0$ implies $x_i = 0$, which eliminates the $x_i - U$ term, the $L - x_i - b_i - c_i$ term can be positive when $a_i = 0$.

This leads to a point of interest to implementers: when there is no lower limit (when $L = 0$), a_i and c_i do not influence the values of these formulas. Also of interest is the fact that when a_i, x_i and c_i are all 0, the formulas reduce to what they would be without history: the formula for $d(C_i)$ becomes

$$d(C_i) = y_i :: max(0, L - y_i, y_i - U)$$

and $d(C_{i-1}) = 0$ since $a_i = 0$. So 0 is a suitable default value for a_i, x_i and c_i.

It is not impossible to extend this work to incorporate information about a resource's future timetable. This would involve redefining x_i to be the number of C's time groups outside w_i which are known to be active, and redefining c_i to be the number of C's time groups outside w_i whose activity is undetermined. The problem is that the formulas $x_{i-1} + y_{i-1} = x_i$ and $c_{i-1} = b_i + c_i$ need detailed adjustment under the new definitions, leading to more complexity.

9.3 Limit active intervals constraints

Suppose now that C is a limit active intervals constraint with minimum limit L and maximum limit U, where $0 \leq L \leq U$.

An *active interval* is a sequence of active time groups; its length is what the constraint constrains. Let the length of active interval Δ be $l(\Delta)$, and let the deviation contributed by Δ be $d(\Delta)$. Then

$$d(\Delta) = max(0, L - l(\Delta), l(\Delta) - U)$$

$L \leq U$ implies that the second and third terms cannot both be positive.

Let C's active intervals in S be $\Delta_1, \ldots, \Delta_m$. One way to define $c(C)$ is

$$c(C) = f(\sum_{j=1}^{m} d(\Delta_j))$$

A total deviation is found and the cost function is applied once, giving a cost.

Conventional though it may be, this definition interacts badly with history: when $f(x)$ is non-linear, C_i needs to know the total deviation of all past active intervals. This is not surprising; after all, the value x_i given to cluster busy times constraints concerns all past time groups. But when constraining total workload it is natural to use total past workload, whereas when constraining the lengths of active intervals it is not natural to use the deviations of active intervals from the distant past. The competition doesn't, for example.

So a different definition of $c(C)$ is made, which interacts better with history:

$$c(C) = \sum_{j=1}^{m} f(d(\Delta_j))$$

This applies the cost function multiple times; but still it is reasonable.

Suppose these quantities are available to C_i for calculating costs with:

b_i	The number of C's time groups in C_i
c_i	The number of C's time groups in $C_{i+1}, \ldots, C_n$
x_i	The number of consecutive active time groups immediately preceding w_i
$\delta_p, \ldots, \delta_q$	The active intervals of C_i's time groups taken in isolation

Cumulative cost cannot be calculated from these values. The adjusted cost is

$$c(C_i) - c(C_{i-1}) = \sum_{j=p}^{q} f(d(\delta_j))$$

although this must be tweaked to take account of active intervals at the ends of w_i which could extend into adjacent weeks. The entire week may be active, giving a single active interval potentially extending in both directions.

First suppose that C_i includes at least one inactive time group, so that its two ends are independent.

Suppose there is an active interval δ_q which includes the last time group of C_i. This interval could extend into w_{i+1} and beyond. Its full length is at least $l(\delta_q)$ and at most $l(\delta_q) + c_i$, so it contributes

$$f(max(0, L - l(\delta_q) - c_i, l(\delta_q) - U))$$

to $c(C_i)$, and the last term of the sum above, $f(d(\delta_q))$, must be replaced by this. If there is no such active interval, then no adjustment is required.

Now suppose that $x_i > 0$. This is the length of an active interval which includes the last time group of C_{i-1}. As just explained, it will have contributed

$$f(max(0, L - x_i - c_{i-1}, x_i - U))$$

to $c(C_{i-1})$, with $c_{i-1} = b_i + c_i$ as usual. This contribution is obsolete and must be subtracted away. Then if the first time group of C_i is not active, the regular cost of an active interval with length x_i must be added:

$$f(max(0, L - x_i, x_i - U))$$

If the first time group of C_i is active, then x_i abuts δ_p, the first active interval of C_i, and their joint contribution to $c(C_i)$ is

$$f(max(0, L - l(\delta_p) - x_i, l(\delta_p) + x_i - U))$$

That ends the $x_i > 0$ case. None of this is needed when $x_i = 0$.

Finally, suppose that C_i has no inactive time groups, so that there is a single active interval δ_q which includes both the first and last time groups of C_i. The analyses for both ends of the week apply to δ_q. If $x_i > 0$, then x_i's obsolete contribution must be subtracted away, and cost

$$f(max(0, L - l(\delta_q) - x_i - c_i, l(\delta_q) + x_i - U))$$

added. If $x_i = 0$, there is nothing to subtract away, but

$$f(max(0, L - l(\delta_q) - c_i, l(\delta_q) - U))$$

must be added.

An implementation of the limit active intervals constraint that does not handle history may be extended to handle history in three steps. First, always subtract a cost from the reported cost:

$$f(max(0, L - x_i - b_i - c_i, x_i - U))$$

when $x_i > 0$, and 0 otherwise. Second, extend the data structure for holding active intervals, and merging and splitting them as time groups become active and inactive, to include an active interval of length x_i (when $x_i > 0$) lying just to the left of the first time group, which participates in interval merges and splits like real intervals do, except that its own (virtual) time groups never become inactive. Third, when comparing an interval length with a lower limit, add c_i to the length when the interval includes the last time group. These three extensions cover everything in the formulas above.

Once again, when $L = 0$, a_i and c_i do not influence the values of these formulas. In fact, a_i has no influence even when $L > 0$, although it is worth recording, since it is an upper limit for x_i. And when a_i, x_i, and c_i are all 0, the formulas reduce to what they would be without history.

Existing models vary in their treatment of sequences of free and busy days at the ends of the cycle. When mimicking such models, artificial values for history may be useful. For example, in [4], minimum limits do not apply to sequences of busy or free days that include the first or last day. This can be handled by assigning value L to a_i, x_i, and c_i in XESTT constraints that impose minimum (but not maximum) limits on the lengths of these sequences.

Again, incorporation of information about a resource's future timetable is not impossible. If a future time group is known to be inactive, the future is irrelevant from there, so c_i may be reduced. Active time groups immediately following w_i may allow $l(\delta_q)$ to increase. Once again, the details are complex.

References

1. Edmund Burke, Patrick De Causmaecker, Sanja Petrovic, and Greet Vanden Berghe, Fitness evaluation for nurse scheduling problems, Proceedings of the Congress on Evolutionary Computation, Seoul, Korea, 1139–1146 (2001).
2. Sara Ceschia, Nguyen Thi Thanh Dang, Patrick De Causmaecker, Stefaan Haspeslagh, and Andrea Schaerf, Second international nurse rostering competition (INRC-II), problem description and rules. oRR abs/1501.04177 (2015). http://arxiv.org/abs/1501.04177
3. Sara Ceschia, Nguyen Thi Thanh Dang, Patrick De Causmaecker, Stefaan Haspeslagh, and Andrea Schaerf, Second international nurse rostering competition (INRC-II) web site, http://mobiz.vives.be/inrc2/
4. Tim Curtois, Employee Shift Scheduling Benchmark Data Sets, http://www.cs.nott.ac.uk/~psztc/NRP/ (2016)
5. Stefaan Haspeslagh, Patrick De Causmaecker, Martin Stølevik, and Andrea Schaerf, First international nurse rostering competition website, http://www.kuleuven-kortrijk.be/nrpcompetition (2010)
6. Stefaan Haspeslagh, Patrick De Causmaecker, Martin Stølevik, and Andrea Schaerf, The first international nurse rostering competition 2010, Annals of Operations Research, 218, 221–236 (2014)
7. Jeffrey H. Kingston, The HSEval High School Timetable Evaluator, http://jeffreykingston.id.au/cgi-bin/hseval.cgi (2010)
8. Jeffrey H. Kingston, KHE web site, http://jeffreykingston.id.au/khe (2014)
9. Jeffrey H. Kingston, Gerhard Post, and Greet Vanden Berghe, A unified nurse rostering model based on XHSTT, PATAT 2018 (Twelfth international conference on the Practice and Theory of Automated Timetabling, Vienna, August 2018)
10. Jeffrey H. Kingston, XESTT web site, http://jeffreykingston.id.au/xestt (2018)
11. Florian Mischek and Nysret Musliu, Integer programming and heuristic approaches for a multi-stage nurse rostering problem, PATAT 2016 (Eleventh international conference on the Practice and Theory of Automated Timetabling, Udine, Italy, August 2016), 245–262 (2016)
12. Haroldo G. Santos, Túlio A. M. Toffolo, Rafael A. M. Gomes, and Sabir Ribas, Integer programming techniques for the nurse rostering problem, Annals of Operations Research 239, 225–251 (2016)
13. Pieter Smet, Fabio Salassa, and Greet Vanden Berghe, Local and global constraint consistency in personnel rostering, International Transactions in Operational Research (2016)