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

Modelling History in Nurse Rostering (DRAFT)

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 that clarity in the area of *history*: how solutions to previous instances affect the current instance. The paper offers improved handling of several issues, including projecting global constraints onto local instances, avoiding double counting of penalties, and constraining consecutive busy times. 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
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 [12] 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 [12], 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 [8,9]. All relevant constraints are formulated using just one kind of constraint, the *cluster busy times constraint*. Although this constraint's universality is only a hypothesis, good evidence for it exists. By applying history to this constraint, 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 constraint. Section 3 defines the history problem and applies the well-known method of counter start values and counter remainder values to the cluster busy times constraint. 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).

2 XESTT and the cluster busy times constraint

This section introduces the XESTT nurse rostering data format and its cluster busy times constraint. For more details, including the XML syntax, see [8,9].

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, a duration, 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 most nurse rostering resource constraints 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. 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* and *prefer resources constraints* for them. Although cover constraints vary somewhat through the literature, they are simple and 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 constraint*. The rest of this section is devoted to this important constraint.

A cluster busy times constraint may apply independently to many resources (for example, to all nurses who share a 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, 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.

There is a close affinity between the cluster busy times constraint and integer programming formulations of the resource constraints [10,11], as is only natural. XESTT has two advantages over integer programming for present purposes. First, if history can be incorporated into the cluster busy times constraint, then, assuming that it 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, Section 6 introduces a constraint for limiting the number of *consecutive* active time groups. This important case is not handled efficiently by the cluster busy times constraint, as will be seen. Integer programming does not lend itself to this optimization.

The cluster busy times constraint offers two functions with Boolean results for each time group: 'at least one busy time' and 'no busy times'. Allowing other functions, such as 'at least two busy times', would not compromise its utility. However, the author knows of no cases which require such functions.

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 $w_1, ..., w_n$, 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 $S_1, ..., S_{i-1}$ are available for all previous weeks $w_1, ..., w_{i-1}$, 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 $w_{i+1}, ..., w_n$.

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 [12] to [1]. This method will be reviewed now, applied 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.

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 [1,12].

On the other hand, the activity of the time groups following w_i is not known, because no timetable exists for those weeks. Let y_i be the number of C's time groups following w_i. This is the *counter remainder value* of [1,12].

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 limits are $min(C)$ and $max(C)$, C_i's limits should be $min(C)-x_i-y_i$ and $max(C)-x_i$ [12]. The x_i omitted active time groups are compensated for by subtracting x_i from both limits. The y_i omitted time groups of unknown activity could turn out to be active too, so the minimum limit must be reduced by y_i to ensure that it is not violated in that case. But they could also turn out to be inactive, so the maximum limit cannot be reduced as well.

In XESTT, the cluster busy times constraint optionally accepts explicit x_i and y_i values for its resources. It increases the number of active time groups by x_i when comparing with a maximum limit, and by $x_i + y_i$ 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 y_i values; it makes x_i and y_i available for another use (Section 4); and it gives the desired result when the cost function is quadratic.

For any global solution containing local solutions $S_1, ..., S_{i-1}$, if C is not violated then neither is C_i. Conversely, for any global solution containing $S_1, ..., S_{n-1}$, if C_n is not violated then neither is C. So the I_i taken together are the same as I: they contain 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 that a constraint C limits the total number of busy times to at most 20. Suppose C's resource is busy for 7 times in each of the four weeks. Taking all weights to be 1, 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 an overall penalty of 9.

The problem is that C_i is in fact a constraint on $I_1, ..., I_i$, so it includes C_{i-1} within itself. There are several ways to handle this *double counting problem.*

Ignore it. When solving I_i, any double counting depends on the solutions to $I_1, ..., I_{i-1}$, 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 $w_1, ..., w_i$ rather than just w_i (Section 7). Then double counting is natural, since I_i includes I_{i-1}. The fundamental problem with this is that users do not want cumulative instances, presumably because they include data that are not currently relevant, such as the coverage constraints from 9 weeks 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 $y_i = 0$ 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 rather differently. It is this paper's preferred method. It works as follows.

For any constraint C, let $c(C)$ be the cost of C in a solution to I, and let $c(C_i)$ be the cost of C_i in a solution to I_i. When C_i is violated, report only $c(C_i) - c(C_{i-1})$. This way, the total cost reported will be

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

as required. This works even when the cost function is quadratic.

A simple way to implement this is to let the cluster busy times constraint accept a cost for each resource which it will subtract from any reported cost violation. However, XESTT deduces this value from x_i and y_i, as follows.

Consider maximum limits first. Given that limits are not adjusted, $c(C_{i-1})$ is a function of the amount by which the number of active time groups from C which lie in $w_1, ..., w_{i-1}$ exceed C's maximum limit. But, from C_i's point

of view, this is just a known function of the amount by which x_i exceeds C_i's maximum limit, easily found by C_i using x_i and other information in C_i.

Minimum limits can also be handled. C_i can find y_{i-1}, the value that C_{i-1} used, by adding its own number of time groups to its own y_i, and so determine whether C_{i-1} reported a violation and what its cost was if so.

However the cost adjustment is determined, it is a constant and a solve platform needs to calculate it only once. It is subtracted from the reported cost each time that cost changes, taking a negligible amount of running time.

This approach is not simple and would not occur naturally to the ordinary user. But it limits that user's task to supplying the well-known x_i and y_i values, it applies uniformly to all resource constraints represented by cluster busy times constraints, and it is simpler than the long case analyses given in the competition documentation [2]. This paper's other main source [12] does not mention the double counting issue.

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

In the example, y_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, y_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.

The cost adjustment method can be applied to heuristic constraints, since the basic idea of a later constraint subsuming an earlier one is still present. However, the resulting 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 end 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 be viable with 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. As before, x_i is not limited to w_{i-1}; it may extend an arbitrary distance to the left. The definition of y_i is as before.

Suppose there is a maximum limit, and consider what should be done about an active interval at the extreme right of w_i. If the length of this interval is above the limit, then a penalty is inevitable but its full value cannot be known until the end of the interval, in w_{i+1} or beyond, is known. Accordingly the

interval is penalized as though it ended at the end of w_i, but its length, x_{i+1}, is passed on to w_{i+1} in anticipation that this cost might be adjusted.

Therefore, at the left end of w_i, the interval of length x_i received from w_{i-1} will have been penalized by C_{i-1}, or not, depending on whether it exceeds the limit. If the first time group of C_i is inactive, this can stand. If the first time group of C_i is active, then x_i's penalty from C_{i-1} (if any) should be subtracted away (unless double counting is being ignored), and the penalty (if any) for a sequence of active time groups whose length is x_i plus the length of the sequence that includes that first active time group should be added.

Now suppose there is a minimum limit, and consider what should be done about an active interval at the extreme right of w_i. If the length of this interval plus y_i is below the limit, then a penalty is inevitable but its value is not known unless $y_i = 0$. In keeping with the policy of penalizing S_i as realistically as possible given the information available, it makes sense to assume that the length of this interval is its current length plus y_i, and penalize it accordingly.

Therefore, at the left end of w_i, the interval of length x_i received from w_{i-1} will have been penalized by C_{i-1}, or not, depending on whether its length plus y_{i-1} is below the limit. If double counting is being ignored, this must be subtracted away. Then, if the first time group of C_i is inactive, it must treat x_i as the length of an unevaluated interval and penalize it, or not, appropriately for its length. If the first time group of C_i is active, then C_i's first sequence has x_i extra length and is evaluated accordingly.

Although this is rather complex, it is quite practicable. It is a matter of accepting that the complexity is really needed, and incorporating it into solve platforms, as the author has done with his KHE platform [7].

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 that I_i should contain its usual times and events plus all times and events from all previous instances, with the previous events' resource demands preassigned, based on previous solutions. Constraints are projected onto I_i as before, except that since I_i contains all the times of $w_1, ..., w_i$, all time groups from these weeks are included, and x_i values are not used. However, y_i values are 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 commonly raised when this is proposed. The first is that it creates management problems by requiring a large amount of historical data to be retained. In this era of big data, this argument cannot carry much weight. Some long-term records must be kept anyway: each resource's total workload and total weekends worked, perhaps, as in the competition. So there must be files holding this information, and those files might as well hold complete

records as partial ones. 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 approach increases the size of instances, causing problems for solvers. If by this is meant that the search space for (say) a simulated annealing solver is increased, that problem can be fixed very easily by making the solver recognize the preassigned areas and exclude them from its search. If the meaning is that the underlying solve platform has to evaluate many more constraints, that is not true of an incremental platform, which only evaluates things that change. For example, if this approach was used with the author's KHE platform [7], the platform would evaluate the parts of the constraints that monitor the preassigned areas just once, as those preassignments are loaded. In effect, the constraints calculate the x_i values for themselves, just once, then carry on as before.

In fact, one can understand this preassignment approach as being merely a different and conceptually simpler way to supply x_i values to the constraints of I_i. The calculations which substitute the x_i for omitted time groups are replaced by standard evaluations of the time groups themselves.

8 Conclusion

The XESTT format accepts x_i and y_i values for its cluster busy times and limit active intervals constraints. The author's KHE solve platform [7], which reads XESTT, uses these values to implement history, including cost adjustment to avoid double counting, as described in this paper. The author's NRConv program [8,9], which converts instances and solutions in several formats into XESTT, adds them when converting weekly instances from the competition. 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, something that the author, being a friend and observer of the nurse rostering research community but not a member, is not well placed to attempt. But standardization is imperative, it will come, and it will include a syntax and semantics for history. This paper shows what such a standard could be.

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, 11391146 (2001).
2. Sara Ceschia, Nguyen Thi Thanh Dang, Patrick De Causmaecker, Stephaan Haspeslagh, and Andrea Schaerf, Second international nurse rostering competition (INRC-II), problem description and rules. oRR abs/1501.04177 (2015). URL http://arxiv.org/abs/1501.04177
3. Sara Ceschia, Nguyen Thi Thanh Dang, Patrick De Causmaecker, Stephaan Haspeslagh, and Andrea Schaerf, Second international nurse rostering competition (INRC-II) web site, URL http://mobiz.vives.be/inrc2/.
4. Tim Curtois, Employee Shift Scheduling Benchmark Data Sets, URL http://www.cs.nott.ac.uk/˜psztc/NRP/ (2016)

5. Stefaan Haspeslagh, Patrick De Causmaecker, Martin Stlevik, and Andrea Schaerf, First international nurse rostering competition website, URL: http://www.kuleuven-kortrijk.be/nrpcompetition (2010)
6. Stefaan Haspeslagh, Patrick De Causmaecker, Martin Stlevik, and Andrea Schaerf, The first international nurse rostering competition 2010, Annals of Operations Research, 218, 221–236 (2014)
7. Jeffrey H. Kingston, KHE web site, `http://www.it.usyd.edu.au/~jeff/khe` (2014)
8. Jeffrey H. Kingston, Home Page of XESTT, `http://www.it.usyd.edu.au/~jeff/xestt` (2017)
9. Jeffrey H. Kingston, Gerhard Post, and Greet Vanden Berghe, A unified nurse rostering model based on XHSTT, to be submitted to PATAT 2018 (Twelfth international conference on the Practice and Theory of Automated Timetabling, Vienna, August 2018)
10. 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)
11. 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)
12. Pieter Smet, Fabio Salassa, and Greet Vanden Berghe, Local and global constraint consistency in personnel rostering, International Transactions in Operational Research (2016)