Improving the Dynamic Programming Algorithm for Nurse Rostering

Jeffrey H. Kingston

School of Information Technologies, The University of Sydney, Australia
jeff@it.usyd.edu.au
http://jeffreykingston.id.au

Abstract. A dynamic programming algorithm for optimally timetabling one nurse appears in papers that perform nurse rostering using column generation. This paper generalizes this algorithm, allowing it to assign an arbitrary (but small) subset of the nurses on an arbitrary subset of the days of the timetable, and handle every nurse rostering constraint used in practice. The paper also presents work in progress on speeding up the algorithm, mainly by improving dominance testing, its key step. The aim is to fit the algorithm for use as the reassignment operator of a VLSN search for nurse rostering.

Keywords: Nurse Rostering · Dynamic Programming · VLSN Search

1 Introduction

Nurse rostering is the problem of assigning shifts to the nurses of a hospital ward, so as to satisfy a given set of hard constraints and minimize the cost of violating a given set of soft constraints.

Papers that use column generation to solve nurse rostering problems have long used a polynomial-time dynamic programming algorithm for finding an optimal timetable for a single nurse. This becomes one column of a set covering integer program which is solved to produce a timetable for the whole ward.

This paper generalizes this dynamic programming algorithm, allowing it to optimally assign an arbitrary (but small) subset of the nurses on an arbitrary subset of the days of the timetable, and handle every nurse rostering constraint used in practice. The paper also describes work in progress on speeding up the algorithm, mainly by improving dominance testing, its key step.

The improved algorithm aspires to be used as the reassignment operator of a very large-scale neighbourhood (VLSN) search for nurse rostering. In VLSN search, a given initial solution is repeatedly improved by unassigning a large part of it and reassigning that part, hopefully in a better way. The dynamic programming algorithm can carry out this reassignment optimally, making it a direct competitor for integer programming, the usual choice here. At present it is not competitive, but the work is ongoing.

Implementing the algorithm presented here turned out to be a major task, running to over 13,000 lines of C code. A detailed 100-page description appears online [11]. This paper focuses on the key points, omitting many details.

Section 2 defines the problem, and presents the relevant literature. Sections 3 and 4 present the original algorithm using our terminology. Sections 5 and 6 present our generalizations and optimizations. Section 7 presents experiments.

2 The nurse rostering and single nurse rostering problems

Nurse rostering is the problem of assigning shifts to the nurses of a hospital ward. Hospitals run 24 hours a day, so nurse rostering problems usually have at least three *shift types*: morning, afternoon, and night. Each *shift* (one shift type on one day) demands a certain number of nurses, often with specified skills. There may be some flexibility in how many nurses to assign to each shift, and the number typically changes from day to day. Nurses are not interchangeable in general, because they have individual contracts (full-time, part-time, and so on), skills (senior nurse, assistant, and so on), and requests for days off.

Perhaps the most characteristic feature of the problem is the large array of constraints that each nurse's timetable must or should satisfy. In practice there are always hard constraints requiring each nurse to work at most one shift per day, and constraints (hard or soft) on each nurse's total workload over the weeks that the problem spans. There are also rules such as 'no day shift immediately following a night shift', 'at most 5 consecutive busy days', 'at most 2 busy weekends', and so on. These vary from one formulation of the problem to another, and from one nurse contract to another.

Nurse rostering is one of the most-studied problems in the discipline of automated timetabling. There are survey papers [1, 4] but they are rather old now. Much of the recent work is inspired by the Second International Nurse Rostering Competition (INRC-II) [2, 3].

The general nurse rostering literature is large, but the problem of optimally assigning a single nurse seems to have been studied only by researchers who use integer programming with column generation. Each column represents a complete timetable for one nurse; to generate a column is to solve a single nurse problem, for which these researchers mostly use dynamic programming. The integer program selects a subset of the columns that solves the full problem. For a survey of this work, going back to 1998, we refer the reader to [12].

These column generation papers often use the resource constrained shortest path problem [6] as a stepping stone. In our experience this does not yield any useful insights into nurse rostering, and so in this paper we move directly from nurse rostering to the dynamic programming algorithm.

The author is aware of one paper which uses dynamic programming to solve a full resource assignment problem, for security guards [5]. As the number of resources increases, the search space expands rapidly, making solving full problems with dynamic programming only viable for small instances.

3 Overview of the dynamic programming algorithm

This section presents our starting point, the existing dynamic programming algorithm for rostering a single nurse r.

On each day, nurse r can be assigned a shift of a given type (morning, afternoon, etc.), or nothing. Let these choices be $\{s_0, s_1, ..., s_a\}$, where s_0 is a special shift type that denotes doing nothing (a day off). We are assuming here, as is usual, that a nurse may take at most one shift per day.

A solution may be an arbitrary set of assignments, but in this paper we consider only solutions of a particular kind. Let the sequence of days for which assignments are required be $\langle d_1, ..., d_n \rangle$. A *solution* is a sequence of k shift types representing assignments for the first k days, where $0 \leq k \leq n$. For example,

s_1	s_1	s_1	s_0	s_0

represents a timetable in which r is assigned shifts of type s_1 on the first three days, followed by two days off; and

s_2	s_2	s_2	s_0	s_0

is similar, with s_2 replacing s_1. A *complete solution* is a solution for all n days.

Let us first consider a tree search algorithm. Each node of its search tree is one solution. For its root, the tree has the unique solution of length 0. Then, for the first day, we try assigning a shift of each type for that day, omitting types for which no shift is available, or for which r is not qualified. This produces one child for each available shift type, being a solution of length 1. For each of these we try assigning a shift of each type for the second day, and so on until all solutions of length n have been tried. The cost of each solution is calculated and the overall result is the best solution of length n.

When all shift types $\{s_0, s_1, ..., s_a\}$ are available on all days $\langle d_1, ..., d_n \rangle$, the tree search tries $(a+1)^n$ complete solutions, an impossibly large number when a timetable is needed over several weeks, as is common in practice. The idea of dynamic programming is to show that some solutions *dominate* others, that is, always produce better complete solutions in the end. Dominated solutions can be discarded, and this can produce major savings. In fact, it leads to running times which are polynomial in n [11], as is well known.

Define P_k to be the set of undiscarded solutions for $\langle d_1, ..., d_k \rangle$. For example, the two solutions above might lie in P_5. For the tree search, P_k contains up to $(a+1)^k$ solutions.

The dynamic programming optimization explores the tree in breadth-first order: it first builds P_0 (just the length 0 solution), then P_1, then P_2, and so on. To proceed from P_k to P_{k+1}, for each solution x in P_k, it constructs all solutions y consisting of x plus one assignment of an available shift type to r on d_{k+1}. Before inserting y into P_{k+1}, it checks whether P_{k+1} contains a solution that dominates y. If so it discards y instead of inserting it. If not, it first removes from P_{k+1} and discards all solutions that are dominated by y, and then inserts y. In this way, P_{k+1} contains only undominated solutions.

For reasons that will become clear later, on the last day, d_n, there will be just one solution in P_n, and that is the desired optimal solution. That completes the algorithm. But we still need a definition of dominance that allows the algorithm to safely discard a large number of solutions.

4 Signatures and dominance

This section defines a dominance relation between solutions which allows many solutions to be safely discarded. As it turns out, this relation is determined by the constraints of the problem instance.

The *signature* of constraint c in solution x, written $s(c, x)$, is a concise but complete representation of x as it affects c, excluding parts of x for which c has already yielded a cost.

Let us consider some examples. Suppose c is a constraint on the total number of shifts worked by nurse r. Then $s(c, x)$ would be the number of shifts worked by r within x. It does not matter to c which shifts they are.

Suppose c is a constraint on the total number of busy weekends (weekends where r works at least one shift). Then for $s(c, x)$ we may choose the number of busy weekends during x, but there is a catch when x's last day is a Saturday: the signature must record whether r works a shift on that day, because that determines whether working a shift on the immediately following Sunday adds to r's number of busy weekends or not. So $s(c, x)$ is an integer plus a Boolean when x's last day is a Saturday, and an integer on other days.

Suppose c is a constraint on the number of consecutive night shifts worked by r. Then $s(c, x)$ is the number of consecutive night shifts ending on the last day of x, or 0 if r is not assigned a night shift on the last day of x. Sequences of consecutive night shifts that terminated earlier have already yielded a cost and do not influence $s(c, x)$.

The reader familiar with history in nurse rostering will have noticed a strong connection between signatures and history values [9]. A history value records what r did that affects c before the first day; a signature records what r did that affects c before and during the last day of x. The idea is the same, although [9] assumes that cases like the Saturday treated above do not occur at the start of the timetable. Here, we cannot assume such cases away.

Suppose solution x is for days $d_1, ..., d_k$. Suppose a constraint c depends only on the assignments on those days. Then c yields its final cost when d_k is assigned, so $s(c, x)$ is empty. Or suppose c depends only on the assignments on days $d_{k+1}, ..., d_n$. Then there is nothing to remember about c on $d_1, ..., d_k$, so again $s(c, x)$ is empty. The only constraints for which $s(c, x)$ is non-empty are those which are affected both by the assignments on at least one of the days $d_1, ..., d_k$, and also by the assignments on at least one of the days $d_{k+1}, ..., d_n$.

The *signature* of a solution x, written $s(x)$, is the concatenation, over all constraints c, of $s(c, x)$. Concretely, it is an array of integers (Booleans are encoded as 0 and 1). The cost of solution x, written $c(x)$, is the sum of all costs already yielded by constraints up to and including x's last day. Each solution x needs

just four fields: a pointer to its parent solution in the search tree; the assignment that x adds to that parent; $s(x)$; and $c(x)$.

Given our definition of $s(c, x)$ as a complete representation of x as it affects c, it is not surprising that as solutions are built up, we can keep $s(c, x)$ up to date, and then on c's last day, convert it into a final cost for c. For example, if c is the number of shifts worked we can take this number from x's parent and add 1 or 0 to it, depending on whether x's assignment is for a shift or a free day. What is perhaps more surprising is that signatures support dominance testing as well as cost calculations, as we will see shortly.

We say that solution x *dominates* solution y if the best complete solution that can be derived from x by further assignments has cost no larger than the best complete solution that can be derived from y by further assignments. Clearly, in that case y can be discarded without risk of losing optimality.

In practice we use a more restricted definition of dominance: we require x and y to have the same last day, we require $c(x) \leq c(y)$, and for each constraint c, we require x to dominate y at c. By this last condition we mean that for each combination of assignments t for later days, when we add those further assignments to x and y, resulting in complete solutions x_t and y_t, the cost of c in x_t must not exceed the cost of c in y_t. It should be clear that every case of this more restricted definition of dominance is a case of the general definition.

There is no need to identify every case of dominance. All that matters is that for each case that we do identify, it is indeed safe to discard y. Of course, the more cases we identify, the faster the algorithm runs.

It is clear now why there is at most one solution on the last day, d_n. The signature array is empty because all constraints have reported their final cost by then, so x dominates y when $c(x) \leq c(y)$, so only one solution will be kept.

To understand dominance, then, it remains to understand how solution x can dominate solution y at constraint c. We answer this question for some kinds of constraints now; in Section 5 we'll see that we have in fact covered all cases.

Many constraints calculate a value that we call the *determinant*: the number of shifts, the number of consecutive night shifts, or whatever. The cost of such a constraint c is a monotone non-decreasing function of the amount by which the determinant exceeds a maximum limit or falls short of a minimum limit. The limits are fixed attributes of c.

The determinant is used as c's signature value on each day a signature value is needed. If c has a maximum limit only, then x dominates y at c when $s(c, x) \leq s(c, y)$. This is because when we add the same further assignments t to x and y, producing complete solutions x_t and y_t, the determinant increases by the same amount in both. So $s(c, x) \leq s(c, y)$ implies $s(c, x_t) \leq s(c, y_t)$, and the cost has this same relation too, because the cost when there is a maximum limit only is a monotone non-decreasing function of the determinant.

If c has a minimum limit only, x dominates y at c when $s(c, x) \geq s(c, y)$. This is because in this case the cost is a monotone non-increasing function of the determinant. If c has both a maximum limit and a minimum limit, then both analyses apply and x dominates y at c when $s(c, x) = s(c, y)$.

As expressed, this argument applies only to constraints on the total number of something, not to constraints on the number of consecutive things. But the argument is easily adapted to constraints on consecutive things, by changing x_t and y_t to the solutions on the first day after the sequence ends. On that day, the cost of the sequence is finalized and added to $c(x_t)$ and $c(y_t)$.

5 Generalizing to arbitrary nurses, days, and constraints

In this section we generalize the dynamic programming algorithm so that, starting from an initial timetable that we want to improve, it can unassign and reassign any (small) number of *selected nurses* on any number of *selected days*. We don't require the selected days to be consecutive, because we want to try oddities like reassigning the eight weekend days of a four-week timetable. We also generalize to arbitrary constraints.

To generalize from a single nurse to a set of selected nurses, the algorithm begins by unassigning the selected nurses on the selected days, then proceeds much as before. Each solution extends its parent solution by adding one assignment (possibly of a free day) on its last day to each selected nurse. If there are m selected nurses and $a + 1$ shift types, there are up to $(a + 1)^m$ ways to do this. The signature of each solution is the concatenation of the signatures of the selected nurses, in some fixed order, plus another signature for cover constraints, which we will come to shortly.

To generalize from all days to selected days, we redefine $\langle d_1, ..., d_n \rangle$ to be the sequence of selected days (in chronological order), not all days. A set of undominated solutions P_k is built for each selected day d_k, not for each day.

The other issue in generalizing this algorithm is to make sure that it can handle every kind of constraint. We do this by supporting the constraints of the XESTT nurse rostering model [7, 8], which have been shown in [8] to encompass everything that occurs in practice.

Nurse rostering constraints are either *cover constraints* (XESTT calls them *event resource constraints*), which require each shift to have a suitable number of suitably qualified nurses, or *resource constraints*, which require individual nurses' timetables to follow the many rules of nurse rostering: at most one shift per day, limits on total shifts, limits on consecutive shifts, and so on.

Although we illustrated the signature idea using resource constraints only, it applies equally well to cover constraints. Cover constraints always seem to constrain the shifts of a single day, and so do not need signature values. But if there was a multi-day cover constraint, for example a constraint requiring the staffing of at least four of the week's night shifts to include a senior nurse, then that could be handled, using a signature value saying how many of the current week's night shifts have been assigned a senior nurse.

XESTT has four event resource constraints, only three of which are used in nurse rostering, and seven resource constraints, only five of which are used. These small numbers are owing to the generality of the constraints: they may reference arbitrary sets of times and nurses. For example, no constraints are inherently

concerned with weekends; instead, an instance would define the set of weekend times and reference that in its constraints.

We do not have space to define all these constraints in detail and explain how they are handled. Instead, we'll divide them into three groups and explain how each group is handled.

The first group contains constraints which find the number of occurrences of something (e.g. busy weekends) and compare that number, which we have called a determinant, with lower and upper limits. As explained in Section 4, these constraints are handled by storing the determinant as the signature value. Assignments on unselected days are constant throughout the solve; their effect here is to add a constant to the signature value. For example, if c constrains the number of shifts worked by nurse r, then at the start of the solve, the signature $s(c, x)$ is the number of shifts worked on unselected days, and it increases from there as the solve proceeds.

The second group contains constraints which do not have determinants and limits. These we transform so that they do. For example, consider a constraint requiring nurse r to be free on Tuesday. Let the determinant be the number of shifts that r works on Tuesday, and apply maximum limit 0.

The third and final group contains constraints which find the total number of *consecutive* occurrences of something and compare that number with lower and upper limits. The signature is the number of consecutive occurrences in the current run, as explained earlier. Handling unselected days is quite complicated, since between any two selected days there may be a sequence of unselected days. Their effect is pre-calculated so that it can be included quickly. For example, suppose we are constraining nurse r's number of consecutive night shifts, and the selected days are the weekend days. Then for each run of consecutive week days we pre-calculate the number of r's consecutive night shifts a at the start of the week, and the number b at the end of the week. Then as the solve proceeds, if the preceding Sunday has a night shift we count that not as 1 night shift but as $1 + a$ night shifts, and so on. For full details we refer the reader to [11].

The definition of dominance at c given above ('$\leq$' when c has a maximum limit only, '$\geq$' when c has a minimum limit only, and '$=$' when c has both) we call *basic dominance*. Our algorithm actually uses *strong dominance*, which is basic dominance with three changes. First, XESTT constraints have an *allow zero flag*, which when set specifies that a determinant with value 0 produces cost 0 regardless of the minimum and maximum limits. Its main use in nurse rostering is in 'complete weekends' constraints, which require a nurse to work both days of a weekend or neither. Strong dominance takes this into account. Second, strong dominance understands that when c has a maximum limit only, $s(c, x)$ may be so small that no further assignments can increase it to the maximum limit, and so x dominates every y at c because the cost of c in every x_t must be 0. The third change is similar, understanding that when c has a minimum limit only, if $s(c, x)$ has already reached it, then x dominates every y at c.

The algorithm discards solutions x such that $c(x)$ equals or exceeds the cost of the original solution (the solution from before the initial unassignments). If

all solutions are discarded, the algorithm has been unable to improve on the original. In that case, the original solution (recorded separately) is reinstated before the algorithm returns. There are two points to note here.

First, constraints do not wait for their last day to report a cost; they report a cost on every day that they are affected by. Each day, they report their cost on that day minus the cost they reported on the previous day. This makes $c(x)$ as large as possible on every day, causing as much discarding as possible.

Second, every cost difference must be non-negative. A negative difference means that there might have been an unjustified discard on the previous day. Cost differences are naturally non-negative when they derive from maximum limits. But costs may decrease as determinants grow towards minimum limits. All constraints know this and only report costs that cannot go away later. Detailed formulas showing how to do this appear online [11].

This algorithm runs in time polynomial in the number of selected days but exponential in the number of selected nurses, according to a straightforward worst-case analysis based on the fact that the solutions in each P_k have distinct signatures [11]. The algorithm will usually do much better than the worst case. For example, the analysis does not take into account the fact that solutions whose cost equals or exceeds the cost of the original solution are discarded, but testing shows that this has a pronounced effect, especially during the last few days. However, testing also shows that selecting a few more days is not particularly costly, but selecting just one more nurse can dramatically increase the running time, confirming the general thrust of the analysis.

6 Optimizations

Our algorithm contains many minor optimizations. But here we focus on three major ones, each of which significantly reduces running time, as we will show.

Generating fewer solutions. For each solution x there are up to $(a+1)^m$ solutions y that extend x for one more day, where a is the number of shift types and m is the number of selected nurses. Our first optimization, which is really several optimizations combined, aims to reduce this large number, using ideas that we expect have been tried before on other problems.

One idea that does not work is to identify equivalent nurses and avoid generating solutions that are symmetrical in those nurses' assignments. As mentioned earlier, nurses have several individual characteristics, and when those are combined with the different timetables that nurses have on unselected days, finding symmetries among the nurses is hopeless.

On the other hand, there are symmetries among the *tasks* that nurses may perform (the variables that nurses may be assigned to) on each day. We have already informally partitioned these tasks into classes—the 'shift types.' But there are subtleties here. There may be a task in the night shift reserved by a hard constraint for a senior nurse; some tasks may have hard constraints requiring them to be assigned; others may have soft constraints requesting that they remain unassigned. We analyse the event resource constraints that apply to

each task, and group the tasks into classes of equivalent tasks, placing the most preferred tasks first in each class. When assigning nurses, there is no need to try all combinations of tasks from one class; instead, each nurse tries only the most preferred of the remaining unassigned tasks of the class.

The analysis may identify some tasks for which assignment is compulsory, because a hard constraint requires it. As nurses are assigned to tasks on one day, if the number of remaining unassigned nurses falls below the number of remaining unassigned compulsory tasks, that line of generation of assignments is abandoned. Compulsory tasks are common so this can be very effective.

As each decision to try assigning a particular nurse to a particular task is made, the cost of that decision is calculated immediately and added to the growing solution cost. If the result equals or exceeds the cost we are trying to improve on, again that line of generation of assignments is abandoned.

We have not included experiments to show the effect of these optimizations on running time. They are cheap to carry out, and some are hard to turn off.

Tradeoff dominance. Our second optimization, which we have not seen elsewhere, improves the dominance test, finding more cases of dominance and shrinking the sets P_k. We previously improved basic dominance into strong dominance (Section 5); now we improve strong dominance into *tradeoff dominance*.

Suppose that solution x fails to dominate solution y, but only at one point along the signature, and only by 1. Suppose that the corresponding constraint has weight w. Then the effect of this failure is that at some point in the future that constraint could have a cost in some x_t which is at most w greater than its cost in the corresponding y_t (assuming the cost function is not quadratic), and this is why dominance fails.

But if $c(x) + w \leq c(y)$, this extra w cannot make x_t cost more than y_t. In other words, x still dominates y even though dominance fails at one point.

This idea easily extends to differences greater than 1, and to multiple points along the signature. It is simply a matter, as we proceed along the signature, of adding to $c(x)$ the cost of ignoring each violation of dominance. Then, if $c(x)$ exceeds $c(y)$ at any point, dominance has failed even with this tradeoff.

Representing sets of solutions by tries. Our third optimization aims to speed up the insertion of a new solution x into the set of undominated solutions P_k. Recall that this involves testing whether P_k contains a solution that dominates x, and if so discarding x; and if not, finding and discarding all solutions in P_k that are dominated by x, then inserting x. The straighforward approach here, taken by all previous authors as far as this author can ascertain, is to take each element of P_k and see whether it dominates x. If all those tests fail, take each element of P_k again and see whether it is dominated by x, making two dominance tests for each element of P_k.

The author experimented with a dominance test he calls *weak dominance*, in which x dominates y if $c(x) \leq c(y)$ and the two signatures are equal. This finds many fewer cases of dominance than strong dominance does, but it allows P_k to be organized as a hash table indexed by signature, converting the search for dominating and dominated solutions into a single hash table retrieval. However,

it turned out that the faster table handling did not compensate for the larger size of the sets P_k. The larger size did not slow down the hash table, but creating the many extra solutions took too much time.

Another failed optimization involved maintaining a *cache* of solutions C_k alongside each usual set of solutions P_k. All solutions y which were extensions of the same solution x were inserted into C_k with the usual dominance testing. When there were no more extensions of x, each surviving element of C_k was inserted into P_k, again with the usual dominance testing, and C_k was cleared ready to receive extensions of another x. The idea was that solutions which are extensions of the same parent are likely to have dominance relations with one another, and these relations can be found quickly within C_k. This is true, and the cache halved some running times, but the improvement disappeared when P_k was organized as a trie.

So let us turn now to an optimization that actually helped. The traditional trie is a symbol table representing some set of values, each associated with a key which is a sequence of characters. The root of the tree contains an array of subtrees. Each subtree contains all values whose keys have the same first character, and its index in the array is the integer value of that character. So to retrieve a value by key, one uses the first character of the key to index the root array to obtain a child, then the second character to index that child's array, and so on. When a subtree contains only a single value, it has a different format: the value and its key are stored, and retrieval compares the key it is looking for with the stored key to see whether the value is the one wanted. There are also null subtrees representing empty sets of values.

Solutions are a natural fit for tries. A solution's key is its signature, a sequence of small integers well suited to array indexing.

To decide whether x is dominated by any solution already in trie T, we proceed as follows. Suppose the first element of x's signature is v, and that it is associated with a maximum limit and so is dominated by any value less than or equal to v. We need to recursively search only those subtrees with indexes in the range 0 to v inclusive, not the whole trie. Similarly if v is associated with a minimum limit, we need to search all subtrees from v to the end of the array of children. If the test is equality, only the subree with index v needs to be searched. This applies at each level of the trie.

Deleting all solutions of T that are dominated by x is similar, with the array ranges swapped: if v is associated with a maximum limit, then all solutions dominated by v lie in the range from v to the end of the array, and so on. Insertion is just the usual trie insertion.

This description applies to basic dominance. Formulas saying exactly where to search under strong dominance are derived online [11]. Tradeoff dominance is a problem, because in principle the entire trie needs to be searched. But instead of that, the algorithm proceeds heuristically: at each level of the trie, it searches each position needed for strong dominance, plus up to two adjacent positions where a tradeoff of w is required.

A signature entry representing workload in minutes can cause efficiency problems for tries, because it is likely to lead to large arrays of children, mainly filled with null entries. The author has not investigated this in detail, but the answer is probably to find the greatest common divisor d of all the workloads and store the workload in minutes divided by d.

7 Experiments

This section offers experiments which show how the algorithm performs on a difficult instance from a standard data set, and how effective the optimizations are. We have not yet tried the algorithm on a wide variety of instances, or tried calling it repeatedly as the reassignment operator of a VLSN search.

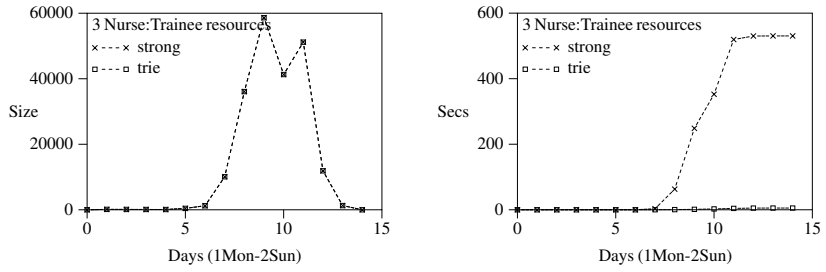

Fig. 1. The effect of introducing the trie data structure. The first graph shows the number of undominated solutions on each of the 14 days during which the selected resources are open to reassignment (almost 60,000 at the peak). This is the same with or without the trie data structure, because the same strong dominance test is used. The second graph shows the running time in seconds. For the ordinary array of undominated solutions, the running time is more than 500 seconds. The trie running time appears negligible. See Fig. 2 for a clearer view of the trie running time.

The experiments all use instance `INRC2-4-030-1-6291.xml`, the XESTT version of a 4-week instance derived from the Second International Timetabling Competition [2, 3] and tested by other authors [12, 13].

This instance divides into two independent parts, one for trainee nurses and one for non-trainee nurses. We show the trainee nurse results, because the algorithm runs more slowly on them. This is probably because trainee nurses can take each other's shifts, whereas non-trainee nurses may have varying skills, and can take some of each other's shifts but not all.

Figures 1 and 2 report on tests that select 3 nurses for reassignment over the first 14 days. The 3 nurses are chosen at random but are the same in all tests, as is the initial solution. This test happens to find an improvement on the initial solution. Fig. 1 shows the effect of changing the way that the solutions are stored in each P_k from an ordinary unsorted array to a trie. The same solutions are found, but running time improves dramatically. Fig. 2 shows the effect of

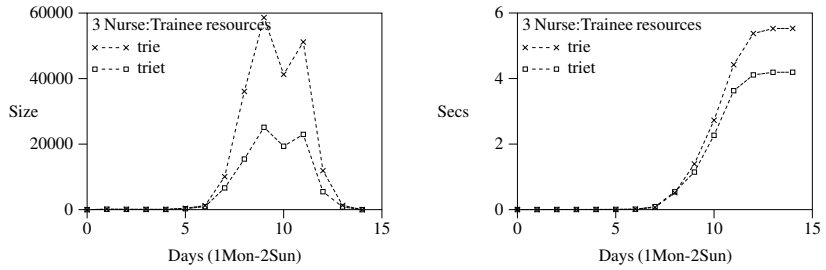

Fig. 2. The effect of introducing tradeoff dominance. This is the same instance as in Fig. 1, and the points labelled 'trie' are the same as in Fig. 1. The points labelled 'triet' are for the trie data structure with tradeoff dominance instead of strong dominance. The table size is reduced by more than half at the peak, and running time is reduced by about 30%.

changing from strong dominance to tradeoff dominance within the trie. The improvement is less dramatic but still useful.

The third figure, Fig. 3, shows what happens when the number of trainee nurses is increased to 4, using tries with tradeoff dominance. This run generates millions of undominated solutions and takes about 6000 seconds (100 minutes) to complete, a very bad result.

8 Conclusion

This paper has generalized the dynamic programming algorithm for optimal nurse rostering, and made some progress in optimizing it, in preparation for using it as the reassignment operator of a VLSN search. The implementation is available online [10] along with a detailed description [11].

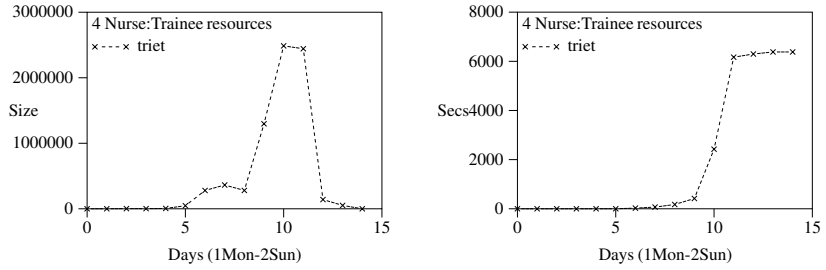

Fig. 3. As before, with tries and tradeoff dominance, but reassigning 4 trainee nurses.

The experiments have shown that the algorithm is able to reassign a fairly large number of days, for example 14. It is not able to reassign a large number

of nurses. In this paper we have been able to reassign 3 nurses quickly, but not 4, so further improvement is needed.

Tries and tradeoff dominance do not interact well, and the implementation tested here does not find every case of tradeoff dominance when tries are used. The author has gathered statistics showing that the number of undominated solutions kept when both are used can be five times larger than when tradeoff dominance is applied to a simple list of undominated solutions. The trie is still faster, but it would be better if both ideas could perform at their best.

A well-known enhancement is to always extend a lowest cost solution next, chosen across all days. If a new best complete solution is found, all incomplete solutions whose cost exceeds its cost will then never be extended. The author has gathered statistics showing that 70% of the solutions created by the solve in Fig. 3 would either not be created or not be extended, if this was done.

Hand analysis shows that even tradeoff dominance is very conservative. By this we mean that there are many cases where one solution is morally certain to dominate another, but the dominance test does not succeed, because it cannot rule out the possibility that a set of highly improbable events will occur which allow the more costly solution to produce a superior extension. Further analysis of dominance testing could produce a major payoff.

As a last resort, we could keep, say, only the best 10,000 undominated solutions on each day. This would provide a very robust upper limit on the running time. In Fig. 3, if the solutions on each day are sorted by increasing cost, the solutions on the path to the optimal complete solution have remarkably low indexes in the sorted lists. For example, on the day with the largest number of undominated solutions, there are 2,486,741 undominated solutions, but the index of the solution on the path to the optimal complete solution is just 25. Of course, keeping only the best 10,000 solutions on each day would remove the guarantee of optimality. But then, the VLSN search that this algorithm is intended to support offers no guarantee of optimality either.

References

1. Edmund K Burke, Patrick De Causmaecker, Greet Vanden Berghe, and H. Van Landeghem, The state of the art of nurse rostering, Journal of Scheduling 7, pages 441–499 (2004). DOI 10.1023/B:JOSH.0000046076.75950.0b
2. Ceschia, S., Nguyen, T. T. D., De Causmaecker, P., Haspeslagh, S., Schaerf, A.: Second international nurse rostering competition (INRC-II), problem description and rules. oRR abs/1501.04177 (2015). http://arxiv.org/abs/1501.04177
3. Ceschia, S., Nguyen T. T. D., De Causmaecker, P., Haspeslagh, S., Schaerf, A.: Second international nurse rostering competition (INRC-II) web site, http://mobiz.vives.be/inrc2/
4. Cheang, B., Li, H., Lim, A., Rodrigues, B.: Nurse rostering problems – a bibliographic survey. European Journal of Operational Research **151**, 447–460 (2003)
5. Elshafei M., Alfares, H. K.: A dynamic programming algorithm for days-off scheduling with sequence dependent labor costs. Journal of Scheduling **11**, 85–93 (2008)
6. Irnich, S., Desaulniers, G., et al.: Shortest path problems with resource constraints. In: Column generation, chap. 2, 33âĂŞ65. Springer US (2005)

7. Kingston, J. H.: XESTT web site, http://jeffreykingston.id.au/xestt (2017)
8. Kingston, J. H., Post, G., Vanden Berghe, G.: A unified nurse rostering model based on XHSTT. In: PATAT 2018 (Twelfth International Conference on the Practice and Theory of Automated Timetabling, Vienna, August 2018), 81–96
9. Kingston, J. H.: Modelling history in nurse rostering. In: PATAT 2018 (Twelfth international conference on the Practice and Theory of Automated Timetabling, Vienna, August 2018), 97–111. Also Annals of Operations Research, https://doi.org/10.1007/s10479-019-03288-x
10. Kingston, J. H.: KHE web site (Version 2.7), http://jeffreykingston.id.au/khe (2022)
11. Kingston, J. H.: The KHE User's Guide (Version 2.7), Appendix C: Resource reassignment using dynamic programming. http://jeffreykingston.id.au/khe (2022)
12. Legrain, A., Omer, J., Rosat, S.: A rotation-based branch-and-price approach for the nurse scheduling problem. Mathematical Programming Computation **2019**, 1–34
13. Ceschia S., Schaerf, A.: Solving the INRC-II nurse rostering problem by simulated annealing based on large neighborhoods. In: PATAT 2018 (Twelfth international conference on the Practice and Theory of Automated Timetabling, Vienna, August 2018), 331–338