Hierarchical Timetable Construction

Jeffrey H. Kingston

School of Information Technologies
The University of Sydney, NSW 2006, Australia
http://www.it.usyd.edu.au/~jeff
jeff@it.usyd.edu.au

Abstract. A hierarchical timetable is one made by recursively joining smaller timetables together into larger ones. Hierarchical timetables exhibit a desirable regularity of structure, at the cost of some limitation of choice in construction. This paper describes a method of specifying hierarchical timetables using mathematical operators, and introduces a data structure which supports the efficient and flexible construction of timetables specified in this way. The approach has been implemented in KTS, a web-based high school timetabling system created by the author.

1 Introduction

The basic timetable construction problem is to assign times and resources (students, teachers, rooms, etc.) to a set of meetings so that the resources have as few timetable clashes as possible. To this basic problem many other constraints are typically added, such as that the times allocated to a meeting be spread evenly through the week, that workload limits placed on some resources not be exceeded, and so on. Timetable construction is an NP-complete problem with an extensive literature [3–7].

Informally, a *regular timetable* is one in which a pattern may be discerned which makes the timetable easy to understand and remember. Regularity may take many forms, but this paper will be chiefly concerned with regularity in the choice of times. For example, North American universities commonly require all courses to occupy three hours per week, offered in one of the sets of time slots Mon-Wed-Fri 9-10am, or Mon-Wed-Fri 10-11am, and so on, producing a very regular timetable.

Even when such a strict rule as this is not possible, still some regularity might be achievable, perhaps by attempting to minimize the number of pairs of meetings that share at least one time, in addition to the usual objectives.

Regular timetables are easy to assign resources to. For example, in the North American university system, each meeting can meet in the same room for all three of its times. This point is particularly significant in high school timetabling, where teachers are assigned as well as rooms. Teacher assignment is the main area where the author's previous work in high school timetabling [8, 10] is deficient. Thus, regularity is more than just an aesthetic consideration.

E. K. Burke, H. Rudová (Eds.): PATAT 2006, pp. 196–208. ISBN 80-210-3726-1.

This paper introduces a method of specifying regular timetables hierarchically, using *timetable expressions* analogous to algebraic expressions, and a data structure, the *layer tree*, which represents these expressions and efficiently supports the basic assignment and deassignment operations on which most timetable construction algorithms are built. This author's KTS timetabling system [11, 12], a free, public web site for high school timetabling, uses layer trees. They are particularly effective when sets of meetings can be identified that must be disjoint in time. In high school timetabling, each set of meetings attended by a given student group satisfies this condition.

Our focus is on the efficient implementation of the basic assignment and deassignment operations, rather than their use with any particular timetable construction algorithm. If these operations are efficient, many algorithms, including construction heuristics, tree searches, and local searches, become available. Although efficiency is a key goal, it has not been considered useful to report running times, since the operations to be presented are all polynomial time, and running times say more about the algorithms built on these operations than the operations themselves. KTS typically produces a good timetable in about ten seconds [12], showing that layer trees can support practical timetabling.

Much of this paper is concerned with constraint propagation, but the emphasis here is on the efficient implementation of a particular set of constraints relevant to timetabling, rather than the use of a general-purpose constraint programming system to solve timetabling problems. Some of the algorithms used here, for example weighted bipartite matching, do not seem to be available in any existing constraint programming system [2, 9], although some recent research into the *all_different* constraint [13], which implements unweighted bipartite matching, is a step in that direction.

The algorithms used here have appeared in previous timetabling work by the author and others [8, 10, 14]. This paper's contribution is to show how these algorithms can be incorporated into a flexible, efficient, hierarchical constraint framework. Section 2 introduces timetable expressions, and Section 3 introduces the layer tree data structure. Section 4 analyses the problem of efficiently propagating constraints related to time through this data structure as assignments and deassignments occur, and Section 5 does the same for resource constraints. Section 6 surveys some other, less fundamental features implemented in KTS.

2 Timetable expressions

The idea of using an expression to specify a problem is well known in logic. Consider a Boolean expression such as

$$(x_1 \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_2} \vee x_3)$$

The expression defines an instance of the satisfiability problem, for which a solution consists of an assignment of values to the variables which satisfies the expression. In the same way, *timetable expressions* will be used to specify timetable construction problems.

The simplest kind of timetable expression is the *time variable*, a variable v whose domain is some subset of the set of available times T. This domain may change as solving proceeds; its value at some moment will be denoted $tdom(v)$, and its initial value, specified when the variable is created, will be denoted $tdom_0(v)$. For example, if v may be assigned any time, then $tdom_0(v) = T$; if v is *preassigned* to a specific time t, $tdom_0(v) = \{t\}$. Other initial domains may constrain times to be during the mornings, or on Mondays, and so on.

The ultimate aim is to assign an element of T to every time variable, just as the aim is to assign a Boolean value to every variable when solving satisfiability problems. However, it turns out that in hierarchical timetabling a more useful basic operation is the unification of one time variable, v, to another, w, with the meaning that v's value is constrained to be equal to w's. Unifying two variables expresses the idea that two meetings are to occur simultaneously, without having to say when.

Thus, our system offers two basic operations: unifying a variable v to one other variable w, and removing the unification of v to w. A variable may be unified to at most one other variable at any moment; but that other variable is free to be unified to a third variable (or not), and many variables may be unified simultaneously to one variable.

Two timetable expressions e_1 and e_2 may be joined using the *concatenation operator*, written e_1e_2, meaning that the times assigned to the variables of e_1 must be disjoint from those assigned to the variables of e_2. For example, a meeting requesting four times may be expressed by the timetable expression $v_1v_2v_3v_4$, where v_1, v_2, v_3, and v_4 are time variables. Concatenation specifies that the times assigned to these four variables must be distinct, as required.

If two meetings request the same resource, and it is a hard constraint that that resource may have no clashes in its timetable, then the expressions representing those two meetings may be concatenated. This is fundamental in the high school timetabling work which motivates this paper: each student group is such a resource, and the meetings it appears in must be disjoint in time.

Two timetable expressions e_1 and e_2 may be joined using the *alternation operator*, written $e_1 + e_2$, meaning that e_1 and e_2 are to appear in the same timetable, but there are no time constraints between their variables. In the high school timetabling application, e_1 might represent the meetings attended by one student group, and e_2 might represent the meetings attended by some other student group. These two sets of meetings have no time interdependencies, so joining them with $+$ is appropriate. If there is a meeting that both student groups attend, then its expression ($v_1v_2v_3v_4$ or whatever) will appear in both subexpressions, and its variables must be assigned times disjoint from those assigned to the variables its expression is concatenated with in both subexpressions.

These operations are named by analogy with the corresponding operators of regular expressions: e_1+e_2 signifies that e_1 and e_2 are alternative activities, while e_1e_2 signifies that one activity must follow after the other. In timetable expressions, however, both operators are associative and commutative. A distributive law, $(a + b)c = (ac + bc)$, also holds.

Finally, there is the *restriction operator*, written

$$w_1 w_2 \ldots w_k : e$$

where $w_1 w_2 \ldots w_k$ is a concatenation of time variables called *restriction variables*, and e is a timetable expression. This specifies that each variable in e must not appear outside e, and must be unified to one of the w_i (which themselves must be assigned disjoint times), restricting e to a timetable using at most k times.

Restriction introduces abstraction into a timetable expression. The expression e may be timetabled into $w_1 w_2 \ldots w_k$ independently of the rest of the problem, after which these variables are indistinguishable from an ordinary concatenation of variables describing a meeting.

Typically, the outermost level of a timetable expression is a restriction expression which limits the timetable to the available times. Letting $T = \{t_1, t_2, \ldots, t_n\}$ be the set of available times, this expression would have the form

$$w_1 w_2 \ldots w_n : e$$

where $tdom_0(w_i) = \{t_i\}$ for all i. Although the operation of assigning a particular time t_i to a variable v is not offered, unifying v to w_i is effectively the same thing.

Variants of the timetabling problem exist in which the exact number of available times is not given; instead, a timetable with as few times as possible is sought, consistent with other requirements. The restriction notation could easily be extended to cover such problems. However, the algorithms appearing later in this paper assume a fixed number of variables, so any such 'extensible restriction' would have to be solved (or at least, its number of variables determined) before incorporation into a larger timetable, forcing a bottom-up solution order.

An example of a small timetable expression appears in Figure 1.

3 The layer tree data structure

A timetable expression such as

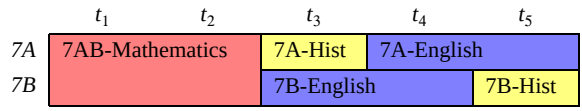
$$(e_1 + e_2)(e_3 + e_4)$$

is difficult to handle, since it is not clear how many of the available times should be allocated to $e_1 + e_2$, and how many to $e_3 + e_4$. While cases of this kind do occur, they are beyond the scope of this paper, and we will now exclude them.

A *simple timetable expression* is one in which each alternation expression $e_1 + \cdots + e_m$ is immediately enclosed in a restriction expression. In such expressions it is easy to determine how many times to allocate to each subexpression. Furthermore, a simple timetable expression can be analysed into a tree (or forest if the root is a concatenation expression) of expressions of the form

$$w_1 w_2 \ldots w_n : (e_{11} e_{12} \ldots e_{1k_1} + \cdots + e_{m1} e_{m2} \ldots e_{mk_m})$$

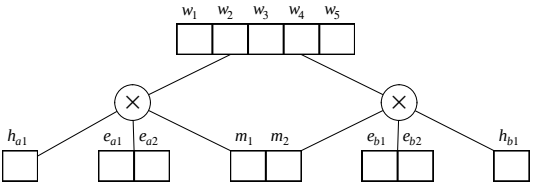
called a *restricted sum of products*. Here m may be 0, in which case the expression just denotes a sequence of variables $w_1 w_2 \ldots w_n$. Each e_{ij} is a restricted sum of



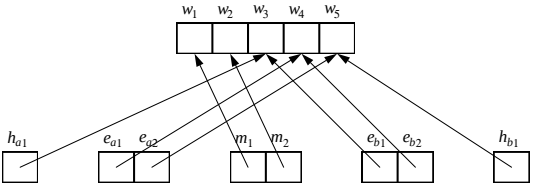
(a) A small timetable, or *tile*, occupying two student groups (7A and 7B) for five times t_1, t_2, t_3, t_4, and t_5.

$$w_1w_2w_3w_4w_5 : m_1m_2h_{a1}e_{a1}e_{a2} + m_1m_2e_{b1}e_{b2}h_{b1}$$

(b) A timetable expression for which (a) is a solution. Here $w_1w_2w_3w_4w_5$ represent the five available times, h_{a1} represents 7A-Hist, $e_{a1}e_{a2}$ represents 7A-English, and so on; m_1m_2, representing 7AB-Mathematics, lies in two subexpressions.



(c) A layer tree corresponding to (b). Variables are shown as labelled boxes; + nodes are shown as concatenations of their variables.



(d) The layer tree of (c), showing unifications representing the timetable of (a). The × nodes have been omitted for clarity.

Fig. 1. Timetables, timetable expressions, layer trees, and unification.

products. Some of the e_{ij} may be shared, i.e. some e_{pq} and e_{rs} may be the same subexpression. To *solve* a restricted sum of products is to unify each of the restriction variables in each e_{ij} to one of the w_i.

One way to solve a timetabling problem represented by a simple timetable expression is to solve its restricted sums of products in bottom-up order. This paper aims for more flexibility, however, in allowing unifications and de-unifications within each restricted sum of products at any moment. For example, this would permit the timetable of a small component to be adjusted (by local search, perhaps) after that component is incorporated into a larger timetable. To achieve this we need a data structure which represents the entire tree of restricted sums of products, with the current state of the unifications of each.

The data structure we will use, which we call a *layer tree*, is essentially just the expression tree corresponding to a simple timetable expression. A layer tree

has two types of nodes: + nodes representing restricted sums of products and containing their restriction variables, and × nodes representing concatenations. Nodes of both types may have any number of children. Figure 1 gives an example of converting a restricted sum of products into a layer tree.

Without loss of generality, we may assume that in every layer tree the root is a + node, its children are × nodes, their children are + nodes, and so on, with the node type alternating between + and × at each level. To bring an arbitrary layer tree into this form, first use the associativity of concatenation to replace every × node whose parent is a × node by its children. Then insert a × node immediately above every + node whose parent is a + node. Finally, if the root is a × node, remove it and solve each of its children independently.

Each variable v within each + node other than the root node requires unification with a variable w in the + node two levels above it. Each such unification is represented by a pointer in v to w (Figure 1d). Eventually, when all these variables are unified in this way, every variable may be said to have been assigned a time, obtainable by following the chain of pointers to its end.

Any set of variables requiring distinct times is called a *layer*. The variables lying in any + node form a layer; the variables lying in all the children of any × node also form a layer.

For example, the author's KTS system builds a layer tree with several levels. Each meeting may contain submeetings which have to be timetabled into the times of the meeting; each such meeting becomes a restricted sum of products. Then small groups of compatible meetings are timetabled together, producing *tiles* such as the one in Figure 1a; each tile is the solution of a restricted sum of products whose child layers contain meetings. Finally, the times of the week form a restricted sum of products whose child layers contain tiles.

4 Time constraints

This section explains how constraints on time assignment are propagated through the layer tree, so that at any moment it is clear for each variable exactly which variables it may be unified to without violating any time constraints.

Since each variable is unified to at most one other variable at any moment, the unifications form a directed forest with edges pointing towards the roots. The current unification of a variable v will be denoted $p(v)$ ('parent of v') when present, and the variable at the root of the tree of unifications containing v (possibly v itself) will be denoted $r(v)$. A *root variable* is a variable w such that $r(w) = w$. Every variable in the root node of a layer tree must be a root variable, but other variables may also be root variables: root variables are just variables that are currently not unified to other variables.

Recall that each time variable v has its *initial domain* $tdom_0(v)$ of times that it may be assigned initially, and its *current domain* $tdom(v)$ of times that it may be assigned to at the current moment. We require

$$tdom(v) \subseteq tdom_0(v)$$

since otherwise the original constraint has been lost.

Each time variable v has a second kind of domain, its *variable domain* $vdom(v)$, which is the set of variables that v may be unified to. Again, $vdom_0(v)$ will denote the initial value of $vdom(v)$, and we require $vdom(v) \subseteq vdom_0(v)$. For each variable v_{ij} in the restricted sum of products

$$w_1 w_2 \ldots w_m : (v_{11} v_{12} \ldots v_{1k_1} + \cdots + v_{m1} v_{m2} \ldots v_{mk_m})$$

we have $vdom_0(v_{ij}) \subseteq \{w_1, w_2, \ldots, w_m\}$.

The two domains are related by the condition

$$w \in vdom(v) \Rightarrow tdom(w) \subseteq tdom(v)$$

($\Rightarrow$ is implication). For example, this prohibits a preassigned variable from being unified to an unpreassigned one; in general, it prevents w from being assigned a time not acceptable to v.

The following formulas show how $tdom(v)$ and $vdom(v)$ may be kept up to date as variables are unified and deunified:

$$tdom(v) = tdom_0(r(v))$$

and

$$vdom(v) = \{w \in vdom_0(v) \mid tdom(w) \subseteq tdom(v)\}$$

These follow easily from the discussion so far. Note that $vdom(v)$ is only needed at moments when v is not unified.

When a variable v is unified to another variable w, the variable domains of all variables concatenated with v need to be reduced by removing w, since unifying any of them with w would violate the constraint that concatenated variables must be assigned distinct times. An efficient method of doing this is as follows.

Let the set of variables lying in the children of one $\times$ node be $v_1, \ldots, v_m$; these variables form a layer which we call L. The variables in the parent of that $\times$ node form another layer, which we call $p(L)$. The variables of L must be unified to the variables of $p(L)$.

For each v_j, define the *child layer set*, $cl(v_j)$, to be the set of $\times$ nodes which are the parents of the $+$ node containing v_j. (As explained earlier, a $+$ node may have several parents, typically because the meeting it represents contains several preassigned resources.) For each w_i, define the *parent layer set*, $pl(w_i)$, to be the union of the child layer sets of all variables unified directly to w_i. Parent layer sets must be maintained dynamically as unifications are done and undone.

Now modify the definition of $vdom(v)$ given above to

$$vdom(v) = \{w \in vdom_0(v) \mid (tdom(w) \subseteq tdom(v)) \wedge (cl(v) \cap pl(w) = \emptyset)\}$$

This excludes w from $vdom(v)$ when some other variable that shares a layer with v is currently unified to w. The set operations may be implemented efficiently using bit vectors.

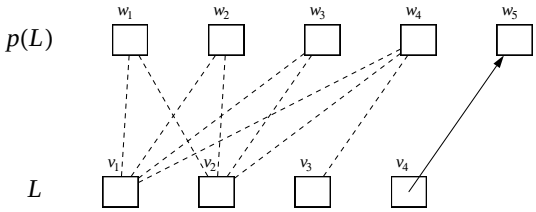


Fig. 2. An example of an unweighted bipartite matching graph between the variables of a child layer L and its parent layer $p(L)$, shown as dashed edges. One unification is already present, from v_4 to w_5, ensuring that $L \in pl(w_5)$ and thus excluding w_5 from $vdom(v)$ for all other $v \in L$. This particular matching could arise when v_3 and w_4 are preassigned to the same time ($tdom(v_3) = tdom(w_4) = \{t_i\}$ for some $t_i \in T$), and the other variables are free to be assigned any time. Note that $w_4 \in vdom(v_1)$ but no maximal matching would unify v_1 to w_4.

Given current values of $vdom(v)$ for all variables v in some layer L, the next question is whether it is possible to unify all the currently un-unified variables of L to variables in $p(L)$. Since the unifications must be to distinct variables, this is an unweighted bipartite matching problem between the currently un-unified variables of L and the variables of $p(L)$, with edges defined by the current values of the domains $vdom(v)$ of the currently un-unified variables of L (Figure 2). We will see in the next section that there are reasons for preferring some unifications to others, converting the unweighted bipartite matching into a weighted one.

5 Resource constraints

In addition to requests for times, meetings contain requests for resources. These may be for particular resources, called *preassigned* resources, or for any resource of a certain type, such as a Science laboratory.

A typical meeting requests one preassigned student group resource, one teacher which may or may not be preassigned, and one room, usually not preassigned. However, it is very common for a whole collection of meetings to be required to run simultaneously, to give the students a choice of activities. Such a collection is modelled as a single large meeting with many resource requests.

A basic question which can be asked of any set of meetings is whether the institution has sufficient resources to allow those meetings to run simultaneously. For example, if the school has only two Music teachers and two Music rooms, then at most two Music meetings may run simultaneously. As is well known, this question can be answered using an unweighted bipartite matching model, called a *resource sufficiency matching* [8], as follows.

For each request for a resource in each of the meetings involved, create one node called a *demand node*. For each resource in the instance of the timetabling problem being solved, create one node called a *supply node*. Connect each demand node to those supply nodes capable of satisfying that demand. For exam-

ple, a demand node for a particular student group resource would be connected to just the supply node representing that resource; a demand node for a Science laboratory would be connected to every supply node representing a Science laboratory. The meetings may run simultaneously if a maximum matching in this graph touches every demand node. The matching defines an assignment of resources to requests which satisfies as many requests as possible.

This model allows supply nodes which are capable of satisfying several kinds of demands: teachers who teach both English and History, rooms which are Science laboratories but are usable as ordinary classrooms, and so on. The obvious simpler method, of comparing the total number of demands of each type with the total supply of resources of that type, fails to handle such cases.

We turn now to the implementation of these ideas within the layer tree. Associated with each time variable is a set of demand nodes, which we call a *demand chunk*. For example, a Music meeting might request student group 7*C*, one Music teacher, and one Music room for four times, and then there will be four variables, each with an associated chunk containing three demand nodes. These chunks happen to be identical, but they are copies, not shared.

Any time variable may have a demand chunk, whether or not it derives from a meeting. The variables of the root layer, for example, have chunks that express resource unavailability: if resource r is unavailable at time t_i, then the chunk associated with root layer variable w_i will contain a demand for r.

The layer tree treats time constraints as hard constraints, in that it is not designed to track the number of violations of these constraints, merely to prohibit them. For resource constraints however we have a free choice of whether to treat them as hard or soft constraints, and we will follow the KTS implementation in treating them as soft constraints. The aim is therefore not to fail when resources are insufficient, but rather to report the number of unmatchable demand nodes. This is calculated by having one bipartite graph for each root variable, in which all the demand chunks of all the variables unified to that root variable directly or indirectly are accumulated (since the unifications have caused these demands to be simultaneous), and supply nodes for all the resources of the instance as usual, and finding a maximum matching in each of these graphs.

The standard algorithm for unweighted bipartite matching has some useful properties which permit matchings to be calculated in an incremental manner. Briefly, one can push and pop demand chunks onto and off a matching graph in stack order (last-in-first-out) without recalculating the matching from scratch. The supply nodes remain constant throughout. Thus, when a unification of v to w is made, one can simply push the demand chunks from v's subtree (that is, the chunks associated with v and every variable currently unified to v, directly or indirectly) onto $r(w)$'s matching graph; when a de-unification of v to w is made, one must pop chunks off $r(w)$'s graph until all v's subtree's chunks are popped, then push back onto $r(w)$'s graph all chunks that were popped off during this process that were not from v's subtree. The KTS implementation uses lazy evaluation, merely recording requests for pushes and pops, and not doing anything until a request for the number of unmatchable nodes is received,

at which point one sequence of pops followed by one sequence of pushes brings the matching up to date.

We return now to the unweighted bipartite matching problem mentioned at the end of the preceding section, between the un-unified variables of a layer L and the variables of its parent layer $p(L)$. For each un-unified variable v of L, we saw that the current domain $vdom(v)$ determines which edges to place in the bipartite graph. Now with each such edge, from v to w say, we can associate a cost: the number of additional unmatched nodes that would occur if v was unified to w, calculated by matching the chunks of v's subtree and $r(w)$'s subtree together without actually making the unification. A maximum matching between L and $p(L)$ of minimum total cost will give a lower bound on the number of additional unmatched demand nodes that will occur when the un-unified variables of L are unified to variables of $p(L)$. This model has been called *weighted meta-matching* in [10], where it provides a valuable forward check.

The KTS implementation recalculates edge costs only when changes to the demands at either end make that necessary. It calculates weighted matchings lazily on demand, but not incrementally. Although a well-known algorithm exists which can do this, by finding negative-cost cycles in the residual graph, it is slow since it requires the use of the Bellman-Ford shortest path algorithm rather than Dijkstra's [1]. Fortunately the graphs are small, since the number of nodes per layer is at most the number of times in the week (typically about 40), so calculating these weighted matchings from scratch is not time consuming.

6 Other features

In this section we briefly survey some other features of the KTS layer tree. They serve as examples of how the basic ideas can be extended.

Time blocks. A sequence of times that follow each other chronologically without a break is called a *time block*. For example, the first four times on Monday might form a time block. Then after a lunch break there might be four more times followed by an end-of-day break. In KTS, meetings may request that their times have a particular *block structure*. For example, a meeting with 6 times might request two doubles (blocks of two times) and two singles.

The KTS layer tree allows time variables to be grouped into blocks. The time variables of a layer of meetings are grouped into blocks defined by the meetings' block structure requests; the time variables of the root layer (representing the times of the week) are grouped into blocks representing the sets of times between the naturally occurring breaks.

An initial problem is to determine whether the time blocks of some layer can be packed into the time blocks of the week, allowing for the fact that (for example) a block of four times on Monday morning can be split into two doubles, or one double and two singles, or whatever is required. This is an NP-complete bin packing problem, but real instances are small and easily solved.

Once such a packing has been found, and the large blocks of the week broken down into smaller blocks that exactly match the meetings' block structure

requests, the layer tree implements a weighted meta-matching between blocks rather than individual variables. Two blocks are connected by an edge if they have the same number of variables and corresponding variables within the blocks would be connected by an edge in the unblocked matching. The cost of a block-to-block edge is the sum of the costs of the variable-to-variable edges it replaces.

The layer tree offers a heuristic algorithm which simultaneously carries out the bin packing and builds the blocked matching. Initially the matching contains all the parent layer blocks as supply nodes, and no child layer demand blocks. The child blocks are introduced into the matching one by one in decreasing width order. If a child block fails to match, a series of repair operations is tried on the parent blocks: larger blocks are split, variables not yet in any block are merged into blocks, and so on. For each type of repair, all possible repairs of that type are tried, and the one which produces a blocked matching of minimum cost is accepted; or if none of them succeed in producing a matching which touches every demand block, the algorithm proceeds to the next, less desirable, kind of repair. As a last resort, one demand block (usually the one just introduced) is dropped and replaced by its variables.

The decisions about how to split parent blocks made by this algorithm depend on the state of resource sufficiency in those blocks' variables. Consequently it is not useful to build a blocked matching for every child layer of a restricted sum initially. Rather, the usual unblocked matchings are built for each child layer, then a child layer's unblocked matching is replaced by a blocked matching as the first step in assigning that layer. The blocked matching is a temporary structure, only in existence while its layer is being assigned.

Blocked matchings suffer from an awkward problem. Suppose a meeting requires one double and one single block. The matching unifies the double to the first two times on Monday; it unifies the single to the third time on Monday. The result is a triple, not a double plus a single. Finding a minimum-cost matching which avoids this problem appears to be NP-complete. KTS's weighted meta-matching algorithm discourages such unifications by artificially increasing the cost of augmenting paths that would produce them. The implementation has been done with care, and runs in time which is often a small constant, and at worst is proportional to the length of the augmenting path being considered. The idea is purely heuristic, to be sure, but it seems to work well.

Many other conditions besides time blocks may be imposed on sets of times. A meeting's times may be required to be spread evenly through the week, the times of the meetings attended by a student group may be required to be *compact* (contain no gaps within any day), and so on. The author has not yet attempted to support such conditions within the layer tree.

Regularity. The layer tree supports regularity by supporting hierarchical timetable construction, but this does not of itself encourage regularity between the child layers of each + node. We mentioned earlier a straightforward way to do this, by partitioning the variables of the parent layer into sets, called *columns*, whose size is a typical meeting size, and assigning meetings to entire columns wherever possible. This was the North American universities' approach.

Columns are supported by the layer tree by allowing temporary reductions in $vdom(v)$. An algorithm might restrict the domains of the variables of a meeting to one column, then check the total resource sufficiency badness of the entire layer tree; if it has not increased, assigning that meeting to that column may be good. The layer tree also maintains, for each set of variables representing one meeting, a count of the number of distinct columns that that meeting's variables are assigned to. The total of all these counts measures the current irregularity.

Evenness. It is desirable for demand for a particular type of resource to be spread evenly across the week, not concentrated at particular times. This is because resource assignment struggles at times when every resource of a particular type is required: there are enough resources, perhaps, but there is little freedom of choice. This property we call *evenness*.

Evenness, like resource sufficiency, depends on the resource demands made at each time, so the layer tree's support for it is very similar to its support for resource sufficiency. (There does not seem to be any efficient way to extract evenness information from the resource sufficiency matchings themselves.) The total demand for each type of resource is maintained in root variables. The sum of the squares of these totals is an effective and easily updated overall measure of unevenness. For example, two root variables each demanding a quantity a of some type of resource contribute $2a^2$ to total unevenness. If the timetable is changed so that one demands quantity $a-1$ and the other demands $a+1$, these less even demands contribute $2a^2+2$ to unevenness. Demands from the same faculty (e.g. Junior English and Senior English) are considered to be the same type of demand, since they typically have many resources in common.

Overall badness. For the convenience of algorithms that use the layer tree, the KTS implementation offers access to the current total badness of the tree, as a triple whose first component is the number of resource sufficiency defects implied by the current state (the total number of unmatched nodes in resource sufficiency matchings, plus the total cost of all meta-matchings), and whose second and third components are the irregularity and unevenness, measured as just described. Each data structure responsible for calculating any badness value at any point in the tree also takes responsibility for reporting any change to this global badness object, or at least reporting itself as out of date and needing recalculation the next time a badness value is requested.

7 Conclusion

This paper has defined a form of hierarchical timetable specification and shown how support for it can be implemented efficiently using the layer tree data structure. Time assignments and deassignments may be carried out at any point in the tree, and an efficient constraint propagation algorithm updates the domains of the variables and reports the consequences for resource sufficiency at each time. Extensions to the basic framework, supporting block structure, regularity, and evennness, have been implemented in the author's KTS system.

Future work will try to add more features to the layer tree without compromising its efficiency. It may be possible to incorporate information about workload limits into the resource sufficiency matchings, for example. A second goal is to design new timetabling algorithms that fully exploit the flexibility of this innovative data structure.

References

1. Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin: Network Flows: Theory, Algorithms, and Applications. Prentice Hall, 1993
2. Krzysztof R. Apt: Principles of Constraint Programming. Cambridge University Press, 2003
3. Edmund Burke and Peter Ross (eds.): Practice and Theory of Automated Timetabling (First International Conference, PATAT'95, Edinburgh, August 1995). Springer Lecture Notes in Computer Science 1153, 1995
4. Edmund Burke and Michael Carter (eds.): Practice and Theory of Automated Timetabling II (Second International Conference, PATAT'97, University of Toronto, August 1997, Selected Papers. Springer Lecture Notes in Computer Science 1408, 1998
5. Edmund Burke and Wilhelm Erben (eds.): Practice and Theory of Automated Timetabling III (Third International Conference, PATAT2000, Konstanz, Germany, August 2000, Selected Papers). Springer Lecture Notes in Computer Science 2079, 2001
6. Edmund Burke and Patrick de Causmaecker (eds.): Practice and Theory of Automated Timetabling IV (Fourth International Conference, PATAT2002, Gent, Belgium, August 2002, Selected Papers). Springer Lecture Notes in Computer Science 2740, 2003
7. Edmund Burke and Michael Trick (eds.): Practice and Theory of Automated Timetabling V (Fifth International Conference, PATAT2004, Pittsburgh, PA, August 2004, Selected Papers). Springer Lecture Notes in Computer Science 3616, 2005
8. Tim B. Cooper and Jeffrey H. Kingston: The solution of real instances of the timetabling problem. The Computer Journal **36**, pages 645–653, 1993
9. Pascal Van Hentenryck: The OPL Optimization Programming Language. MIT Press, 1999
10. Jeffrey H. Kingston: A tiling algorithm for high school timetabling. In Proceedings of the 5th International Conference on the Practice and Theory of Automated Timetabling, Pittsburgh, PA, pages 233-249, August 2004
11. Jeffrey H. Kingston: The KTS high school timetabling web site (Version 1.3), October 2005. http://www.it.usyd.edu.au/~jeff
12. Jeffrey H. Kingston: The KTS high school timetabling system. Submitted to 6th International Conference on the Practice and Theory of Automated Timetabling, Brno, Czech Republic, August 2006
13. W. J. van Hoeve: A hyper-arc consistency algorithm for the soft alldifferent constraint. Tenth International Conference on Principles and Practice of Constraint Programming (CP 2004), Springer-Verlag Lecture Notes in Computer Science 3258, pages 679-689, 2004
14. D. de Werra: An introduction to timetabling. European Journal of Operational Research **19**, pages 151-162, 1985