10th International Conference of the Practice and Theory of Automated Timetabling
PATAT 2014, 26-29 August 2014, York, United Kingdom

KHE14: An Algorithm for High School Timetabling

Jeffrey H. Kingston

Abstract This paper presents work in progress on KHE14, an algorithm for high school timetabling. Most of KHE14's components have been published previously, so are described only briefly. A few components, notably many of the augment functions called by the ejection chain algorithm, are new, so are described in detail. The paper includes experiments on the standard XHSTT-2013 data set, performed in February 2014.

Keywords High school timetabling · Ejection chains

1 Introduction

High school timetabling is one of the three major timetabling problems found in academic institutions, along with university course timetabling and examination timetabling. Automated methods for its solution have been studied from the early days of computers [19] to the present day [15].

An XML format called XHSTT was introduced recently to represent real instances and solutions of the high school problem [8, 16]. XHSTT was used in the Third International Timetabling Competition [18], the first competition to include high school timetabling. An XHSTT data set called XHSTT-2013 is currently being promoted as a benchmarking standard [17]. It contains 24 instances from real high schools in 11 countries around the world.

This paper presents work in progress on KHE14, an algorithm for high school timetabling, with experiments on XHSTT-2013 made in February 2014.

KHE14 is built on the author's KHE high school timetabling platform [12], and distributed with it. It has many parts, developed by the author in a series of papers over the last ten years [6, 7, 9–11]. All this cannot be repeated here. So although this paper describes KHE14 completely, it does so only at a high

J. Kingston
School of Information Technologies, The University of Sydney, Australia
E-mail: jeff@it.usyd.edu.au

level. Details are explained only when they are new and significant; otherwise they are just mentioned, with a reference to the papers just cited, or to the KHE documentation [12], which has full details. The main innovations are in Sect. 7, where new repairs for several kinds of defects are given.

Sect. 2 gives a brief specification of the problem. Sects. 3–7 present the components of KHE14, with experiments related to the components. Sect. 8 brings the components together into the full KHE14 algorithm and contains experiments that evaluate it generally.

All experiments use the XHSTT-2013 data set [16], as downloaded on 4 February 2014, and were performed on the author's desktop machine, an Intel i5 quad-core running Linux. Individual solutions were produced single threaded; multiple solutions for one instance were produced in parallel. The DK-HG-12 instance is not tested, because it is not realistic; it has 411 demand defects (Sect. 4) before solving starts (connected with resources Student539 and R36, for example). Instances NK-KP-05 and NK-KP-09 are not tested because their run times are inconveniently long at present. KHE14 does solve these instances, but it takes approximately one hour on each.

2 Problem specification

This paper uses the XHSTT specification of the high school timetabling problem. XHSTT instances contain four parts: the *cycle*, which is the chronological sequence of times that may be assigned to events; *resources*, which are entities that attend events (teachers, rooms, students, or classes, where classes are groups of students who mostly attend the same events); *events*, which are meetings, each of fixed duration (number of times), and containing any number of *event resources*, each specifying one resource that attends the event; and *constraints*, which specify conditions that solutions should satisfy, and the penalty costs to impose when they don't.

XHSTT currently offers 16 constraint types (Table 1). Whatever its type, each constraint may be marked *required*, in which case it is called a *required* or *hard* constraint, and its cost (a non-negative integer) contributes to a total called the *infeasibility value* in XHSTT, and the *hard cost* here. Otherwise the constraint is called *non-required* or *soft*, and its cost contributes to a different total called the *objective value* in XHSTT and the *soft cost* here.

A solver assigns starting times to events, except for *preassigned* events (events whose starting time is given by the instance), trying to minimize first hard cost and then soft cost. It may also be required to split events of long duration into smaller events, called *sub-events* in XHSTT and *meets* in KHE and in this paper. And it may be required to assign resources to unpreassigned event resources: often rooms, occasionally teachers, never (in practice) students or classes. A full specification appears online [8]; further details are given as needed throughout this paper.

Table 2 gives some idea of the instances of the XHSTT-2013 data set. They vary greatly in difficulty, in ways that such a table cannot fully capture.

Table 1 The 16 constraints, with informal definitions, grouped by what they apply to.

Event constraints	
Split Events constraint	Split event into limited number of meets
Distribute Split Events constraint	Split event into meets of limited durations
Assign Time constraint	Assign time to event
Prefer Times constraint	Assign time from given set
Spread Events constraint	Spread events evenly through the cycle
Link Events constraint	Assign same time to several events
Order Events constraint	Assign times in chronological order
Event resource constraints	
Assign Resource constraint	Assign resource to event resource
Prefer Resources constraint	Assign resource from given set
Avoid Split Assignments constraint	Assign same resource to several event resources
Resource constraints	
Avoid Clashes constraint	Avoid clashes involving resource
Avoid Unavailable Times constraint	Make resource free at given times
Limit Idle Times constraint	Limit resource's idle times
Cluster Busy Times constraint	Limit resource's busy days
Limit Busy Times constraint	Limit resource's busy times each day
Limit Workload constraint	Limit resource's total workload

Table 2 The number of times, teachers, rooms, classes (groups of students), individual students, and events in the instances of XHSTT-2013. There are 24 instances altogether.

Instance	**Times**	**Teachers**	**Rooms**	**Classes**	**Students**	**Events**
AU-BG-98	40	56	45	30		387
AU-SA-96	60	43	36	20		296
AU-TE-99	30	37	26	13		308
BR-SA-00	25	14		6		63
BR-SM-00	25	23		12		127
BR-SN-00	25	30		14		140
DK-FG-12	50	90	68		279	1120
DK-HG-12	50	100	70		523	1471
DK-VG-09	60	46	52		163	928
ES-SS-08	35	66	4	21		225
FI-MP-06	35	25	25	14		280
FI-PB-98	40	46	34	31		387
FI-WP-06	35	18	13	10		172
GR-H1-97	35	29		66		372
GR-P3-10	35	29		84		178
GR-PA-08	35	19		12		262
IT-I4-96	36	61		38		748
KS-PR-11	62	101		63		809
NL-KP-03	38	75	41	18	453	1156
NL-KP-05	37	78	42	26	498	1235
NL-KP-09	38	93	53	48		1166
UK-SP-06	25	68	67	67		1227
ZA-LW-09	148	19	2	16		185
ZA-WD-09	42	40		30		278

3 Timetabling structures

KHE evaluates constraints continuously as the solution changes during solving, using efficient incremental methods, and makes the resulting costs available to solvers, which use them to guide the solve as usual. If requested, KHE can also add structures to the solution which ensure that violations of some constraints cannot occur, and it can add other structures which encourage *regularity*: patterns of assignment that make timetables more uniform. Regularity has no direct effect on cost, but it may make good solutions easier to find [11].

KHE14's first, *structural* phase, is mainly devoted to adding the structures explained in this section. These are all optional as far as the KHE platform is concerned; KHE14 chooses to use them, but other algorithms need not.

KHE14 does not use any information that could be called metadata. For example, sets of times may be identified in XHSTT as days, but KHE14 does not use that information. Nor does it treat student resources (say) differently merely because they are called students. Instead, it examines which resources are preassigned, which sets of times and resources appear in constraints, and so on, taking its cues from the structure alone.

Many elements of the instance influence KHE14's structures: avoid clashes constraints (constraining events which share a preassigned resource to be disjoint in time), time preassignments, link events constraints, split events and distribute split events constraints, spread events constraints (influencing how many meets events split into), prefer times constraints, prefer resources constraints, and avoid split assignments constraints. These are taken in decreasing cost order; each either influences the structures, or is ignored if inconsistent with previous elements. The KHE documentation [12] explains in detail how they affect the result and how they interact. It would take too long to repeat that here. Instead, the following explains the structures that emerge.

Courses are sets of events during which the same students meet the same teacher to study the same subject. Spread events constraints may be present to encourage a course's meets to spread evenly through the cycle, and avoid split assignments constraints may be present to encourage those meets to be assigned the same teacher (if not preassigned) or room.

XHSTT offers a spectrum of ways to define courses. At one extreme, the exact set of events required is given. For example, if a Mathematics course needs to occur five times per week in events of durations 2, 1, 1, 1, and 1, then five events with these durations would be given, along with split events constraints which ensure that each event produces one meet. At the other extreme, a single event of the total duration required is given, along with split events and distribute split events constraints which say how that total may be split into meets. In the Mathematics example, a single event of duration 6 would be given, along with constraints saying that meets of duration 1 or 2 are required. This handles a situation frequently found in real instances, where the total duration is fixed, but how it is to be split up is more flexible.

The structural phase splits events into meets whose durations depend on the parts of the instance listed above, and groups the meets into sets that KHE calls *nodes*. One node contains the meets of one course, at least to begin with. The structural phase creates nodes heuristically, as follows. Meets derived from the same event go into the same node. When two events contain the same preassigned resources and are connected by a spread events or avoid split assignments constraint, they are taken to belong to the same course, so their meets also go into the same node. Grouping meets into nodes does not constrain their assignments, but it acts as a hint to solvers that the meets should be assigned times together, and opens the door to various methods of promoting regularity, which work with nodes, not meets.

Link events constraints, specifying that certain events should be assigned the same times, give rise to a different structure. KHE allows one meet to be assigned to another instead of to a time, meaning that any time assigned to the other meet is in fact assigned to both. The structural phase makes assignments of meets to other meets which ensure that link events constraints cannot be violated. Meets assigned to other meets are not included in nodes, which (by convention) tells solvers that their assignments should not be changed.

Assigning one meet to another supports *hierarchical timetabling*, in which a timetable for a few meets is built and later incorporated into a larger one. This promotes regularity, so the structural phase spends time searching for useful hierarchical structures, as described in [6,7].

Each meet contains a set of times called its *domain*. Only times from its domain may be assigned to a meet. KHE14 chooses domains based on prefer times constraints. The duration of a meet also affects its domain: a meet of duration 2 cannot be assigned the last time in the cycle as its starting time, and so on. KHE represents domains both as bit sets, for efficient assignability testing, and as lists of times, for efficient iteration over all legal assignments.

A meet contains one *task* for each event resource in the event that it is derived from. Each task is a demand for one resource at each of the times the meet is running, either preassigned (the usual case for student and class tasks) or not (the usual case for room tasks). Unpreassigned tasks specify the type of resource required (teacher, room, etc.), and there are usually prefer resources constraints which encourage the solution to assign a specific kind of resource, such as a Mathematics teacher or a Science laboratory.

Each task contains a set of resources called its *domain*. Only resources from its domain may be assigned to a task. KHE14 chooses domains based on prefer resources constraints.

Avoid split assignments constraints, which specify that certain tasks should be assigned the same resources, are handled structurally by KHE14. One of the tasks is chosen to be the *leader task*, and the others are assigned it instead of a resource, meaning that whatever resource is assigned to the leader task is to be considered as assigned to them too.

The XHSTT specification says that hard constraint violations, while permitted, should be few in good solutions, but that soft constraint violations are normal and to be expected [8]. So additional structures must be used with caution, especially when derived from soft constraints. KHE14 uses an heuristic strategy: it includes them at first, but removes them towards the end, so

Table 3 Encouraging regularity between forms: -RF and +RF denote without it and with it. KHE14 uses +RF. In all tables in this paper, columns headed C: contain solution costs. Hard costs appear to the left of the decimal point; soft costs appear as five-digit integers to the right of the point. The minimum costs in each row are highlighted. Columns headed T: contain run times in seconds. All tables and graphs (including captions) were generated by KHE and incorporated unchanged. They can be regenerated by any user of KHE.

Instance	C:-RF	C:+RF	T:-RF	T:+RF
AU-BG-98	**9.00583**	12.00491	17.1	25.4
AU-SA-96	**4.00015**	16.00014	35.1	72.2
AU-TE-99	**1.00158**	4.00124	0.8	3.4
BR-SA-00	1.00090	**1.00057**	0.7	0.8
BR-SM-00	30.00123	**29.00093**	4.4	6.3
BR-SN-00	5.00249	**4.00243**	1.6	2.8
DK-FG-12	0.02370	**0.02248**	94.0	180.9
DK-HG-12				
DK-VG-09	**12.03206**	12.03349	393.9	368.4
ES-SS-08	0.02367	**0.01362**	4.4	15.6
FI-MP-06	**0.00118**	3.00132	1.5	8.0
FI-PB-98	**0.00051**	6.00039	1.7	6.3
FI-WP-06	0.00086	**0.00078**	1.7	1.3
GR-H1-97	**0.00000**	**0.00000**	0.5	5.9
GR-P3-10	4.00078	**2.00088**	16.6	22.7
GR-PA-08	**0.00029**	0.00040	2.4	6.9
IT-I4-96	0.00602	**0.00494**	4.6	8.1
KS-PR-11	0.00160	**0.00150**	10.7	77.1
NL-KP-03	**0.03825**	0.04774	86.4	193.1
NL-KP-05				
NL-KP-09				
UK-SP-06	0.00196	**0.00102**	14.0	29.5
ZA-LW-09	29.00000	**26.00000**	16.9	18.9
ZA-WD-09	**5.00000**	9.00000	6.3	24.2
Average	**4.00681**	5.00660	34.0	51.3

that later repair operations are not limited by them. The original constraints are not forgotten: even when violations are allowed, they are still penalized.

The author has used a structural phase like the one described here for many years [6,7]. KHE14's version, described briefly here and fully in the KHE documentation [12], is more robust than its predecessors: it resolves conflicting requirements using priorities as explained above, and it takes full account of all interactions between requirements.

Testing the effectiveness of adding structures that encourage regularity is complicated by the fact that there are several kinds of regularity and several ways to encourage it [11], not all of which can be disabled at present. Table 3 investigates regularity between forms. For example, if the classes of the Year 11 form attend English for 6 times per week in meets of durations 2, 1, 1, 1, and 1, and the classes of the Year 12 form attend Mathematics for 6 times per week in meets of the same durations, then encouraging regularity between forms encourages these two courses (or others with the same meet durations) to be simultaneous. The results show no clear advantage in cost, and a clear disadvantage in running time. It is too soon to abandon regularity between forms, but the evidence of Table 3 is tending against it.

4 The global tixel matching

A timetabling problem is a market in which resources are demanded by events and supplied to them. The unit of supply is one resource at one time, called a *supply tixel*. The term 'tixel' has been coined by the author by analogy with the 'pixel', one cell of a graphical display.

Each event demands a number of tixels of certain types. For example, a typical event called *7A-English*, in which class *7A* studies English for 6 times per cycle, demands 18 tixels: six tixels of class resource *7A*, six tixels of teachers qualified to teach English, and six of ordinary classrooms. This event is said to contain 18 *demand tixels*.

The market is represented by an unweighted bipartite graph. Each demand tixel is a node; each supply tixel is a node. An edge joins demand tixel d to supply tixel s when s may be assigned to d. For example, a demand tixel demanding class resource *7A* would be connected to the supply tixels for resource *7A* (one for each time in the cycle). A demand tixel demanding an English teacher would be connected to each supply tixel of each English teacher.

Each demand tixel requires only one supply tixel. Each supply tixel can be assigned to only one demand tixel, otherwise there would be a timetable clash. Accordingly, a set of assignments is a *matching* in this graph: a set of edges such that no two edges share an endpoint. There is an efficient algorithm for finding a maximum matching (one with as many edges as possible) [14].

There may be many maximum matchings, but they all fail to assign supply tixels to the same number of demand tixels, and since that number is the important thing, it is convenient to pretend that there is just one maximum matching. The author calls it the *global tixel matching*. The important number is a lower bound on the number of unassigned demand tixels in any solution, given the decisions already made. The matching defines an assignment which maximises the number of tixels assigned, but it is not useable directly, because it violates many constraints.

When a meet is assigned, the sets of edges connected to its demand tixels (their *domains*) shrink. For example, the six tixels demanding resource *7A* in the meets of event *7A-English* are initially connected to all the supply tixels for *7A* (one for each time of the cycle), but after times are assigned, each becomes associated with a particular time, and is connected to just the supply tixel for *7A* at that time. Tixel domains also change when the domain of a meet or task is changed. KHE keeps them up to date automatically.

Use of the global tixel matching is optional. KHE14 installs it during its structural phase and retains it until the end. It uses it in two ways. First, while times are being assigned, each unmatched demand tixel adds hard cost 1 to the total cost, guiding the solver away from assignments that would lead to problems later. For example, assigning 6 Science meets to some time, demanding 6 Science laboratories then, will be penalized if there are only 5 Science laboratories, even though no room assignments are made. Second, while resources are being assigned, only assignments that do not increase the number of unmatched demand tixels are permitted (until the end, when a last-ditch attempt is made to assign any remaining unassigned tasks). The rationale is that unmatched demand tixels lead inevitably to defects, and if their number is allowed to increase, there is little hope of reducing it again.

Additional demand tixels are added based on hard unavailable times, limit busy times, and limit workload constraints. For example, if teacher Smith is limited to at most 7 busy times out of the 8 times on Monday, then one demand tixel demanding Smith at a Monday time is added. This section is adapted from [10], which contains much more detail: how these additional tixels are defined, how to implement the matching efficiently, and so on.

5 Polymorphic ejection chains

Like most timetabling solvers, KHE14 first constructs, then repairs. Most of the repair work is done by *ejection chains*. An ejection chain is a sequence of one or more *repair operations* (also called *repairs*), which are often simple operations such as moves and swaps. The first repair removes one *defect* (a specific fault in the solution) but may introduce another; the next repair removes that defect but may introduce another; and so on. A key point is that the defects that appear as a chain grows are not known to have resisted attack before. It might be possible to repair one of them without introducing another, bringing the chain to a successful end.

Ejection chains are not new. They are the augmenting paths of matching algorithms, and they occur naturally to anyone who tries to repair a timetable by hand. They were brought into focus and named by Glover [3], in work on the travelling salesman problem. In timetabling, they have been applied to nurse rostering [2], resource assignment [10], and time repair [4,5,11].

A key insight of [11] is that ejection chains are naturally *polymorphic*: each defect along one chain can have a different type from the others, calling for a correspondingly different type of repair. Thus, any number of types of defects, and any number of types of repairs, can be handled together. In KHE, there is one defect type for each constraint type, representing one specific point in the solution where a constraint of that type is not satisfied, plus one defect type representing one specific unmatched demand tixel in the global tixel matching.

An ejection chain algorithm uses a set of functions, one for each kind of defect, called *augment functions* since they are based on the function for finding an augmenting path in bipartite matching [14]. An augment function is passed a specific defect of the type it handles, and it tries a set of alternative repairs on it. Each repair removes the defect, but may create new defects. If no significant new defects appear, the augment function terminates successfully, having reduced the solution cost. If one significant new defect appears (one whose removal would reduce the solution cost below its value when the chain began; it may cost more than the previous defect), it makes a recursive call to the appropriate augment function for that defect in an attempt to remove it. In this way a chain of coordinated repairs is built up. If the recursive call does not succeed in improving the solution, or was not attempted because two

Table 4 Effectiveness of variants of KHE14's ejection chain algorithm. Each pair of characters represents one complete restart of the algorithm: a digit denotes a maximum chain length (u means unlimited); + denotes allowing entities to be revisited along one chain, and - denotes not allowing it. KHE14 uses 1+,u-. Other details as previously.

Instance	C:u-	C:1+,u-	C:1+,2+,u-	T:u-	T:1+,u-	T:1+,2+,u-
AU-BG-98	9.00571	12.00491	**8.00500**	**22.6**	25.4	24.4
AU-SA-96	**14.00024**	16.00014	17.00027	81.1	**72.6**	76.2
AU-TE-99	2.00130	4.00124	**2.00090**	3.9	3.5	**3.1**
BR-SA-00	1.00072	1.00057	**1.00054**	1.0	**0.8**	1.5
BR-SM-00	**25.00135**	29.00093	27.00135	10.1	**6.4**	10.8
BR-SN-00	4.00252	**4.00243**	5.00246	3.7	2.9	**2.0**
DK-FG-12	0.02390	**0.02248**	0.02347	**160.8**	180.2	302.9
DK-HG-12						
DK-VG-09	**12.03206**	12.03349	12.03498	544.1	**375.8**	756.8
ES-SS-08	1.02081	**0.01362**	1.02639	17.1	**16.0**	43.1
FI-MP-06	4.00120	3.00132	**3.00117**	12.8	**7.8**	9.0
FI-PB-98	5.00051	6.00039	**4.00056**	5.5	6.3	**4.7**
FI-WP-06	**0.00049**	0.00078	0.00079	1.5	1.3	**1.2**
GR-H1-97	**0.00000**	**0.00000**	**0.00000**	6.2	**5.9**	5.9
GR-P3-10	2.00047	2.00088	**0.00009**	37.5	22.6	**12.4**
GR-PA-08	**0.00035**	0.00040	0.00040	**6.2**	7.2	6.9
IT-I4-96	1.01111	**0.00494**	0.00512	**7.8**	8.1	8.0
KS-PR-11	**0.00130**	0.00150	0.00135	77.6	**77.0**	78.2
NL-KP-03	**0.03707**	0.04774	0.04547	232.2	**197.8**	226.2
NL-KP-05						
NL-KP-09						
UK-SP-06	2.00144	**0.00102**	0.00182	46.8	**30.0**	34.9
ZA-LW-09	**25.00000**	26.00000	26.00000	**18.1**	20.2	18.7
ZA-WD-09	11.00000	**9.00000**	**9.00000**	26.5	24.0	**18.9**
Average	5.00678	**5.00660**	5.00724	63.0	**52.0**	78.4

or more significant new defects appeared, the augment function undoes the repair and continues with alternative repairs. It could try to remove a whole set of new defects, but that would rarely succeed in practice.

The main loop of the algorithm repeatedly iterates over the defects of the solution, or over a subset of them that it is expedient to target, and calls the appropriate augment function on each. It terminates when one complete pass over all defects yields no reduction in solution cost.

KHE offers two methods for preventing the tree of repairs searched by an augment function from growing to exponential size: either the length of the chains is limited to at most some fixed constant, or else it is unlimited, but entities visited while searching for one chain are marked, and revisiting them is prohibited, limiting the size of one search to the size of the solution.

Table 4 investigates these options for limiting the search. KHE14's choice has the lowest average cost and run time, but there is no clear signal.

Entities are marked in a way that prevents repairs on other chains, or further along the same chain, from targeting that entity, while allowing any number of alternative repairs of the entity to be tried where it is marked. Only the first entity changed by each repair is marked. Other entities changed by it are neither checked for being marked nor marked. This is important, since one

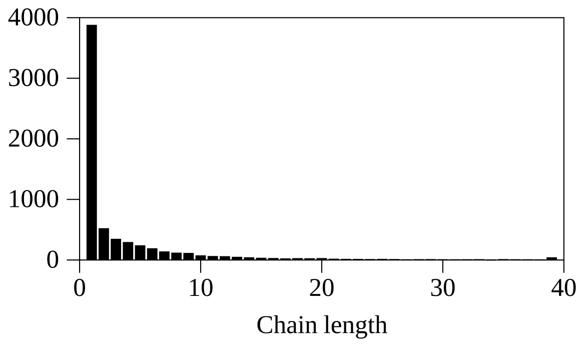

Fig. 1 For each chain length, the number of successful chains of that length found during time repair, over all tested instances of XHSTT-2013. There were 6322 successful chains altogether, and their average length was 3.5. All successful chains of length greater than 39 are shown as having length 39. The longest successful chain had length 140.

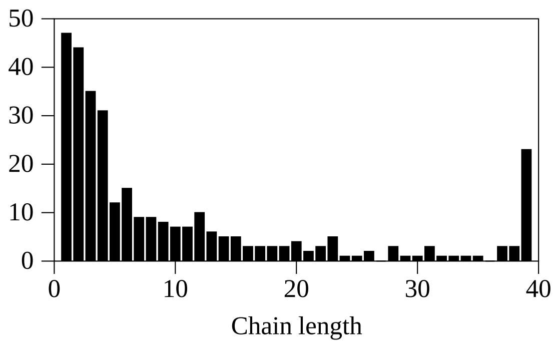

Fig. 2 For each chain length, the number of successful chains of that length found during resource repair, over all tested instances of XHSTT-2013. There were 321 successful chains altogether, and their average length was 11.5. All successful chains of length greater than 39 are shown as having length 39. The longest successful chain had length 105.

of them is likely to be targeted next. As long as at least one entity is marked by each repair, the size of the search will be limited as desired.

KHE14 makes two kinds of calls to the ejection chain algorithm: *time repair* calls, which repair time assignments, and *resource repair* calls, which repair resource assignments. Fig. 1 shows how long successful time repair chains are, and Fig. 2 does the same for resource repair. Most are short, but some are long. The average length of resource repair chains is surprisingly high. It was shown in [11] that chain lengths tend to increase as the algorithm progresses.

The text of this section is adapted from [11], which also explains how ejection chains are implemented in KHE. The user writes one augment function for each defect type, which iterates over the alternative repairs, applying and unapplying each in turn. KHE supplies the main loop, chaining together of individual repairs, testing for success, and dynamic dispatch by defect type.

6 Repair operations

A *repair operation*, or just *repair*, is a change to the solution made in the hope of removing a specific defect. This section presents the repairs used by KHE14's ejection chain algorithms. A detailed description of two unusual repairs is given first, followed by a brief description of the full set. Sect. 7 explains how they are used to repair the various kinds of defects.

Node swaps. A node is a set of meets, grouped together because they make up one course (Sect. 3). Suppose two nodes have meets of the same durations (one of duration 2 and four of duration 1, say). A node swap swaps the starting times of corresponding meets in those nodes.

Pairs of swappable nodes are common, since it simplifies planning if many courses have equal duration, and courses of equal duration often split into meets of equal durations. Nodes are only swapped when they have the same preassigned resources, so swapping avoids introducing clashes involving those resources. Swapping nodes has some advantages over swapping meets: it preserves regularity, and tends not to create new spread events defects.

Kempe meet moves. These begin with the move of a meet from its current time t_1 to some other time t_2. If that causes clashes between preassigned resources at t_2, the other meets involved in the clashes are moved to t_1, any clashes produced by those moves cause more meets to be moved to t_2, and so on until there are no new clashes and no more moves.

When a Kempe meet move succeeds, the result is usually a simple move or swap. Having a single operation which could turn out to be either is convenient, since it allows a solver to try moving a problem meet to each time t_2, whether the affected resources are free then or not.

Node swaps and Kempe meet moves are implemented more generally than described here, including support for hierarchical timetabling and preserving regularity. Kempe meet moves can swap meets of different durations when they are adjacent in time. A full description appears in [11], except for one recent improvement, in how regularity is preserved [12].

Turning now to the full list of repair operations, let a *variable* be a meet or a task, considered as an entity requiring a time or resource to be assigned to it. An *assignment* is a change to a variable from unassigned to assigned. A *move* is a change to a variable from one assignment to a different assignment. An *unassignment* is a change to a variable from assigned to unassigned.

When the change is an assignment or move, the new value of the variable is likely to create conflicts (timetable clashes) with other variables. There are at least four ways to handle these conflicts. The *basic* way is to do nothing, leaving it to the ejection chain algorithm to notice the resulting defects and try to repair them. The *ejecting* way is to unassign conflicting variables. This will be better than the basic way if it produces a single defect (an assign time or assign resource defect) rather than several defects whose common cause may not be clear to the ejection chain algorithm. The *swap* way, applicable only to moves, is to move the conflicting variables in the opposite direction. The

Kempe way, also applicable only to moves, is to continue swapping back and forth to remove conflicts, as explained above for Kempe meet moves.

Ignoring unassignment, which seems to be not useful alone, this makes six operations altogether: *basic assignment*, *ejecting assignment*, *basic move*, *ejecting move*, *swap*, and *Kempe move*. Applying them to meets and tasks gives twelve operations. KHE14 uses most of them, plus two operations which are sets of these ones treated as a unit: node swaps and *ejecting task-set moves*, which are sets of ejecting task moves to a common resource.

An ejecting move is a Kempe move that ends early, as soon as the variables to be moved in the opposite direction are unassigned. It often makes sense to first try a Kempe move, then fall back on an ejecting move; this is similar to trying a particular reassignment of the unassigned variables first. The term *Kempe/ejecting move* refers to a sequence of one or two repairs, first a Kempe move, then an optional ejecting move with the same parameters. The ejecting move is omitted when the Kempe move (successful or not) does not try to move anything back in the opposite direction, since the two repairs are identical then.

Kempe meet moves are useful because instances often contain preassigned class resources which are busy for all or most of the cycle. Moving a meet containing such a resource practically forces another meet to move the other way, so it makes sense to get on and do it. Kempe task moves are less useful because they apply to unpreassigned resources, such as teachers and rooms, which are less constrained. Ejecting moves seem more appropriate for them.

Table 5 investigates various combinations of Kempe, ejecting, and basic meet moves. The results here are clear: Kempe meet moves are helpful, and the Kempe/ejecting combination is better than the Kempe/basic alternative when run time is included in the comparison.

These repair operations are not original to this paper. The author has used node swaps and Kempe meet moves before [11], and ejecting task moves [10], but not ejecting meet moves.

7 Repairing defects

This section explains how the ejection chain algorithm repairs defects using the repair operations of Sect. 6. For each kind of defect, this section defines a set of repairs. The augment function for that kind of defect applies the first of these repairs, calls a KHE function to test for success and recurse, then either returns 'success' immediately or unapplies that repair and tries the next, and so on, returning 'no success' when all repairs are tried without success.

KHE has objects called *monitors*, each monitoring one point of application of one constraint, or one demand tixel in the global tixel matching. Each monitor contains a cost. When some part of the solution changes, KHE notifies the monitors affected by that part, and they revise their evaluation and perhaps change their cost. Any cost changes are reported and cause the overall solution cost to change. For example, when a time is assigned to a meet, any affected

Table 5 Kempe, ejecting, and basic moves during time assignment. Where the main text states that Kempe meet moves are tried, K means to try them and X means to omit them. Where it states that ejecting meet moves are tried, E means to try them and B means to try basic meet moves instead. KHE14 uses KE. Other details as previously.

Instance	C:KE	C:KB	C:XE	C:XB	T:KE	T:KB	T:XE	T:XB
AU-BG-98	12.00491	19.00390	**11.00490**	19.00473	26.5	39.5	18.8	41.9
AU-SA-96	16.00014	**4.00013**	25.00106	34.00060	73.9	417.6	52.3	112.2
AU-TE-99	**4.00124**	9.00133	8.00172	11.00153	3.5	10.1	4.0	3.3
BR-SA-00	**1.00057**	3.00075	1.00081	6.00066	0.8	0.6	1.3	0.3
BR-SM-00	29.00093	28.00093	32.00120	**26.00084**	6.4	1.5	4.8	1.2
BR-SN-00	**4.00243**	7.00258	10.00267	11.00231	2.8	1.8	4.5	0.8
DK-FG-12	0.02248	**0.01893**	0.02359	0.02565	192.9	370.0	122.5	88.1
DK-HG-12								
DK-VG-09	12.03349	**12.03077**	12.03424	14.03683	384.5	698.4	297.6	133.8
ES-SS-08	**0.01362**	0.02662	0.01771	0.04053	15.8	30.4	13.0	15.0
FI-MP-06	3.00132	**1.00123**	2.00107	6.00120	7.5	9.7	8.3	8.1
FI-PB-98	6.00039	**2.00173**	7.00022	6.00144	6.3	9.9	7.4	5.9
FI-WP-06	0.00078	1.00069	**0.00036**	3.00102	1.2	3.4	2.8	1.6
GR-H1-97	**0.00000**	**0.00000**	2.00000	2.00000	5.9	5.9	6.2	6.0
GR-P3-10	2.00088	**0.00074**	12.00150	1.00105	22.7	9.0	59.2	7.0
GR-PA-08	0.00040	**0.00035**	0.00043	2.00111	5.7	6.4	8.1	5.5
IT-I4-96	**0.00494**	1.00711	0.00605	0.00651	8.4	7.6	8.0	5.9
KS-PR-11	0.00150	0.00294	**0.00149**	2.01257	77.6	82.1	79.6	80.2
NL-KP-03	0.04774	0.04111	**0.03340**	0.04635	205.5	922.5	327.0	161.8
NL-KP-05								
NL-KP-09								
UK-SP-06	**0.00102**	2.00084	5.00324	12.00608	28.9	146.5	176.7	28.7
ZA-LW-09	**26.00000**	30.00000	29.00000	31.00000	18.0	13.7	18.2	15.1
ZA-WD-09	**9.00000**	11.00000	9.00001	17.00000	23.6	34.9	88.0	10.9
Average	**5.00660**	6.00679	7.00646	9.00909	53.3	134.4	62.3	34.9

assign time and prefer times monitors are notified, and they change their cost accordingly. Concretely, a defect is a monitor whose cost is non-zero.

Monitors may be *grouped*: joined into sets treated as single monitors whose cost is the sum of the individual costs. KHE14's grouped monitors have the same kind and monitor the same thing in reality (examples appear below), and are repaired by repairing any one member of the group. Monitors may also be *detached*: fixed to cost 0 regardless of their true cost. Grouping and detaching are used to prevent the algorithm from being confused by apparently distinct defects which really point to the same problem. Such defects could cause a chain to end when there is a worthwhile repair to continue with.

(One application of ejection chains to timetabling [4] found a way to exploit a coarser grouping than any used here: it groups all defects related to one resource. The elements of such a group may have different types, so it does not make sense to perform repairs specific to any one type. Instead, all moves of a meet to which the resource is assigned to a time when the resource is free are tried. From those moves which introduce at most one new conflict, the 20 best are selected and used as the set of repairs for the group.)

As mentioned earlier, there are two categories of calls on the ejection chain repair function: *time repair* calls, which repair the assignments of times to meets, and *resource repair* calls, which repair the assignments of resources to tasks. Each category has its own augment functions, and its own way of grouping and detaching monitors. Both categories are defined below.

Demand defects. These are cases of unmatched demand tixels in the global tixel matching (Sect. 4), usually indicating that the demand for some set of resources at some time exceeds their supply then. The defect is not really the one unmatched tixel, but rather the whole set of demand tixels that contribute to the excess demand. Given the nondeterminism of the global tixel matching, any one of these could be reported as the defect. Repair operations use KHE functions to visit them all, and, in effect, repair the set, not the individual.

During time repair, demand monitors lying within tasks of the same meet are grouped, so that multiple demand defects that can be repaired by moving that meet are perceived as a single defect. Defects are handled by trying all repairs that move any of the meets contributing to the excess demand away from the problem time. Kempe/ejecting meet moves are tried first, then node swaps between nodes with similar preassigned resources. For example, 6 Science meets running simultaneously when there are only 5 Science laboratories will generate one demand defect which will cause repairs to be tried that move any of the 6 meets away from the problem time. Simple clashes also produce demand defects, and are repaired in the same way.

Demand monitors derived from the same avoid unavailable times, limit busy times, or workload limit monitor are grouped. If any of these are involved in a demand defect (if they contribute to the excess demand), then the repairs just given are not well targeted, because they could move a contributing meet to a different time within the times whose limit has been exceeded, or swap a meet back into those times. So different repairs are tried in that case: ejecting meet moves that move any of the contributing meets away from the times whose limit has been exceeded. For example, if teacher Smith is preassigned to tasks which give him 8 busy times on Monday when he is limited to 7, the demand monitor derived from this limit will contribute to the excess demand, causing ejecting moves to be tried that move meets preassigned to Smith away from Monday. Similarly, a demand monitor derived from an avoid unavailable times monitor will cause ejecting moves away from the unavailable times.

During resource repair, demand defects are handled differently. They do not contribute to the cost of the solution, and they are not repaired. Instead, ejection chains which increase their number are not applied (except right at the end, when a last-ditch attempt is made to assign any remaining unassigned tasks). The reason for this is as follows.

At the start of each call to the resource repair ejection chain algorithm, the number of demand defects is equal to what it was at the end of time assignment. This is because no resource assignments are accepted which increase this number, and resource assignments which decrease it are impossible, since a resource assignment reduces the domains of demand nodes, reducing the choice of matchings. (If resource repairs changed meet assignments, that could reduce the number of demand defects; but it is not likely to, because similar changes were tried during time repair, at a point when fewer resources were assigned so more choices were open.) Repair of demand defects is not attempted, then, because it is doomed to failure. This also explains why chains which increase the number of demand defects are not applied: such an increase would be almost impossible to reverse later. This argument is from [10].

Split events defects and distribute split events defects. These are cases of events split into too few or too many meets, or into meets whose durations are not wanted. Event splitting is handled by the structural phase (Sect. 3), and these defects are ignored during time and resource repair.

Assign time defects. These are cases of meets not assigned a time. There are usually none when time repair begins, because the initial time assignment usually assigns a time to every meet; but ejecting meet moves create them. They are handled during time repair only, by trying all ejecting meet assignments to times in the meet's domain. Kempe meet moves are not possible (there is no original time). Assign time monitors whose events are joined by link events constraints handled structurally are grouped; they monitor the same thing.

It is important to assign a time to every meet, so assign time defects are also treated during time repair like demand defects are treated during resource repair: ejection chains may unassign meets temporarily as they go, but no chain is applied which, in the end, increases the total cost of assign time defects.

Prefer times defects. These are cases of meets assigned unwanted times: for example, a Physics meet that prefers a morning time but is assigned an afternoon time. They are handled during time repair only, by trying all Kempe/ejecting meet moves of the meet to a preferred time. Prefer times monitors whose events are joined by link events constraints handled structurally are grouped when they request the same times.

Spread events defects. These are cases where the meets of a course are spread unevenly through the cycle. For example, two might occur on Monday, and none on Thursday. They are handled during time repair only, by trying all Kempe/ejecting meet moves which remove the defect (in the example, all moves of a Monday meet to a Thursday time). Spread events monitors whose events are joined by link events constraints handled structurally are grouped.

Link events defects. These are cases where events which should occur at the same time do not. Like event splitting, event linking is handled structurally (Sect. 3), and these defects are ignored during time and resource repair.

Order events defects. These are cases where events should appear in a particular time order, but they don't. KHE14 ignores these defects, because there are no order events constraints in the XHSTT-2013 data set. However, it is easy to find repairs that remove them, so there will be no problem in handling them in future. Order events monitors whose events are joined by link events constraints handled structurally are grouped.

Assign resource defects. These are cases where a task is not assigned a resource. They are handled during resource repair, by trying all ejecting task assignments to resources in the task's domain. Assign resource monitors whose tasks are joined by avoid split assignments constraints handled structurally are grouped while those structures are present.

Prefer resources defects. These are cases where a task is assigned a resource it does not prefer: for example, the room task of a Science meet assigned an ordinary classroom instead of a Science laboratory. They are handled during resource repair, by trying all ejecting task moves to the preferred resources. Kempe task moves are possible, but (as discussed above) they seem unlikely to be useful and have not been tried. Prefer resources monitors whose tasks are joined by avoid split assignments constraints handled structurally are grouped while those structures remain in place, if they request the same resources.

Avoid split assignments defects. These are cases where tasks are assigned different resources, when they want the same resource. For example, a Music event split into five meets, four taught by Smith and one taught by Brown, is an avoid split assignments defect, also called a *split assignment.*

Structures which prohibit split assignments are present for most of KHE14, but near the end they are removed and split assignments are constructed (they are usually better than nothing). These split assignments are repaired by taking each pair of resources assigned to the tasks, finding the subset of the tasks assigned those resources, and trying each ejecting task-set move of that subset to a resource from the domain of one of them. These are the smallest repairs capable of reducing cost, which is prudent, given the difficulty.

A promising alternative kind of repair, not yet implemented, is to pick one of the resources currently assigned to some of the tasks, whose workload limit permits it to be assigned to all of them, and try to move the meets of the other tasks to times when that resource is free. Previous phases would need to ensure that split assignments were concentrated in meets that demand few resources, making them more likely to be movable.

Avoid clashes defects. These are cases where a resource attends two meets at the same time. During time repair, avoid clashes monitors are detached: demand monitors do their job and more. During resource repair, they are handled by trying all ejecting task moves of the clashing tasks to resources in their domains. There is no confusion with demand defects, because of the way that demand monitors are handled during resource repair (see above).

Avoid clashes monitors for resources of the same type are grouped when they are derived from the same constraint, all the event resources of their type are preassigned, and the resources are preassigned to the same events, so follow the same timetable. The saving can be significant: in the NL-KP-03 instance, for example, there are 453 resources representing individual students, but only 297 groups, or 285 when link events constraints are taken into account.

Avoid unavailable times defects. These are cases where a resource attends a meet at a time when it is unavailable. An avoid unavailable times constraint is the same as a limit busy times constraint whose set of times is the unavailable times, and whose `Maximum` attribute is 0. So these defects are handled as described below for limit busy times defects, including grouping.

Limit idle times defects. These are cases where a resource's timetable contains an idle time: a time when the resource is not *busy* (attending an event), but is busy earlier that day and later. During time repair, they are handled by taking each meet assigned to the resource which occurs at the beginning or end of one of its days, and trying each ejecting move of it to a time that reduces the number of idle times and does not cause clashes. A move to any non-clashing time between the first and last busy times on any day is acceptable. If the meet is adjacent to an idle time, then moving it far away will remove that idle time, so it may also be moved to just before the first busy time of any day, and just after the last busy time of any day, provided that first or last busy time is not one when the meet itself is currently running. Limit idle times monitors are grouped in the same way as avoid clashes monitors. During resource repair, limit idle times defects are ignored; repairing them then is future work.

Cluster busy times defects. These are cases where a resource is busy on too few or too many days. During time repair, if the problem is too few days, then all ejecting moves are tried which move a meet from a day in which it is not the only meet to a day in which it is. If the problem is too many days, then ejection chains struggle, because cost is reduced only when a day becomes completely free, which may require several meet moves. The present repair tries all ejecting moves of meets which are alone in their day to days when other meets are present. Some defects can be repaired in this way, but many cannot; they are future work, as are repairs during resource repair. Cluster busy times monitors are grouped in the same way as avoid clashes monitors.

Limit busy times defects. These are cases where a resource is overloaded or underloaded during some set of times, typically one day. For example, teacher Jones might expect to be busy for only at most 7 of the 8 times on any day; anything more is a limit busy times defect.

Break each limit busy times monitor into two monitors, one monitoring underloads and the other overloads. During time repair, overload monitors that give rise to demand monitors that do all their work are detached. Other overload monitors and all underload monitors are not. They are handled by trying all ejecting meet moves of one of the resource's meets from inside the set of times to outside it, or vice versa, depending on whether the defect is an overload or an underload.

During resource repair, all these monitors are attached. There is no confusion with demand monitors, as explained above for avoid clashes defects. Overloads are handled by trying all ejecting task moves which unassign the resource from any meet running during the set of times. Underloads are ignored; repairing them during resource repair is future work. Limit busy times monitors are grouped in the same way as avoid clashes monitors.

Limit workload defects. These are similar to limit busy times defects whose times are the whole cycle, and are handled in the same way, including grouping.

Repairs targeted at specific defects are rare in the timetabling literature. About half of the above is previous work [10, 11] and half is new. Disentangling new from old would be tedious, but this paper is the first to tackle anything approaching a complete set of defect types with polymorphic ejection chains.

Which augment functions are most effective? Finding a good measure of effectiveness is not easy. For example, virtually any defect can be removed if enough mayhem is visited on its surroundings, so success in removing defects, taken in isolation, is a poor measure. One simple approach, not claimed to be

Table 6 Effectiveness of time augment functions. For each time augment function, the number of calls to the function, the number of successful calls, and the ratio of the two as a percentage, over all tested instances of XHSTT-2013. Only non-zero rows are shown.

Augment function	Total	Successful	Percent
Assign time	3770681	12890	0.3
Spread events	440720	3723	0.8
Ordinary demand	177379	797	0.4
Workload demand	8439	63	0.7
Avoid unavailable	48428	367	0.8
Limit idle	847642	3447	0.4
Cluster busy	253699	333	0.1
Limit busy	652935	737	0.1

Table 7 Effectiveness of resource augment functions. For each resource augment function, the number of calls to the function, the number of successful calls, and the ratio of the two as a percentage, over all tested instances of XHSTT-2013. Only non-zero rows are shown.

Augment function	Total	Successful	Percent
Assign resource	500023	2032	0.4
Avoid splits	6946	182	2.6
Avoid clashes	94	0	0.0
Limit busy	24523	133	0.5
Limit workload	170855	1343	0.8

perfect, is to say that a call on an augment function is effective when it returns `true`, meaning that it lies on a chain that improved the solution, and ineffective when it returns `false`. The ratio of effective to effective plus ineffective calls, expressed as a percentage, measures the effectiveness of the function.

Table 6 presents the number of calls on each time repair augment function used by KHE14 when solving XHSTT-2013, and their effectiveness, measured as just described. Table 7 does the same for resource repair. Interpretation is problematical, but it seems, for example, that the time repair functions for cluster busy times and limit busy times defects are relatively ineffective.

Which repairs are most effective? Again, finding a good measure is not easy. For example, on any given defect one kind must be tried first, and this gives it more opportunities to both succeed and fail than the others. Again, a simple approach is used: the successful calls on a given augment function are attributed to the repairs that caused the successes.

Table 8 is like Table 6 except that it contains one row for each kind of repair of each kind of time defect, measured in this way. Some of the results are quite interesting: the tiny number of successful node swaps, for example. The corresponding results for resource repairs are omitted, since each resource repair augment function tries just one kind of repair operation.

8 The algorithm

This section describes the KHE14 algorithm at a high level. An implementation is available online (function `KheGeneralSolve2014` of [12]).

Table 8 Effectiveness of time repair operations. For each time augment function and repair operation, the number of calls on that repair operation made by that augment function, the number of successful calls, and the ratio of the two as a percentage, over all tested instances of XHSTT-2013. Only non-zero rows are shown.

Augment function : Repair operation	Total	Successful	Percent
Assign time : Basic meet assignment	3	3	100.0
Assign time : Ejecting meet assignment	3770675	12884	0.3
Assign time : Basic meet move	3	3	100.0
Spread events : Basic meet move	1	0	0.0
Spread events : Ejecting meet move	254101	249	0.1
Spread events : Kempe meet move	186618	3474	1.9
Ordinary demand : Basic meet move	27	2	7.4
Ordinary demand : Ejecting meet move	121656	64	0.1
Ordinary demand : Kempe meet move	54942	726	1.3
Ordinary demand : Node swap	754	5	0.7
Workload demand : Ejecting meet move	8439	63	0.7
Avoid unavailable : Ejecting meet move	48428	367	0.8
Limit idle : Ejecting meet move	847642	3447	0.4
Cluster busy : Ejecting meet move	253699	333	0.1
Limit busy : Ejecting meet move	652935	737	0.1

KHE14 proceeds in *phases* (major steps). First comes the *structural phase.* It constructs an initial solution with no time or resource assignments, converts resource preassignments (in the instance) into resource assignments (in the solution), adds additional structure as described in Sect. 3, and adds the global tixel matching as described in Sect. 4.

Next comes the *time assignment* phase, which assigns a time to each meet. It has been described fully elsewhere [6, 7, 11]; here is an overview. For each resource to which a hard avoid clashes constraint applies it builds one *layer*, the set of all nodes containing meets preassigned that resource. After discarding layers that are redundant because they are subsets of other layers, and sorting so that (heuristically) the most difficult layers come first, it assigns times to the meets of each layer in turn. The algorithm for assigning times to the meets of one layer is heuristic and complex. It tries for regularity with previously assigned layers, and exploits the fact that the meets of one layer should not overlap in time, by maintaining a minimum-cost matching of meets to times.

A node may lie in several layers, if its meets contain several preassigned resources. Such a node is handled with the first layer it lies in, and the result is not changed when assigning subsequent layers. So when a layer's turn comes to be assigned, all its nodes may be already assigned. Such layers are skipped.

After each unskipped layer is assigned, an ejection chain repair (Sect. 5) is applied. Its main loop is targeted at the defects of the layer just assigned, but its recursive calls may spread into earlier layers. After all layers have been assigned and repaired, another ejection chain repair is carried out, targeted at the entire time assignment. Then the structures which enforce regularity in time are removed and yet another ejection chain time repair is run.

Next come the *resource assignment* phases, one for each type of resource (teacher, room, etc.). These phases are sorted heuristically so that the most

Table 9 Effectiveness of KHE14 and KHE14x8. Details as previously. Different solutions to one instance vary in run time, so finding eight solutions on a quad-core machine often takes more than twice as long as finding one.

Instance	C:KHE14	C:KHE14x8	T:KHE14	T:KHE14x8
AU-BG-98	12.00491	**4.00524**	24.9	43.8
AU-SA-96	16.00014	**6.00006**	72.2	172.3
AU-TE-99	4.00124	**2.00140**	3.6	7.6
BR-SA-00	1.00057	**1.00051**	0.8	1.9
BR-SM-00	29.00093	**22.00129**	6.4	15.2
BR-SN-00	**4.00243**	**4.00243**	2.9	6.7
DK-FG-12	0.02248	**0.02046**	175.5	404.0
DK-HG-12				
DK-VG-09	12.03349	**12.03257**	373.1	919.4
ES-SS-08	0.01362	**0.01287**	14.8	31.1
FI-MP-06	3.00132	**0.00125**	7.7	16.4
FI-PB-98	6.00039	**1.00024**	6.1	12.9
FI-WP-06	0.00078	**0.00041**	1.2	5.4
GR-H1-97	**0.00000**	**0.00000**	6.2	13.2
GR-P3-10	2.00088	**0.00006**	22.8	51.8
GR-PA-08	0.00040	**0.00021**	7.2	19.5
IT-I4-96	0.00494	**0.00197**	7.9	20.1
KS-PR-11	0.00150	**0.00116**	76.4	173.3
NL-KP-03	0.04774	**0.03919**	190.4	698.7
NL-KP-05				
NL-KP-09				
UK-SP-06	0.00102	**0.00056**	30.3	88.2
ZA-LW-09	26.00000	**16.00000**	21.7	34.5
ZA-WD-09	9.00000	**6.00000**	25.0	50.1
Average	5.00660	**3.00580**	51.3	132.7

difficult come first. In practice, teachers are assigned first (if needed), then rooms; students and classes are not assigned, since they are all preassigned, and so were assigned during the structural phase.

Each resource assignment phase has three parts. In the first part, which in practice assigns most tasks, violations of avoid split assignments and prefer resources constraints are prohibited structurally, and assignments that increase the number of demand defects (Sect. 4) are rejected. A resource assignment algorithm is called that tries to assign a resource to each unpreassigned task of the current type. If there are avoid split assignments constraints, a *resource packing* algorithm which follows a bin packing paradigm is used. Otherwise a constructive heuristic is used, more effectively than usual because of the guidance provided by the global tixel matching. These algorithms are described in detail in [10], where resource packing was found to be the best of three plausible resource assignment algorithms for teacher assignment. This first part ends with a call on the ejection chain resource repair algorithm, targeted at the event resource and resource defects of the current type.

The second part of the phase is only carried out for those types of resources whose event resources are subject to avoid split assignments constraints. It removes structures that prevent split assignments, finds split assignments for unassigned tasks (using a construction heuristic specialized for them), and

Table 10 Event defects in the solutions produced by KHE14x8. Each column shows the number of defects of one kind of event constraint. A dash indicates that the instance contains no constraints of that type. The columns appear in the same order as the rows of Table 1.

Instance	SS	DS	AT	PT	SE	LE	OE
AU-BG-98	0	0	0	0	30	0	-
AU-SA-96	0	0	0	0	6	0	-
AU-TE-99	0	0	0	-	8	0	-
BR-SA-00	0	0	0	0	0	-	-
BR-SM-00	0	0	4	0	2	-	-
BR-SN-00	0	0	0	0	0	-	-
DK-FG-12	-	-	0	-	41	0	-
DK-HG-12							
DK-VG-09	-	-	0	-	81	0	-
ES-SS-08	0	-	0	-	88	-	-
FI-MP-06	0	-	0	0	0	-	-
FI-PB-98	0	-	0	0	1	-	-
FI-WP-06	0	-	0	-	0	-	-
GR-H1-97	-	-	0	-	0	0	-
GR-P3-10	0	-	0	0	0	0	-
GR-PA-08	-	-	0	-	5	0	-
IT-I4-96	0	-	0	0	0	-	-
KS-PR-11	0	0	0	0	0	-	-
NL-KP-03	0	-	0	0	0	0	-
NL-KP-05							
NL-KP-09							
UK-SP-06	-	-	0	-	0	0	-
ZA-LW-09	0	-	3	0	-	0	-
ZA-WD-09	-	-	1	0	0	0	-
Total	0	0	8	0	262	0	

calls ejection chain repair again. Then it tries two VLSN search algorithms [1,13] which sometimes find small improvements. One rearranges the resource assignments within a given set of times using min-cost flow, the other unassigns and optimally reassigns pairs of resources [9]. The details are in [12] as usual; they are omitted here because these algorithms are peripheral to the main thrust of this paper, which is already overlong.

The third part is a last-ditch attempt to assign as many of the remaining unassigned tasks as possible. It removes all structural prohibitions, removes the prohibition on increasing the number of unmatched demand tixels, and calls the ejection chain repair algorithm a third time.

The final *cleanup phase* carries out some minor tidying up. Whenever two meets derived from the same event have ended up adjacent in time, this phase merges them into one when that is possible and reduces cost. It also unassigns tasks and meets when that reduces cost.

Table 9 shows the overall performance of KHE14 and its variant KHE14x8, which runs KHE14 8 times in parallel and keeps a best solution. Random numbers are not used; instead, each run is given a different *diversifier* (a small fixed integer). It is used in several places, to vary the starting point of list traversals, and to break ties. For example, ejection chain algorithms sort their initial defects by decreasing cost; the diversifier influences the order of defects

Table 11 Event resource and resource defects produced by KHE14x8. Details as previously.

Instance	AR	PR	AS	AC	AU	LI	CB	LB	LW
AU-BG-98	2	0	42	2	0	-	-	18	0
AU-SA-96	1	0	0	3	0	-	-	0	0
AU-TE-99	0	0	12	2	0	-	-	1	0
BR-SA-00	-	-	-	1	0	8	2	-	-
BR-SM-00	-	-	-	11	1	8	11	-	-
BR-SN-00	-	-	-	4	0	16	16	-	-
DK-FG-12	0	0	-	0	0	56	113	64	-
DK-HG-12									
DK-VG-09	0	0	-	2	-	52	47	32	-
ES-SS-08	0	0	-	0	5	-	0	0	-
FI-MP-06	-	-	-	0	10	20	-	6	-
FI-PB-98	-	-	-	0	0	12	-	0	-
FI-WP-06	-	-	-	0	-	13	-	8	-
GR-H1-97	-	-	-	0	0	-	-	-	-
GR-P3-10	-	-	-	0	0	0	-	3	-
GR-PA-08	-	-	-	0	0	7	-	0	-
IT-I4-96	-	-	-	0	6	31	1	2	-
KS-PR-11	-	-	-	0	0	54	-	0	-
NL-KP-03	0	0	-	0	7	334	15	51	-
NL-KP-05									
NL-KP-09									
UK-SP-06	0	-	-	0	0	25	-	-	-
ZA-LW-09	-	-	-	1	-	-	-	-	-
ZA-WD-09	-	-	-	3	0	-	-	-	-
Total	3	0	54	29	29	636	205	185	0

of equal cost. These solutions are available from the KHE web page [12]. An analysis of the remaining defects appears in Tables 10 and 11.

9 Conclusion

No overall conclusion can be drawn yet: KHE14 is the author's first solver to address such a wide range of constraint types; it is work in progress and has not reached its full potential; and solutions to XHSTT-2013 made by other solvers were not available at the time of writing. Individual solutions are available, but comparing with them ignores an essential requirement of a good solver, namely robustness over many instances, so is deliberately not done here.

The author is currently exploring ideas for improvements in all phases of KHE14. Recently, he modified the structural phase to restrict meet domains to discourage cluster busy times defects. This produced a solution to instance NL-KP-03 with 9 cluster busy times defects and cost 0.02804, much better than the 15 and 0.04774 reported in the tables. The algorithm that assigns times to the meets of one layer was designed without regard to cluster busy times and limit idle times constraints, so it needs revision. That should make solving the Dutch instances run faster, since at present they are overwhelming the time repair algorithm with hundreds of limit idle times defects.

References

1. R. Ahuja, Ö. Ergun, James B. Orlin, and A. Punnen, A survey of very large-scale neighbourhood search techniques, Discrete Applied Mathematics, 123, 75–102 (2002)
2. Kathryn A. Dowsland, Nurse scheduling with tabu search and strategic oscillation, European Journal of Operational Research, 106, 393–407 (1998)
3. Fred Glover, Ejection chains, reference structures and alternating path methods for traveling salesman problems, Discrete Applied Mathematics, 65, 223–253 (1996)
4. Peter de Haan, Ronald Landman, Gerhard Post, and Henri Ruizenaar, A case study for timetabling in a Dutch secondary school, Practice and Theory of Automated Timetabling VI (Springer Lecture Notes in Computer Science 3867), 267–279, (2007)
5. Myoung-Jae Kim and Tae-Choong Chung, Development of automatic course timetabler for university, Proceedings of the 2nd International Conference on the Practice and Theory of Automated Timetabling (PATAT'97), 182–186 (1997)
6. Jeffrey H. Kingston, A tiling algorithm for high school timetabling, Practice and Theory of Automated Timetabling V (Springer Lecture Notes in Computer Science 3616), 208–225 (2005)
7. Jeffrey H. Kingston, Hierarchical timetable construction, Practice and Theory of Automated Timetabling VI (Springer Lecture Notes in Computer Science 3867), 294–307 (2007)
8. Jeffrey H. Kingston, The HSEval High School Timetable Evaluator, URL `http://www.it.usyd.edu.au/~jeff/hseval.cgi` (2010)
9. Jeffrey H. Kingston, Timetable construction: the algorithms and complexity perspective, Annals of Operations Research, DOI 10.1007/s10479-012-1160-z (2012)
10. Jeffrey H. Kingston, Resource assignment in high school timetabling, Annals of Operations Research, 194, 241–254 (2012)
11. Jeffrey H. Kingston, Repairing high school timetables with polymorphic ejection chains, Annals of Operations Research, DOI 10.1007/s10479-013-1504-3
12. Jeffrey H. Kingston, KHE web site, `http://www.it.usyd.edu.au/~jeff/khe` (2014)
13. Carol Meyers and James B. Orlin, Very large-scale neighbourhood search techniques in timetabling problems Practice and Theory of Automated Timetabling VI (Springer Lecture Notes in Computer Science 3867), 24–39 (2007)
14. Christos. H. Papadimitriou and Kenneth Steiglitz, Combinatorial Optimization: Algorithms and Complexity, Prentice-Hall (1982)
15. Nelishia Pillay, An overview of school timetabling research, PATAT10 (Eighth international conference on the Practice and Theory of Automated Timetabling, Belfast, August 2010), 321–335 (2010)
16. Samad Ahmadi, Sophia Daskalaki, Jeffrey H. Kingston, Jari Kyngäs, Cimmo Nurmi, Gerhard Post, David Ranson, and Henri Ruizenaar, An XML format for benchmarks in high school timetabling, Annals of Operations Research, 194, 385–397 (2012)
17. Gerhard Post, XHSTT web site, `http://www.utwente.nl/ctit/hstt/` (2011)
18. Gerhard Post, Luca Di Gaspero, Jeffrey H. Kingston, Barry McCollum, and Andrea Schaerf, The Third International Timetabling Competition, PATAT 2012 (Ninth international conference on the Practice and Theory of Automated Timetabling, Son, Norway, August 2012), 479–484 (2012)
19. G. Schmidt and T. Ströhlein, Timetable construction—an annotated bibliography, The Computer Journal, 23, 307–316, (1980)