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

KHE14: An Algorithm for High School Timetabling

Jeffrey H. Kingston

Received: date / Accepted: date

Abstract This paper presents an algorithm called KHE14 for solving the high school timetabling problem. KHE14 builds its timetables one student form at a time, and repairs them using ejection chains. Many of its components have been published previously, and so are described here only briefly. A few of its components, notably most of the augment functions called by the ejection chain algorithm, are new, so are described in detail. Experiments using the XHSTT-2014 data set, conducted in August 2014, are included.

Keywords High school timetabling · Ejection chains

1 Introduction

High school timetabling is one of the three major timetabling problems found in academic institutions, the others being university course timetabling and examination timetabling. Automated methods for solving it have been studied from the early days of computers [20] to the present day [16].

An XML format called XHSTT was introduced recently to represent real instances and solutions of the high school problem [8, 18]. XHSTT was used in the Third International Timetabling Competition [19], the first competition to include high school timetabling. An XHSTT data set called XHSTT-2014 is currently being promoted as a benchmarking standard [17]. It contains 25 instances taken from real high schools in 12 countries around the world.

This paper presents KHE14, an algorithm for high school timetabling, with experiments using XHSTT-2014 conducted in August 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.

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

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

So although this paper describes KHE14 completely, it does so only at a high 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 types 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-2014 data set [17], as downloaded on 18 August 2014, and were performed on the author's desktop machine, an Intel i5 quad-core running Linux. Each individual solution is produced on a single processor; tests that produce multiple solutions utilize all four processors.

2 Problem specification

The XHSTT specification of the high school timetabling problem is used here. An XHSTT instance contains four parts: the *cycle*, which is the chronological sequence of times that may be assigned to events; a set of *resources*, which are entities that attend events (usually either teachers, rooms, students, or classes, where a class is a group of students who mostly attend the same events); a set of *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 a set of *constraints*, which specify conditions that solutions should satisfy, and the penalty costs to impose when they don't.

Table 2 The number of times, teachers, rooms, classes (groups of students), individual students, and events in the instances of XHSTT-2014. There are 25 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	69		279	1077
DK-HG-12	50	100	71		523	1235
DK-VG-09	60	46	53		163	918
UK-SP-06	25	68	67	67		1227
FI-PB-98	40	46	34	31		387
FI-WP-06	35	18	13	10		172
FI-MP-06	35	25	25	14		280
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		1148
ZA-LW-09	148	19	2	16		185
ZA-WD-09	42	40		30		278
ES-SS-08	35	66	4	21		225
US-WS-09	100	134	108			628

XHSTT currently offers 16 constraint types (Table 1), specifying preferred times for events, unavailable times for resources, and so on. 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. (Sect. 3 has more on this.) And it may be required to assign resources to unpreassigned event resources: often rooms, occasionally teachers, never (in practice) classes or students. 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-2014 data set. They vary greatly in difficulty, in ways that such a table cannot fully capture. The five instances with resources representing individual students seem particularly challenging, because there are hundreds of these resources, each with its own timetable and constraints.

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 [10].

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: preassigned resources with avoid clashes constraints (constraining events 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 and 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 their effects in detail. It would take too long to repeat all that here. Instead, what follows is a description of 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 Science 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 Science example, a single event of duration 6 would be given, along with constraints which place limits on the number and duration of the meets it is to be split into. This handles a situation often 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 this set may be assigned to a meet, either directly, or indirectly via an assignment to another 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 prefer resources constraints are usually present 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 this set 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, at least to begin with. 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 violations of hard constraints, while permitted, should be few in good solutions, but 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 a heuristic

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	**12.00764**	13.00520	24.1	14.3
AU-SA-96	**2.00011**	4.00022	39.3	91.4
AU-TE-99	**2.00134**	5.00152	1.6	2.5
BR-SA-00	**0.00042**	0.00044	0.9	0.7
BR-SM-00	**8.00131**	12.00128	2.9	2.4
BR-SN-00	**0.00134**	0.00145	2.9	2.5
DK-FG-12	**0.03336**	0.03391	368.0	350.6
DK-HG-12	**13.05467**	14.05094	734.0	805.1
DK-VG-09	**2.04393**	3.04433	927.2	948.2
UK-SP-06	**29.00926**	33.01108	374.0	377.3
FI-PB-98	**0.00025**	3.00031	6.9	9.5
FI-WP-06	**0.00024**	**0.00024**	3.5	8.4
FI-MP-06	**0.00123**	0.00147	3.5	4.6
GR-H1-97	**0.00000**	**0.00000**	0.6	5.9
GR-P3-10	**0.00011**	**0.00011**	2.7	7.8
GR-PA-08	**0.00012**	0.00016	10.4	13.6
IT-I4-96	0.00145	**0.00054**	7.7	14.9
KS-PR-11	0.00025	**0.00020**	195.1	194.6
NL-KP-03	**0.01229**	0.01487	440.0	416.9
NL-KP-05	21.07252	**15.07401**	396.5	373.6
NL-KP-09	16.09330	**16.07930**	95.8	85.3
ZA-LW-09	**19.00022**	20.00018	4.5	9.7
ZA-WD-09	**13.00000**	26.00000	4.3	13.6
ES-SS-08	0.01142	**0.01117**	26.4	27.1
US-WS-09	**0.00651**	0.00758	30.5	48.7
Average	**5.01413**	6.01362	148.1	153.2

strategy: it includes them at first, but removes them towards the end, so that later repair operations are not limited by them. The original constraints are not forgotten: even when violations are allowed, they are still penalized.

This structural phase is similar to previous structural phases described by the author [6,7]. It 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 presence of several kinds of regularity and several ways to encourage it [10], not all of which can be disabled at present. Table 3 examines regularity between forms. For example, if the classes of the Year 11 form attend English 6 times per week in meets of durations 2, 1, 1, 1, and 1, and the classes of the Year 12 form attend Science 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 author is not ready to abandon regularity between forms, but the evidence of Table 3 is tending against it, at least as currently implemented (Sect. 8).

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) [15].

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 one supply tixel: the one 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. 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 [9], which has much more detail: how to define the additional tixels, how to implement the matching efficiently, and so on.

5 Polymorphic ejection chains

Like most timetabling solvers, KHE14 first constructs, then repairs. The repair work is mostly done by *ejection chains*. An ejection chain is a sequence of one or more *repair operations* (also called *repairs*), which are small changes to the solution. 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. Importantly, 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 [9], and time repair [4, 5, 10].

A key insight of [10] 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 incorporates a set of functions, one for each defect type, called *augment functions* after the function for finding augmenting paths in bipartite matching [15]. An augment function is passed a defect of the type it handles. 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 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 removed defect), it calls the appropriate augment function for that defect. In this way a chain of coordinated repairs is built up. If that call does not succeed, or was not tried because two or more significant new defects appeared, the function undoes the repair and continues with alternative repairs.

Alternatively, if two or more new defects appear, the algorithm could try to remove them all by finding a whole set of ejection chains, one for each new defect. This *ejection tree* approach seldom succeeds, so the author only uses it in a few cases, described in Sect. 7, where there seems to be nothing better.

The algorithm's main loop repeatedly iterates over the solution's defects, or over a subset of them that it is expedient to target, calling the appropriate augment function on each. It terminates when one pass over all these defects yields no reduction in solution cost. A *main loop defect* is a defect iterated over by the main loop; a *main loop repair* is a repair of a main loop defect.

After each pass over the main loop defects, the wall clock time since the solution was created is compared with a *soft time limit*. If the soft time limit has been reached, repair is terminated as though no improvement was found on

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	**4.00752**	13.00520	13.00431	19.7	14.4	72.0
AU-SA-96	5.00009	4.00022	**2.00021**	47.2	87.8	111.5
AU-TE-99	6.00151	5.00152	**5.00105**	2.9	2.5	3.0
BR-SA-00	0.00045	**0.00044**	0.00049	0.5	0.7	1.0
BR-SM-00	11.00102	12.00128	**10.00097**	2.6	2.5	3.7
BR-SN-00	**0.00128**	0.00145	0.00135	2.1	2.4	2.7
DK-FG-12	0.06727	0.03391	**0.02546**	365.3	350.2	433.7
DK-HG-12	14.09212	14.05094	**12.03956**	722.0	810.3	1296.3
DK-VG-09	7.07526	3.04433	**2.03506**	685.9	951.4	1716.7
UK-SP-06	35.01096	33.01108	**29.01062**	313.1	376.3	489.8
FI-PB-98	**0.00018**	3.00031	1.00038	7.0	9.4	13.5
FI-WP-06	**0.00021**	0.00024	0.00025	8.2	8.4	11.3
FI-MP-06	**0.00101**	0.00147	0.00120	5.5	4.6	6.1
GR-H1-97	**0.00000**	**0.00000**	**0.00000**	6.0	5.9	6.0
GR-P3-10	2.00019	**0.00011**	2.00019	9.6	7.8	7.3
GR-PA-08	**0.00014**	0.00016	0.00015	11.8	13.6	17.6
IT-I4-96	**0.00048**	0.00054	1.00054	14.6	14.9	13.9
KS-PR-11	0.00021	**0.00020**	0.00034	266.7	201.5	142.2
NL-KP-03	0.01818	0.01487	**0.01469**	362.7	417.2	480.7
NL-KP-05	38.09413	**15.07401**	18.04507	385.1	373.3	412.9
NL-KP-09	12.14925	16.07930	**9.10155**	136.2	86.6	85.1
ZA-LW-09	20.00020	**20.00018**	22.00022	10.2	9.7	9.9
ZA-WD-09	27.00000	26.00000	**24.00000**	14.0	13.5	16.1
ES-SS-08	**0.00582**	0.01117	0.01106	28.5	27.5	23.0
US-WS-09	**0.00675**	0.00758	0.00748	58.4	49.1	70.6
Average	7.02136	6.01362	**6.01208**	139.4	153.7	217.9

the pass just ended. Since this allows each run of the ejection chain algorithm after the soft time limit one pass over its defects, and places no limit on other code, it does not enforce a hard time limit; but, since ejection chains consume most of KHE14's running time, it does cap running time in practice.

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 two methods. KHE14's choice trades off cost and running time quite well, but there is no simple signal. KHE14 also limits the number of calls on augment functions per search to 120, because tests not reported here in detail show that successes after that are very rare.

Another way to vary the scope of the search is to reopen the whole solution for visiting, not just before each main loop defect, but before each main loop repair. KHE14 does this. Yet another way, worth trying in practice only for teacher assignment, is to allow repairs of resource assignments to alter time assignments. KHE14 as presented here does not do this, since it can increase run time by a factor of 4 or more; but it is available as an option.

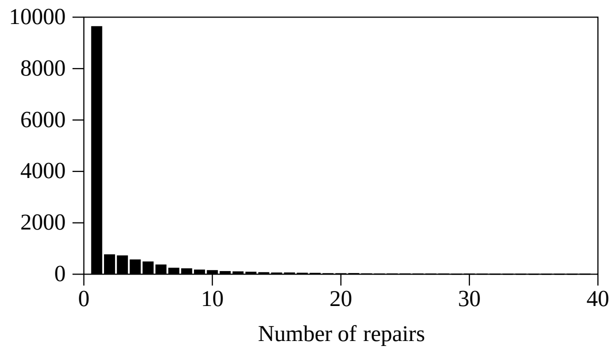

Fig. 1 For each number of repairs, the number of improvements (successful chains or trees) with that number of repairs found during time repair, over all instances of archive XHSTT-2014. There were 13880 improvements altogether, and their average number of repairs was 2.7. All improvements with more than 39 repairs are shown as having 39 repairs. The longest improvement had 49 repairs.

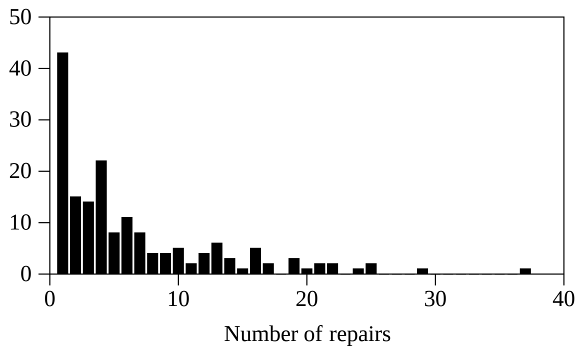

Fig. 2 For each number of repairs, the number of improvements (successful chains or trees) with that number of repairs found during resource repair, over all instances of archive XHSTT-2014. There were 170 improvements altogether, and their average number of repairs was 6.5.

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 quite long. It was shown in [10] that chain lengths tend to increase as the algorithm progresses.

The text of this section is adapted from [10], which also describes the extensive support for ejection chains provided by KHE. The user writes one augment function for each defect type, which iterates over the alternative repairs, applying each in turn. KHE supplies the main loop, chaining together of individual repairs, testing for success, unapplying, and dynamic dispatch by defect type. It also offers many options for varying the behaviour. For example, for each repair independently it allows the caller to choose to continue with either an ejection chain or an ejection tree.

6 Repair operations

A *repair operation*, or just *repair*, is a change intended to remove a defect. This section gives an overview of the repairs used by KHE14's ejection chain algorithm. Sect. 7 explains how they are used to repair defects.

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 from one assignment to a different assignment. An *unassignment* is a change 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 fourth way, also applicable only to moves, is the *Kempe* way. For example, a Kempe meet move begins 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.

This makes 7 repairs on variables: *unassignment*, *basic assignment*, *basic move*, *ejecting assignment*, *ejecting move*, *swap*, and *Kempe move*. Applying them to both meets and tasks gives 14 operations. KHE14 uses most of them.

When a Kempe meet move succeeds, the result is usually a simple move or swap. A single operation that could be either allows a solver to try moving a meet to each t_2, whether its resources are free then or not. Tests not reported here in detail show that the median number of meets moved by one Kempe meet move is 2, although 20 or more meets move in rare cases.

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.

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 ejecting move with the same parameters, the ejecting move being omitted when the Kempe move (successful or not) does not try to move anything in the opposite direction, since the two repairs are identical then.

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	13.00551	11.00626	**10.00758**	12.00580	14.6	44.0	23.3	72.5
AU-SA-96	4.00022	**1.00024**	15.00103	26.00081	76.2	319.2	135.3	132.1
AU-TE-99	5.00152	**5.00101**	9.00209	12.00172	2.1	11.3	8.3	6.8
BR-SA-00	0.00044	1.00063	**0.00039**	1.00068	0.7	1.1	1.0	0.6
BR-SM-00	12.00128	16.00088	**3.00123**	18.00072	2.2	4.3	7.4	1.9
BR-SN-00	0.00145	2.00175	**0.00118**	4.00193	2.3	4.5	3.7	2.9
DK-FG-12	0.03132	**0.03039**	0.03513	0.03835	366.0	583.3	353.6	325.8
DK-HG-12	12.05069	12.04959	**12.04866**	12.05460	724.0	1445.9	605.7	772.1
DK-VG-09	3.04709	**2.04271**	2.04954	2.05395	874.6	1213.1	503.6	524.2
UK-SP-06	32.01188	**31.00918**	49.01156	41.00734	420.9	910.2	335.8	353.3
FI-PB-98	4.00028	3.00053	**1.00023**	3.00042	10.2	23.5	8.2	9.7
FI-WP-06	**0.00024**	0.00045	0.00027	1.00027	8.4	29.7	6.6	8.2
FI-MP-06	**0.00110**	0.00118	0.00127	5.00124	6.8	12.3	5.6	6.2
GR-H1-97	**0.00000**	**0.00000**	**0.00000**	**0.00000**	5.9	5.7	5.7	5.6
GR-P3-10	**0.00011**	0.00032	6.00046	0.00033	7.8	12.3	12.4	8.8
GR-PA-08	0.00016	0.00014	**0.00013**	0.00017	13.6	16.9	7.2	6.8
IT-I4-96	0.00150	0.00067	**0.00056**	0.00068	13.0	16.3	11.9	15.3
KS-PR-11	0.00020	0.00022	**0.00018**	0.00028	131.4	156.7	170.3	136.3
NL-KP-03	0.01792	0.01641	0.01954	**0.01414**	444.0	1102.9	427.2	523.5
NL-KP-05	16.07919	15.04026	16.07289	**14.05057**	375.0	638.5	354.5	325.1
NL-KP-09	33.07335	**8.10415**	41.15690	18.08255	74.3	215.8	127.0	89.7
ZA-LW-09	**16.00004**	19.00014	21.00018	17.00016	9.4	11.9	9.8	9.2
ZA-WD-09	26.00000	**19.00000**	21.00000	38.00000	13.5	28.9	22.1	12.0
ES-SS-08	**0.01117**	0.01167	0.01731	0.02407	27.2	88.0	21.9	20.5
US-WS-09	0.00784	0.00746	0.00738	**0.00718**	41.3	47.8	38.6	41.9
Average	7.01378	**5.01304**	8.01742	8.01391	146.6	277.8	128.3	136.4

Kempe meet moves are implemented more generally than described here. They support hierarchical timetabling and preserving regularity, and swap meets of different durations in some cases. For these details, see [10] and [12].

Table 5 compares Kempe, ejecting, and basic meet moves. The data are noisy, but Kempe/ejecting meet moves seem best when both cost and running time are taken into account. When evaluating alternatives the author prefers not to reduce noise by averaging several runs, since what is needed are ideas powerful enough to make themselves heard above the noise. Table 5 shows that Kempe/ejecting meet moves are marginal by this standard.

KHE also uses *combined repairs* which are sets of repairs treated as a unit: for a combined repair to succeed, all the repairs that make it up must succeed.

For example, suppose two nodes (Sect. 3) have meets of the same durations (one of duration 2 and four of duration 1, say). A *node swap* is a combined repair which swaps the starting times of corresponding meets in those nodes. Swappable nodes are common, since courses of equal duration are common and 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 also preserves regularity and tends to not create new spread events defects.

Meet splitting and merging are mostly done during the structural phase, but they are occasionally useful during repair. A *meet split* splits a meet in two. A *split move* combines a meet split with a Kempe meet move of one fragment. Split moves are tried after Kempe meet moves in some cases. A *meet merge* merges two meets. Mergeable meets are rarely adjacent in time, so the *merge move* is more useful in practice. It combines a Kempe meet move of a meet to alongside a meet it can merge with, with a merge of the two meets.

Most of these repairs are not new. The author has used Kempe meet moves, node swaps, and ejecting task moves before [9,10]. Others have used ejecting meet moves ([14], for example). Split and merge moves, and the 'unassign and reduce' repairs described in Sect. 7, seem to be new.

7 Repairing defects

This section explains how the ejection chain algorithm repairs defects using the repair operations of Sect. 6. For each type of defect, this section defines a set of repairs. The augment function for that type 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 have been tried without success.

One lesson that the author has learned is the importance of *precision* in augment functions: ensuring that they include only repairs that reduce the cost of the defects they are supposed to target, and that all repairs that do this are included, subject to reasonable limits on the number and complexity of the repairs, and to common-sense assessments of their likelihood of success. In the course of this work, improving the precision of an augment function has usually led to improved solution cost and running time.

The repairs tried by an augment function are basically unordered, but some attention to order can be helpful. One reason for this is diversity. For example, when each repair assigns one resource from a list to a task, KHE14 chooses the starting point in the list randomly. Another reason is to try first those repairs which seem most likely to succeed. For example, when repairing limit idle times defects, KHE14 tries meet moves in order of increasing meet duration, since meets of small duration are more likely to fit into idle times.

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 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 groups monitors when they have the same type and monitor the same thing in reality (examples appear below), and repairs a group by repairing any one of its members. 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] groups all defects related to one resource. The elements of such a group may have different types, so repairs specific to one type are not used. 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.)

Calls on the ejection chain algorithm are either *time repair* calls, which repair assignments of times to meets, or *resource repair* calls, which repair assignments of resources to tasks. These differ very little. The set of defects targeted by the main loop is different. The augment functions are the same, although options passed to them change their behaviour. For example, KHE14 uses options which prevent changes to resource assignments during time repair and to time assignments during resource repair. This needs to be kept in mind when reading the descriptions of augment functions below.

Here now is the full list of defect types and how KHE14 repairs them. The author has tried some quite complex repairs, including resource repairs that also change time assignments, but KHE14 mostly uses only the simplest repairs that remove the defects they are given, possibly extended to increase their chance of success (as meet moves are extended into Kempe meet moves, for example). One reason for this is that complex repairs have proved to be slow and subject to diminishing returns. Another is a desire to build a relatively simple and coherent foundation for future work.

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.

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 taken to be 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 swaps of nodes with the same preassigned resources. For example, 6 Science meets running simultaneously when there are only 5 Science laboratories will produce one demand defect, causing all repairs to be tried that move any of the 6 meets away from the problem time, even before any room assignments are made. Simple clashes also produce demand defects, and are repaired in the same way.

Demand monitors derived from avoid unavailable times, limit busy times, and limit workload monitors are not grouped. If any of those 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: those described below for avoid unavailable times, limit busy times, and limit workload monitors.

Split events defects and distribute split events defects. These are cases of events split into too few or too many meets, or meets of unwanted durations. Event splitting is handled by the structural phase (Sect. 3), making these defects rare during repair. Nevertheless, they occur in some instances, so they are carefully analysed and all single meet splits and merge moves that remove them are tried. Split events and distribute split events monitors whose events are joined by link events constraints handled structurally are grouped.

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 by trying all ejecting meet assignments to times in the meet's domain. Assign time monitors whose events are joined by link events constraints handled structurally are grouped.

It is important to assign a time to every meet, so, although ejection chains may unassign meets temporarily as they go, no chain or tree is accepted 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 Sport meet that prefers afternoons but is assigned a morning time. They are handled by trying all Kempe/ejecting moves of the meet to preferred times. Prefer times monitors whose events are joined by link events constraints handled structurally are grouped if they request the same times.

Spread events defects. These are cases where too few or too many meets derived from a given set of events begin in a given set of times, usually one day. They are handled by trying all Kempe/ejecting meet moves of the meets from outside the set of times to inside it, or vice versa, depending on whether the problem is too few meets or too many. 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. Event linking is handled structurally (Sect. 3), and link events defects are ignored during repair.

Order events defects. These are cases where one event should appear earlier in the cycle than another, but it doesn't. KHE14 does not repair them, because the XHSTT-2014 data set has no order events constraints. In future, it will be easy to add meet move repairs for them. Order events monitors whose events are joined by link events constraints handled structurally are grouped.

Assign resource defects. These are cases of unassigned tasks, handled 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: an ordinary classroom instead of a Science laboratory, for

Table 6 Effectiveness of augment functions during time repair. For each augment function, the number of calls to the function during time repair, the number of successful calls, and the ratio of the two as a percentage, over all instances of archive XHSTT-2014. Only non-zero rows are shown.

Augment function	Total	Successful	Percent
Ordinary demand	440330	1974	0.4
Split events	104	31	29.8
Assign time	5317024	22922	0.4
Spread events	4975247	8850	0.2
Avoid unavailable times	42369	227	0.5
Limit idle times	3645128	16599	0.5
Cluster busy times	119048	884	0.7
Limit busy times	381587	1996	0.5

Table 7 Effectiveness of augment functions during resource repair. For each augment function, the number of calls to the function during resource repair, the number of successful calls, and the ratio of the two as a percentage, over all instances of archive XHSTT-2014. Only non-zero rows are shown.

Augment function	Total	Successful	Percent
Ordinary demand	3711	53	1.4
Assign resource	173270	4805	2.8
Prefer resources	6487	0	0.0
Avoid split assts	5008	189	3.8
Limit busy times	17062	106	0.6
Limit workload	10596	711	6.7

example. They are handled by trying all ejecting task moves of the task to its preferred resources. 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, three taught by Smith and two taught by Brown, creates an avoid split assignments defect, also called a *split assignment*.

Most defects can be repaired by assigning or moving one meet or task, but several tasks may need to be moved in order to reduce the cost of an avoid split assignments defect, making repair difficult in general. Accordingly, structures that prohibit split assignments are present for most of KHE14. Near the end of the solve the prohibitions are removed and split assignments are constructed, since they are usually better than nothing.

Given one of these split assignments, the ejection chain algorithm tries one repair for each distinct resource assigned to the tasks involved. The repair attempts to remove that resource from the split assignment without adding any new resources, as follows. All involved tasks assigned that resource are unassigned, all involved tasks' domains are reduced to the other participating resources only, and a whole set of ejection chains is tried, each aiming to reassign one of the unassigned tasks, making an ejection tree (Sect. 5) rather than an ejection chain. The repair succeeds only if all the chains succeed. We call this general idea *unassign and reduce*.

Ejection tree repairs (including others described below) are expensive and unlikely to work, so are only tried on main loop defects or when only one task or meet needs to be unassigned. It is important to give them every chance, since there is often nothing better to try; so when the defect is a main loop defect the whole solution is reopened for visiting before each sub-chain.

Avoid clashes defects. These are cases where a resource attends two meets at the same time. Avoid clashes monitors are detached, since demand monitors do their job and more; so there are no avoid clashes defects.

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, with an upper limit of 0 on the number of those times that the resource may be busy. 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 has an *idle* time: a time when the resource is not *busy* (attending an event), but such that it is busy both earlier that day and later. They are handled by trying each ejecting move of a meet assigned the resource at the start or end of a day to a time that reduces idle times.

Limit idle times 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: the NL-KP-03 instance, for example, has 453 resources representing individual students, but only 297 groups, or 285 when event linking is taken into account.

Cluster busy times defects. These are cases where a resource is busy on too few or too many days. When the problem is too few days, all ejecting moves are tried which move a meet to an empty day. When the problem is too many days, an unassign and reduce repair is tried, like the one for avoid split assignments defects except that, instead of encouraging a set of tasks to be assigned one less resource, a set of meets is encouraged to be assigned one less day. For each busy day there is one repair. It unassigns all the resource's meets on that day, reduces all the resource's meets' domains to exclude that day and all other empty days, and tries a whole set of ejection chains, each aiming to reassign one of the unassigned meets. Cluster busy times monitors are grouped in the same way as limit idle times monitors.

Limit busy times defects. These are cases where a resource is underloaded or overloaded during some set of times, typically one day. For example, teacher Jones might expect to be busy for between 3 and 7 of the 8 times on any day; if not, that is a limit busy times defect. When the problem is an underload, first, all ejecting meet moves of one of the resource's meets which increase the meet's overlap with the day are tried; and second, since a completely empty day is by definition not a defect, an ejection tree repair similar to the one for cluster busy times defects is tried, to remove all meets from the day. When the problem is an overload, first, all ejecting task moves of unpreassigned tasks contributing to the overload are tried, to other resources in their domains; and

Table 8 Effectiveness of repair operations during time repair. For each augment function and repair operation, the number of calls on that repair operation made by that augment function during time repair, the number of successful calls, and the ratio of the two as a percentage, over all instances of archive XHSTT-2014. Only non-zero rows are shown.

Augment function : Repair operation	Total	Successful	Percent
Ordinary demand : Kempe meet move	120254	1702	1.4
Ordinary demand : Ejecting meet move	307619	238	0.1
Ordinary demand : Basic meet move	76	2	2.6
Ordinary demand : Node swap	1524	8	0.5
Ordinary demand : Split move	10857	24	0.2
Split events : Merge move	104	31	29.8
Assign time : Ejecting meet assignment	5317024	22922	0.4
Spread events : Kempe meet move	1372346	5959	0.4
Spread events : Ejecting meet move	3602641	2890	0.1
Spread events : Split move	260	1	0.4
Avoid unavailable times : Ejecting meet move	42369	227	0.5
Limit idle times : Kempe meet move	1289812	14985	1.2
Limit idle times : Ejecting meet move	2350474	1586	0.1
Limit idle times : Split move	4842	28	0.6
Cluster busy times : Kempe meet move	28270	540	1.9
Cluster busy times : Ejecting meet move	50327	18	0.0
Cluster busy times : Cluster unassign and reduce	40451	326	0.8
Limit busy times : Kempe meet move	105867	1705	1.6
Limit busy times : Ejecting meet move	269390	260	0.1
Limit busy times : Split move	47	1	2.1
Limit busy times : Limit busy unassign and reduce	6283	30	0.5

Table 9 Effectiveness of repair operations during resource repair. For each augment function and repair operation, the number of calls on that repair operation made by that augment function during resource repair, the number of successful calls, and the ratio of the two as a percentage, over all instances of archive XHSTT-2014. Only non-zero rows are shown.

Augment function : Repair operation	Total	Successful	Percent
Ordinary demand : Ejecting task move	3711	53	1.4
Assign resource : Ejecting task assignment	173270	4805	2.8
Prefer resources : Ejecting task move	6487	0	0.0
Avoid split assts : Split tasks unassign and reduce	5008	189	3.8
Limit busy times : Ejecting task move	17062	106	0.6
Limit workload : Ejecting task move	10596	711	6.7

second, all ejecting meet moves of one of the resource's meets which decrease the meet's overlap with the day are tried. A form of partial detachment ensures that only defects not detected as demand defects are handled. Limit busy times monitors are grouped in the same way as limit idle times 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.

Which augment functions are most effective? Measuring 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 perfect, is to say that a call on an augment function is effective when it lies on a chain that improved the solution, and ineffective otherwise. The ratio of effective to effective plus ineffective calls, given as a percentage, measures the function's effectiveness.

Table 6 presents the number of calls on each time repair augment function when solving the instances of XHSTT-2014, and their effectiveness, measured as just described. Table 7 does the same for resource repair. Interpretation is problematical, but all the augment functions seem to be making a reasonable contribution. The apparently poor results on prefer resources defects may be due to anomalous data in instance US-WS-09.

Which repairs are most effective? Again, finding a good measure is not easy. For example, on any given defect one type 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.

Tables 8 and 9 are like Tables 6 and 7 except that they contain one row for each type of repair of each type of defect. Some of the results are quite suggestive: the tiny number of successful node swaps, for example.

Repairs targeted at specific defects are rare in the timetabling literature. The above is partly old [9,10] and partly new. Disentangling new from old would be tedious, but the ejection tree repairs are new, and this paper is the first to repair a large set of defect types with polymorphic ejection chains.

8 The algorithm

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

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 [6,7,10]; here is an overview. For each resource to which a hard avoid clashes constraint applies it builds a *layer*, the set of nodes containing meets preassigned that resource. After merging layers wherever one's nodes are a subset of the other's, 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.

The minimum-cost matching approach to meet assignment assumes that the cost of each meet assignment in one layer is independent of the others. This is true for most kinds of constraints, but false for a few, notably cluster busy times, limit idle times, and limit busy times constraints. So monitors for these constraints are detached while finding minimum-cost matchings. Recently, the

Table 10 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. The anomalous result for DK-VG-09 (where the best of 8 is worse than one) may be due to nondeterminism in imposing the time limit.

Instance	C:KHE14	C:KHE14x8	T:KHE14	T:KHE14x8
AU-BG-98	13.00520	**3.00608**	13.2	41.3
AU-SA-96	4.00022	**2.00016**	75.6	189.1
AU-TE-99	5.00152	**2.00152**	2.2	6.3
BR-SA-00	0.00044	**0.00031**	0.7	2.1
BR-SM-00	12.00128	**6.00112**	2.4	5.1
BR-SN-00	0.00145	**0.00113**	2.3	6.1
DK-FG-12	0.03317	**0.03310**	345.8	737.8
DK-HG-12	12.05364	**12.04759**	712.5	1744.3
DK-VG-09	**2.04097**	2.04600	987.7	1853.2
UK-SP-06	33.01100	**28.01140**	366.5	775.6
FI-PB-98	3.00031	**0.00015**	9.4	20.7
FI-WP-06	0.00024	**0.00016**	8.7	17.7
FI-MP-06	0.00147	**0.00093**	4.5	12.3
GR-H1-97	**0.00000**	**0.00000**	5.8	14.4
GR-P3-10	0.00011	**0.00002**	7.7	18.5
GR-PA-08	0.00016	**0.00009**	14.1	26.8
IT-I4-96	0.00054	**0.00046**	14.3	31.5
KS-PR-11	0.00020	**0.00012**	194.0	382.1
NL-KP-03	0.01439	**0.01286**	426.0	1005.3
NL-KP-05	18.05189	**8.06250**	400.5	787.5
NL-KP-09	16.07930	**10.05125**	89.1	224.4
ZA-LW-09	20.00018	**13.00016**	10.8	23.9
ZA-WD-09	26.00000	**16.00000**	13.6	36.7
ES-SS-08	0.01117	**0.00616**	26.2	55.0
US-WS-09	0.00758	**0.00677**	50.0	111.2
Average	6.01265	**4.01160**	151.3	325.2

author has tried several ideas for taking these constraints into account during the initial time assignment [12]. In KHE14 as presented here, some reductions of meet domains before time assignment begins are used to prevent some cluster busy times defects. Other plausible reductions (removing the first or last time of a day of a limit idle times constraint, for example) are tried while assigning one layer. This produces multiple minimum-cost matchings. Each is evaluated with the troublesome monitors re-attached, and the best is chosen.

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 still said to have been assigned, but the assignment algorithm does nothing.

After each layer is assigned, a call is made to the ejection chain time repair algorithm (Sect. 5). Its main loop is targeted at the event defects of the layer and the resource defects of the layer's preassigned resources, but its recursive calls may spread into earlier layers. After all layers have been assigned and repaired, another ejection chain time repair call is made, targeted at all layers. Then the structures that encourage regularity in time are removed, and a second all-layers time repair call is made.

Table 11 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	4	0	-
AU-SA-96	0	0	0	0	13	0	-
AU-TE-99	0	0	0	-	8	0	-
BR-SA-00	0	4	0	0	0	-	-
BR-SM-00	0	21	3	0	0	-	-
BR-SN-00	0	13	0	0	0	-	-
DK-FG-12	-	-	0	-	85	-	-
DK-HG-12	-	-	3	-	115	-	-
DK-VG-09	-	-	1	-	108	-	-
UK-SP-06	-	-	2	-	7	0	-
FI-PB-98	0	-	0	0	0	-	-
FI-WP-06	0	-	0	0	0	-	-
FI-MP-06	0	-	0	0	0	-	-
GR-H1-97	-	-	0	-	0	0	-
GR-P3-10	0	-	0	0	0	0	-
GR-PA-08	-	-	0	-	2	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	0	-	1	0	4	0	-
NL-KP-09	0	-	0	0	4	0	-
ZA-LW-09	3	-	2	0	-	0	-
ZA-WD-09	-	-	1	0	0	0	-
ES-SS-08	0	-	0	-	74	-	-
US-WS-09	0	-	0	0	57	-	-
Total	3	38	13	0	481	0	

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

During resource assignment (including repair), changes which reduce the number of demand defects are almost impossible to find. Resource assignments cannot do it, since they reduce the domains of demand nodes, reducing the choice of matchings. Even when resource repairs change meet assignments (not done in KHE14 as presented here), reductions are not likely, because similar changes were tried during time repair, when fewer resources were assigned so more choices were open. Since reductions are almost impossible and demand defects lead to real problems in the end (unassigned meets or tasks, or clashes, and so on), changes during resource assignment that increase the number of demand defects are rejected, except at the end, when a last-ditch attempt is made to assign all unassigned tasks. This idea comes from [9].

Each resource assignment phase has three parts. In the first part, which assigns most tasks in practice, violations of avoid split assignments and prefer resources constraints are prohibited structurally, and assignments that increase

Table 12 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	52	0	1	-	-	21	0
AU-SA-96	1	0	0	1	0	-	-	1	0
AU-TE-99	0	0	14	2	0	-	-	2	0
BR-SA-00	-	-	-	0	0	6	0	-	-
BR-SM-00	-	-	-	1	1	5	8	-	-
BR-SN-00	-	-	-	0	0	10	6	-	-
DK-FG-12	0	0	-	0	0	105	122	88	-
DK-HG-12	5	0	-	1	-	153	154	95	-
DK-VG-09	0	0	-	0	-	92	47	37	-
UK-SP-06	0	-	-	13	-	76	-	-	-
FI-PB-98	-	-	-	0	0	9	-	0	-
FI-WP-06	-	-	-	0	-	7	-	6	-
FI-MP-06	-	-	-	0	11	12	-	5	-
GR-H1-97	-	-	-	0	0	-	-	-	-
GR-P3-10	-	-	-	0	0	0	-	1	-
GR-PA-08	-	-	-	0	0	3	-	0	-
IT-I4-96	-	-	-	0	4	12	0	2	-
KS-PR-11	-	-	-	0	0	6	-	0	-
NL-KP-03	0	0	-	0	0	174	3	48	-
NL-KP-05	0	0	-	2	45	100	8	77	-
NL-KP-09	1	0	-	2	24	20	8	-	-
ZA-LW-09	-	-	-	3	-	-	-	-	-
ZA-WD-09	-	-	-	4	0	-	-	-	-
ES-SS-08	0	0	-	0	0	-	0	0	-
US-WS-09	0	168	-	0	-	-	-	-	-
Total	9	168	66	29	86	790	356	383	0

the number of demand defects are rejected as just described. An algorithm is called that tries to assign a resource to each unpreassigned task of the current type. If avoid split assignments constraints are present, a *resource packing* algorithm which follows a bin packing paradigm is used. In other cases a constructive heuristic is used, more effectively than usual because of the guidance provided by the global tixel matching. Both these algorithms come from [9], where resource packing was found to be the best of three 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 types of resources whose event resources have avoid split assignments constraints. It removes structures that prevent split assignments, finds split assignments for unassigned tasks using a specialized construction heuristic, and 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 reassigns pairs of resources [11]. The details are in [12] as usual; they are omitted here because they are peripheral to the main ideas of this paper, and it 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 prohibitions and calls ejection chain repair yet again. The third part for each resource type is delayed until after the first and second parts are complete for all resource types.

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.

As described in Sect. 5, ejection chain repair offers the option of a soft time limit which helps to cap running time, without enforcing any hard limit. KHE14 as presented here has a soft time limit of 300 seconds. The author would prefer ejection chain repair to proceed to its natural end, but at present that leads to run times of several hours on some of the larger instances. A glance at Table 10 reveals which solves are affected by the soft time limit.

Table 10 shows the overall performance of KHE14 and its variant KHE14x8, which runs KHE14 8 times in parallel and keeps a best solution. Instead of using random numbers, each run is given a different *diversifier*, which is 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 of equal cost. These solutions are available from the KHE web page [12]. A listing of the remaining defects appears in Tables 11 and 12.

9 Conclusion

KHE14 is the first timetabling solver to apply polymorphic ejection chains to a wide variety of types of defects. It has also pioneered several repair operations, including three ejection tree repairs.

As of the time of writing (August 2014), slower variants of KHE14 have produced best known solutions for several XHSTT-2014 instances, as reported by [17]. These include solutions for AU-BG-96 with cost 1.00386, for IT-I4-96 with cost 0.00040, and for NL-KP-03 with cost 0.00617. The solutions reported in Table 10 for DK-FG-12, DK-VG-09, and US-WS-09 are also new bests.

The author's main goal has always been to build a *robust* solver: one that finds very good solutions to a wide range of instances quickly—say, within ten seconds, or sixty seconds for instances with hundreds of student resources.

KHE14x8 is not at this standard yet. With respect to solution quality the situation is not clear, in part because solutions to the full set of XHSTT-2014 instances by other solvers were not available at the time of writing. It would be easy to compare KHE14x8 with the best known solutions, as reported by [17], but that would not be fair, because those solutions were found by a variety of solvers, including very slow solvers and solvers that run to optimality on small instances and produce nothing useful on large ones. The remaining defects (Tables 11 and 12) define the agenda for future work here.

In running time, KHE14 is dominated by the ejection chain algorithm, which can be slow and has had to be artificially limited in this paper. Better repairs can dramatically improve its running time, but they are becoming hard to find. Construction of initial solutions with fewer defects, and faster computer hardware, produce more modest improvements, but seem more practicable at this point. Hand analyses, aimed, for example, at understanding the remaining spread events defects in the Danish (DK) instances, are also needed.

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, Resource assignment in high school timetabling, Annals of Operations Research, 194, 241–254 (2012)
10. Jeffrey H. Kingston, Repairing high school timetables with polymorphic ejection chains, Annals of Operations Research, DOI 10.1007/s10479-013-1504-3
11. Jeffrey H. Kingston, Timetable construction: the algorithms and complexity perspective, Annals of Operations Research, 218, 249–259 (2014) DOI 10.1007/s10479-012-1160-z
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. Keith Murray, Tomás Müller, and Hana Rudová Modeling and solution of a complex university course timetabling problem, Practice and Theory of Automated Timetabling VI (Springer Lecture Notes in Computer Science 3867), 189–209 (2007)
15. Christos. H. Papadimitriou and Kenneth Steiglitz, Combinatorial Optimization: Algorithms and Complexity, Prentice-Hall (1982)
16. 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)
17. Gerhard Post, XHSTT web site, `http://www.utwente.nl/ctit/hstt/` (2011)
18. 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)
19. 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)
20. G. Schmidt and T. Ströhlein, Timetable construction—an annotated bibliography, The Computer Journal, 23, 307–316, (1980)