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

Repairing High School Timetables with Polymorphic Ejection Chains

Jeffrey H. Kingston

Received: date / Accepted: date

Abstract This paper introduces polymorphic ejection chains, and applies them to the problem of repairing time assignments in high school timetables while preserving regularity. An ejection chain is a sequence of repairs, each of which removes a defect introduced by the previous repair. Just as the elements of a polymorphic list may have different types, so in a polymorphic ejection chain the individual repairs may have different types. Methods for the efficient realization of these ideas, implemented in the author's KHE framework, are given, and some initial experiments are presented.

Keywords High school timetabling · Ejection chains

1 Introduction

Most work in timetabling utilizes two phases: a *construction* phase, in which an initial solution is built, for example by a construction heuristic, and a *repair* phase, in which the solution is improved, for example by local search.

Local search works well initially, but its effectiveness declines, for two reasons. The solution's *defects* (specific points where it is deficient) become few and isolated, but local search continues to change parts of the solution where there are no defects to remove. And a point is reached where the small changes it typically makes (moves and swaps) have all been tried and have little chance of improving the solution.

Some repair methods attempt to avoid these problems. One well-known example is very large-scale neighbourhood search (Ahuja et al. 2002; Meyers and Orlin 2007). It repeatedly deassigns and reassigns many related variables. It can be targeted at specific defects by building neighbourhoods around them (Ryan and Rezanova 2010), and it may make many more changes than one move or swap.

Jeffrey H. Kingston
School of Information Technologies
The University of Sydney
E-mail: jeff@it.usyd.edu.au

This paper repairs time assignments in high school timetables using *ejection chains*. An ejection chain is a sequence of *repair operations* (also called *repairs*), which are usually but not necessarily moves and swaps. The first repair removes one defect but introduces another; the next removes that defect but introduces another; and so on. Starting at defects and coordinating repairs like this avoids the problems with local search identified above. A key point is that defects that appear as the 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. Augmenting paths, found in matching algorithms, are examples of them, and they occur naturally to anyone who tries to repair a timetable by hand. They were named by Glover (1996), who applied them to the travelling salesman problem. In timetabling, they have been used in nurse rostering (Dowsland 1998), teacher assignment (Kingston 2008), and time repair (Kim and Chung 1997; de Haan et al. 2007). Kim and Chung (1997) is very cryptic, unfortunately.

This paper tells three interwoven stories. The first story is concerned with repair operations for high school timetables which improve them without disrupting their *regularity*. Informally, a regular timetable is one whose events occur at similar times. For example, the well-known arrangement followed by many North American universities, in which each course occupies one of the sets of times $\{Mon1, Wed1, Fri1\}$, $\{Mon2, Wed2, Fri2\}$, and so on, is perfectly regular. Although this paper uses these repairs as steps in ejection chains, they could equally well be used in the conventional way, to define neighbourhoods for local search algorithms.

The second story concerns the design of ejection chain software. This paper seems to be the first to explicitly recognize the polymorphism inherent in ejection chains: each repair in the chain may have a different type. This insight leads to a framework in which any number of different types of defects can be repaired together, bringing the method to a level of generality (in the sense of applicability to many combinatorial optimization problems) approaching that of metaheuristics.

The third story concerns the implementation of these ideas within the author's KHE high school timetabling framework (Kingston 2010a), including data structures for expressing regularity and marshalling defects for repair. Ejection chain algorithms are *white-box algorithms* (Parkes 2010): they need access to more information about the current solution than just its cost, so the implementation effort is quite high.

This paper uses the recently developed XML format for high school timetabling as its specification of the high school timetabling problem (Kingston 2010b; Post et al. 2012, 2011). This specification will not be repeated here, since it is enough to understand that the problem is to assign times and resources (teachers, rooms, and so on) to a collection of events so as to avoid clashes and satisfy a number of other constraints, as far as possible. More detail is given as needed throughout the paper.

2 Defects

Local search algorithms need access to the cost of the solution and to the repair operations that may be used to change it (moves, swaps, and so on). Ejection chain algorithms also need access to its *defects*: the specific points where problems lie.

In high school timetabling, as in most real-world combinatorial optimization problems, there are several types of defects (clashes, events scheduled at undesirable times, and so on), and several types of repairs. This inherent polymorphism is partly obscured in most timetabling work: repair types may be recognized, but the defects are lumped together into a single number, the overall cost. Fully polymorphic repair, in which defects are repaired in ways specific to their types, is rare in the timetabling literature. One recent paper used it to improve staff rosters (Ásgeirsson 2010).

Surprisingly, for the time repair problem which is the focus of this paper, there are only three defect types. The XML format has 15 constraint types, and there is one defect type for each constraint type, but only two of them matter here: the *prefer times defect*, which occurs when an event is assigned an undesirable time (for example, an afternoon time when the event is supposed to occur during the morning), and the *spread events defect*, which occurs when the events for one subject are supposed to be spread evenly through the *cycle* (the chronologically ordered sequence of all times when events may occur), but instead some of them occur close together in time.

The third defect type is not derived from any constraint in the XML format, but it is nevertheless the most important type of all. Suppose a time assignment assigns six Science events, each requiring one Science laboratory, to the third time on Tuesdays, and suppose the school has only five Science laboratories. Then when rooms are assigned later on, one of these six events must miss out.

As is well known, such problems can be detected by building a biparite matching graph for each time of the cycle. Each graph contains one *supply node* for each resource in the instance, and one *demand node* for each demand for a resource made by the events running at the graph's time. Each demand node is connected to the supply nodes that represent resources capable of satisfying the demand. To decide whether the demands of the events running at one time can all be satisfied, find a maximum matching in this graph. If it fails to touch every demand node, there is a problem.

For example, the six Science events produce six demand nodes connected to the five supply nodes representing the five Science laboratories (plus other demands for student groups and teachers). One of the six demand nodes will fail to match.

KHE has matching graphs, and considers each unmatched demand node to be a defect called a *demand defect*. Demand defects include clashes and use of resources at times when they are unavailable as special cases. For example, a clash turns up as two demand nodes, both linked only to a single supply node representing the resource in contention. So two of the constraints of the XML format that apply to resources, the *avoid clashes* and *avoid unavailable times* constraints, are taken account of in this way. Some other constraints on the timetables of resources are relevant to time assignment when the resources involved are preassigned. They are not treated here, but the polymorphic approach makes it straightforward to do so.

3 Time assignment in KHE

This section explains how time assignment is modelled by the author's KHE platform. There is a lot of detail, and it will be necessary to introduce some of KHE's jargon.

In KHE, an *instance* of the high school timetabling problem (a case of the problem for a given school in a given year or semester) is a structured object which remains immutable after creation. A *solution* of an instance is a mutable structured object containing the time and resource assignments which define the solution. Keeping solutions separate from instances has several advantages. For example, it helps when constructing multiple solutions in parallel.

The lessons for one subject usually need to be spread evenly through the cycle. For example, a class might attend Mathematics for a total of six times, which are to be spread through the five days of the week. On the day when the class meets for two times, it is usually best for those times to be adjacent, forming one lesson of twice the usual duration. It is common for the total number of times devoted to one subject to be rigidly prescribed, but for the way in which that total is split into individual lessons of varying duration to be more flexible. This is handled as follows.

An instance contains *events* of fixed duration, each representing a single subject, plus constraints saying how they may be split into lessons. For the Mathematics example just given, there might be one Mathematics event of duration 6, and constraints saying that 5 or 6 lessons are required, whose durations may be 1 or 2.

In KHE, the solution contains the individual lessons, which are called *meets*. Each meet has a fixed integer *duration*, meaning that it runs for that many consecutive times; a *starting time*, which is a variable requiring assignment; and a set of *tasks*, each of which is a variable requiring the assignment of one resource, such as a student group, a teacher, or a room. Tasks contain demand nodes and are the source of demand defects. Meets and tasks may be preassigned.

Although the aim is to assign a starting time to each meet, KHE does this in an indirect way, by assigning one meet to another. The meaning is that the two meets must have the same starting time, but that time is yet to be determined. Assignment is directed (it assigns one meet to another, not two meets to each other) and includes an offset. For example, suppose meet m_1 has duration 1 and meet m_2 has duration 2. Then m_1 may be assigned to m_2 with offset 0, meaning that m_1 starts at the same time as m_2, or with offset 1, meaning that it starts at the second time of m_2. The assigned meet may not run outside the interval of time that the meet it is assigned to is running. For example, m_2 may not be assigned to m_1 at any offset.

Assigning one meet to another supports *hierarchical timetabling*, in which a few meets are timetabled together, then the whole assembly is timetabled into a larger context, and so on until the complete timetable is built. When this is done, a sequence of assignments builds up, from one meet to another, from that meet to a third, and so on. Special meets called *cycle meets* are available such that assignment to a cycle meet effectively assigns a time. When every assignment sequence ends at a cycle meet, every meet has a starting time. There is usually one cycle meet for each sequence of adjacent times in the cycle not spanning a meal break or the end of a day.

One use for hierarchical timetabling is to link meets whose events are required to run simultaneously. For example, suppose there are five Mathematics events of equal duration, one for each of five Year 8 student groups. These events are required to run simultaneously so that the Year 8 students can be regrouped by ability at Mathematics. To handle this, break the events into meets of the same durations, choose one student group to be the leader, and assign the meets of the non-leader student groups to corresponding meets of the leader student group, with offset 0. This forces the events to be simultaneous. The common starting times of the meets will be determined later, when assignments are made to the leader student group's meets.

There are good reasons to remember that certain meets are derived from one event. For example, they usually need to be spread evenly through the cycle, which might involve finding assignments for them all at the same point during solving. In KHE, meets may be grouped together into sets called *nodes*. Usually, one node holds the meets derived from one event, but there are exceptions, such as *runaround nodes*, which hold small timetables in which several student groups attend several subjects. There is also a *cycle node* holding the cycle meets.

By convention, a non-cycle meet lies in a node if and only if its assignment may be changed. The meets of the leader student group in the example above would lie in a node, but the other meets, which already have final assignments, would not.

In addition to holding meets, each node except the cycle node usually has a parent node, forming the nodes into a tree rooted at the cycle node that the author has called the *layer tree* (Kingston 2006). When a meet lies in a node, it may only be assigned to meets in the parent of that node, forcing all sequences of assignments to eventually end in cycle meets as desired. Although it is not forbidden, there is an assumption that meets which share a node will not be assigned so as to overlap in time. For example, it is not desirable for two meets derived from the same event to overlap in time.

4 Regularity

Regularity has two forms. *Meet regularity* occurs when the sets of times at which two meets are running are either disjoint, or one is a subset of the other. For example, two meets of duration 2, one starting at the first time on Wednesdays, the other at the second time, are not regular. When all meets have duration 1, meet regularity is automatic. In practice, most meets have duration 1 or 2, and it only takes a little care to achieve very good meet regularity.

Node regularity occurs when the sets of times at which the meets of two nodes are running are either disjoint, or one is a subset of the other. Since most nodes contain several meets, node regularity is harder to achieve than meet regularity.

Node regularity is important. In the author's experience, based on Australian high schools, the main type of defect left over at the end of solving is the *split assignment*, in which one teacher attends some of the meets of an event, and another teacher attends the others. Split assignments are permitted, but they are undesirable. Some of their causes are inherent in instances: part-time teachers who are difficult to utilize effectively, and tight teacher workload limits that force every teacher to be used to capacity. But node irregularity also causes split assignments, and it is the one cause that solvers can do something about.

Because node regularity depends on coordinating the assignments of many meets (perhaps five derived from one event and five from another), it is not likely to arise by chance in the course of repairing the assignments made to individual meets, not even if the solver explicitly measures irregularity and favours assignments that reduce it. (KHE does not do this at present, and the argument just given explains why it is not a priority.) On the other hand, when constructing a time assignment node by node to begin with, it is easy to find previously assigned nodes with compatible sets of meets whose assignments can be re-used.

The author's current strategy for achieving node regularity, then, is as follows. When constructing the initial time assignment, give priority to node regularity, even ahead of minimizing demand defects. KHE offers a function that does this. Its algorithm is a descendant of the tiling algorithm published some years ago by this author (Kingston 2005), so it will not be described in detail here (the KHE User's Guide has a full description). Then repair the initial assignment, but restrict the repairs to operations which do not disrupt such node regularity as is already present.

The algorithm which constructs the initial time assignment begins by assigning the nodes of whatever student form seems likely to serve best as a template for assigning the others (usually the most senior form). It tries hard to find a very good assignment for this form; since no other forms have been timetabled yet, it should be possible to assign it with no defects at all. For each node of this first form, the set of times that its meets are running is called a *zone*. The algorithm stores these zones permanently in the cycle node (the common parent of the forms' nodes). When assigning subsequent forms, it tries to ensure that each node's meets are assigned entirely within one zone, as far as possible.

This approach is only effective if the chosen form attends classes at every time of the cycle, since if not, some times will receive no zone, or at any rate no guidance on which zone they should lie in. A more general approach, not implemented yet, would be to look through the instance and decide on a set of zones in advance. This is effectively what North American universities do when they define zones $\{Mon1, Wed1, Fri1\}$, $\{Mon2, Wed2, Fri2\}$, and so on.

5 Repair operations that preserve regularity

The picture of the data that a time repair algorithm has to work with is now complete: meets grouped into nodes and assigned to meets in parent nodes with zones. The assignment of a node is considered regular if it places all its meets into one zone. Deeper in the tree there will be nodes with no zones; they may be considered to have a single zone holding all their meets.

Several repair operations preserve existing node regularity within this structure.

Given two nodes which are children of the same parent node and which have meets with the same durations, the assignments of corresponding meets may be swapped. Each node will be as regular after the swap as the other was before it. This *node swap* repair operation will usually be neutral with respect to prefer times and spread defects, and it could well reduce demand defects. For example, if the Year 12 students attend both Mathematics and English for 6 times per week, in lessons of equal durations, then the times they attend Mathematics can be swapped with the times they attend English without reducing regularity.

The assignments of two meets of equal duration may be swapped when they are assigned to the same zone, whether or not they lie in the same node. For example, suppose some zone includes a double time containing the first two times on Wednesday, and that the Year 10 students attend History at the first of these times and Science at the second. Then these two events may be swapped without reducing regularity. This *meet swap* repair operation is also available, in a slightly different form, when the two meets' durations differ, provided their assignments are adjacent in time. When the meets are derived from the same event, a meet swap accomplishes nothing and would not be tried. Two meets may also be swapped if they are the only meets in their nodes, but in that case the operation is better classified as a node swap.

It is possible to go further when irregularity is already present. For example, if a meet lies in a different zone from all the others in its node, it can be moved to any zone without increasing irregularity. The algorithm presented in this paper does not yet check for such cases.

Node and meet swaps can be applied at any level of the layer tree. Here is one surprisingly high-level application. Make a new node with the same number of meets as the cycle node, and the same durations. Make the new node a child of the cycle node and assign its meets to the corresponding meets of the cycle node. Let the new node have the same zones as the cycle node, and delete the zones of the cycle node. Then move all the other child nodes of the cycle node so that they become children of the new node, and assign all their meets to the meets of the new node corresponding to their previous assignments. This reorganization does not change the timetable, nor its node regularity as measured by zones; it merely interposes a redundant node between the cycle node and its child nodes.

Apply meet swaps to the meets of the new node. The cycle node now has no zones, so these meets are free to swap with each other whenever they have the same durations or are adjacent in time. Since the cycle node has one meet for each set of consecutive times not spanning a break, one swap might swap the entire Wednesday morning timetable with the entire Thursday morning timetable, for example. Depending on how meet-regular the timetable is, it may also be possible to break these meets into smaller ones, and swap half-mornings and so on.

In the Australian instances which are this author's main focus, virtually all student group resources are preassigned to events whose total duration equals the number of times in the cycle. Under those circumstances, the only practical repair is the meet swap (or several meet swaps, as in the node swap) between meets containing the same preassigned student group resources. However, there are European instances in which students attend for fewer times, and even in Australian instances there are staff meetings which just need to occur whenever the teachers involved are most available. Thus, there is a need for a repair operation which moves one meet to a new time. This time should be one when the meet's preassigned resources are not busy—just the opposite of what is required when swapping.

Meet moves and swaps can be unified into a single well-known repair operation, here called the *Kempe meet move*. It starts by moving one meet, say from time t_1 to time t_2. If that causes clashes with other meets, those other meets are moved from t_2 to t_1. If that in turn causes clashes with other meets, the other meets are moved from t_1 to t_2, and so on for as long as new clashes appear. A Kempe meet move could fail for several reasons, but when it succeeds, it has moved the meet without increasing the overall number of clashes or the number of cases where a preassigned resource attends a meet at a time when it is unavailable.

All the meets moved are required to have the same duration, except in the special case where the second meet moved is adjacent to the first in time. In that case, all the meets moved on odd-numbered steps are required to have the same duration, all the meets moved on even-numbered steps are required to have the same duration, and each meet moves to the other end of the block of adjacent times during which the first two meets to be moved were originally running.

The ejection chain algorithm of this paper uses two repair operations: node swaps and Kempe meet moves. Kempe meet moves conveniently avoid the clumsiness of having one repair operation which swaps a meet to some times and another which moves it to the rest. Node regularity is preserved by allowing only repairs that do not increase the number of zones to which the meets of any affected node are assigned.

6 From defects to repairs

The repairs just defined may be applied in the traditional way to build neighbourhoods for local search which preserve node regularity. However, if they are to be used to repair defects, a path must be defined from each defect to a set of alternative repairs, each of which removes that defect.

Given a defect, the first step is to find the *contributing meets*: those meets whose assignments contribute to the defect and may be changed. For a prefer times defect, look through the meets derived from its event to find those assigned undesirable times. For a spread events defect, look through the monitored meets to find those for which a move to some other day would reduce the spread cost. For a demand defect, query the matching graph to find the demand nodes that are competing for the insufficient supply. (KHE offers an operation for this. For the Science laboratories example given some time ago, it would return all six Science laboratory demand nodes.) From each demand node, proceed to its task and from there to the task's meet.

For each meet identified by these steps, ascend its sequence of assignments to other meets. Any non-cycle meet on this sequence that lies in a node is a contributing meet: changing its assignment is permitted and might fix the defect.

For each contributing meet *m*, a Kempe meet move is possible to each legal offset in each meet of the parent node of *m*'s node (except *m*'s current meet and offset), provided the move does not change *m*'s zone. Ejection chains work best when repairs do actually remove the defects that provoked them, so each of these moves is only tried if it will do this: when repairing prefer times defects, only moves that will give the meet a preferred time are tried; when repairing spread events defects, only moves which will reduce the spread cost are tried; and when repairing demand defects, all moves are tried, since they all move *m* away from the insufficient supply.

KHE offers a *layer* data structure which groups together nodes with the same parent node and the same preassigned resources. After all Kempe meet moves of *m* have been tried without success, node swaps are tried between *m*'s node and the other nodes of its layer which have meets of the same durations as *m*'s node. Node swaps are not likely to be very effective at removing prefer times defects and spread events defects, since they merely shift these defects from one event to another, but they are potentially very valuable for removing demand defects.

7 Polymorphic ejection chains

The previous section showed how to identify the meets whose assignments contribute to a defect, and a set of alternative repairs that remove that defect. But it is very likely that removing one defect will create another. This is where ejection chains enter the picture: they chain repairs together, with each repair removing a defect created by the previous repair, until, with luck, a repair occurs which creates no new defects. This section introduces ejection chains and shows how to implement them polymorphically, that is, so that any number of types of defects, and any number of types of repairs, can be incorporated in a uniform way.

The heart of the ejection chain algorithm is a function that will be called `Augment`, since it is based on the well-known function for finding an augmenting path in bipartite matching. `Augment` targets one defect and tries a set of alternative repairs on it. Each repair removes the defect, but may create new defects. If no new defects of significant cost appear, `Augment` terminates successfully. If one significant new defect appears, `Augment` calls itself recursively in an attempt to remove that defect; in this way a chain of coordinated repairs is built up. If two or more significant new defects appear, `Augment` undoes the repair and continues with alternative repairs. It could try to remove all the new defects, but that would rarely succeed in practice.

The author's formulation of `Augment` has the following interface:

```
bool Augment(Defect d, Solution s, Cost c);
```

It has precondition

```
cost(s) >= c && cost(s) - cost(d) < c
```

where `cost(s)` is the current overall cost of solution `s`, and `cost(d)` is the contribution to this cost made by `d`, which is one of `s`'s defects.

If `Augment` can change `s` so as to reduce `cost(s)` to less than `c`, it does so and returns `true`; otherwise it leaves `s` unchanged and returns `false`. The second part of the precondition implies that removing `d` without adding any new defects, if that can be done, would achieve success. Here is an abstract implementation:

```
bool Augment(Defect d, Solution s, Cost c)
{
  repair_set = RepairsOf(d);
  for( each repair r in repair_set )
  {
    new_defect_set = Apply(s, r);
    if( cost(s) < c )
      return true;
    for( each e in new_defect_set )
      if( cost(s) - cost(e) < c && Augment(e, s, c) )
        return true;
    UnApply(s, r);
  }
  return false;
}
```

It begins by finding a set of repairs for `d`. For each of those, it applies the repair and receives the set of new defects introduced by that repair, checks for success, then if success has not been achieved it unapplies the repair and continues with the next repair, returning `false` when all repairs have been tried without success.

Success could come in two ways. Either one of the repairs reduces `cost(s)` to below `c`, or some new defect `e` has cost large enough to ensure that removing it alone would constitute success, and a recursive call targeted at `e` succeeds. Notice that `cost(s)` may grow without limit as the chain deepens, provided that there is a single defect `e` whose removal would reduce the cost of the solution to less than `c`.

When there are several defect types, several `Augment` algorithms are needed, one for each defect type, dynamically dispatched on the type. Ejection chains are naturally polymorphic: repairing a demand defect could create a spread defect, repairing that defect could create a prefer times defect, and so on. Repairs can usually be generated and applied directly, rather than being represented as a set of objects as above.

The tree searched by `Augment` as presented may easily grow to exponential size, which is not the intention. The author has tried two methods of limiting its size, both of which seem to be useful. They may be used separately or together.

The first method is to limit the depth of recursion to a fixed constant, perhaps 3 or 4. The maximum depth is passed as an extra parameter to `Augment`, and reduced by one on each recursive call, with value 0 preventing further recursion. Not only is this method attractive in itself, it also supports *iterative deepening*, in which `Augment` is called several times on the same defect, with the depth parameter increased each time. Another idea is to use a small depth limit on the first iteration of the main loop (see below), and increase it on later iterations.

The second method is the one used by the augmenting path method from bipartite matching. Just before each call on `Augment` from the main loop, the entire solution is marked unvisited (by incrementing a single global visit number, not by traversing the entire solution). When a repair changes some part of the solution, that part is marked visited. Repairs that change parts of the solution that are already marked visited are tabu. In this way, the size of the tree is limited to at most the size of the solution.

Given a solution and the set of all its defects, or a subset of its defects that it is expedient to target, the main loop cycles through the set repeatedly, calling `Augment` on each defect in turn, with `c` set to `cost(s)`. The set of defects and `cost(s)` change with each successful `Augment`. The main loop exits when `Augment` has been tried on every defect since the last successful call to `Augment`. At that point, no further successful augments are possible, assuming that `Augment` contains no randomness. This is a very clear-cut stopping criterion compared with, say, the stopping criteria used by metaheuristics. Under reasonable assumptions, it ensures that the whole algorithm runs in polynomial time, for the same reason that hill-climbing does.

8 Realizing ejection chains in KHE

This section sketches how the author's KHE platform supports ejection chains. Full details are available in KHE's documentation.

KHE has *monitor* objects, each of which monitors one point of application of one constraint, or one demand node of one matching graph. Each monitor contains a cost, which KHE keeps up to date as the solution changes. For example, for each set of meets which are required to spread evenly through the cycle there is a monitor. This monitor is notified whenever the starting time of any of its meets changes, triggering it to update its cost and notify any change.

Group monitors may be created to monitor other monitors. These other monitors notify their group monitor of any change in cost, rather than notifying the solution directly. The cost of a group monitor is the total cost of the monitors it monitors, so when any of its monitors' costs change, the group monitor's cost does too, and it must notify its own group monitor, and so on.

Group monitors are useful when several nominally distinct monitors monitor the same thing in reality. Take the example of several events required by a link events constraint to run simultaneously. These are usually handled by a preprocessing step which links their meets together so that only simultaneous assignment is possible thereafter. Each event may have its own spread events monitor, but these all monitor the same thing in reality and are best treated as a single monitor, which is done by grouping them. It is the group monitor that appears on lists of monitors, with the monitors it groups hidden from view below it.

The object representing the solution as a whole is (among other things) a group monitor. Provided each monitor reports its cost to this special group monitor, either directly or via intermediate group monitors, this special group monitor will hold the current total cost of all non-group monitors, which is the cost of the solution.

A defect (of type `Defect` in the algorithm above) is realized in KHE as a monitor, often a group monitor, of non-zero cost. For each type of defect (or group of related defects), the user has to write an implementation of `Augment` specialized to that type of defect, and register it with an *ejector* object defined by KHE, which also holds other useful information, such as the desired method of limiting depth. Each of these functions iterates over a set of repairs of the user's choice, applying and unapplying each repair in turn, and calling a function supplied by KHE which handles the testing for success and the recursive call, dynamically dispatching it to one of the functions registered with the ejector. KHE also supplies an implementation of the main loop. So the user only has to implement the code that converts a defect into a set of repairs and applies and unapplies each repair in turn; the rest comes for free.

Each group monitor makes available the set of its child monitors whose cost is non-zero. Keeping this set up to date requires a small constant amount of work each time a monitor reports a change in its cost from zero to non-zero or vice versa. The set of defects iterated over by the main loop of the ejection chain algorithm is just this set of monitors, for some particular group monitor given to the ejector object by the user. This group monitor could be the special solution group monitor, in which case every defect in the solution is open to repair, or it could be a group monitor whose children are just some of the monitors, in which case only defects among those children are open to repair. For example, the time repair ejector object is given a group monitor whose children are those monitors concerned with time assignment.

While a repair is being applied, KHE's *tracing* feature records which children of the ejector's group monitor changed in cost during the repair, and by how much. The subset of these monitors whose cost increased is the new defect set used by `Augment`. KHE also offers a *transactions* feature which allows the operations carried out since some starting point to be recorded, and subsequently undone and redone. This makes it easy to undo a complex repair, such as a Kempe meet move, as required by the `UnApply` step of `Augment`. Transactions are also used by a variant of the algorithm which tries all ejection chains, remembers the best, and redoes it at the end.

9 Experiments

This paper has been concerned with explaining how ejection chains can be applied to real-world timetabling problems, in particular to repairing time assignments while preserving regularity. It is not empirical in orientation. Accordingly, the experiments of this section have very modest aims: to show that the algorithm for repairing time assignments produces a reasonable result in a reasonable time, and to shed light on some design choices, without claiming to be definitive.

The algorithm tested here is KHE's standard ejection chain solver, configured to use a simple form of iterative deepening (Sect. 7), and loaded with augment functions that make Kempe meet moves and node swaps after carrying out the mapping from defects to repairs described in Sect. 6.

In the XML format, each constraint has an integer weight which is multiplied by the number of defects to produce a cost. Each constraint also has a Boolean *required* attribute. If this attribute is true, the constraint is 'hard' and its cost is added to a total called the *infeasibility value* $i(s)$ of the solution s. If the attribute is false, the constraint is 'soft' and its cost is added to a different total called the *objective value* $o(s)$ of the solution. A solver aims to minimize the ordered pair $(i(s), o(s))$. The experiments reported here are confined to one difficult real-world instance (BGHS98 from the XHST2011 archive (Post 2011)). It has no prefer times constraints, and its spread constraints are soft with weight 1. Demand defects are treated as hard with weight 1. So, in these experiments, the hard cost is the number of demand defects, and the soft cost is the number of spread defects.

A *student form*, or just *form*, is a set of resources, each representing one group of students from the same age cohort. The construction algorithm assigns times to the meets of one form at a time. The first question, then, is whether repair is needed at all, and if so, whether it is needed after assigning each form. Repair is certainly needed after assigning the first form: it is important to assign that form as well as possible, because its assignments guide the assignments of the other forms. Accordingly, two runs were performed, the first repairing after the first and last forms only, taking 10.0 seconds (all times include both construction and repair), and the second after each form, taking 16.1 seconds.

The results appear in Fig. 1. Repair after each form is superior, producing 17 fewer demand defects in the end, and fewer still at intermediate points. This is not surprising: more repairs succeed when the timetable is only partly assigned and resource demands are not pressing.

The difference between the final number of demand defects when only the first form is repaired (104) and when every form is repaired (64) is important, because

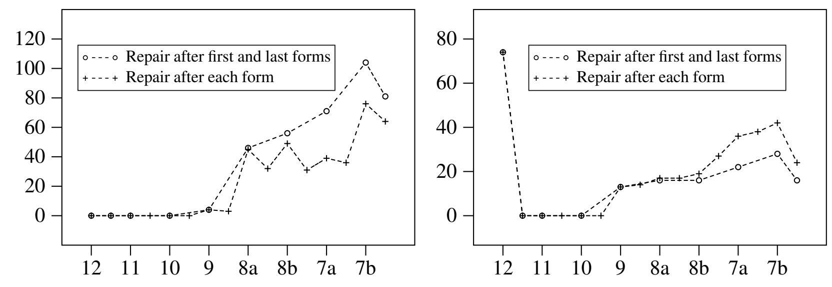

Fig. 1 Effectiveness of repair. The vertical axis represents hard cost in the first graph, soft cost in the second. The horizontal axes have one label for each student form, in order of assignment. For each form there is one data point representing the cost after constructing the time assignment of that form, possibly followed by a second point representing the cost after repair. Some repairs increase soft cost, because decreasing hard cost takes priority. The first four forms (12, 11, 10, and 9) are complete forms, the next four (8a, 8b, 7a, and 7b) are half-forms. Other 'forms' containing staff meetings follow these forms, but they add virtually no cost so have been omitted.

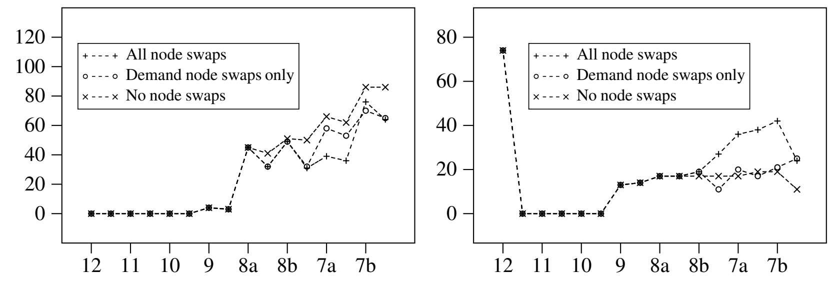

Fig. 2 Effectiveness of node swaps. Details as in Fig. 1.

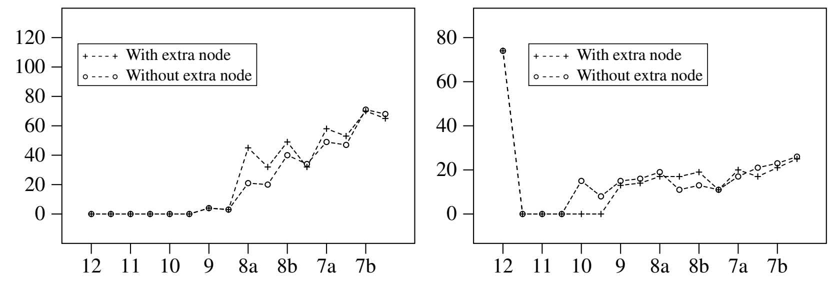

Fig. 3 Effectiveness of an extra node under the cycle node. Details as in Fig. 1.

repairs that sacrifice regularity will need to be applied to these remaining defects. After the runs reported here, the author's solver removes all zones and interior nodes, thereby removing all requirements for regularity, and runs the repair algorithm again. Demand defects then drop dramatically, typically to just 5 or 6, at the cost of some loss of regularity that shows up in split resource assignments later.

The author's long-term goal is to solve high school instances reliably in about 10 seconds, including resource assignment, which follows after time assignment and takes several seconds. This raises the question (also interesting for its own sake) of whether there are unproductive parts of the algorithm that could be removed, saving time without degrading performance.

Node swaps are one possibility. As explained previously, they seem likely to be effective at removing demand defects, but unlikely to be effective at removing spread defects. This is investigated by continuing with repair after assigning each form, but now trying it with all node swaps (taking 16.1 seconds as before), node swaps when repairing demand defects only (12.9 seconds), and no node swaps (23.3 seconds). The apparent contradiction in the last run time, of doing less work but taking longer, is quite common with ejection chains. It is a sign that node swaps are removing defects effectively, so that the algorithm struggles without them.

The results appear in Fig. 2. Omitting all node swaps is inferior, producing significantly more demand defects from form 8a onwards. There is some support for using node swaps on demand defects only (the end results, at least, are indistinguishable), and it does save time (3.2 seconds).

Another possibility for reducing run time is to omit the extra node directly under the cycle node. Its presence more than doubles the number of Kempe meet moves available to most repairs. This is investigated by repairing after each form, with node swaps for demand defects only, but now trying with this extra node (taking 12.9 seconds as before) and without it (8.0 seconds).

The results appear in Fig. 3. The extra node leads to 3 fewer demand defects in the end, but may not be worth its cost in run time (4.9 seconds). More data are needed.

10 Conclusion

This paper introduces polymorphic ejection chains and shows that they are effective for repairing time assignments in high school timetables while preserving regularity. Unlike local search, polymorphic ejection chains target specific defects and build sets of coordinated repairs. Based on the results in this paper and other papers cited earlier, it seems likely that they would be effective in improving solutions to many real-world combinatorial optimization problems.

There is little more to do in the area of time repairs that preserve regularity, but other applications beckon. Within high school timetabling, removing resource defects such as split assignments and unwanted gaps in teachers' timetables requires complex repairs that reassign times and resources together. That promises to be an even richer area of application for polymorphic ejection chains than the one explored here.

References

R. Ahuja, Ö. Ergun, J. Orlin, and A. Punnen, A survey of very large-scale neighbourhood search techniques, Discrete Applied Mathematics, 123, 75–102 (2002)

Eyjólfur Ingi Ásgeirsson, Bridging the gap between self schedules and feasible schedules in staff scheduling, PATAT10 (Eighth international conference on the Practice and Theory of Automated Timetabling, Belfast, August 2010)

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)

Kathryn A. Dowsland, Nurse scheduling with tabu search and strategic oscillation, European Journal of Operational Research, 106, 393–407 (1998)

Fred Glover, Ejection chains, reference structures and alternating path methods for traveling salesman problems, Discrete Applied Mathematics, 65, 223–253 (1996)

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, 182–186 (1997)

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)

Jeffrey H. Kingston, Hierarchical timetable construction, Practice and Theory of Automated Timetabling VI, Springer Lecture Notes in Computer Science 3867, 294–307 (2007)

Jeffrey H. Kingston Resource assignment in high school timetabling, PATAT08 (Seventh international conference on the Practice and Theory of Automated Timetabling, Montreal, August 2008)

Jeffrey H. Kingston, The KHE High School Timetabling Engine, http://www.it.usyd.edu.au/˜jeff/khe (2010)

Jeffrey H. Kingston, The HSEval High School Timetable Evaluator, http://www.it.usyd.edu.au/˜jeff/hseval.cgi (2010)

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)

Andrew J. Parkes, Combined Blackbox and AlgebRaic Architecture (CBRA), PATAT2010 (Eighth international conference on the Practice and Theory of Automated Timetabling, Belfast, August 2010)

Gerhard Post, Samad Ahmadi, Sophia Daskalaki, Jeffrey H. Kingston, Jari Kyngäs, Cimmo Nurmi, and David Ranson, An XML format for benchmarks in high school timetabling, Annals of Operations Research 194, 385–397, 2012

Gerhard Post, Jeffrey H. Kingston, Samad Ahmadi, Sophia Daskalaki, Christos Gogos, Jari Kyngäs, Cimmo Nurmi, Haroldo Santos, Ben Rorije and Andrea Schaerf, An XML format for benchmarks in high school timetabling II, Annals of Operations Research (online), http://10.1007/s10479-011-1012-2, 2011

Gerhard Post, High school timetabling web site, http://wwwhome.math.utwente.nl/˜postgf (2011)

David M. Ryan and Natalia J. Rezanova, The train driver recovery problem – solution method and decision support system framework, PATAT2010 (Eighth international conference on the Practice and Theory of Automated Timetabling, Belfast, August 2010)