

# Mergeable Schedules for Lazy Matching

*Leslie De Koninck*

*Report CW 505, December 2007*



Katholieke Universiteit Leuven  
Department of Computer Science  
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

# Mergeable Schedules for Lazy Matching

*Leslie De Koninck*

*Report CW505, December 2007*

Department of Computer Science, K.U.Leuven

## **Abstract**

This paper presents a new data structure, based on circular linked lists and the union-find algorithm, for the purpose of incremental, lazy pattern matching for rule based languages, with storage of partial matches. Our approach consists of incrementally generating a schedule of matches. It extends previous work in that it allows merging of such schedules without increasing the overall complexity. Schedule merges are needed in the context of non-ground data. Therefore, our mergeable schedules can be used to implement matching for the Constraint Handling Rules (CHR) language. Our technique forms an alternative to the matching algorithm that is most commonly used by current CHR implementations and which does not keep track of partial matches, but does keep a history of rule firings.

**Keywords :** Schedules, lazy pattern matching, rule based languages.

**CR Subject Classification :** E.1 [Data Structures], F.2.2 [Analysis of Algorithms and Problem Complexity] Nonnumerical Algorithms and Problems — Pattern matching, F.2.2 [Analysis of Algorithms and Problem Complexity] Nonnumerical Algorithms and Problems — Sequencing and scheduling

# Mergeable Schedules for Lazy Matching

Leslie De Koninck\*

Department of Computer Science, K.U.Leuven, Belgium

*FirstName.LastName@cs.kuleuven.be*

**Abstract** This paper presents a new data structure, based on circular linked lists and the union-find algorithm, for the purpose of incremental, lazy pattern matching for rule based languages, with storage of partial matches. Our approach consists of incrementally generating a schedule of matches. It extends previous work in that it allows merging of such schedules without increasing the overall complexity. Schedule merges are needed in the context of non-ground data. Therefore, our mergeable schedules can be used to implement matching for the Constraint Handling Rules (CHR) language. Our technique forms an alternative to the matching algorithm that is most commonly used by current CHR implementations and which does not keep track of partial matches, but does keep a history of rule firings.

## 1 Introduction

In this paper, we propose a new data structure and algorithm for the following task. Consider a set of  $n$  jobs  $J = \{J_1, \dots, J_n\}$ , and a set of  $m$  machines  $M = \{M_1, \dots, M_m\}$ . The task at hand is to maintain a schedule  $\langle J, M \rangle$  that generates in a demand driven way (i.e., lazily), a series of matches of the form  $J_i - M_j$  such that each such match is generated exactly once for each  $i \in [1..n]$  and  $j \in [1..m]$ . In the above description, a match  $J_i - M_j$  represents the fact that job  $J_i$  is executed on machine  $M_j$ . The schedule should support the following operations: adding a new job or a new machine, removing a job or a machine, and merging the schedule with another *disjoint* schedule. Two schedules are called disjoint if both their sets of jobs and their sets of machines are disjoint. After merging disjoint schedules  $\langle J, M \rangle$  and  $\langle J', M' \rangle$ , the resulting schedule should (eventually) generate all matches of jobs from  $J \cup J'$  with machines from  $M \cup M'$  such that no match is generated more than once before or after the schedule merge.

**Pattern matching in rule-based languages** This work is situated in the context of pattern matching in a rule-based language, in particular Constraint Handling Rules (CHR) [5]. Pattern matching for multi-headed rules (like the ones in CHR) has traditionally been studied in the context of production rule systems. Notable developments in this area are the RETE algorithm by Forgy [4], and

---

\* Research funded by a Ph.D. grant of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen)

the TREAT [7] and LEAPS [8] algorithms, both by Miranker. The former two algorithms implement eager (*data driven*) matching, while the latter implements lazy (*demand driven*) matching. The difference is that in eager matching, all matches are generated and then one is chosen, whereas in lazy matching, matches are generated incrementally and on demand, one at a time.

The RETE algorithm keeps track of partial rule matches in so-called *beta memories*. Whenever a new element is inserted, it is used to extend already found partial matches into either larger partial matches or full matches. If rules cannot have negated heads (which is the case in CHR), then an element deletion is handled by removing those partial and full matches in which the deleted element participates. After all added and removed elements have been processed, a full match is selected and the corresponding rule instance is fired.

The TREAT algorithm works in a similar fashion, except that it does not store partial matches. Instead, full matches are generated by iteratively adding heads starting from a newly inserted element, which is often called the *dominant object* in this role. Deletion then only requires removing those *full* matches in which the deleted element occurs. Finally, LEAPS is like TREAT, but stops generating matches as soon as a full match is found, after which the corresponding rule instance is fired immediately. Afterwards, the matching process continues if the dominant object has not been deleted by then. In LEAPS, no partial or full matches are stored. The main advantage of LEAPS, and to a lesser extent also of TREAT, are the lower memory requirements by not having to store partial and/or full matches. However, the laziness of LEAPS in itself has the extra advantage that no computation is wasted on producing partial and full matches that are never used because some of the constituent facts are removed before we get to the point of using them. This suggests that also a lazy version of RETE (i.e., lazy matching *with* storage of partial matches that are computed on the way) would make sense. Still, in general, a lazy version of RETE matching would consume considerably more memory than LEAPS matching.

**Matching in Constraint Handling Rules** In the Constraint Handling Rules language, rules need to be matched with a multi-set of *CHR constraints*, which more or less correspond to the elements in production rule systems. An interesting aspect of CHR is that it supports non-ground CHR constraints, i.e., constraints whose arguments contain variables. The variables represent a form of partial information that can become more precise in a later state by means of updates through so-called built-in constraints that constrain (or instantiate) the variables further. Such updates could be implemented by first removing the affected CHR constraints, and then reasserting the updated versions of them.

Most current CHR implementations use a variant of the LEAPS algorithm to perform the rule matching task. Instead of actually removing and reasserting facts that are affected by a built-in constraint, these facts are ‘reactivated’ after the update, i.e., they become dominant objects again. Under the condition that each rule instance that was applicable before an update, remains applicable after the update, this more or less has the same effect as a removal followed by a

reinsertion. However, the operational semantics of CHR requires that each rule can fire only once for the same combination of constraints.

To avoid that a reactivated constraint fires a rule instance that has fired before, a so-called *propagation history* is maintained, which contains tuples representing the constraints that have fired a given rule. In the worst case, this propagation history takes space proportional to the maximal number fireable rule instances at any time.<sup>1</sup> Also note that if a rule instance is found that has fired before, then the matching work leading to this event has been redundant.

So in the context of CHR, a lazy version of the RETE algorithm may be interesting because it is no longer necessarily suboptimal memorywise, but moreover may be able to avoid redundant work when dealing with reactivated constraints. In absence of built-in constraints and reactivations, such a lazy RETE matching can be implemented by using a scheduling data structure consisting of a stack of active constraints (dominant objects), each of which contains a pointer into a list of partial matches. Such a pointer indicates which partial matches have already been matched with the active constraint in question, and which partial matches still need to match with it. We can think of partial matches as jobs and active constraints as machines using the terminology used above to describe our scheduling problem. We assume here that for each active constraint, we have a way to select those partial matches which it extends. In the ground context, this selection can be done using a hash table lookup.

In general, we have a set of partial matches and a set of active constraints such that each partial match in the first set, is to be extended with each constraint in the second set. These sets map on the set of jobs and the set of machines from our initial description of the problem. Now, a built-in constraint may require merging sets of partial matches and sets of active constraints. It is here that mergeable schedules are needed. While in absence of merging, the implementation of a data structure for our scheduling problem is relatively straightforward, schedule merges complicate this implementation considerably. The main purpose of this paper is to show how mergeable schedules can be implemented efficiently.

**Overview** The rest of this paper is organized as follows. In Section 2, we first show how to implement schedules in absence of schedule merges, and then extend this approach towards mergeable schedules, leading to a first, albeit suboptimal implementation. Next, in Section 3, an optimized implementation is given, which is amongst others based on an extension of the optimal union-find algorithm. The correctness of this optimized implementation is established in Section 4, and its complexity is analyzed in Section 5. We briefly look at matching in Constraint Handling Rules as application area in Section 6 and conclude in Section 7.

## 2 Towards Mergeable Schedules

In this section, we give a first, though not yet optimal, version of our scheduling algorithm. In Section 2.1, we deal with the case of non-mergeable schedules as a

---

<sup>1</sup> If the propagation history is not cleared of invalid tuples, it may even become larger.

first step, in the context of the job shop problem described in the introduction. Next, in Section 2.2, we show how the approach for the non-mergeable case can be extended to support schedule merges. Finally, Section 2.3 gives a first, but suboptimal, implementation of mergeable schedules, and illustrates some of the problems we will have to deal with in an optimal implementation.

## 2.1 A Job Shop Problem

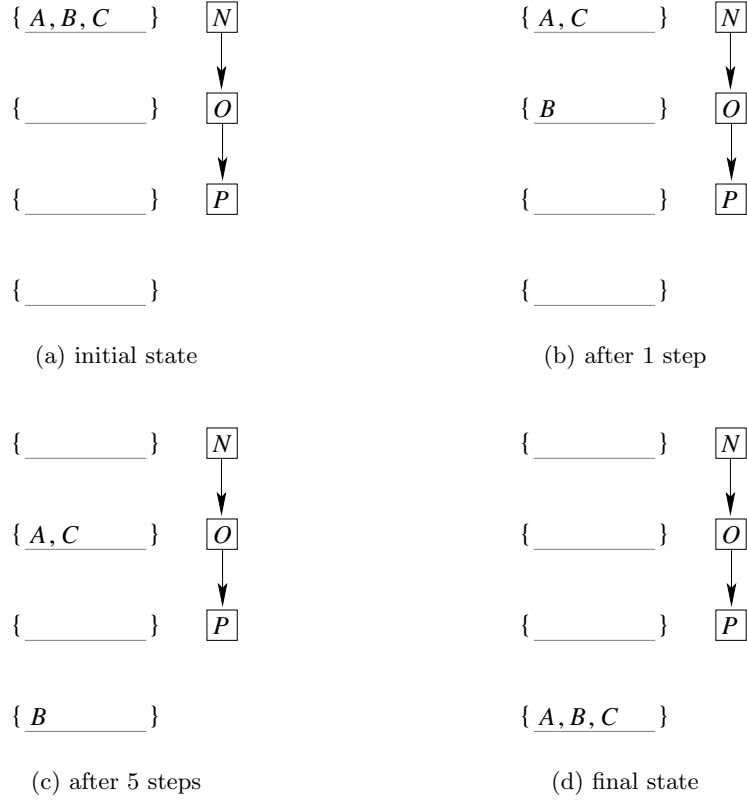
In the introduction, we already briefly described the problem of scheduling a set of jobs and a set of machines such that each job is executed on each machine exactly once. This problem resembles the classical job shop scheduling problem, which consists of finding an optimal schedule, given a cost function that assigns a score to each schedule. In our context, we do not care about optimality of the schedule: any schedule will do. However, we do require that the schedule is built incrementally, assigning one job to one machine in each step. Moreover, new jobs and machines can be added during this process, while other jobs and machines may be removed. Finally, two *job shops* may be merged, leading to many new combinations of jobs and machines. In this subsection, we first demonstrate how this problem can be tackled in absence of schedule merges, and then show how schedule merges complicate the task at hand.

A job shop consists of a set of jobs  $J = \{J_1, \dots, J_n\}$  and a set of machines  $M = \{M_1, \dots, M_m\}$ . In the standard job shop problem, each job  $J_i$  ( $1 \leq i \leq n$ ) is to be matched with each machine  $M_j$  ( $1 \leq j \leq m$ ) exactly once. Since we allow deletion of jobs and machines, some of these jobs and machines may have been deleted before all of their matches are generated. Therefore, we relax our requirement as follows: each job is to be matched with each machine *at most* once, and all jobs and machines that exist in a *final* state, are matched with each other *exactly* once. A final state is a state in which no more matches can be generated. Figure 1 depicts a data structure to manage the generation of these matches. Given  $n$  jobs and  $m$  machines, the data structure consists of  $m + 1$  sets (e.g., represented by lists) on the left hand side, in which the  $n$  jobs are partitioned, and a list of length  $m$  on the right hand side, containing the  $m$  machines. Jobs and machines can be switched here of course. The following semantics is used: the jobs in the  $i^{th}$  left hand side set have already been matched with the first  $i - 1$  machines in the right hand side list, and still have to be matched with the remaining machines in that list.

Initially, all jobs are in the first set. This is shown in Figure 1(a).<sup>2</sup> Now at each subsequent step, a job from the  $i^{th}$  left hand side set (for some  $i \in \{1, \dots, m\}$ ) is selected to be matched with the  $i^{th}$  machine in the right hand side list, and is then moved to the  $i + 1^{th}$  set. For example, we can choose job  $B$  from the  $1^{st}$  set, to be matched with machine  $N$ , after which it moves to the  $2^{nd}$  set. This is depicted in Figure 1(b). Figure 1(c) shows a possible state after 5 steps. In this state, jobs  $A$  and  $C$  have already been matched with machine  $N$ , and job  $B$  with machines  $N$ ,  $O$  and  $P$ . Note that the data structure is quite non-deterministic:

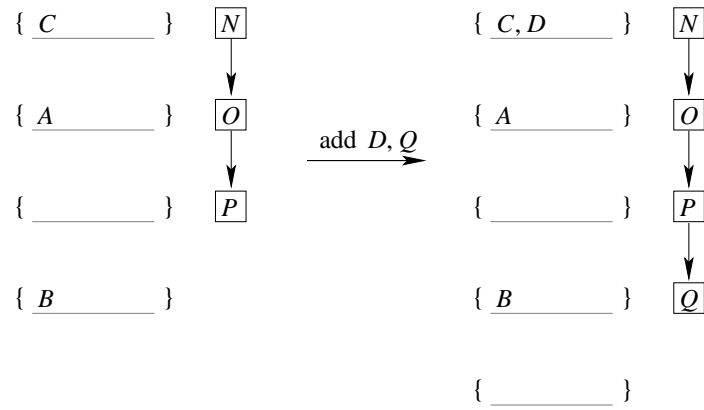
<sup>2</sup> In the following, we use the letters  $A, \dots, M$  for jobs and  $N, \dots, Z$  for machines.

we can choose any job from any non-empty set. Eventually, all jobs end up in the  $m + 1^{th}$  left hand side set. This is depicted in Figure 1(d). When a new

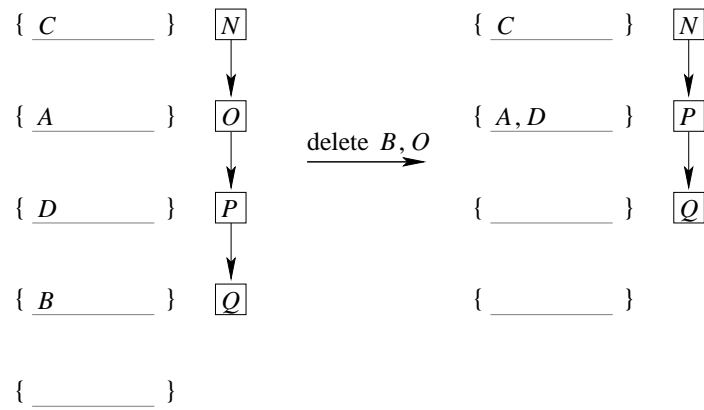


**Figure1.** A data structure for non-mergeable schedules

job is added, it is inserted into the first of the left hand side sets. If a new machine is added, it is appended at the end of the right hand side list, and a new set is created and appended at the end of the left hand side sets. Figure 2 depicts the (simultaneous) addition of job  $D$  and machine  $Q$ . Note that all jobs in the previously final left hand side set, now are scheduled to be matched with the newly added machine. When a job is to be removed from the job shop, it is removed from whichever left hand side set it was in. When a machine is to be removed, it is removed from the right hand side list. Let  $i$  be the removed machine's position in the right hand side list before the removal, then the  $i^{th}$  and  $i + 1^{th}$  left hand side sets are merged. Figure 3 shows the (simultaneous) removal of job  $B$  and machine  $O$ .



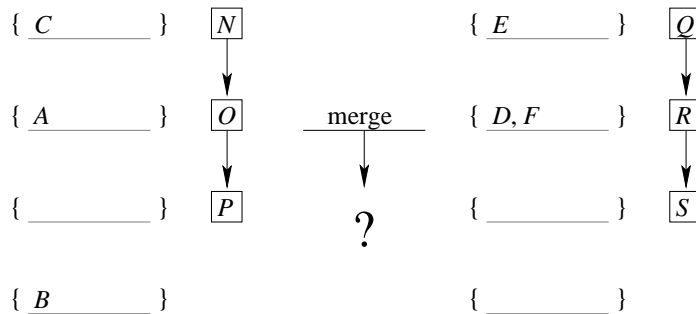
**Figure2.** Adding a new job  $D$  and machine  $Q$



**Figure3.** Removing job  $B$  and machine  $O$

The data structure described so far is similar to what is used in [6] for finding rule instances in a bottom-up logic programming language. In a more generalized setting, we refer to the left hand side items (which are stored in sets) as the *elements* and to the right hand side items (which together form a (linked) list) as the *nodes*. We return to our job shop one last time to add one more operation, which complicates matters considerably.

Let there be other, similarly organized job shops. Now assume that at some point, two job shops are to be merged. In general, either of them could be in a non-final state (i.e., a state in which there exist at least one job that has not been matched yet with at least one machine). Figure 4 shows a possible state of the schedules for both job shops at the point of merging. In this case, job *A* still has to be matched with machines *O*, *P*, *Q*, *R* and *S*; *B* with *Q*, *R* and *S*; *C* with *N*, *O*, *P*, *Q*, *R* and *S*; *D* and *F* with *R*, *S*, *N*, *O* and *P*; and *E* with *Q*, *R*, *S*, *N*, *O* and *P*. One way to deal with this situation, is to keep the two data structures and append respectively *Q*, *R* and *S*, and *N*, *O* and *P*, to the right hand side list of the first and second data structure. This implies that from that point on, we have to update two data structures for each subsequent insertion and removal of a machine. Meanwhile, new merges may take place, increasing the amount of work to an unacceptable level.



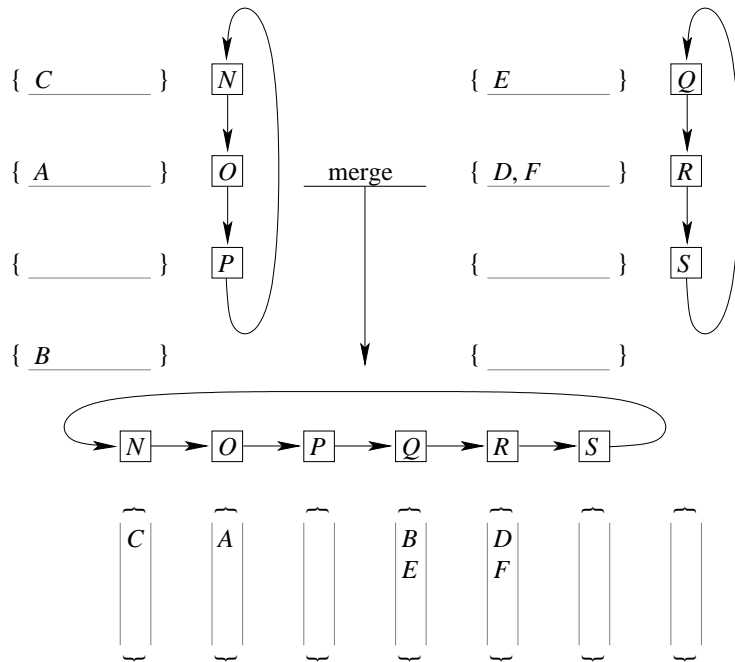
**Figure4.** Merging two job shops

## 2.2 Mergeable Schedules

This subsection describes a new data structure to handle our scheduling problem in the context of schedule merges. In the previous subsection, we observed that after a merge operation, the elements of one schedule have to match with the nodes of another and vice versa. This observation inspired the use of a circularly linked list to represent the nodes. In its simplest form, it works as follows. Every element contains a pointer to the node corresponding to the first set in which it is scheduled. We refer to this node as the *start* node. In any state, the node corresponding to the set an element belongs to in that state, is called the element's

*current* node for that state. When an element matches with a node and moves on to the next one, it is checked whether this next node equals the start node. If so, the element has been matched with all nodes and is made passive.

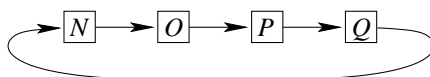
Now, when a merge takes place, we link both schedules as is shown in Figure 5. For now, we assume all elements in a given schedule have the same start node. After merging this is obviously not true anymore. In the figure, element *B* was



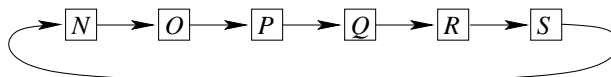
**Figure5.** Linking schedules

passive before the merge. After the merge, it is joined with *E* to be matched with node *J*. While *B* will become passive after having matched with *L*, *E* will continue until node *I*. The fact that after a merge, elements may have different start nodes, becomes a problem when new nodes are added. Say a new node *M* is added in between *L* and *G*. In that case, if element *E* has passed node *L*, it will never see *M* because it will become passive after matching with *I*. The problem here is that new nodes are inserted before a node that is different from the start node. We call this node the *insertion node*. There is exactly one insertion node per schedule. When an element's current node becomes equal to its schedule's insertion node, either because the current node is deleted, or because the element is *activated* (i.e., it is selected to match with its current node) and the next node is the insertion node, it is said to have passed its schedule's insertion point.

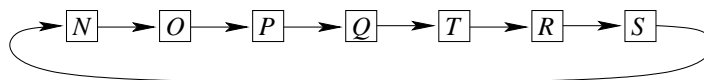
To solve the problem of nodes that were missed by some element  $e$  because they were added after  $e$  passed its insertion point, we store the last node seen before passing the insertion point, as well as the insertion node at that time. These nodes are called respectively the *last* node and the *first* node. If any new nodes are added after passing the insertion point, they must appear in between these two nodes. Note that in some later state, the insertion node may change. However, it can only change to a node that was previously not part of the schedule, and hence it must also appear in between the last and first nodes.



(a) Initially;  $A$  is the insertion node



(b) After merge; insertion node is changed to  $E$



(c) After inserting  $G$  right before  $E$

**Figure6.** Merge and Insertion

Figure 6 shows the nodes of a schedule (a) initially, (b) after merging with the nodes  $R$  and  $S$ , and (c) after inserting node  $T$ . Consider an element  $A$  whose start node is  $P$  (e.g., resulting from a merge of nodes  $N$  and  $O$  with nodes  $P$  and  $Q$ ) and current node is  $N$ . Let  $N$  be the insertion node at this point. Since  $A$  has passed its insertion node, it records this fact by storing both  $N$  and the node right before  $N$ , which is  $Q$  in the schedule of Figure 6(a). Now, when  $A$  is about to match its start node again, instead of being made passive, we *jump back* to the node right after  $Q$  and check whether this node still equals  $N$ . If so, no new nodes have been added since passing  $N$  and  $A$  can be made passive. In the schedule of Figure 6(b), nodes  $R$  and  $S$  have been added in between  $Q$  and  $N$ . In this case, the jump back changes the current node of  $A$  to  $R$  and changes its start node to  $N$ . Assume that  $R$  is the schedule's new insertion node, then because the insertion point is different from the start node (and hence all nodes

inserted before it will not be seen before reaching the start node), we again store the last node seen before  $R$ , namely  $Q$ . Finally, after reaching the (new) start node  $N$ , we again jump back, now to the node right after  $Q$ . In the schedule of Figure 6(c), this is node  $T$ . We change the start node to  $R$  which is still the insertion node.

### 2.3 Basic Algorithm

This subsection presents a simplified version of the algorithm. The algorithm as it is presented here is far from optimal, but we give it because on the one hand it is still relatively simple, while on the other hand, it already shows some of the complications we have to deal with. A considerably more complicated, but quasi optimal algorithm is given in Section 3.

**Representation** A schedule  $s$  is a structure with fields `insert`, `scheduled_set` and `passive_set`. The first field points to the insertion node of the schedule. The second and third fields contain respectively the set of all scheduled elements and the set of all passive elements (i.e., those that have already been matched with each node in the schedule). A node is a structure with two fields: `prev` and `next`, which point to respectively the previous and next node in the circular list of nodes. Finally, an element is a structure with four fields `start`, `current`, `first` and `last` which point to the corresponding fields cfr. previous subsection.

**Operations** The schedule supports the following operations: inserting and deleting a node, inserting and deleting an element, activating an element, and merging two schedules. In the remainder of this section, we show how each of these operations can be implemented in a correct albeit suboptimal way. For each of the operations, a listing in pseudo code is given.

**Node Insertion** A new node  $n$  is inserted right before the insertion node of its schedule  $s$ , if such a node exists. Otherwise, the new node becomes the schedule's insertion node. Any element that was passive in  $s$ , is rescheduled by making its start node equal to the insertion node, and its current node equal to the newly added node.

```

1  procedure insert_node(Node n, Schedule s)
2  {
3      if (s.insert = null) // no nodes in schedule
4      {
5          // initialize prev and next pointers
6          n.prev := n.next := n;
7          // set insertion node
8          s.insert := n;
9      }
10     else
11     {
```

```

12         // insert new node right before insertion node
13         n.prev := s.insert.prev;
14         n.next := s.insert;
15         n.prev.next := s.insert.prev := n;
16     }
17     // reschedule passive elements
18     foreach e in s.passive_set
19     {
20         e.start := e.current := n;
21         e.first := e.last := null;
22     }
23     s.scheduled_set := set_union(s.scheduled_set,s.passive_set);
24     s.passive_set := empty_set;
25 }

```

**Node Deletion** If the *only* node of a schedule is deleted, then all the elements of the schedule become passive. Otherwise, if a node  $d$  is deleted, all elements whose current node, first node or start node equals the deleted node, have their affected fields set to point to the node *after* the deleted one. All elements whose last node equals the deleted node, have their last field set to point to the node *before* the deleted one. If an element's current node is deleted and becomes equal to its start node, then a conditional jump back is performed (see also **Element Activation**). If the deletion causes some element to pass its schedule's insertion point, the element's last and first node are set to respectively the node *before* the deleted one, and the node *after* the deleted one. Finally, if an element's start and last node are equal and are subsequently deleted, then this element's start node is changed to its first node, and its first and last fields are reset.

```

26 procedure delete_node(Node d, Schedule s)
27 {
28     if (d.next = d)
29     {
30         // last node in schedule: make all elements passive
31         s.passive_set := set_union(s.passive_set,s.scheduled_set);
32         s.scheduled_set := empty_set;
33         // reset insertion node
34         s.insert := null;
35     }
36     else
37     {
38         foreach e in s.scheduled_set
39         {
40             if (d = e.current and d.next = e.start)
41             {
42                 // conditional jump back
43                 if (e.last.next = e.first)
44                 {
45                     // no unseen nodes: make e passive

```

```

46         add e to s.passive_set;
47         remove e from s.scheduled_set;
48     }
49     else
50     {
51         // jump back
52         e.start := e.first;
53         e.current := e.last.next;
54         if (e.last.next = s.insert) e.first := s.insert;
55         else e.first := e.last := null;
56     }
57 }
58 else if (d = e.current and d.next = s.insert)
59 {
60     // e passes insertion point
61     e.current := e.first := d.next;
62     e.last := d.prev;
63 }
64 else
65 {
66     if (d = e.current) e.current := d.next;
67     if (d = e.first) e.first := d.next;
68     if (d = e.start = e.last)
69     {
70         // all nodes between start and last are deleted
71         e.start := e.first;
72         e.first := e.last := null;
73     }
74     else
75     {
76         if (d = e.start) e.start := d.next;
77         if (d = e.last) e.last := d.prev;
78     }
79 }
80 }
81 // update links of circular list of nodes
82 d.prev.next := d.next;
83 d.next.prev := d.prev;
84 }
85 }

```

**Merging Schedules** When two schedules  $s_1$  and  $s_2$  are merged, the circular lists of nodes of both schedules are crosslinked at their respective insertion points. Elements that were passive in schedule  $s_1$  are rescheduled by setting their start node equal to  $s_1$ 's insertion node, and their current node equal to  $s_2$ 's insertion node. Similarly, the elements that were passive in schedule  $s_2$  are rescheduled by setting their start node equal to  $s_2$ 's insertion node, and their current node equal to  $s_1$ 's insertion node. The resulting schedule  $s$  gets the same insertion node as the input schedule  $s_1$ . Therefore, to avoid

that the rescheduled elements from schedule  $s_2$  are not matched with nodes inserted after the schedule merge, these elements have their first and last fields appropriately instantiated.

```

86 function merge_schedules(Schedule s1, Schedule s2) = Schedule
87 {
88     // crosslink circular lists
89     p1 := s1.insert.prev;
90     p2 := s2.insert.prev;
91     s1.insert.prev := p2;
92     p2.next := s1.insert;
93     s2.insert.prev := p1;
94     p1.next := s2.insert;
95     // reschedule passive elements of schedule s1
96     foreach e in s1.passive_set
97     {
98         e.start := s1.insert;
99         e.current := s2.insert;
100        e.first := e.last := null;
101    }
102    s2.scheduled_set := set_union(s2.scheduled_set,s1.passive_set);
103    // reschedule passive elements of schedule s2
104    foreach e in s2.passive_set
105    {
106        e.start := s2.insert;
107        e.current := s1.insert;
108        // initialize first and last because s1.insert will be
109        // the resulting schedule's insertion point
110        e.first := s1.insert;
111        e.last := p2;
112    }
113    s1.scheduled_set := set_union(s1.scheduled_set,s2.passive_set);
114    // initialize fields of resulting schedule
115    s.insert := s1.insert;
116    s.scheduled_set := set_union(s1.scheduled_set,s2.scheduled_set);
117    s.passive_set := empty_set;
118    return s;
119 }

```

**Element Insertion** A new element  $e$  is inserted at the insertion node of its schedule  $s$  if such a node exists. This is done by making  $e$ 's current and start node equal to the insertion node. If the schedule contains no nodes, and hence also no insertion node, then the element is made passive instead.

```

120 procedure insert_element(Element e, Schedule s)
121 {
122     if (s.insert = null) // no nodes in schedule
123     {
124         // make element passive

```

```

125         add e to s.passive_set;
126     }
127     else
128     {
129         // schedule element
130         e.start := e.current := s.insert;
131         e.first := e.last := null;
132         add e to s.scheduled_set;
133     }
134 }

```

**Element Deletion** An element  $e$  is deleted from its schedule  $s$  by removing it from the schedule's passive set or scheduled set, whichever contains  $e$ .

```

135 procedure delete_element(Element e, Schedule s)
136 {
137     if (e in s.passive_set) remove e from s.passive_set;
138     else remove e from s.scheduled_set;
139 }

```

**Element Activation** When an element  $e$ , belonging to schedule  $s$ , is activated, it is matched with its current node, and has its current node subsequently advanced to the next node. If this next node is also its start node, then depending on whether there are (unseen) nodes in between its last and first node, the element is made passive, or a jump back is performed. Such a jump back consists of setting the element's current node to the node after its last node, and its start node to its first node. If after this,  $e$ 's current node equals the schedule's insertion node,  $e$ 's first and last node are set to respectively this insertion node and the node before it. Similarly, if a normal activation (i.e., when the next node is not  $e$ 's start node) causes  $e$  to pass its schedule's insertion point, its first and last fields are instantiated appropriately.

```

140 procedure activate(Element e, Schedule s)
141 {
142     match e with e.current;
143     if (e.current.next = e.start)
144     {
145         // conditional jump back
146         if (e.last.next = e.first)
147         {
148             // no unseen nodes: make e passive
149             add e to s.passive_set;
150             remove e from s.scheduled_set;
151         }
152     }
153     else
154     {
155         // jump back
156         e.start := e.first;
157         e.current := e.last.next;

```

```

157         if (e.last.next = s.insert) e.first := s.insert;
158         else e.first := e.last := null;
159     }
160 }
161 else
162 {
163     e.current := e.current.next;
164     if (e.current = s.insert)
165     {
166         // e has passed insertion point
167         e.first := e.current;
168         e.last := e.current.prev;
169     }
170 }
171 }

```

The above description is far from optimal. In particular, it is too expensive to manually update element pointers at each node deletion. In the next section, a much more complicated, but quasi optimal implementation is presented and subsequently proven correct.

### 3 An Optimized Implementation of Mergeable Schedules

In this section, we describe an optimized implementation of the scheduling data structure that in particular makes node deletion considerably cheaper compared to the description in Section 2.3. The proposed implementation is quasi optimal in the sense that each of the supported operations takes quasi constant time. The correctness of the proposed implementation is established in Section 4 and its time and space complexity are analyzed in Section 5.

#### 3.1 Union-Find with Distances

For complexity reasons we use an extended version of the union-find algorithm for pointer management. The union-find algorithm is used to represent disjoint sets under the set union operation. Our description is roughly based on [11,12] and as such makes use of *named* (also called *labeled*) sets.<sup>3</sup> The algorithm offers the following functions and operations:

- $e = \text{make}(n)$  which creates a new set with name  $n$ , containing a single element  $e$
- $n = \text{find}(e)$  which returns the name of the set containing  $e$ ; if two elements  $e_1$  and  $e_2$  are in the same set, then  $\text{find}(e_1) = \text{find}(e_2)$
- $\text{union}(e_1, e_2)$  which combines the set containing element  $e_1$  with the set containing element  $e_2$ . The resulting set has the same name as the original set of  $e_1$ .

---

<sup>3</sup> Further on, we use the names to store pointers.

In an optimized implementation, all these operations take quasi constant time (amortized).<sup>4</sup> Note that our union operation is asymmetric because the name of the first argument's set is used for the resulting set, whereas the name of the second argument's set is discarded. This is similar to the description in [12].

For the purpose of our scheduling data structure, we extend the union-find data structure by introducing the concept of the *distance* of elements. These distances are defined recursively as follows:

- For a set consisting of only one element  $e$ , the distance of  $e$  equals 0.
- Given two elements  $e_1$  and  $e_2$  belonging to disjoint sets  $S_1$  and  $S_2$ , then after a call to `union( $e_1, e_2$ )`, we have that each element  $e'_1$  of  $S_1$  keeps its distance from before the `union` call, whereas each element  $e'_2$  of  $S_2$  has its distance from before the call increased by one plus the distance of  $e_1$ .

One way to think of the distances is as the maximal number times an element's set name is changed under any order of the union operations. For example, given elements  $e_1$ ,  $e_2$  and  $e_3$ , and union operations `union( $e_1, e_2$ )` and `union( $e_2, e_3$ )`, then the distances are as follows:  $e_1$  has a distance of zero,  $e_2$  has a distance of one, and  $e_3$  has a distance of two.

The union-find operations and functions are extended to support distances as follows. The `make` function is extended with an extra argument representing the distance, and now has two input arguments:  $e = \text{make}(n, d)$  where  $n$  is the set name (as before) and  $d$  is the initial distance which is 0 by default. The `find` function is extended to also return the distance  $d$ , next to the set name  $n$ :  $\langle n, d \rangle = \text{find}(e)$ . To increase readability, we write  $n = \text{value}(e)$  and  $d = \text{dist}(e)$  where  $\langle n, d \rangle = \text{find}(e)$ . We also use the following shorthand notation  $U_1 := \text{make\_and\_union}(U_0, D)$  to denote the combination of  $U_1 := \text{make}(\cdot, -1 - \text{dist}(U_0) + D)$  and `union( $U_0, U_1$ )` which makes the new union find element  $U_1$  point to the same set as element  $U_0$  with distance  $D$ .

### 3.2 Representation

This subsection describes how nodes, elements and schedules are represented.

**Nodes** Nodes are kept in a doubly linked circular list. Instead of having a direct pointer to the next and previous node, we add an extra level of indirection by using the extended union-find algorithm as presented above. The extra level of indirection allows us to update a set of pointers in quasi constant time (see further). A node is represented as a structure containing two fields: `prev` and `next`. Both fields contain a union-find element whose set name is respectively a pointer to the previous node and a pointer to the next node. Finding the previous or next node of a given (current) node  $c$  is done by applying the find operation on the respective element and following the pointer which is stored as the set name, e.g., for the previous node  $p$ :  $p = \text{value}(c.\text{prev})$ .

<sup>4</sup> More precisely, for a series of  $m$  operations on  $n$  elements, the amortized time complexity is  $\mathcal{O}(m + n\alpha(n))$  where  $\alpha$  is the inverse Ackermann function which can be considered a constant for all practical purposes.

**Elements** An element is represented as a structure with six fields. The first four, namely `start`, `current`, `first` and `last`, contain union-find elements and indirectly point to the corresponding nodes cfr. previous section. The last two, `start_to_last` and `last_to_first`, are used to store distances: `start_to_last` contains the distance between the (original) start node and the last node, and `last_to_first` contains the distance between the last and first node. Both distances are recorded at the moment the element passes the insertion point. Therefore, the actual distances may differ from the recorded ones in a later state.

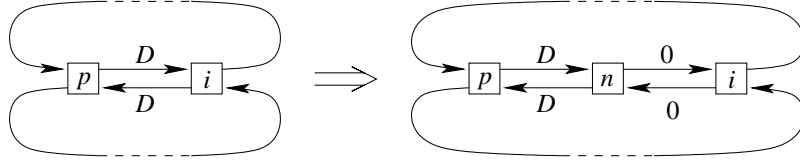
**Schedules** A schedule is represented as a structure with two fields: `insert` and `round_trip`. The `insert` field contains a union-find element pointing to the schedule's insertion node. The `round_trip` field contains the round trip distance of the schedule, which is defined as follows for a given schedule  $S$ . Let  $N_{i+1} = \text{value}(N_i.\text{next})$  for  $1 \leq i \leq n$  where  $N_1 = N_{n+1} = \text{value}(S.\text{insert})$  and  $N_j \neq N_1$  for  $1 < j \leq n$ , i.e., the nodes  $N_1, \dots, N_n$  form one cycle of the circular list of nodes of  $S$ , starting from the insertion node of  $S$ . The round trip distance  $S.\text{round\_trip} = n + \sum_{i=1}^n \text{dist}(N_i.\text{next})$ . Intuitively, it corresponds to the distance an element's current pointer would have after having made a round trip through all nodes (starting at the schedule's insertion node, or any other node).

### 3.3 Operations

This subsection shows how different events on nodes, schedules and elements are dealt with. In particular, nodes and elements can be inserted and deleted, schedules can be merged, and elements can also be activated which might lead to a conditional jump back, or cause the elements in question to become passive.

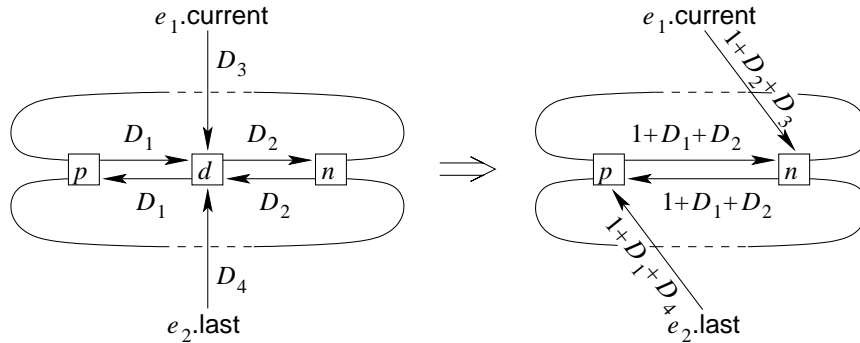
**Node Insertion** If the schedule in which the new node  $n$  is to be inserted, contains no nodes (or no such schedule exists), then we initialize the new node's fields as follows:  $n.\text{prev} := \text{make}(n, 0)$  and  $n.\text{next} := \text{make}(n, 0)$ . Note that we create two union-find elements in two disjoint sets that share the same name and distance. This is necessary for dealing with node deletion. We also initialize the fields of the corresponding schedule  $s$ : its insertion node is pointed to  $n$  using a next pointer:  $s.\text{insert} := n.\text{next}$ , and its round trip distance is set to one:  $s.\text{round\_trip} := 1$ .

If the schedule already contains nodes, then let  $i$  be its insertion node,  $p = \text{value}(i.\text{prev})$  and  $D = \text{dist}(i.\text{prev})$ . The fields of the new node  $n$  are initialized as follows:  $n.\text{next} := \text{make\_and\_union}(p.\text{next}, 0)$  and  $n.\text{prev} := i.\text{prev}$ . Furthermore, the insertion node  $i$  and its (former) previous node  $p$  are updated:  $i.\text{prev} := \text{make}(n, 0)$  and  $p.\text{next} := \text{make}(n, D)$ . Again, we keep the previous and next pointers in separate sets, even though they point to the same node.



In either case, all elements that were passive in the given schedule, are rescheduled. Let  $e$  be such an element. We set  $e.start := make\_and\_union(n.next, 0)$  and  $e.current := make\_and\_union(p.next, s.round\_trip)$  where  $s.round\_trip$  is the schedule's round trip distance. By using these distances, it is ensured that  $e.current$  and  $e.start$  have the correct distances for the purpose of initializing the `start_to_last` field and for detecting whether  $e$  has lapped its start node or not. Afterwards, the value of  $s.round\_trip$  is increased by one.

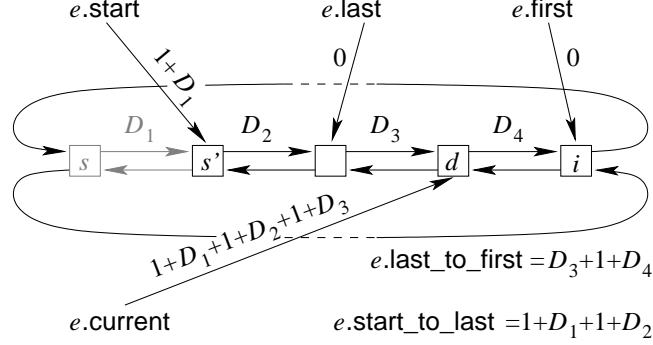
**Node Deletion** Let  $d$  be a node that is to be deleted. Let  $p$  be the previous node in the schedule and let  $n$  be the next one, i.e.,  $p = value(d.prev)$  and  $n = value(d.next)$ . If  $d$  is the only remaining node in the schedule (i.e.,  $p = d = n$ ), then we make all elements in the schedule passive. Otherwise, we make use of union operations to update the affected pointers:  $union(d.next, p.next)$  and  $union(d.prev, n.prev)$ . Note that because of the asymmetric union operation, the result is that  $p = value(n.prev)$  and  $n = value(p.next)$ . The result is that all schedule elements whose `current`, `first` or `start` field pointed to  $d$ , now have these fields pointing to  $n$  and all schedule elements whose `last` field pointed to  $d$ , now have this field pointing to  $p$ . The figure below illustrates this and also shows how the distances are updated.



Deletion may cause elements to pass their schedule's insertion point. For complexity reasons, we only consider those elements that have not passed the insertion point more than once since their last jump back or (re-)scheduling.<sup>5</sup> We distinguish two cases: elements that pass the insertion point for the first time, and those that pass it for the second time. In the following, we assume that the union operations shown above, have not been performed yet. Then for each element  $e$  in the set of elements that pass the insertion point for the first time, the

<sup>5</sup> In Section 4.2, such elements are said to be in Degenerate Case 2 or 3.

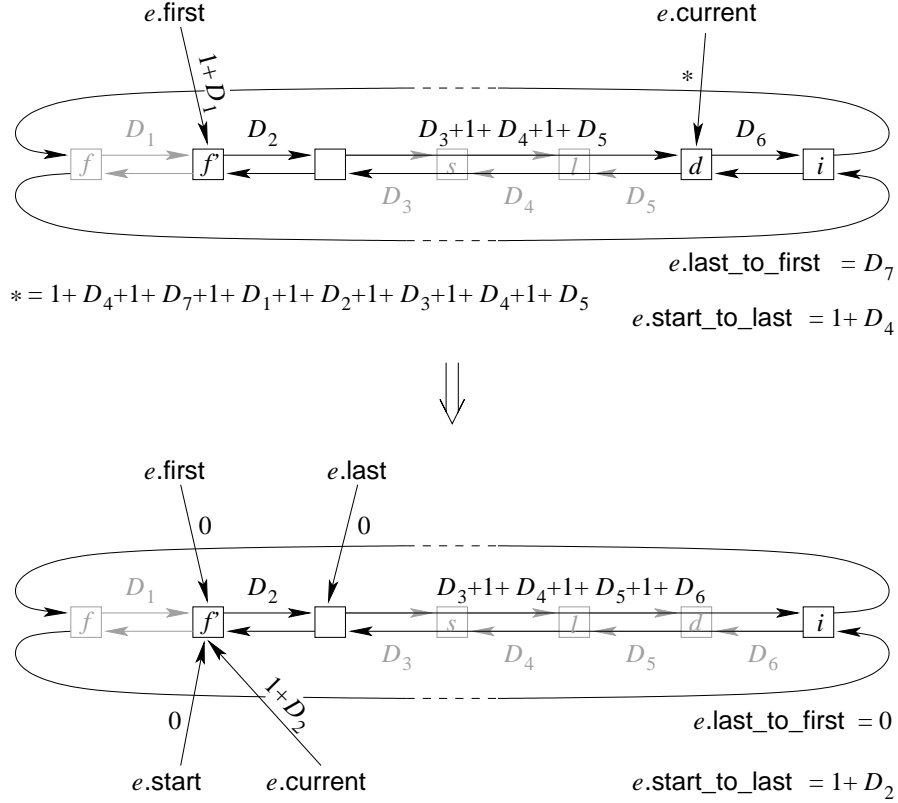
first and last fields are initialized as follows:  $e.first := \text{make\_and\_union}(d.next, 0)$  and  $e.last := \text{make\_and\_union}(d.prev, 0)$ . An exception is the degenerate case that  $\text{value}(d.next) = \text{value}(e.start)$  in which case we initialize the first node by  $e.first := e.start$  instead. We furthermore initialize the values of  $e.start\_to\_last$  and  $e.last\_to\_first$  as follows:  $e.start\_to\_last := \text{dist}(e.current) - 1 - \text{dist}(d.prev)$  and  $e.last\_to\_first := \text{dist}(d.prev) + 1 + \text{dist}(d.next)$ . The figure below illustrates the relation between the  $start\_to\_last$  and  $last\_to\_first$  values and the distances of the start, current, first and last pointers.



In the figure, node  $s$  is the original start node of  $e$  but has meanwhile been deleted, making  $s'$  the new start node. Also shown are the deleted node  $d$  and insertion node  $i$ . The value of  $start\_to\_last$  denotes the distance between the original start node ( $s$ ) and the original last node. At any point in time, the distance between the actual start and last node equals  $e.start\_to\_last - \text{dist}(e.start) - \text{dist}(e.last)$ . If this value is negative, this means that at some earlier state, the start node was equal to the last node and was deleted. Note that as long as the value is positive, no new nodes can be added in between the start and last node. The distance between the original first node and the current node equals  $\text{dist}(e.current) - e.start\_to\_last - e.last\_to\_first$ . To find the distance between the actual first node and the current node, we further need to subtract  $\text{dist}(e.first)$  from this value.

Those elements that pass the insertion point for the second time (the insertion point may have changed meanwhile), are in the above mentioned degenerate case in which  $e.start\_to\_last - \text{dist}(e.start) - \text{dist}(e.last) < 0$ . In this case, the element may see nodes that have been added after it passed the insertion point for the first time. Let  $D_f = \text{dist}(e.first)$ . First, we make the start node equal to the first node and reset the distance of both the `start` and `first` pointers:  $e.start := \text{make\_and\_union}(e.first, 0)$  and  $e.first := e.start$ . Next, we make the last node point to the node before the deleted one (i.e., right before the insertion point after deletion):  $e.last := \text{make\_and\_union}(d.prev, 0)$ . Again, the pointer distance is zero. The distances  $start\_to\_last$  and  $last\_to\_first$  are updated as follows:  $e.start\_to\_last := \text{dist}(e.current) - e.start\_to\_last - 1 - e.last\_to\_first - D_f - 1 - \text{dist}(d.prev)$ , and  $e.last\_to\_first := 0$ . Finally, we update the current node using the new value of  $e.start\_to\_last$  if this value is positive:  $e.current := \text{make}(\_, e.start\_to\_last)$ ,

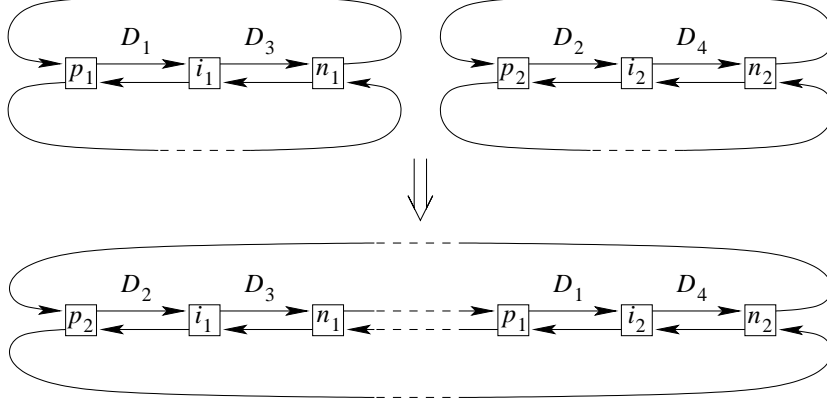
or  $e.current := make(-, 0)$  if  $e.start\_to\_last < 0$ , followed by a union operation:  $union(e.start, e.current)$ . After this,  $value(e.current) = value(e.start)$  and  $dist(e.current) > dist(e.start)$  and so a conditional jump back will be performed as soon as  $e$  is activated (see further). We do not perform this conditional jump back immediately to ensure the cost of the deletion can be amortized on previous operations. The figure below illustrates how the various fields are updated.



In the figure,  $f$  denotes the original first node, which meanwhile has been deleted. The new first node is  $f'$ . The original start node  $s$  and last node  $l$ , as well as all nodes in between, have also been deleted, causing the degeneracy. The value of  $dist(e.current)$  before the deletion is  $(1 + D_4) + (1) + (D_7) + (1 + D_1) + (1 + D_2) + (1) + (D_3 + 1 + D_4 + 1 + D_5)$  which is  $e.start\_to\_last + 1 + e.last\_to\_first + D_f + (1 + D_2) + 1 + dist(d.prev)$ . Note that if  $value(e.first) = value(e.current)$  then before the deletion,  $dist(e.current) = e.start\_to\_last + 1 + e.last\_to\_first + D_f$  and after deleting  $d$ ,  $e.start\_to\_last = -1 - dist(d.prev) < 0$  and hence is in any event smaller than  $e.start + e.last$ .<sup>6</sup>

<sup>6</sup> The current node cannot have lapped the first node without first having passed either the start or insertion node again.

**Merging Schedules** Let  $s_1$  and  $s_2$  be the schedules to merge, and let  $i_1$  and  $i_2$  be the (dereferenced) insertion nodes of the respective schedules. Let  $p_j = \text{value}(i_j.\text{prev})$  and  $D_j = \text{dist}(i_j.\text{prev})$  for  $j \in \{1, 2\}$ . The merge operation consists of the following simultaneous updates:<sup>7</sup>  $p_1.\text{next} := \text{make\_and\_union}(p_2.\text{next}, D_1)$ ,  $p_2.\text{next} := \text{make\_and\_union}(p_1.\text{next}, D_2)$ ,  $i_1.\text{prev} := i_2.\text{prev}$ , and  $i_2.\text{prev} := i_1.\text{prev}$ .



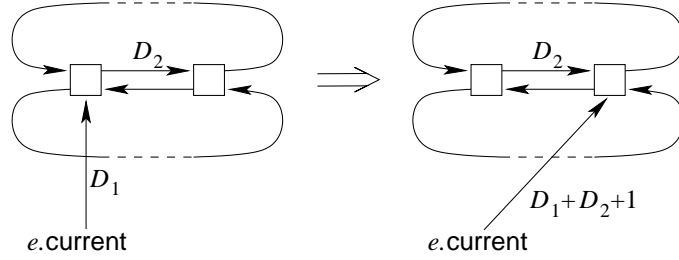
The resulting schedule  $s$  has its insertion node and round trip distance set as follows:  $s.\text{insert} := s_1.\text{insert}$  and  $s.\text{round\_trip} := s_1.\text{round\_trip} + s_2.\text{round\_trip}$ . All elements that were passive in schedule  $s_1$  are rescheduled as follows: let  $e$  be such an element, then  $e.\text{current} := \text{make\_and\_union}(p_1.\text{next}, s_1.\text{round\_trip})$  and  $e.\text{start} := \text{make\_and\_union}(p_2.\text{next}, 0)$ . The passive elements in schedule  $s_2$  are treated in a similar way. The reasoning is similar to the case of a (single) node insertion. However, the rescheduled elements of schedule  $s_2$  also need to have their `first`, `last`, `start_to_last` and `last_to_first` fields initialized to avoid that they are not matched with nodes inserted after the schedule merge, which would be inserted in between the rescheduled elements' start and current node.<sup>8</sup> The initialization is as follows:  $e.\text{first} := \text{make\_and\_union}(p_2.\text{next}, 0)$ ,  $e.\text{last} := \text{make\_and\_union}(i_1.\text{prev}, 0)$ ,  $e.\text{start\_to\_last} := s_2.\text{round\_trip} - 1 - D_2$  and  $e.\text{last\_to\_first} := D_2$ . The result is that the elements' first node and last node are respectively  $i_1$  and  $p_2$ .

**Element Insertion** A new element  $e$  is always inserted at the schedule's insertion node  $i$  if such a node exists. Otherwise, the element is made passive. Assume  $i$  exists and let  $p$  be the node right before  $i$ , i.e.,  $p = \text{value}(i.\text{prev})$ . The element's fields are initialized as follows:  $e.\text{current} := e.\text{start} := \text{make\_and\_union}(p.\text{next}, 0)$ . Note that the `start` and `current` fields share their union-find element. This is allowed because they are both forward pointers. Finally, the `first`, `last`, `start_to_last` and `last_to_first` fields remain uninitialized, i.e.,  $e.\text{first} = e.\text{last} = \text{null}$  and  $e.\text{start\_to\_last} = e.\text{last\_to\_first} = 0$ .

<sup>7</sup> In practice, some auxiliary variables are needed.

<sup>8</sup> Initially, the current node of the rescheduled elements equals the resulting schedule's insertion node.

**Element Activation** When an element  $e$  is activated, it is first checked whether a conditional jump back is required. This is the case if both  $\text{value}(e.\text{current}) = \text{value}(e.\text{start})$  and  $\text{dist}(e.\text{current}) > \text{dist}(e.\text{start})$ . In the next paragraph, we show how this situation is dealt with. If no conditional jump back is required, the element  $e$  is matched with its (dereferenced) current node. In the general case where the next node is not equal to the schedule's insertion node, i.e.,  $\text{value}(s.\text{insert}) \neq \text{value}(\text{value}(e.\text{current}).\text{next})$  where  $s$  is the schedule, we update  $e$ 's current field as follows:  $e.\text{current} := \text{make}(\_, \text{dist}(e.\text{current}))$  and  $\text{union}(\text{value}(e.\text{current}).\text{next}, c)$ . Intuitively, the current node is advanced to the next node, and its distance is updated as shown in the figure below.



In case the next node is the insertion node, we again distinguish between the case that it passes the insertion point for the first time, and the case that it passes this point for the second time. Apart from some minor details which follow from the fact that the current node is not deleted, this event is dealt with in the same way as deleting the current node if it appears right before the insertion node.

**Conditional Jump Back** Activating an element  $e$  can lead to a conditional jump back if  $\text{value}(e.\text{current}) = \text{value}(e.\text{start})$  and  $\text{dist}(e.\text{current}) > \text{dist}(e.\text{start})$ . In the normal case, a conditional jump back works as follows: if  $\text{value}(e.\text{first}) = \text{value}(\text{value}(e.\text{last}).\text{next})$ , then  $e$  is made passive. Otherwise, we set the current node to the node after the last node, set the start node to the first node, and reset the first and last nodes, as well as the two distance fields. More precisely, this is done as follows. To make sure the value of the  $e.\text{start\_to\_last}$  field is correctly computed in states following a jump back, the current field is updated such that its distance corresponds to the actual distance between the start and current node. This distance equals  $D = \text{dist}(e.\text{current}) - \text{dist}(e.\text{start}) - \text{dist}(e.\text{last}) - 1 - e.\text{last\_to\_first} - \text{dist}(e.\text{first})$  if  $e.\text{start\_to\_last} - \text{dist}(e.\text{start}) - \text{dist}(e.\text{last}) \geq 0$  and  $D = \text{dist}(e.\text{current}) - e.\text{start\_to\_last} - 1 - e.\text{last\_to\_first} - \text{dist}(e.\text{first})$  otherwise. Note that  $D$  can be zero in which case we are in the degenerate case described further on. If  $D > 0$  then the current field is updated by  $e.\text{current} := \text{make}(\_, D)$  and  $\text{union}(\text{value}(e.\text{last}).\text{next}, e.\text{current})$  which causes it to point to the node after the former last node, with a distance equal to  $D$  plus the distance from the last node to its next node. The start field is updated by  $e.\text{start} := \text{make\_and\_union}(e.\text{first}, 0)$  which causes it to point to the former first node with distance zero. If the new current node equals the schedule's insertion node, then the remaining fields are initialized as follows:  $e.\text{first} := \text{make\_and\_union}(c, 0)$ ,

$e.last := \text{make\_and\_union}(e.last, 0)$ ,  $e.start\_to\_last := D$  and  $e.last\_to\_first := \text{dist}(c)$  where  $c = \text{value}(e.last).next$ . Any new nodes will indeed be inserted in between the last and first nodes now. If the new current node is not equal to the schedule's insertion node, then we reset the remaining fields:  $e.first := e.last := \text{null}$  and  $e.start\_to\_last := e.last\_to\_first := 0$ . After these updates,  $e$  is activated as shown in the previous paragraph.

In the degenerate case that  $e.start\_to\_last - \text{dist}(e.start) - \text{dist}(e.last) < 0$ ,  $\text{value}(e.first) = \text{value}(e.start)$  and  $\text{dist}(e.first) \geq \text{dist}(e.start)$ , all nodes with which  $e$  has matched so far, have been deleted. Therefore, we reset all fields as in the case of an element insertion, followed by an element activation.

### 3.4 Element Schedule

So far we have not shown how to determine which elements still need to be activated and which are passive. In this subsection, we describe a method to do this efficiently for the case that it is not important in which order the elements are activated. In particular, we can use a stack of elements to be activated. In the following, we refer to this data structure as the *element schedule*.

For complexity reasons, we need to make the data structure somewhat more complicated. On the one hand, we need to ensure that the deletion of an element is efficient. On the other hand, we want to make sure that it is cheap to make a set of elements passive whenever a node is deleted, as well as initializing the elements' *first* and *last* fields when they pass the insertion node. The efficient deletion can be achieved by maintaining for each element, a pointer to the next and previous element in the stack. For the other requirement, we do the following.

The element schedule is a stack containing both scheduled elements and representatives for *local schedules*. A local schedule is also represented as a stack and contains elements that have been simultaneously scheduled (because of a node insertion or schedule merge) and have not been activated since. A local schedule's representative is a structure with the same fields as an element, plus a reference to the stack it represents. The elements in a local stack have their fields pointing to the corresponding fields of their representative via a union-find element. We also keep track of the passive elements by storing them in a schedule, called the *passive schedule*. The passive schedule is again a stack. Whenever an element becomes passive, it is added on top of this stack. The elements in a passive stack have their fields all pointed to the same value by means of an extra level of indirection via union-find elements.

Now, a local schedule's representative is dealt with in the same way as a regular element, except when it is activated, which happens if it appears on top of the element stack. In that case, if the result of activation as a regular element would make it passive, then the local stack is appended to the schedule's passive stack (this can be done in constant time), and the fields of the local schedule's elements are pointed to the same value as those in the passive schedule by using a *union* operation. Afterwards, the representative is discarded. If on the other hand, treatment as a regular element would result in a new match, then the

top element of the local stack is popped, its fields are dereferenced,<sup>9</sup> and the element is activated as usual. If a local stack becomes empty, its representative is removed. The deletion of an element that appears in a local schedule, is dealt with similarly to a regular element deletion, but may also result in the removal of the local schedule’s representative in case this schedule becomes empty.

Using the above approach, elements that were simultaneously rescheduled because of a node insertion or schedule merge, and that need to be made passive on their first activation, can be dealt with in (quasi) constant time.

## 4 Correctness

In this section, the correctness of our algorithm is shown. The intended meaning of correctness is that every element is eventually matched with every node in its schedule, and no node is matched with twice by the same element. In Section 4.1, we give some lemmas concerning distances for the purpose of detecting degeneracies. The main correctness theorem is given in Section 4.2.

In the following, we denote a sequence of nodes as  $[N_1, \dots, N_n]$ . The empty sequence is denoted by  $\epsilon$  and sequence concatenation by the  $++$  operator. We also write  $X \preceq Y \preceq Z$  to denote that the shortest path of non-zero length from node  $X$  to node  $Z$  using forward pointers only, contains node  $Y$ . We write  $X \prec Y$  instead of  $X \preceq Y$  to denote the additional fact that the subpath from  $X$  to  $Y$  contains at least one link.<sup>10</sup>

### 4.1 Distances and Lapping

The following lemma shows how we can determine whether some element’s current node has ‘lapped’ its start node, or whether its start node has ‘caught up’ with its current node.

**Lemma 1.** *If in some state, a given element  $e$ ’s start and current node are equal, i.e.,  $\text{value}(e.\text{start}) = \text{value}(e.\text{current})$ , then if  $\text{dist}(e.\text{start}) < \text{dist}(e.\text{current})$ , the last event that made  $e$ ’s start and current node equal, is either the deletion of  $e$ ’s current node, or the activation of  $e$ .<sup>11</sup> If on the other hand  $\text{dist}(e.\text{start}) \geq \text{dist}(e.\text{current})$ , then the last event that made the nodes equal, is the deletion of  $e$ ’s start node or the initialization of  $e$ ’s start and current fields.*

*Proof.* We start by introducing two distances,  $D_s^c$  and  $D_c^s$ , which are defined below. Let  $N_{i+1} = \text{value}(N_i.\text{next})$  for  $1 \leq i \leq n$  where  $N_1 = N_{n+1} = \text{value}(e.\text{start})$  and  $N_j \neq N_1$  for  $1 < j \leq n$ , i.e.,  $[N_1, \dots, N_n]$  represents one cycle of the

<sup>9</sup> More precisely, only the extra indirection with respect to regular elements is removed.

<sup>10</sup> Although  $X$  and  $Y$  can be equal in the case of a schedule consisting of only one node  $N$ . In that case, both  $N \prec N \preceq N$  and  $N \preceq N \prec N$ , but not  $N \prec N \prec N$ .

<sup>11</sup> If  $e$ ’s start and current node are equal, and  $e$  is successfully activated without changing the current node, then we consider this activation as the last event that made the start and current node equal.

circular list of nodes, starting from  $e$ 's start node. Let  $c$  be such that  $N_c = \text{value}(e.\text{current})$ . The distances  $D_s^c$  and  $D_c^s$  are defined as follows:

$$D_s^c = \left( \sum_{i=1}^{c-1} 1 + \text{dist}(N_i.\text{next}) \right) + \text{dist}(e.\text{start})$$

$$D_c^s = \left( \sum_{i=c}^n 1 + \text{dist}(N_i.\text{next}) \right) + \text{dist}(e.\text{current})$$

Intuitively,  $D_s^c$  represents the (directional) distance between  $e$ 's start and current node, whereas  $D_c^s$  represents the distance between  $e$ 's current and start node. Both distances include the union-find pointer distance of respectively  $e$ 's start field and  $e$ 's current field.

Initially, when  $e.\text{start}$  and  $e.\text{current}$  are initialized (after  $e$  is inserted, reactivated, or has jumped back), we have that  $\text{dist}(e.\text{current}) = D_s^c$  and  $\text{dist}(e.\text{start}) = 0$ .<sup>12</sup> If in this initial state, the start and current node are equal, then  $D_s^c = 0$  and clearly  $\text{dist}(e.\text{start}) \geq \text{dist}(e.\text{current})$ . The theorem holds in this case as the last event making both nodes equal is their initialization. Assume that the theorem holds in a given state in which  $e$ 's start and current node are equal. We now show that as long as the nodes remain equal, the theorem still holds.

The values of  $\text{dist}(e.\text{start})$  and  $\text{dist}(e.\text{current})$  can change in the following ways. A first possibility is that the start (and current) node is deleted. In this case (assuming that the schedule is not empty at this point), both  $\text{dist}(e.\text{start})$  and  $\text{dist}(e.\text{current})$  are increased by the value of  $\text{dist}(\text{value}(e.\text{start}).\text{next})$  before the deletion and hence the relation between both distances remains the same. An exception is the case that the deletion causes  $e$  to pass its schedule's insertion point for the second time. In that case, it is ensured that  $\text{dist}(e.\text{start}) < \text{dist}(e.\text{current})$  after this event, which simulates  $e$  lapping its start node. If  $\text{dist}(e.\text{start}) \geq \text{dist}(e.\text{current})$ , then another possibility is that  $e$  is (successfully) activated. If the result of this activation is that  $e$ 's start and current node are still equal, then the activation increases the value of  $\text{dist}(e.\text{current})$  with  $1 + s.\text{round.trip}$  with  $s$  the schedule of  $e$ , and so in the result  $\text{dist}(e.\text{start}) < \text{dist}(e.\text{current})$  and the last event that made  $e$ 's start and current node equal (according to the interpretation of 'last event' given in the theorem) is the activation of  $e$ .

In the remainder of this proof, we show that whenever the start and current node are not equal,  $D_s^c \geq \text{dist}(e.\text{current})$  and  $D_c^s > \text{dist}(e.\text{start})$ . Initially, if both nodes are different, then since by initialization,  $\text{dist}(e.\text{start}) = 0$ ,  $\text{dist}(e.\text{current}) = D_s^c$ , and both  $D_s^c$  and  $D_c^s$  are strictly positive, both the invariants trivially hold. If in some state, both nodes are equal, and in a next state, they become distinct, then an activation of the element  $e$  must have taken place. Before the activation, we must have that  $\text{dist}(e.\text{start}) \geq \text{dist}(e.\text{current})$ , or the activation would have made  $e$  passive. After activation,  $\text{dist}(e.\text{current})$  increases by the value of  $1 + \text{dist}(\text{value}(e.\text{current}).\text{next})$  before activation. We then have that  $D_s^c = \text{dist}(e.\text{start}) + 1 + \text{dist}(\text{value}(e.\text{start}).\text{next}) \geq \text{dist}(e.\text{current})$  after activation.

<sup>12</sup> The initialization of the current pointer distance is explained in Section 3.3.

ing  $e$ . Also  $D_c^s$  is at least one more than  $\text{dist}(e.\text{start})$ . Hence, whenever the start and current node become distinct, the invariants hold.

Next, assume that the start and current node are distinct and remain so after an event, then one of the following applies:

- If the start node is deleted, then  $\text{dist}(e.\text{start})$  as well as  $\text{dist}(p.\text{next})$  increase by the value of  $1 + \text{dist}(\text{value}(e.\text{start}).\text{next})$  before deletion, where  $p = \text{value}(\text{value}(e.\text{start}).\text{prev})$ . Hence  $D_c^s$  increases by the same amount, and  $D_s^c$  remains the same, so the invariants remain valid because  $D_c^s$  and  $\text{dist}(e.\text{current})$  remain the same, and  $D_c^s$  and  $\text{dist}(e.\text{start})$  are both increased by the same value.
- If the current node is deleted, or  $e$  advances to the next node, then distance of  $e$ 's current pointer increases by the value  $1 + \text{dist}(\text{value}(e.\text{current}).\text{next})$  before the deletion or activation. The distance  $D_c^s$  remains as before (the path between the current and start node contains one node less, but this is compensated by the increased distance for the current node) and  $D_s^c$  increases by the same amount as the current node's distance. Again, the invariants remain valid.
- If some other node is deleted, then  $D_c^s$  and  $D_s^c$ , as well as the distances for the start and current nodes, remain as before.
- If some node is added in between the start and current nodes, then  $D_s^c$  is increased by one and  $D_c^s$  remains as before. If some node is added in between the current and start nodes, then  $D_c^s$  increases with one and  $D_s^c$  remains as before. Either way, the invariants remain.

Finally, given that the start and current node are distinct, but become equal after an event, one of the following applies:

- If  $e$ 's start node appears right before its current node and is subsequently deleted, then after this event, the distance of  $e$ 's **start** pointer increases by the value of  $1 + \text{dist}(\text{value}(e.\text{start}).\text{next})$  before the deletion. This means that  $\text{dist}(e.\text{start})$  becomes equal to the value of  $D_s^c$  before the deletion, and hence since the value of  $\text{dist}(e.\text{current})$  remains unchanged, the invariants imply that  $\text{dist}(e.\text{current}) \leq \text{dist}(e.\text{start})$ . Since the last event making the start and current node equal, is the deletion of the start node, the theorem holds in this case.
- If  $e$ 's current node appears right before its start node, and either the current node is deleted or  $e$  is activated, then after this event, the distance of  $e$ 's **current** pointer increases by the value of  $1 + \text{dist}(\text{value}(e.\text{current}).\text{next})$  before the deletion. This means that  $\text{dist}(e.\text{current})$  becomes equal to the value of  $D_c^s$  before the deletion, and hence since the value of  $\text{dist}(e.\text{start})$  remains unchanged, the invariants imply that  $\text{dist}(e.\text{current}) > \text{dist}(e.\text{start})$ . Since the last event making the start and current node equal, is either the deletion of the current node, or the activation of  $e$ , the theorem also holds in this case.
- If  $e$ 's current node appears right before its schedule's insertion node,  $e.\text{first}$  and  $e.\text{last}$  are already initialized, and either  $e$ 's current node is deleted or  $e$  is activated, then after this event,  $e$ 's current node equals its start node

and  $\text{dist}(e.\text{current}) > \text{dist}(e.\text{start})$  by initialization. In this case, the last event making the start and current node equal, is the simulated event of  $e$  lapping its start node, and so the theorem holds in this case as well.

Since these are all the possibilities, this concludes our proof.  $\square$

The next result allows us to detect whether the sequence of nodes

$$[\text{value}(e.\text{start}), \dots, \text{value}(e.\text{last})]$$

has become empty by a deletion of the start and last node, after both nodes have become equal.

**Lemma 2.** *If in some state, for a given element  $e$  we have that  $e.\text{start\_to\_last} - \text{dist}(e.\text{start}) - \text{dist}(e.\text{last}) < 0$ , then in some earlier state the node  $\text{value}(e.\text{start}) = \text{value}(e.\text{last})$  is deleted. This includes the case where the last node is deleted during initialization of the last pointer.*

*Proof.* New nodes are added right before the insertion node. Therefore, as long as  $e.\text{start\_to\_last}$  is uninitialized, we have that no new nodes are added in between  $\text{value}(e.\text{start})$  and  $\text{value}(e.\text{current})$ . Let  $N_{i+1} = \text{value}(N_i.\text{next})$  for  $1 \leq i \leq n$  where  $N_1 = \text{value}(e.\text{start})$ ,  $N_n = \text{value}(e.\text{last})$  and  $N_j \neq N_1$  for  $1 \leq j \leq n$ . At the moment that  $e.\text{last}$  is initialized, we have that

$$e.\text{start\_to\_last} = \text{dist}(e.\text{start}) + \left( \sum_{i=1}^{n-1} 1 + \text{dist}(N_i.\text{next}) \right) + \text{dist}(e.\text{last}) \quad (1)$$

If the initialization takes place because the start node is deleted while appearing right before the insertion node, then  $e.\text{start\_to\_last} < 0$  and the theorem trivially holds. As long as  $\text{value}(e.\text{start}) = \text{value}(e.\text{last})$  is not deleted, no new nodes are added in between the start and last node. If the start node is deleted while being different from the last node, the new value of  $\text{dist}(e.\text{start})$  is increased by  $1 + \text{dist}(N_1.\text{next})$ . Similarly, if the last node is deleted while being different from the start node, the new value of  $\text{dist}(e.\text{start})$  is increased by  $1 + \text{dist}(N_{n-1}.\text{next})$  (which also equals  $1 + \text{dist}(N_n.\text{prev})$ ). Finally, if some node in between the start and last node is deleted, say  $N_i$  with  $1 < i < n$ , then the distance of  $N_{i-1}.\text{next}$  increases by  $1 + \text{dist}(N_i.\text{next})$ . In any case, the increase in distance is compensated by a reduction of the sum in (1).

Now if the start and last node have become equal, (1) becomes

$$e.\text{start\_to\_last} = \text{dist}(e.\text{start}) + \text{dist}(e.\text{last}) \quad (2)$$

If this remaining node is subsequently deleted, then the distances of the start and last node are both increased by at least one, and hence because of (2),  $e.\text{start\_to\_last} - \text{dist}(e.\text{start}) - \text{dist}(e.\text{last}) < 0$ . In all later states, the distances of the start and last node can only increase, whereas the value of  $e.\text{start\_to\_last}$  remains the same. Furthermore, because no nodes could have been added in between  $\text{value}(e.\text{start})$  and  $\text{value}(e.\text{last})$  as long as both nodes were not deleted while being equal, we have that  $e.\text{start\_to\_last} - \text{dist}(e.\text{start}) - \text{dist}(e.\text{last}) < 0$  implies that in some earlier state,  $\text{value}(e.\text{start}) = \text{value}(e.\text{last})$  was deleted.  $\square$

A final result allows us to detect whether the sequence

$$[\text{value}(e.\text{first}), \dots, \text{value}(\text{value}(e.\text{start}).\text{prev})]$$

has become empty by a deletion of the first node if it appears right before the start node.

**Lemma 3.** *If in some state, for a given element  $e$  we have that  $\text{value}(e.\text{first}) = \text{value}(e.\text{start})$  and  $\text{dist}(e.\text{first}) \geq \text{dist}(e.\text{start})$ , then in some earlier state the node  $\text{value}(e.\text{first}) = \text{value}(\text{value}(e.\text{start}).\text{prev})$  is deleted. This includes the case where the first node is deleted during initialization of the first pointer.*

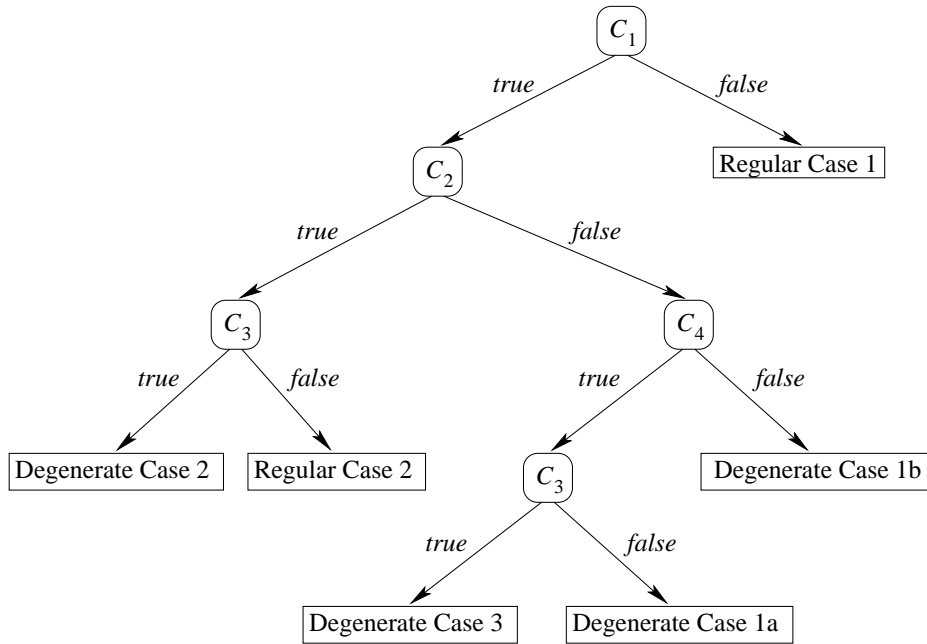
*Proof.* If  $e$ 's first pointer is initialized because  $e$ 's current node is deleted or  $e$  is activated while  $\text{value}(\text{value}(e.\text{current}).\text{next}) = \text{value}(e.\text{start}) = \text{value}(e.\text{first})$ , then by initialization,  $\text{dist}(e.\text{first}) = \text{dist}(e.\text{start})$ . Otherwise, initially  $\text{dist}(e.\text{first}) = 0$ . The rest of the proof is similar to the proof of Lemma 1.  $\square$

## 4.2 Main Correctness Result

In this subsection, the main correctness theorem is given. We first introduce four conditions on the fields of an element  $e$ :

$$\begin{aligned} C_1 &\iff e.\text{first} \neq \text{null} \wedge e.\text{last} \neq \text{null} \\ C_2 &\iff e.\text{start\_to\_last} - \text{dist}(e.\text{start}) - \text{dist}(e.\text{last}) \geq 0 \\ C_3 &\iff \text{value}(e.\text{first}) = \text{value}(e.\text{start}) \wedge \text{dist}(e.\text{first}) \geq \text{dist}(e.\text{start}) \\ C_4 &\iff \text{value}(e.\text{current}) \preceq \text{value}(e.\text{start}) \prec \text{value}(s.\text{insert}) \end{aligned}$$

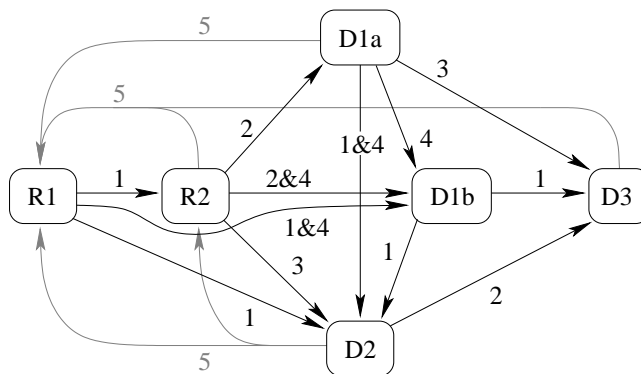
Depending on the truth value of these conditions, each scheduled element is in one of the following cases: Regular Case 1 or 2, or Degenerate Case 1a, 1b, 2 or 3. The decision tree below shows how the conditions determine the case.



An element can transition from one case to another if one of the following events happens:

1. the element's current node passes its schedule's insertion point, either because the node is deleted, or because the element is activated;
2. the element's start node equals its last node and is deleted;
3. the element's first node appears right before its start node and is deleted;
4. the element's start node appears right before its schedule's insertion node and is deleted;
5. the element is activated and a conditional jump back is performed.

The figure shows which events can cause which case transitions, where R1 and R2 are Regular Cases 1 and 2, and D1a, D1b, D2 and D3 are the respective Degenerate Cases.



In the remainder of this subsection, we consider an element  $e$  belonging to schedule  $s$ , and use the following shorthand notation in this context:  $X = \text{value}(e.X)$  for  $X \in \{\text{start}, \text{current}, \text{first}, \text{last}\}$  (e.g.,  $\text{start} = \text{value}(e.\text{start})$ ) and  $\text{insert} = \text{value}(s.\text{insert})$ . We also write  $\text{next}(X)$  and  $\text{prev}(X)$  to denote respectively  $\text{value}(X.\text{next})$  and  $\text{value}(X.\text{prev})$ . Each element has its own view of the schedule. We denote the view of element  $e$  by  $e.\text{view}$  and define it as follows.

- If  $e$  is in Regular Case 1 then

$$e.\text{view} = [\text{start}, \dots, \text{prev}(\text{start})]$$

- If  $e$  is in Regular Case 2 then

$$e.\text{view} = [\text{start}, \dots, \text{last}] ++ [\text{first}, \dots, \text{prev}(\text{start})] ++ \\ [\text{next}(\text{last}), \dots, \text{prev}(\text{first})]$$

- If  $e$  is in Degenerate Case 1a then

$$e.\text{view} = [\text{first}, \dots, \text{last} = \text{prev}(\text{start})] ++ [\text{start} = \text{next}(\text{last}), \dots, \text{prev}(\text{first})]$$

- If  $e$  is in Degenerate Case 1b then

$$e.\text{view} = [\text{first}, \dots, \text{prev}(\text{insert})] ++ [\text{insert}, \dots, \text{prev}(\text{first})]$$

- If  $e$  is in Degenerate Case 2 then

$$e.\text{view} = [\text{first} = \text{start}, \dots, \text{last}] ++ [\text{next}(\text{last}), \dots, \text{prev}(\text{first})]$$

- If  $e$  is in Degenerate Case 3 then

$$e.\text{view} = [\text{insert}, \dots, \text{prev}(\text{insert})]$$

Now the main correctness result is the following.

**Theorem 1.** *Each element is matched with each node of its schedule at most once. If an element is passive, there exists no node in its schedule with which it has not been matched.*

*Proof.* The following invariants hold on schedule views:

1. A schedule view contains all nodes in the schedule and no node appears in it more than once.
2. The schedule view can be written as  $\text{Seen} ++ \text{Unseen}$  where  $\text{Seen}$  are the nodes that  $e$  has already been matched with, and  $\text{Unseen}$  are the nodes with which  $e$  still needs to match. If  $\text{current} \neq \text{start}$  or  $\text{dist}(e.\text{current}) \leq \text{dist}(e.\text{start})$  then  $\text{Unseen}$  starts with  $\text{current}$ . Otherwise, if  $\text{current} = \text{start}$  and  $\text{dist}(e.\text{current}) > \text{dist}(e.\text{start})$  then  $\text{Unseen} = [\text{next}(\text{last}), \dots, \text{prev}(\text{first})]$ <sup>13</sup> except for Degenerate Case 3 in which  $\text{Unseen} = [\text{insert}, \dots, \text{prev}(\text{insert})]$  (and  $\text{Seen} = \epsilon$ ).

<sup>13</sup> In case  $\text{next}(\text{last}) = \text{first}$  then  $\text{Unseen}$  is considered to be empty.

The first invariant holds by definition. For the second invariant, we show that all schedule operations preserve the invariant. Initially, when an element  $e$  is inserted, the invariant holds because we start in Regular Case 1 and no nodes have been matched with  $e$ . Similarly, when an element is rescheduled after inserting a new node  $n$ , this element starts in Regular Case 1, and the *Unseen* part of the schedule consists of the new node only, i.e.,  $n = \text{current} = \text{prev}(\text{start})$ . Finally, in case of a schedule merge, the rescheduled elements are in Regular Case 1 or 2. In either case, all nodes of the schedule to which such an element  $e$  belonged before the merge, are in the *Seen* part of the schedule, and all nodes of the other schedule, are in the *Unseen* part.

In the following, we consider a scheduled element  $e$ . We analyze each of the cases  $e$  can be in (i.e., Regular Case 1 and 2, and Degenerate Case 1a, 1b, 2 and 3), and consider each of the following operations: a node insertion, a node deletion, and the activation of element  $e$ . A schedule merge is similar to a node insertion. We only consider those node deletions and element activations that change the case an element is in. The other ones are trivial. An element activation can be a regular activation or a conditional jump back. A conditional jump back satisfies the invariant because it consists of making the current node equal to the first node of the *Unseen* part of the schedule (if such a node exists) and potentially reorders the *Seen* part of the schedule. Therefore, in the remainder of the proof, we only consider regular element activations.

**Regular Case 1** New nodes are inserted right before the schedule's insertion node, and hence after the current node. If an element activation causes  $e$  to pass the insertion node, we either end up in Regular Case 2, or in Degenerate Case 2:

- If  $\text{current} \neq \text{prev}(\text{start})$  then

$$[\text{start}, \dots, \text{current}, \text{insert}, \dots, \text{prev}(\text{start})]$$

becomes

$$[\text{start}, \dots, \text{last}] ++ [\text{first} = \text{current} = \text{insert}, \dots, \text{prev}(\text{start})] ++ \epsilon$$

and  $e$  is in Regular Case 2 because by initialization,  $e.\text{start\_to\_last} - \text{dist}(e.\text{start}) - \text{dist}(e.\text{last}) \geq 0$  and because  $\text{value}(e.\text{first}) \neq \text{value}(e.\text{start})$ .

- Otherwise, if  $\text{current} = \text{prev}(\text{start})$  then

$$[\text{start} = \text{insert}, \dots, \text{current} = \text{prev}(\text{start})]$$

becomes

$$[\text{first} = \text{start} = \text{insert} = \text{current}, \dots, \text{last} = \text{prev}(\text{start})] ++ \epsilon ++ \epsilon$$

and  $e$  is in Degenerate Case 2 because by initialization,  $\text{dist}(e.\text{first}) = \text{dist}(e.\text{start})$ . Also, by Lemma 1,  $\text{dist}(e.\text{current}) > \text{dist}(e.\text{start})$  and so  $e$  is waiting for a conditional jump back.

Deleting the current node leads to Regular Case 2, Degenerate Case 1b or Degenerate Case 2, if the current node appears right before the insertion node:

- If  $\text{current} \neq \text{start}$  and  $\text{start} \neq \text{insert}$ , then

$$[\text{start}, \dots, \underline{\text{current}}, \text{insert}, \dots, \text{prev}(\text{start})]$$

becomes

$$[\text{start}, \dots, \text{last}] ++ [\text{first} = \text{current} = \text{insert}, \dots, \text{prev}(\text{start})] ++ \epsilon$$

where the first and last nodes are respectively the node right after and right before the deleted one. In this case  $e$  is in Regular Case 2 because by initialization  $e.\text{start\_to\_last} - \text{dist}(e.\text{start}) - \text{dist}(e.\text{last}) \geq 0$  and because  $\text{value}(e.\text{first}) \neq \text{value}(e.\text{start})$ .

- If  $\text{current} = \text{start}$  and  $\text{start} \neq \text{insert}$ , then

$$[\underline{\text{start} = \text{current}}, \text{insert}, \dots, \text{prev}(\text{start})]$$

becomes

$$\epsilon ++ [\text{first} = \text{current} = \text{insert} = \text{start}, \dots, \text{prev}(\text{start}) = \text{last}] ++ \epsilon$$

and  $e$  is in Degenerate Case 1b because by initialization  $e.\text{start\_to\_last}$  is negative,  $\text{value}(e.\text{first}) = \text{value}(e.\text{start})$  and  $\text{dist}(e.\text{first}) < \text{dist}(e.\text{start})$  (because  $\text{dist}(e.\text{first}) = 0$  and  $\text{dist}(e.\text{start}) > 0$  after the delete).

- If  $\text{current} \neq \text{start}$ , but  $\text{start} = \text{insert}$ , then

$$[\text{start} = \text{insert}, \dots, \underline{\text{current}} = \text{prev}(\text{start})]$$

becomes

$$[\text{first} = \text{start} = \text{insert} = \text{current}, \dots, \text{last} = \text{prev}(\text{start})] ++ \epsilon ++ \epsilon$$

and  $e$  is in Degenerate Case 2 because  $e.\text{start\_to\_last} - \text{dist}(e.\text{start}) - \text{dist}(e.\text{last}) \geq 0$  and  $\text{dist}(e.\text{first}) = \text{dist}(e.\text{start})$ , both by initialization. Also, by Lemma 1,  $\text{dist}(e.\text{current}) > \text{dist}(e.\text{start})$  and so  $e$  is waiting for a conditional jump back.

- If  $\text{current} = \text{start}$  and  $\text{start} = \text{insert}$  then since  $\text{current} = \text{prev}(\text{start})$  this means that the last node of the schedule is being deleted, making all elements passive.

**Regular Case 2** Any new nodes are added right before the insertion node, and hence appear in  $[\text{next}(\text{last}), \dots, \text{prev}(\text{first})]$ . Deleting the start node leads to Degenerate Case 1a or 1b if it is equal to the last node:

$$[\underline{\text{start} = \text{last}}] ++ [\text{first}, \dots, \text{prev}(\text{start})] ++ [\text{next}(\text{last}), \dots, \text{prev}(\text{first})]$$

leads to Degenerate Case 1a if  $\text{next}(\text{last}) \neq \text{insert}$ :

$$\epsilon ++ [\text{first}, \dots, \text{prev}(\text{start}) = \text{last}] ++ [\text{next}(\text{last}) = \text{start}, \dots, \text{prev}(\text{first})]$$

and to Degenerate Case 1b if  $\text{next}(\text{last}) = \text{insert}$ :

$$\epsilon \text{ ++ } [\text{first}, \dots, \text{prev}(\text{start}) = \text{last}] \text{ ++ } [\text{next}(\text{last}) = \text{insert} = \text{start}, \dots, \text{prev}(\text{first})]$$

In particular, the latter also holds if  $\text{next}(\text{last}) = \text{first}$  in which case the schedule looks as follows:

$$\epsilon \text{ ++ } [\text{first} = \text{start} = \text{insert}, \dots, \text{prev}(\text{start}) = \text{last}] \text{ ++ } \epsilon$$

In either case, because of Lemma 2 and because the conditions of Lemma 3 do not apply, the conditions for Degenerate Case 1a or 1b are satisfied. Deleting the first node leads to Degenerate Case 2 if it is right before the start node:

$$[\text{start}, \dots, \text{last}] \text{ ++ } [\text{first} = \text{prev}(\text{start})] \text{ ++ } [\text{next}(\text{last}), \dots, \text{prev}(\text{first})]$$

becomes

$$[\text{start} = \text{first}, \dots, \text{last}] \text{ ++ } \epsilon \text{ ++ } [\text{next}(\text{last}), \dots, \text{prev}(\text{first})]$$

and  $e$  is in Degenerate Case 2 because of Lemma 3 and because the conditions of Lemma 2 do not apply. Note that if  $\text{value}(e.\text{current}) = \text{value}(e.\text{first})$  before the deletion, then because of Lemma 1, after this operation,  $\text{dist}(e.\text{current}) > \text{dist}(e.\text{start})$  while also  $\text{value}(e.\text{current}) = \text{value}(e.\text{start})$  and so  $e$  will be waiting for a conditional jump back (while in Degenerate Case 2). Otherwise,  $\text{current}$  must have been equal to  $\text{start}$  already and the same conditions hold. Finally, activating element  $e$  cannot change the case  $e$  is in.

**Degenerate Case 1a** Since we have that  $\text{first} \preceq \text{start} \prec \text{insert}$ , all new nodes added while in Degenerate Case 1a, appear in  $[\text{next}(\text{last}), \dots, \text{prev}(\text{first})]$  and hence after the current node (for which holds that  $\text{first} \preceq \text{current} \preceq \text{start}$ ). Deleting the first node leads to Degenerate Case 3 if it is right before the start node:

$$\epsilon \text{ ++ } [\text{first} = \text{last} = \text{prev}(\text{start})] \text{ ++ } [\text{next}(\text{last}), \dots, \text{prev}(\text{first})]$$

becomes

$$\epsilon \text{ ++ } \epsilon \text{ ++ } [\text{start}, \dots, \text{prev}(\text{start})]$$

and  $e$  is in Degenerate Case 3 because of Lemma 2 which remains applicable, and because of Lemma 3. If  $\text{value}(e.\text{current}) = \text{value}(e.\text{first})$  before deletion, then afterwards, because of Lemma 1,  $\text{dist}(e.\text{current}) > \text{dist}(e.\text{start})$  while also  $\text{value}(e.\text{current}) = \text{value}(e.\text{start})$  and so  $e$  will be waiting for a conditional jump back (while in Degenerate Case 3). Otherwise,  $\text{current}$  must have been equal to  $\text{start}$  already and the same conditions hold.

Deleting the start node while it appears right before the insertion node, leads to Degenerate Case 1b if  $\text{start} \neq \text{current}$ :

$$\epsilon \text{ ++ } [\text{first}, \dots, \text{last} = \text{prev}(\text{start})] \text{ ++ } [\text{next}(\text{last}) = \text{start}, \text{insert}, \dots, \text{prev}(\text{first})]$$

becomes

$$\epsilon ++ [\text{first}, \dots, \text{last} = \text{prev}(\text{insert})] ++ [\text{insert} = \text{start}, \dots, \text{prev}(\text{first})]$$

and  $e$  is in Degenerate Case 1b because Lemma 2 still applies,  $\text{first} \prec \text{insert} \preceq \text{start}$  and Lemma 3 does not apply. If on the other hand  $\text{start} = \text{current}$  (and  $\text{dist}(\text{start}) < \text{dist}(\text{current})$  because  $\text{start} \prec \text{insert} \preceq \text{first}$ ), then

$$\epsilon ++ [\text{first}, \dots, \text{last} = \text{prev}(\text{start})] ++ \underline{[\text{next}(\text{last}) = \text{start} = \text{current}, \text{insert}, \dots, \text{prev}(\text{first})]}$$

becomes

$$[\text{start} = \text{first}, \dots, \text{last}] ++ \epsilon ++ [\text{next}(\text{last}), \dots, \text{prev}(\text{first})]$$

with  $e$  in Degenerate Case 2 because of the actions taken after  $e$  passes the insertion point for the second time. Afterwards,  $e$  is still waiting for a conditional jump back, now in Degenerate Case 2.

Finally, again, activating element  $e$  cannot change the case  $e$  is in.

**Degenerate Case 1b** New nodes added while in Degenerate Case 1b appear right before the schedule's insertion node, and hence after the current node. Deleting the current node leads to Degenerate Case 2 or 3 if it appears right before the schedule's insertion node. Lemma 2, together with the actions taken when the current node passes the schedule's insertion point, ensures this is the case. If  $\text{current} \neq \text{first}$  then

$$\epsilon ++ [\text{first}, \dots, \underline{\text{current} = \text{prev}(\text{insert})}] ++ [\text{insert}, \dots, \text{prev}(\text{first})]$$

becomes (Degenerate Case 2)

$$[\text{first} = \text{start} = \text{current}, \dots, \text{last}] ++ \epsilon ++ \underline{[\text{next}(\text{last}) = \text{insert}, \dots, \text{prev}(\text{first})]}$$

where  $\text{last}$  is the node right before the deleted one and  $\text{dist}(e.\text{current}) > \text{dist}(e.\text{start})$  by initialization, so  $e$  is waiting for a conditional jump back. Moreover,  $\text{value}(e.\text{first}) = \text{value}(e.\text{start})$  and  $\text{dist}(e.\text{first}) = \text{dist}(e.\text{start})$ . If  $\text{current} = \text{first}$  then

$$\epsilon ++ \underline{[\text{first} = \text{current} = \text{prev}(\text{insert})]} ++ [\text{insert}, \dots, \text{prev}(\text{first})]$$

becomes (Degenerate Case 3)

$$\epsilon ++ \epsilon ++ [\text{start} = \text{current} = \text{first} = \text{insert}, \dots, \text{prev}(\text{start})]$$

and by initialization, both  $\text{dist}(\text{first}) = \text{dist}(\text{start})$  and  $e.\text{start\_to\_last} < 0$ . Also, again  $\text{dist}(\text{current}) > \text{dist}(\text{start})$  causing  $e$  to be waiting for a conditional jump back. If  $e$  is activated while its current node is right before the insertion node, then Degenerate Case 1b becomes Degenerate Case 2. This case is similar to the above case of deleting the current node while it is right before the insertion node and different from the first node.

**Degenerate Case 2** New nodes added while in Degenerate Case 2 appear in  $[\text{next}(\text{last}), \dots, \text{prev}(\text{first})]$ . Deleting the start node leads to Degenerate Case 3 if it is equal to the last node:

$$[\text{start} = \text{last} = \text{first} = \text{current}] \text{ ++ } \epsilon \text{ ++ } [\text{next}(\text{last}), \dots, \text{prev}(\text{first})]$$

leads to

$$\epsilon \text{ ++ } \epsilon \text{ ++ } [\text{start}, \dots, \text{prev}(\text{start})]$$

and  $e$  is in Degenerate Case 3 because of Lemma 2, and because Lemma 3 still applies. In Degenerate Case 2, the conditions of Lemma 1 apply, and this remains so in Degenerate Case 3. Activating element  $e$  keeps the element in Degenerate Case 2.

**Degenerate Case 3** This case is detected because of Lemmas 1, 2 and 3. No operations on nodes (apart from deleting the last node in the schedule) has an effect in this case. In case of a conditional jump back, the degenerate case is transformed into Regular Case 1 (because the `first` and `last` pointers are reset). Note that no regular element activation is possible while  $e$  is in Degenerate Case 3.

Since the invariants ensure that all nodes with which  $e$  has been matched, are in the *Seen* part of the schedule view, and all other nodes are in the *Unseen* part, and furthermore no element can become passive if this *Unseen* part is not empty, the invariants imply the theorem.  $\square$

## 5 Complexity Analysis

In this section, we analyze the time complexity of the different operations supported by the described data structure, as well as its space complexity.

### 5.1 Time Complexity

The main time complexity result for our algorithm is as follows:

**Theorem 2.** *Each of the following operations takes amortized quasi constant<sup>14</sup> time: inserting a new node, deleting a node, inserting a new element, deleting an element, activating an element, and merging two schedules. Furthermore, the number of times an element is unsuccessfully activated (i.e., it is unable to match with a node), is also constant (amortized).*

*Proof.* We look at each of the operations in detail.

**Node Insertion** This operation requires looking up the new node's schedule, or creating such a schedule if no such exists. A new node data structure is created, the schedule's insertion node, as well as the node right before

<sup>14</sup> They take constant time under the assumption that the inverse Ackermann function is constant, and that all operations on distances take constant time (see Section 5.2).

it, are retrieved (`find`). Then, the `prev` and `next` pointers of the insertion node, the node right before it, and the new node, are updated using `make` and `union` operations. Finally, all elements that were passive for the new node's schedule, are rescheduled. This requires setting the elements' `start` and `current` pointers to the appropriate values using `make_and_union` operations. Because of the element schedule data structure, we can update these pointers for all rescheduled elements at once, using only two `make_and_union` calls.

**Node Deletion** When a node is deleted, its previous and next node are retrieved (`find`) and its `prev` and `next` pointers are subsequently updated using `union` calls. If the deleted node is right before the schedule's insertion node, the elements whose current node equals the deleted node, and which have not passed the insertion point more than once since their last jump back or (re-)scheduling, are updated as follows. For those elements that pass the insertion point for the first time, the `first` and `last` pointers are initialized using `make_and_union` calls. Also, the `start_to_last` and `last_to_first` fields are initialized, the first of which requires retrieving the distance of the `current` pointer (`find`). The elements that pass the insertion point for the second time have their `start`, `current`, `first` and `last` fields updated using `make` and `union` operations, requiring the distances of the `first` and `current` pointers (`find`). The `start_to_last` and `last_to_first` fields are updated in a similar way. Each of the above updates (of element fields) can be done in quasi constant time for each set of elements that have been simultaneously rescheduled and have not been activated since. Hence, for those sets, we can amortize the cost to the event that caused the rescheduling (a node insertion or schedule merge). For the other elements, we can amortize the cost to at least one activation of each element in question. Note that it is crucial for complexity that only those elements that have not passed their insertion point more than once (since the last jump back or (re-)scheduling) are considered. This can be accomplished using a data structure similar to the element schedule, that assigns elements that have passed their insertion point at most once, to their current node. Such a data structure can be maintained easily. Finally, if the deleted node is the last node in the schedule, all scheduled elements are made passive. Like above, the cost of this operation can be amortized to earlier node insertions, schedule merges and element activations.

**Merging Schedules** Merging two schedules requires retrieving their insertion nodes and the nodes right before them (`find`). Then, these nodes have their `prev` and `next` pointers updated, using `make` and `union` operations. As with node insertion, all elements that were passive in one schedule have to be rescheduled for the other, and vice versa. This requires setting the affected elements' `start` and `current` pointers to the appropriate values using `make` and `union` operations. Because of the element schedule data structure, we can update these pointers for all rescheduled elements at once, using only two `make_and_union` calls per schedule.

**Element Insertion** For this operation, we create a new element data structure, lookup the relevant schedule for this element and retrieve its insertion node (`find`). The new element's `start` and `current` fields are initialized using

forward pointers to the insertion node which requires two `find` operations, as well as a `make_and_union` call. The element is also inserted into the element schedule data structure, and the list of elements whose current node equals the schedule's insertion node.

**Element Activation** When activating an element, we first check whether this element should be matched with its current node, or a conditional jump back is to be performed. This check consists of comparing the start and current node and the distances of the respective pointers (`find`). The next paragraph deals with the case of a conditional jump back. This paragraph deals with the normal case. A normal element activation creates a new match between the activated element and its current node, the cost of which is application dependent and outside the scope of our complexity analysis. It furthermore requires advancing the current pointer which is done using a `make`, `union` and `find` operation. If the activation causes the element in question to pass its schedule's insertion point, the same is done as in the corresponding case of node deletion, i.e., the `first` and `last`, and/or `start` and `current` pointers are updated using a constant number of `make`, `union` and `find` operations, and the `start_to_last` and `last_to_first` fields are updated using some `find` calls.

**Conditional Jump Back** If an element performs a conditional jump back, it is checked whether it is in Degenerate Case 3 or not (see Section 4.2) by comparing the distances of its `start` and `last` fields with its `start_to_last` field, and by comparing the (distances of the) `first` and `start` fields (`find`). In this degenerate case, we perform the same actions as in the case of an element insertion, followed by an element activation.

In the normal case, it is checked whether the element in question needs to be made passive or not, by comparing the node after the last node, and the first node (`find`). If the element is not made passive, an actual jump back is performed, which consists of updating the element's `start` and `current` fields using `make`, `union`, and `find` operations, and resetting the other fields.

In either case, if the element performing the conditional jump back, has not been successfully activated since it was (re-)scheduled, all elements that were simultaneously scheduled are dealt with in the same way (apart from the activation following a jump back), using a single operation which hence also takes quasi constant time.

We conclude that each of the operations can be performed in quasi constant amortized time. We note that each element that is activated without being able to match with a node, is made passive. If such an element has not been successfully activated since it was (re-)scheduled, then all elements that were simultaneously scheduled, are made passive as well (in a single operation). Therefore, we can amortize the cost of unsuccessful activations to either the event that caused the element in question to be (re-)scheduled (i.e., a node insertion or schedule merge), or to a previous successful element activation. This proves the second part of the theorem.  $\square$

## 5.2 Space Complexity

In our analysis of the data structure’s space complexity, we make the following assumptions: the integers used to represent distances, weights and ranks<sup>15</sup> all take constant space, and the memory used by union-find elements that have become garbage, can be reclaimed instantly. Under these assumptions, the total space complexity of our data structure equals  $\mathcal{O}(N + E)$  where  $N$  is the number of nodes and  $E$  is the number of elements in the schedule. The reasoning is as follows. Each node and each element only has a constant number of fields, each of which contains a single union-find element or distance. Each union-find element contains a pointer or a name, a rank, and a weight (see Appendix A), all of which take constant space under the above assumptions. Unfortunately, these assumptions do not always hold because on the one hand, distances may grow to unbounded size over time (which has its effects on time complexity as well), and on the other hand, the memory used by discarded union-find elements cannot always be reclaimed as other union-find elements may still refer to them. However, our results are still valid (on any practical machine) if we perform at regular intervals, the memory optimization procedure described below.

To clean up discarded union-find elements, we can traverse the union-find data structure, applying the find operation on each element, after which all unused union-find elements can be removed.<sup>16</sup> We can mark these union-find elements the moment they become garbage.

The distances can be reduced as follows. In a first phase, we traverse all scheduled elements and ensure that they are in a regular (non-degenerate) case by checking the degeneracy conditions (Section 4.2) and performing the necessary updates (i.e., making the element in question passive, performing a jump back, or resetting the `first` and `last` fields). The cost of this first phase is  $\mathcal{O}(E)$  where  $E$  is the number of elements in the schedule. Next, we traverse the circular list of nodes, assigning consecutive numbers to each node encountered. The distances of the nodes’ next and previous pointers are reset to zero during this process. The cost of the second phase is  $\mathcal{O}(N)$  where  $N$  is the number of nodes in the schedule. In a third phase, all elements are traversed again. For each of the elements, the distances of the `start`, `current`, `first` and `last` node pointer, as well as the `start_to_last` and `last_to_first` fields are recomputed by using the numbers assigned to each of the nodes.<sup>17</sup> We also update the schedule’s round trip distance in a similar way. In total, the third phase takes  $\mathcal{O}(N)$  time to complete.

The runtime complexity of both clean-up operations is  $\mathcal{O}(N_a + U)$  where  $N_a$  is the number of active (non-deleted) nodes in the schedule, and  $U$  is the number

---

<sup>15</sup> The latter two are used internally by the extended union-find algorithm, see Appendix A for more details.

<sup>16</sup> More precisely, all but one for each union-find set with at least one used element, if the root of this set is unused.

<sup>17</sup> We can easily change a union-find element’s distance after having cleaned up the discarded elements, since at that point all remaining union-find elements directly point to the same (root) element. However, a new union-find element needs to be created when updating the distance of this root element.

of union-find elements, including unused ones. Since each schedule element corresponds to at least two union-find elements, we have that  $E = \mathcal{O}(U)$ . In absence of the memory optimization step, the memory overhead caused by discarded union-find elements and unbounded distances is  $\mathcal{O}(U_a \log N_d + U_d \log N)$  where  $U_a$  are the active (used) union-find elements,  $N_d$  is the number of deleted nodes (since the last clean-up),  $U_d$  is the number of discarded union-find elements,  $N = N_a + N_d$  and  $U = U_a + U_d$ . The first component of the memory overhead, namely  $U_a \log N_d$ , is caused by using suboptimal distances for the active union-find elements. Note that no distance can be larger than the optimal one by more than the number of deleted nodes.<sup>18</sup> The second component, namely  $U_d \log N$ , is caused by the discarded union-find elements. The remaining memory requirement for the distances after clean-up is  $\mathcal{O}(U_a \log N_a)$ .

Now, let  $N_d + U_d = U_a$ , then because each non-deleted node requires at least two (active) union-find elements for its previous and next pointers,  $N_a = \mathcal{O}(U_a)$ , and so we have that the cost of performing a clean-up ( $\mathcal{O}(N_a + U_a + U_d)$ ) is proportional to the number of operations performed since the last clean-up (which is at least  $N_d + U_d$ ). In this case, the memory overhead before clean-up is  $\mathcal{O}(U_a \log U_a)$  because also  $N_d = \mathcal{O}(U_a)$ . Since  $U_a$  is bounded by the available memory,  $\log U_a$  is bounded by the word size used for storing memory addresses, which is a constant on any practical machine.<sup>19</sup>

Given the fact that there are at most four active union-find elements per schedule element (pointed to by the `start`, `current`, `first` and `last` fields), and at most two per node (for the `prev` and `next` fields), we have that the memory use is  $\mathcal{O}(N + E)$  on any machine with a bounded address space. That is, if we perform the above memory optimization whenever the number of deleted nodes and discarded union-find elements becomes larger than the number of actively used union-find elements.

## 6 Application: Matching for Constraint Handling Rules

Constraint Handling Rules (CHR [5]) is a rule based language, originally designed for the implementation of Constraint (Logic) Programming systems, and increasingly used as a general purpose programming language. CHR runs on top of a host language, which offers an underlying constraint solver that implements the *built-in* constraints. CHR rules are multi-headed and operate on a multi-set database called the *constraint store*. Most current implementations of CHR use a lazy matching technique, similar to that of the LEAPS algorithm [8]. CHR

<sup>18</sup> Also, because each union-find element has a distance between zero and  $N$ , and each weight used internally in the union-find algorithm (see Appendix A) represents the difference between two such distances, all weights are in the range  $[-N, N]$ .

<sup>19</sup> Note that similar assumptions are implicitly made in the analysis of the union-find algorithm in [12], where it is shown that the space needed for storing ranks or sizes is respectively  $\mathcal{O}(\log \log n)$  and  $\mathcal{O}(\log n)$  (with  $n$  the number of union-find elements) whereas comparing and adding these numbers is considered to take constant time.

constraints are not necessarily ground and so the assertion of a new built-in constraint may create new rule instances.

When using CHR on top of Prolog, the basic built-in constraint is (syntactic) equality between Herbrand terms, implemented using the Prolog unification algorithm. After a unification, all affected constraints are reactivated. Each of them starts a search for partner constraints, trying again rule instances that have fired before. To prevent the actual firing from happening more than once, a so-called *propagation history* is maintained, which consists of the identifiers of all combinations of constraints that have already fired a particular rule.

In [1], it is proposed to use our scheduling data structure for the implementation of a lazy RETE-style matching algorithm for CHR with rule priorities [2]. The approach taken there consists of creating a representation of the RETE matching network as CHR constraints, and using the scheduler to incrementally create new partial and full matches, by combining already found partial matches with CHR constraints. Some of the advantages of the approach include a considerably cheaper treatment of built-in constraints and the ability to reuse previously generated partial matches. One of the disadvantages is that the join order [3], i.e., the order in which the different heads of a multi-headed rule are matched with, is fixed on a per rule basis instead of per rule head.

## 7 Conclusion

We have proposed a new data structure and algorithm to maintain schedules for a specific matching problem (somewhat resembling a job shop problem), that supports the addition and deletion of elements to be matched, and merging of such schedules. The results can be used to support a form of lazy matching with storage of partial matches in the context of non-ground data. The data structure is designed such that all relevant operations can be performed in quasi constant time, and therefore it is in a sense quasi-optimal. However, a considerable amount of overhead is introduced, in the form of extra indirections and information maintenance, in order to achieve this optimality. Therefore, it is likely that a (complexity-wise) suboptimal relaxation of this work yields better results in the average case. Still, the results are interesting because they show what is feasible.

In the specific context of matching for the Constraint Handling Rules language, we can get rid of the bookkeeping involving distances by using a propagation history instead. This propagation history deviates from the normal one in that it is kept at the level of partial matches. Instead of the various tests involving distances when activating an element, we need to check whether the resulting match has been previously generated or not. In fact this approach would cause the data structure for elements to be reduced to only two fields, namely *current* and *last*. The approach presented in this paper is somewhat more complicated, but on the other hand also more general in that it works independent of the semantics of matching an element with a node.

## 7.1 Related Work

In the context of the production rule system Drools [10], a lazy version of RETE has been proposed in [9], which resembles our approach in absence of schedule merges, but presumably needs some extensions to support negated heads. A similar proposal is made in [6] for a bottom-up logic programming language that is in a sense a restriction of Constraint Handling Rules with rule priorities with ground constraints only (as shown in [1]). Schedule merges seem to be needed only in the context of non-ground data. We have not found any other work that proposes a lazy RETE style of matching in this context.

To the best of our knowledge, our data structure is the first to implement the incremental, demand driven, generation of a schedule (i.e., a sequence of matches) for problems like the job shop problem (in absence of a cost function), supporting the insertion and deletion of elements, as well as schedule merges, during schedule generation. We note that the data structure is designed with a very specific application in mind, and that we are yet to find application areas outside of this scope. However, some of the ideas that were used, in particular the implementation of simultaneous updates to a set of pointers by means of the (optimal) union-find algorithm, can be applied to a wider context, for example in the implementation of iterators over linked lists.

## 7.2 Future Work

In future work, we intend to implement the data structure and algorithm in the context of matching for the Constraint Handling Rules language, and investigate the trade-off between constant factors and optimality. This will allow a comparison between the usual LEAPS-style matching used by most CHR implementations, and a lazy RETE-style matching. While it is to be expected that eager (regular) RETE matching is suboptimal, this is not necessarily so for a lazy version of the algorithm, in particular if the alternative (i.e., LEAPS matching) requires the use of a propagation history anyway.

## References

1. Leslie De Koninck. Logical Algorithms meets CHR: A meta-complexity result for Constraint Handling Rules with rule priorities. Submitted to Theory and Practice of Logic Programming, 2007.
2. Leslie De Koninck, Tom Schrijvers, and Bart Demoen. User-definable rule priorities for CHR. In Micheal Leuschel and Andreas Podelski, editors, *9th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 25–36. ACM Press, 2007.
3. Leslie De Koninck and Jon Sneyers. Join ordering for Constraint Handling Rules. In *4th Workshop on Constraint Handling Rules*, pages 107–121, 2007.
4. Charles L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17–37, 1982.
5. Thom Frühwirth. Theory and practice of Constraint Handling Rules. *Journal of Logic Programming*, 37(1-3):95–138, 1998.

6. Harald Ganzinger and David A. McAllester. Logical algorithms. In *18th International Conference on Logic Programming*, volume 2401 of *Lecture Notes in Computer Science*, pages 209–223. Springer, 2002.
7. Daniel P. Miranker. TREAT: A better match algorithm for AI production system matching. In *6th National Conference on Artificial Intelligence*, pages 42–47. AAAI Press, 1987.
8. Daniel P. Miranker, David A. Brant, Bernie Lofaso, and David Gadbois. On the performance of lazy matching in production systems. In *8th National Conference on Artificial Intelligence*, pages 685–692. AAAI Press / The MIT Press, 1990.
9. Mark Proctor. Rete with lazy joins. <http://blog.athico.com/2006/11/rete-with-lazy-joins.html>, 2006.
10. Mark Proctor, Michael Neale, Michael Frandsen, Sam Griffith, Jr, Edson Tirelli, Fernando Meyer, and Kris Verlaenen. *Drools Documentation, Version 4.0.3*, 2007. <http://www.jboss.com/products/rules>.
11. Robert E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975.
12. Robert E. Tarjan and Jan van Leeuwen. Worst-case analysis of set union algorithms. *Journal of the ACM*, 31(2):245–281, 1984.

## A Implementation of Optimal Union-Find with Distances

This section shows how the optimal union-find algorithm can be extended so that it also maintains distances without increasing the overall complexity. We first review the union-find algorithm with link by rank and compression.

Each disjoint set is represented as a tree whose nodes represent the elements in the set. A non-root node is a data structure with the following fields: `parent` and `rank`. The `parent` field contains a pointer to the node's parent in the tree, and the `rank` field contains the *rank* of the node, which is defined as the height of the subtree of which the node is the root. A root node has a `name` field instead of a `parent` field, the contents of which is the name of the set to which it belongs.

In the naive union-find algorithm, the `make` function creates a new root node, the `find` function is implemented by following parent pointers to the root node and returning the name of this node, and finally, the `union` operation consists of finding the root nodes of both sets, and linking the root node corresponding to the second input argument, to the one corresponding to the first. The optimal union-find algorithm uses two optimizations. The first is called *compression* and consists of directly pointing all nodes traversed to their root node. The second one, called *link by rank*, links the root node with the smallest rank to the one with the highest rank in each union operation. Below, the optimal union-find algorithm is given in pseudo code.

```
1  function make(Name name) = Node
2  {
3      RootNode root;
4      root.name := name;
5      root.rank := 0;
6      return root;
7  }
8  function findRoot(Node node) = RootNode
9  {
10     if (root(node))
11     {
12         return node;
13     }
14     else
15     {
16         RootNode root := findRoot(node.parent);
17         node.parent := root; // compression
18         return root;
19     }
20 }
21 function find(Node node) = Name
22 {
23     return findRoot(node).name;
24 }
25 procedure union(Node node1, Node node2)
26 {
```

```

27     RootNode root1 := findRoot(node1);
28     RootNode root2 := findRoot(node2);
29     // link by rank
30     if (root1.rank < root2.rank)
31     {
32         Name name := root1.name;
33         make_non-root(root1);
34         root1.parent := root2;
35         root2.name := name; // keep name of first set
36     }
37     else
38     {
39         make_non-root(root2);
40         root2.parent := root1;
41         root1.rank := max(root1.rank,root2.rank + 1);
42     }
43 }

```

We now show how the above code is extended to support distances. We implement distances by assigning a *weight* to each union-find node. The *distance* of a node then corresponds to the sum of the *weights* of the nodes on the path from the given node to its root node. When a new set is made, its root node gets as weight the distance given. A find operation may change this path due to compression, and so it may require updating the weights of the nodes on the path to compensate for the skipped nodes. Finally, when a union operation takes place, the weights are updated such that all nodes in the set to which the first argument belongs keep their distance and all nodes in the set to which the second argument belongs have their distance increased by the distance of the first argument. This only requires an appropriate weight for the original root nodes. The resulting code is shown below.

```

1  function make(Name name, int distance) = Node
2  {
3      RootNode root;
4      root.name := name;
5      root.rank := 0;
6      root.weight := distance; // initialize weight
7      return root;
8  }
9  function findRoot(Node node) = <RootNode,int>
10 {
11     // also returns the distance of the node
12     if (root(node))
13     {
14         return <node,node.weight>;
15     }
16     else
17     {
18         RootNode root;
19         int distance;

```

```

20     <root,distance> := findRoot(node.parent);
21     node.parent := root; // compression
22     node.weight := node.weight + distance - root.weight;
23     return <root,node.weight + root.weight>;
24 }
25 }
26 function find(Node node) = <Name,int>
27 {
28     RootNode root;
29     int distance;
30     <root,distance> := findRoot(node);
31     return <root.name,distance>;
32 }
33 procedure union(Node node1, Node node2)
34 {
35     RootNode root1, root2;
36     int distance1;
37     <root1,distance1> := findRoot(node1);
38     <root2,_> := findRoot(node2);
39     // link by rank
40     if (root1.rank < root2.rank)
41     {
42         Name name := root1.name;
43         make_non-root(root1);
44         root2.name := name; // keep name of first set
45         root2.weight := root2.weight + distance1 + 1;
46         root1.weight := root1.weight - root2.weight;
47         root1.parent := root2;
48     }
49     else
50     {
51         make_non-root(root2);
52         root2.parent := root1;
53         root2.weight := root2.weight + distance1 + 1 - root1.weight;
54         root1.rank := max(root1.rank,root2.rank + 1);
55     }
56 }

```

It is easy to see that the above code has the same time complexity as the original code since each of the distance calculations can be performed in constant time and because the distances do not influence the rest of the control flow.

## B Pseudo Code

In this section, we give a listing in pseudo code of the complete algorithm apart from those aspects related to the element schedule. We make use of the union-find data structure as presented in Section 3.1.

```

1 // Node Insertion
2 procedure insert_node(Node n, Schedule s)

```

```

3 {
4   Node p;
5   if (s.insert = null) // no nodes in schedule
6   {
7     // initialize prev and next fields
8     n.prev := make(n,0);
9     n.next := make(n,0);
10    // initialize insertion node
11    s.insert := n.next;
12    s.round_trip := 1;
13    p := n;
14  }
15  else
16  {
17    // insert new node right before insertion node
18    Node i := value(s.insert);
19    p := value(i.prev);
20    int d = dist(i.prev);
21    n.next := make_and_union(p.next,0);
22    n.prev := i.prev;
23    i.prev := make(n,0);
24    p.next := make(n,d);
25    s.round_trip := s.round_trip + 1;
26  }
27  // reschedule passive elements
28  foreach e in s.passive_set
29  {
30    e.start := make_and_union(n.next,0);
31    e.current := make_and_union(p.next,s.round_trip - 1);
32  }
33  n.current_set := s.passive_set;
34  s.passive_set := empty_set;
35 }
36 // Node Deletion
37 procedure delete_node(Node d, Schedule s)
38 {
39   Node p := value(d.prev);
40   Node n := value(d.next);
41   if (n = d) // last node in schedule: make all elements passive
42   {
43     foreach e in d.current_set
44     {
45       e.start := e.current := e.first := e.last := null;
46       e.start_to_last := e.last_to_first := 0;
47     }
48     s.passive_set := set_union(s.passive_set,d.current_set);
49     // reset insertion node
50     s.insert := null;
51   }
52   else

```

```

53     {
54         if (n = value(s.insert)) // elements pass insertion point
55         {
56             foreach e in d.current_set
57             {
58                 if (e.first = null) // pass for the first time
59                 {
60                     if (n = value(e.start)) // degenerate case 1b
61                         e.first := e.start;
62                     else // regular case 2
63                         e.first := make_and_union(d.next,0);
64                     e.last := make_and_union(d.prev,0);
65                     e.start_to_last := dist(e.current) - 1 -
66                         dist(d.prev);
67                     e.last_to_first := dist(d.prev) + 1 + dist(d.next);
68                     add e to n.current_set;
69                 }
70                 else // pass for the second time: degenerate case 1b
71                 {
72                     // new start_to_last distance equals
73                     // former distance from first node to the node
74                     // before the deleted one
75                     e.start_to_last := dist(e.current) -
76                         e.start_to_last - 1 - e.last_to_first -
77                         dist(e.first) - 1 - dist(d.prev);
78                     e.last_to_first := 0;
79                     e.start := e.first := make_and_union(e.first,0);
80                     e.last := make_and_union(d.prev,0);
81                     if (e.start_to_last >= 0) // degenerate case 2
82                         e.current := make(_,e.start_to_last);
83                     else // degenerate case 3: ensure distance is positive
84                         e.current := make(_,0);
85                     union(e.start,e.current);
86                     // now dist(e.start) < dist(e.current)
87                     // do not add e to its current node's current set
88                 }
89             }
90         }
91         else // elements do not pass insertion point
92             n.current_set := set_union(n.current_set,d.current_set);
93         // finally link nodes p and n
94         union(d.next,p.next);
95         union(d.prev,n.prev);
96     }
97 }
98 // Merging Schedules
99 function merge_schedules(Schedule s1, Schedule s2) = Schedule
100 {
101     Node i1 := value(s1.insert);
102     Node i2 := value(s2.insert);

```

```

103     Node p1 := value(i1.prev);
104     Node p2 := value(i2.prev);
105     int d1 := dist(i1.prev);
106     int d2 := dist(i2.prev);
107     // crosslink circular lists
108     UnionFindElement p1next := p1.next;
109     UnionFindElement p2next := p2.next;
110     p1.next := make_and_union(p2next,d1);
111     p2.next := make_and_union(p1next,d2);
112     UnionFindElement i1prev := i1.prev;
113     UnionFindElement i2prev := i2.prev;
114     i1.prev := i2prev;
115     i2.prev := i1prev;
116     // reschedule passive elements of schedule s1
117     foreach e in s1.passive_set
118     {
119         e.current := make_and_union(p1.next,s1.round_trip);
120         e.start := make_and_union(p2.next,0);
121         e.first := e.last := null;
122         e.start_to_last := e.last_to_first := 0;
123     }
124     i2.current_set := set_union(i2.current_set,s1.passive_set);
125     // reschedule passive elements of schedule s2
126     foreach e in s2.passive_set
127     {
128         e.current := make_and_union(p2.next,s2.round_trip);
129         e.start := make_and_union(p1.next,0);
130         e.first := make_and_union(p2.next,0);
131         e.last := make_and_union(i1.prev,0);
132         e.start_to_last := s2.round_trip - 1 - d2;
133         e.last_to_first := d2;
134     }
135     i1.current_set := set_union(i1.current_set,s2.passive_set);
136     // create resulting schedule and initialize its fields
137     Schedule s;
138     s.insert := s1.insert;
139     s.round_trip := s1.round_trip + s2.round_trip;
140     s.passive_set := empty_set;
141     return s;
142 }
143 // Element Insertion
144 procedure insert_element(Element e, Schedule s)
145 {
146     if (s.insert = null) // no nodes in schedule
147         add e to s.passive_set; // make element passive
148     else
149     {
150         // schedule element
151         Node i := value(s.insert);
152         Node p := value(i.prev);

```

```

153         e.start := e.current := make_and_union(p.next,0);
154         add e to i.current_set;
155     }
156 }
157 // Element Deletion
158 procedure delete_element(Element e, Schedule s)
159 {
160     if (e in s.passive_set) // element is passive
161         remove e from s.passive_set;
162     if (e in value(e.current).current_set)
163         remove e from value(e.current).current_set;
164 }
165 // Element Activation
166 procedure activate(Element e, Schedule s)
167 {
168     if (e in value(e.current).current_set)
169         remove e from value(e.current).current_set;
170     if (value(e.current) = value(e.start) and
171         dist(e.current) > dist(e.start))
172     {
173         // conditional jump back
174         if (e.start_to_last - dist(e.start) - dist(e.last) < 0 and
175             value(e.first) = value(e.start) and
176             dist(e.first) >= dist(e.start))
177         {
178             // degenerate case 3: reset all fields and reactivate
179             Node i := value(s.insert);
180             Node p := value(i.prev);
181             e.start := e.current := make_and_union(p.next,0);
182             add e to i.current_set;
183             activate(e,s);
184         }
185     else
186     {
187         if (value(value(e.last).next) = value(e.first))
188         {
189             // no unseen nodes: make e passive
190             e.start := e.current := e.first := e.last := null;
191             e.start_to_last := e.last_to_first := 0;
192             add e to s.passive_set;
193         }
194     else
195     {
196         // jump back
197         // new distance for the current pointer equals
198         // former distance from first to last node (via start)
199         // plus 1 + distance from last node to next to last node
200         int d;
201         if (e.start_to_last - dist(e.start) - dist(e.last) >= 0)
202         {

```

```

203         // regular case 2 or degenerate case 2
204         d = dist(e.current) - dist(e.start) - dist(e.last) -
205             1 - e.last_to_first - dist(e.first);
206     }
207     else
208     {
209         // degenerate case 1a
210         d = dist(e.current) - e.start_to_last - 1 -
211             e.last_to_first - dist(e.first);
212     }
213     e.current := make(_,d);
214     union(value(e.last).next,e.current);
215     e.start := make_and_union(e.first,0);
216     if (value(e.current) = value(s.insert))
217     {
218         // jump back to right after insertion point
219         e.first := make_and_union(e.current,0);
220         e.last := make_and_union(e.last,0);
221         e.start_to_last := d;
222         e.last_to_first := dist(value(e.last).next);
223     }
224     else
225     {
226         e.first := e.last := null;
227         e.start_to_last := e.last_to_first := 0;
228     }
229     add e to value(e.current).current_set;
230     activate(e,s);
231 }
232 }
233 }
234 else
235 {
236     // regular activation
237     Node c := value(e.current);
238     Node n := value(c.next);
239     match e with c;
240     if (n = value(s.insert))
241     {
242         // element passes insertion point
243         if (e.first = null) // pass for the first time
244         {
245             // initialize fields
246             e.first := make_and_union(c.next,0);
247             e.last := make_and_union(n.prev,0);
248             e.start_to_last := dist(e.current);
249             e.last_to_first := dist(c.next);
250             // advance element to next node
251             remove e from c.current_set;
252             e.current := make(_,dist(e.current));

```

```

253         union(c.next,e.current);
254         add e to n.current_set;
255     }
256     else // pass for the second time
257     {
258         // degenerate case 1b becomes degenerate case 2
259         // new start_to_last distance equals
260         // former distance from first to current node
261         e.start_to_last := dist(e.current) - e.start_to_last -
262             e.last_to_first - 1 - dist(e.first);
263         e.last_to_first := 0;
264         e.start := e.first := make_and_union(e.first,0);
265         e.last := make_and_union(n.prev,0);
266         e.current := make(_,e.start_to_last);
267         union(e.start,e.current);
268         // do not add e to its current node's current set
269     }
270 }
271 else
272 {
273     e.current := make(_,dist(e.current));
274     union(c.next,e.current);
275     add e to n.current_set;
276 }
277 }
278 }

```