MISTA 2009

A local approach to automated correction of violated precedence and resource constraints in manually altered schedules

Roman Barták • Tomáš Skalický

Abstract Users of automated scheduling systems frequently require an interactive approach to scheduling where they can manually modify the schedules. Because of complexity and cohesion of scheduling relations, it may happen that manual modification introduces flaws to the schedule, namely the altered schedule violates some constraints such as precedence relations or limited capacity of resources. It is useful to automatically correct these flaws while minimizing other required changes of the schedule. In this paper we suggest a fully automated approach to correcting violated precedence and unary resource constraints. The presented techniques attempt to alter minimally the existing schedule by doing the changes only locally in the area of the flaw.

1 Introduction

Fully automated scheduling seems like the Holy Grail of scheduling community, but most practitioners frequently require the freedom of manually altering the generated schedules, for example, to introduce some aspects of the particular area that were hard to formalize and hence are not reflected in the automatically generated schedule. Despite the high experience of human schedulers, there is a high probability that after a manual modification some flaws are introduced to the schedule. This probability is higher if the density of scheduling constraints is large and the constraints are highly coupled. For example, delaying one activity may delay other dependent activities due to precedence constraints between them or due to limited capacity of resources. It might be enough just to detect such violations and report them to the user who will be responsible for manual correction. Nevertheless, such manual corrections may be boring and sometimes very hard because of interconnectivity of the constraints (correction of one flaw introduces other flaws etc.). Hence we suggest a fully automated ("push button") approach to correcting schedules after manual modification. Namely, we address the problem of correcting precedence constraints and unary resource constraints by shifting locally the affected activities in time.

Roman Barták
Charles University in Prague, Faculty of Mathematics and Physics
E-mail: bartak@ktiml.mff.cuni.cz

Tomáš Skalický
Charles University in Prague, Faculty of Mathematics and Physics
E-mail: skalicky.tomas@gmail.com

This paper studies the problem of correcting general schedules consisting of a set of non-interruptible activities with fixed durations; each activity is allocated to one or several unary resources (at most one activity can be processed at any time by a unary resource, but the activity may require more resources at the same time) and the activities are connected via precedence constraints (activity can start only after all its predecessors finished). The primary motivation for the research is providing an automated tool for correcting manually altered schedules in interactive Gantt chart environments (a "push button" approach). Nevertheless, the proposed repair techniques can also be used for example in the intensification stage of scheduling algorithms based on genetic algorithms or meta-heuristics. From another perspective, the proposed techniques belong to the group of re-scheduling algorithms.

We assume that the initial allocation of all activities to time (an initial schedule) is known. This time allocation may violate some precedence constraints (activity starts before some of its predecessors finishes) or some resource constraints (two or more activities are processed at the same time by the same resource). The goal is to correct the schedule (re-schedule) by shifting the activities in time, that is, to find a feasible schedule that does not violate any constraint. Moreover, the new schedule should not differ a lot from the initial time allocation of activities. Note that finding a feasible schedule is always possible unless there is a loop in the precedence constraints – activities can always be shifted to future as there are no deadlines. To minimize the number of changes between the initial and final schedule we apply a local approach, where particular flaws are repaired by local changes of affected activities rather than generating a completely new schedule from scratch. A local repair may introduce other flaws in the neighborhood which spread like a wave until all flaws are resolved. We use a three-step approach to repair a schedule. In the first step, loops of precedence constraints are detected and the user is asked to break each loop by removing at least one precedence constraint from it. In the second step, we repair all precedence constraints; two methods are suggested for this repair. Finally in the third step we repair violation of resource capacity constraints while keeping the precedence constraints valid. Each repair is realized by shifting affected activities locally in time.

The paper is organized as follows. First, a formal definition of the scheduling problem is given. Then we discuss some existing techniques that can be applied to schedule repair. After that, our three-stage approach to schedule repair is described in detail and proof of soundness is given. In conclusion we briefly discuss possible future steps.

2 Problem formulation

We use a fairy general description of the scheduling problem that fits especially fields such as construction and production scheduling. The Resource-Constrained Project Scheduling Problem (RCPSP) [3] is probably the closest classical scheduling problem though there are some differences as described below.

We assume a finite set of activities *Act*, each activity $A \in Act$ has a fixed duration d_A and it is non-interruptible (activity must run from its start till its end without interruption). Let s_A be the start time of activity A – the minimum start time of any activity is zero (schedule start), but there is no deadline. There is a set *Prec* of precedence constraints between the activities in the form $(A \rightarrow B)$; A is called a predecessor of B and B is called a successor of A. Formally for each precedence relation $(A \rightarrow B) \in Prec$ the following constraint must hold:

$$s_A + d_A \leq s_B \tag{1}$$

Let *Res* be a finite set of unary resources, that is, each resource can process at most one activity at any time. For each activity $A \in Act$ there is a set of required resources $r(A) \subseteq Res$. Activity A requires all resources from the set r(A) for processing, that is, A occupies each resource $R \in r(A)$ for the time period $\langle s_A, s_A + d_A \rangle$. The resource constraints can be formally expressed in the following way:

$$\forall A,B \in Act \text{ s.t. } r(A) \cap r(B) \neq \emptyset: s_A + d_A \leq s_B \vee s_B + d_B \leq s_A \tag{2}$$

The above resource constraint says that two activities A and B sharing the same resource cannot overlap in time (either A precedes B or B precedes A).

A *schedule* is a particular allocation of activities to time, formally it is a mapping of all variables s_A to natural numbers N_0 (including zero). A *feasible schedule* is a schedule that satisfies constraints (1) and (2). Notice that resource allocation is not part of the problem (activities are already allocated to resources). It is easy to prove that a feasible schedule always exists provided that there is no loop in the precedence constraints (for example $A \rightarrow B \rightarrow C \rightarrow A$). It is possible to topologically order all activities respecting the precedence constraints (precedence constraints define the partial ordering of activities) and then to allocate activities in this order to earliest possible times while respecting the precedence (1) and resource (2) constraints (activity can always be shifted to future if resource is not available at some time).

We do not assume any particular objective function in our scheduling problem. Makespan is a typical objective for RCPSP and other scheduling problems and our techniques for schedule repair try not to extend the makespan a lot beyond the makespan of the initial schedule. Though keeping the schedule compact (with small makespan) we are not really optimizing makespan or any other objective function. One of the reasons is that real-life objectives are frequently different from makespan (for example on-time-in-full is required) and it is hard or even impossible to formally express the real objective. Hence, we use the assumption that the initial schedule has a good quality from the user point of view and so the repaired schedule should not differ a lot from the initial schedule. Nevertheless, minimizing the difference between the schedules is more the nature of the presented repair techniques (repairing flaws locally) than formal minimization of the difference between the schedules as for example studied in [5,2].

The problem that we are solving in this paper can be stated as follows: given some schedule S, find a feasible schedule S' that does not differ a lot from S. The difference between schedules S and S' can be formalized in the following way:

$$\text{difference}(S,S') = \Sigma_{A \in Act} \mid s_A - s'_A \mid \qquad (3)$$

where s_A is the start time of activity A in S and s'_A is the start time of A in S'. Notice that the only way to modify the schedule is via changing values of variables s_A. As mentioned above, we are not strictly minimizing the objective (3), we are trying to achieve a good value of difference(S,S') by changing the values of s_A as little as possible (locally) to repair a violated constraint.

3 Related works

Dynamic scheduling is not a new area and though most scheduling research still focuses on static problems that do not change over time, there is an enlarging interest in studying dynamic aspects of scheduling [6]. This is mainly due to real-life scheduling problems that are primarily dynamic – machines break down, deliveries are delayed, workers become ill etc. As mentioned in the introduction, our motivation is slightly different and it goes from the area of interactive scheduling, where we need to repair manually modified schedules to satisfy all constraints. Nevertheless, the used technology is very similar.

The simplest approach to re-scheduling is generating a new schedule from scratch, also called total re-scheduling. This is the best technique when there are many disruptions in the schedule, but it has the disadvantage of generating a schedule completely different from the original schedule. We focus more on local repairs of the schedule with the motivation to keep the schedule similar to the original schedule. The method of generating a new schedule from scratch can be "localized" by removing some elements from the schedule and then adding them back without violation of constraints. Iterative Flattening Search [8] is an example of such a method where in the relaxation step all violated (and some other) constraints are removed from the problem (typically decisions about the ordering of activities) and then in the flattening stage the possible conflicts in the schedule are resolved by adding the constraints

back. Naturally, the question is how to relax the schedule to be able to resolve all possible conflicts (in the extreme case, all decision constraints are removed). Iterative Forward Search [7] uses a similar approach but it removes activities participating in violated constraints and then schedules these activities again. It has been proposed originally for timetabling problems with limited temporal constraints but for scheduling problems with temporally connected activities it may require re-scheduling many activities as it works with partial consistent schedules. Nevertheless, thanks to iterative improvements of schedules, this approach has been successfully applied to minimal perturbation problems (MPP) introduced in [5] and redefined in [2]. MPP focuses on finding a solution to a modified problem with minimal differences (perturbations) from the solution of the original problem. Hence, it is a primarily optimization problem with a specific objective function defined by the original solution. MPP can be solved from scratch like other optimization problems, for example using search techniques [2].

The approach studied in this paper belongs to heuristic-based repair algorithms where right shift rescheduling and affected operation rescheduling are the most frequently used techniques. *Right shift rescheduling* [4] repair is performed by globally shifting all remaining activities forward in time by the amount of disruption time. This introduces a gap in the schedule and it is not really appropriate for our type of problem. *Affected operations rescheduling* [1] reschedules only the activities that are directly or indirectly affected by the disruption. This heuristic was proposed to repair machine breakdowns but its generalization called modified Affected Operations Rescheduling [9] has been proposed to repair other typical disruptions seen in a job shop. We follow the idea of affected operations rescheduling but rather that assuming specific repair rules and repair operations such as insertion or deletion of activity, we allow only shifting the existing activity in time both forward and backward. Moreover, our techniques are designed to repair any number of constraint violations in the schedule. Recall that we are repairing manually modified schedules with flaws rather than reacting to instant disruptions from the job shop. Hence there might be more flaws spread in the schedule and we need to repair all of them while keeping the schedule as similar as possible to the original schedule.

4 Re-scheduling (repair) algorithm

As we already mentioned, we assume a typical scenario, where the human scheduler modifies an automatically generated schedule to reflect better the peculiarities of particular environment. The modification can affect any part of the scheduling problem introduced above – it is possible to change duration of activities, their position in time and required resources, to add or delete precedence constraints or even to add or delete activities and resources (in case of changing the set of activities, it is necessary to introduce a different measure of schedule difference, see for example [2]). By these modifications, it is quite easy to obtain an infeasible schedule where some of precedence or resource constraints are violated (we call the violated constraint a *flaw*). Though it is easy to detect and visualize the violated constraints (see Figure 1), it is frequently more complicated to repair them without introducing other flaws.

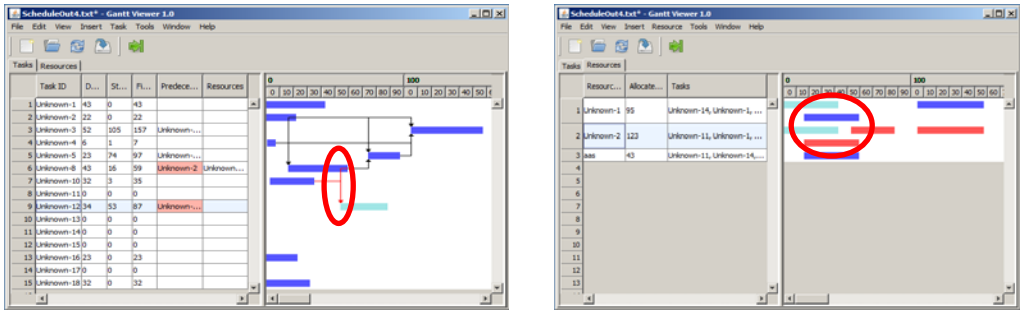


Fig. 1. Gantt charts visualization of violated precedence (left) and resource (right) constraint.

We suggest a schedule-repair method that mimics the behavior of a human scheduler by repairing flaws via local changes of time allocation of activities participating in the flaw. Naturally, this may introduce other flaws which need to be repaired and hence a systematic approach is necessary to prevent an infinite number of repairs (repairing one flaw introduces another flaw whose repair brings back the original flaw etc.). While such a systematic approach may be boring for a human, it is easy for a computer. The suggested method consists of three stages:

- detecting and breaking loops of precedence constraints,
- repairing violated precedence constraints,
- repairing violated resource constraints.

By modifying the set of precedence constraints, the user may unwittingly introduce a cycle between activities which prevents existence of the feasible schedule. Hence the first stage is detecting such loops and asking the user to remove some precedence constraint from each such loop. This is the only stage where user intervention is necessary[1]; the other two repair stages are fully automated. Recall that if there are no loops of precedence constraints then a feasible schedule always exists. In the following sections we will describe each stage in more details.

4.1 Loop detection

We represent the scheduling problem as a directed graph G = (E,V), where the set V of nodes equals the set *Act* of activities and there is edge (A,B) in E if and only if $(A \rightarrow B) \in Prec$. There exist standard methods for finding cycles in graphs and we adopted one of them. The method is based on repeating the following three steps until an empty graph is obtained:

1. repeatedly delete all nodes N from the graph such that there is either no incoming edge $(X,N) \in E$ or no outgoing edge $(N,X) \in E$; after this step, each remaining node in the graph has at least one predecessor and one successor
2. select any node from the graph (we use the node with the largest number of successors) and find a loop by depth-first search going in the direction of edges
3. present the loop to the user (in terms of activities) and remove the edge(s) suggested by the user (the particular precedence is also removed from *Prec*).

It is easy to prove that the above method removes all loops from the precedence constraints. In particular, by DFS in step 2 we must find a loop because each node has a successor (after step 1) and there is a finite number of nodes so at some time some node must be visited for the second time so the path between the first and second visit forms a loop. As we stop with an empty graph, no loops remain in the graph.

4.2 Precedence repair techniques

The goal of the second stage of the repair algorithm is to remove violation of all precedence constraints (1). This is possible for any schedule that does not contain loops in precedence relations which is exactly the schedule resulting from the first stage described in the previous section. We ignore violation of resource capacity constraints (2) at this stage.

The precedence $(A \rightarrow B) \in Prec$ is violated if $s_A + d_A > s_B$. The size of violation can be described by the following variable:

$$diff(A,B) = s_A + d_A - s_B.$$

[1] It is possible to randomly remove some precedence constraint from each loop or even to minimize the number of removed precedence constraints to break all loops, but in our opinion, the human decision is more appropriate.

Cleary, diff(A,B) is positive if and only if precedence (A→B) is violated. To locally repair the violated precedence (A→B) we can shift A backward in time (decrease s_A) or shift B forward in time (increase s_B) or shift together A backward and B forward. Naturally, if we do not want to stretch the schedule (increase makespan) then decreasing s_A as much as possible (but no more than constraint (1) requires) is the preferred way of repair. To find out how much time is available for shifting A backward we introduce the following variable:

$$\text{freeOnLeft}(A) = \begin{cases} s_A & \text{if A has no predecessors} \\ s_A - (s_{lp(A)} + d_{lp(A)}) & \text{if A has some predecessor and lp(A) denotes the latest predecessor of A in the schedule; } lp(A) = \operatorname{argmax}_{C:\,(C\to A)\,\in\, Prec} (s_C + d_C). \end{cases}$$

The straightforward technique of repairing a violated precedence constraint (A→B) is shifting A backward as much as possible and then shifting B forward if necessary. This can be formally described by the following assignments:

$$s_A \leftarrow s_A - \min(\text{freeOnLeft}(A), \text{diff}(A,B))$$
$$s_B \leftarrow s_A + d_A$$

Clearly, after making the suggested modification of start times s_A and s_B, the precedence relation (A→B) is satisfied ($s_B = s_A + d_A$). However, does it always realize the idea of shifting A backward? If all precedence constraints (C→A) are satisfied then freeOnLeft(A) ≥ 0 because $s_A \geq 0$ (definition of s_A) and $\forall C \in Act$ s.t. $(C\to A) \in Pre$: $s_A \geq s_C + d_C$ (according to (1)). However, if some (C→A) is violated then freeOnLeft(A) < 0 which actually means that A is shifted forward (s_A is increased) when repairing precedence (A→B). To prevent this unwanted behavior, it is enough to ensure that all precedence constraints (C→A) are satisfied before repairing the precedence constraint (A→B). Moreover, in such a case all precedence constraints (C→A) still remain valid after repairing (A→B). Hence, if we repair the violated precedence constraints in the right order, namely from left to right, then it is enough to repair each violation exactly once. Let us assign a unique index i to each precedence (A→B), denoted $(A\to B)_i$, in such a way that if we have two indexed precedence relations $(A\to B)_i$ and $(B\to C)_j$ then i < j. This can be easily realized by sorting first the activities according to the topological order satisfying the partial order defined by the precedence relations (this is always possible as there are no cycles) and then indexing the precedence relations according to this order (see Figure 2).

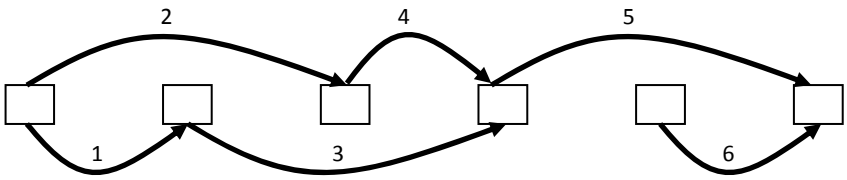


Fig. 2. Possible topological ordering of precedence constraints.

After defining the order of precedence relations the following pseudo-code *PrecRep* describes the repair algorithm:

```
   algorithm PrecRep
1      while any precedence is violated do
2          select violated precedence (A→B)_i such that i is minimal
3          s_A ← s_A – min( freeOnLeft(A), diff(A,B) )
4          s_B ← s_A + d_A
5      end while
   end PrecRep
```

Proposition 1: Algorithm *PrecRep* is sound and complete with respect to producing a schedule without violation of precedence constraints.

Proof: Clearly, if the algorithm stops then there are no violated precedence constraints in the schedule (because of the condition in the while loop). Hence it is enough to show that the algorithm stops after a finite number of steps. In each iteration the algorithm repairs (at least) one precedence constraint. The precedence relations are repaired in the order of their indexes so when $(A \rightarrow B)_i$ is being repaired then all $(X \rightarrow Y)_j$ such that $j < i$ are satisfied (have already been repaired). Moreover, no such relation $(X \rightarrow Y)_j$ where $j < i$ is violated by the repair of $(A \rightarrow B)_i$. Note that only relations $(C \rightarrow A)$ and $(B \rightarrow D)$ are influenced by the repair of $(A \rightarrow B)$ because we can only shift A backward and B forward. Satisfaction of other precedence relations involving A or B, namely $(A \rightarrow C)$ and $(D \rightarrow B)$, is not influenced by the repair. As we already discussed, relations $(C \rightarrow A)$ are not violated by the repair. According to the ordering of precedence relations, relations $(B \rightarrow D)_k$ have larger index than $(A \rightarrow B)_i$ $(k > i)$. In summary, after each iteration of the while loop we increase index k such that $\forall j \leq k$ $(X \rightarrow Y)_j$ is satisfied by at least one. Hence, after at most *m* iterations, where *m* is the number of precedence constraints in the schedule (the largest index), we repair all precedence relations. ■

Algorithm *PrecRep* represents a straightforward way of repairing precedence constraints. Unfortunately, it can shift activities forward more than necessary and hence it can increase makespan and make the schedule less compact. Though the algorithm can shift activity A backward when repairing $(A \rightarrow B)$, it can shift A at most as the latest predecessor lp(A) of A allows (see the definition of FreeOnLeft). Hence lp(A) may block shifting A backward even if there is time. In particular, it might be possible to shift lp(A) backward as well and hence to increase the time available for A (see Figure 3). To improve this behavior, we suggest a modification of the repair algorithm called *PrecRep-2* that exploits better available time on the left of activity A by shifting it beyond the horizon defined by lp(A).

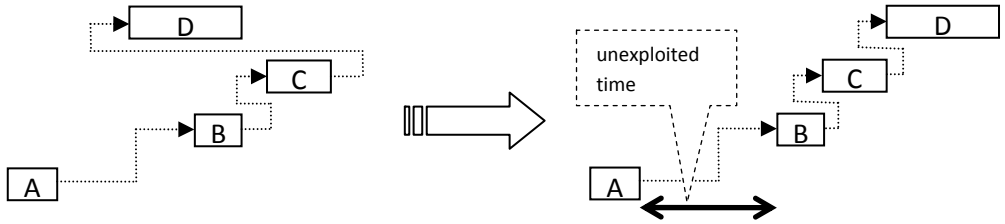


Fig. 3. Algorithm *PrecRep* does not exploit fully available time on left of D.

The idea of *PrecRep-2* algorithm is to shift A backward similarly to *PrecRep*, but if this is not enough to satisfy the constraint $(A \rightarrow B)$ (diff(A,B) is still positive) then we shift A backward slightly more, in particular by *truncate*(diff(A,B)/2), where *truncate*(X) is the closest integer between X and 0, for example *truncate*(3.7) = 3. This way, we violate the constraint $(lp(A) \rightarrow A)$ which can be repaired later by shifting lp(A) backward and so on. By this process, we can exploit better available time by shrinking the schedule. We only ensure that we do not violate the constraint $0 \leq s_A$ so the schedule does not stretch beyond the schedule start.

```
   algorithm PrecRep-2
1      while any precedence is violated do
2          select violated precedence (A→B)_i such that i is minimal
3          s_A ← s_A – min( freeOnLeft(A), diff(A,B) )
4          s_A ← max( 0, s_A – truncate(diff(A,B)/2) )
5          s_B ← s_A + d_A
6      end while
   end PrecRep-2
```

Lemma 1: If algorithm *PrecRep-2* repairs precedence $(A \rightarrow B)_i$ then after a finite number of iterations, the algorithm reaches a situation when all $(X \rightarrow Y)_j$ such that $j \leq i$ are satisfied. Moreover, the final start time s_A is not greater than the start time of A before repair of $(A \rightarrow B)$.

Proof: We shall prove the lemma by induction on the index of the repaired precedence. For $i = 1$ the lemma trivially holds, because the precedence $(A \rightarrow B)_1$ is repaired by the algorithm and s_A is not increased. When the algorithm started the repair of precedence $(A \rightarrow B)_i$, $i > 1$, all $(X \rightarrow Y)_j$ such that $j < i$ were satisfied. If these precedence relations are still satisfied after the repair then the lemma holds.

Assume that some precedence $(C \rightarrow A)$ has been violated by the backward shift of A. This may happen only after the assignment at line 4 so let old_s_A be the value of s_A before processing line 4 and new_s_A be the value of s_A after processing line 4. Note that if A starts at old_s_A then no precedence constraint $(C \rightarrow A)$ is violated. Let us take the violated precedence constraint $(C \rightarrow A)_i$ with the smallest index i that will be repaired next. According to the induction assumption, it is possible to repair this constraint and the value of s_C does not increase. Hence the new value of s_A is not greater than old_s_A because old_s_A satisfied the constraint $(C \rightarrow A)$ and s_C did not increase above its original value. In the same way, we can repair all violated precedence constraints $(C \rightarrow A)$ to reach the situation when all $(X \rightarrow Y)_j$ such that $j < i$ are satisfied again. The final value of s_A will not be greater than old_s_A.

Unfortunately, the constraint $(A \rightarrow B)_i$ may be violated again (when s_A increases) and we need to repair it. Nevertheless, one should realize that diff(A,B) is now strictly smaller than it was before we repaired the constraint $(A \rightarrow B)$ for the first time. Note that when A starts at new_s_A then the constraint $(A \rightarrow B)$ is satisfied – we repaired it this way. However, after repairing all $(C \rightarrow A)$, the value of s_A might increase as much as to old_s_A ($new\_s_A < old\_s_A$) so the new $diff(A,B) \leq old\_s_A - new\_s_A$. From line 4 of the algorithm, we can see that new diff(A,B) is at most half of the original diff(A,B). Anyway, we need to repeat the above process again and next time diff(A,B) will be even smaller. In the worst case, we stop when $diff(A,B) = 1$ because then the assignment at line 4 does not change s_A and hence no constraint $(C \rightarrow A)$ is violated so we reached the situation when all $(X \rightarrow Y)_j$ such that $j \leq i$ are satisfied. Moreover, we can see that s_A never went above old_s_A, which is not larger than the original value of s_A (old_s_A may be smaller than s_A because s_A might be decreased at line 3). ■

Proposition 2: Algorithm *PrecRep-2* is sound and complete with respect to producing a schedule without violation of precedence constraints.

Proof: If the algorithm stops then there are no violated precedence constraints in the schedule (because of the condition in the while loop) so the algorithm is sound. To prove completeness, it is necessary to show that the algorithm stops after a finite number of iterations (recall that a feasible schedule always exists). The precedence relations are repaired in the order of their indexes. According to lemma 1 after $(A \rightarrow B)_i$ is repaired then in a finite number of iterations the algorithm reaches a situation when all $(X \rightarrow Y)_j$ such that $j \leq i$ are satisfied. Hence, in the next step we will be repairing some precedence $(C \rightarrow D)_k$ where $k > i$ (if any such violated precedence still exists). Again, according to lemma 1 we can reach the situation when all $(X \rightarrow Y)_j$ such that $j \leq k$ are satisfied. We can continue this way until we reach the last index *m*. In summary, after a finite number of iterations the algorithm repairs all precedence constraints – the algorithm reaches a situation when all $(X \rightarrow Y)_j$ such that $j \leq m$ are satisfied. ■

Algorithm *PrecRep-2* exploits better available time (see Figure 4) but it is slower than *PrecRep* due to repeated "shrink-and-stretch" stages after violating the already repaired constraints. The open question is if the time complexity of *PrecRep-2* can be improved for example by memorizing how much time is actually available for backward shifts (to prevent the stretch stage).

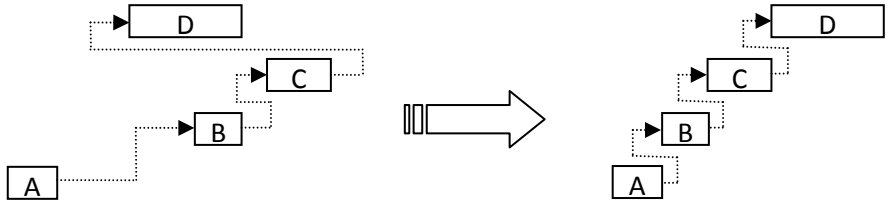


Fig. 4. Algorithm *PrecRep-2* exploits better available time on left of D.

4.3 Resource capacity repair technique

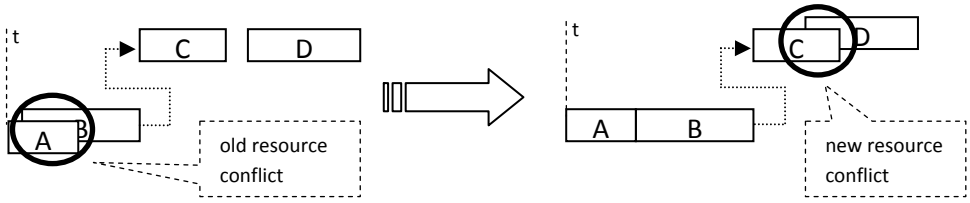
The final stage of the proposed repair algorithm consists of repairing resource conflicts. Recall that activities require for their processing unary resources; it is possible that an activity requires more than one resource (for example machine, tool, and worker). There is a resource conflict if two (or more) activities require the same resource at the same time.

From the previous stage we have a schedule that does not violate precedence constraints so it is either feasible or some resource constraints are violated. We now present a technique that repairs resource conflicts while keeping the precedence constraints satisfied. This technique resolves the conflict by shifting one of the activities forward in time. The algorithm *ResRep* iteratively repairs resource conflicts and each time a new precedence conflict is introduced then all precedence conflicts are repaired before continuing to the next resource conflict. By sweeping the schedule from past to future we remove all violated constraints (recall that there are no deadlines so any activity can be shifted forward).

```
   algorithm ResRep
1      while any constraint is violated do
2          if precedence is violated then
3              select violated precedence (A→B) with smallest s_A
3          else
4              let A,B be activities violating resource constraint (2)
5              such that s_A ≤ s_B and s_A is smallest among such pairs
6          end if
7          s_B ← s_A + d_A
8      end while
   end ResRep
```

Proposition 3: Algorithm *ResRep* is sound and complete (produces a feasible schedule).

Proof: If the algorithm stops then there are no violated constraints in the schedule (because of the condition in the while loop) so the final schedule is feasible and the algorithm is sound. To prove completeness, it is necessary to show that the algorithm stops after a finite number of iterations (recall that a feasible schedule always exists). We prove the proposition by showing that the number of possible conflicts decreases as we sweep the schedule from past to future. At the beginning, no precedence constraints were violated so some resource constraint must be violated. Let t be the smallest time such that for some activity A with $s_A = t$ there is a violated resource constraint (2) between A and B. The constraint is repaired by shifting B forward to start after activity A (line 7 of the algorithm), which may violate some precedence constraints (B→C). These violated precedence constraints will be repaired in next iterations of the algorithm (repairing precedence constraints is preferred to repairing resource conflicts) by shifting C forward etc. It may happen that during these repairs some new resource conflicts are introduced. However, all these new resource conflicts "start" after time t; formally, for any new resource conflict between activities C and D, $t < s_C$ and $t < s_D$ hold. The reason is that the conflicting activity, say C, that was shifted forward by the precedence repair steps now starts after $s_A + d_A$. The other activity D must start after t as well because otherwise there was already a resource conflict between C and D.



In summary, after repairing the resource conflict between A and B starting at time t and "propagating" it to all precedence constraints, no other resource conflict starting at or before t was added. As there is a limited number of resource conflicts, after a finite number of iterations, all resource conflicts starting at t are repaired and one of the former conflicting activities still starts at t. Now we can move to the next resource conflict which starts at time t' > t. Because the number of activities starting at or after t' is strictly smaller than the number of activities starting at or after t, the maximal number of possible conflicts is also smaller. This upper bound can be simply the number of all pairs of activities starting at or after t. Hence by shifting forward in time, the upper bound on the number of possible conflicts decreases and therefore the algorithm repairs all conflicts after a finite number of iterations. ■

5 Conclusions

The paper proposes a novel local repair technique for correcting violated precedence and resource constraints in RCPSP-like scheduling problems. By doing local repair steps, the schedule changes only locally in the area of the flaw, which is the main advantage over re-scheduling from scratch. Moreover, the proposed techniques are fully automated and problem independent so it is not necessary to describe specific repair rules for the problem as for example in modified Affected Operations Rescheduling [9]. To repair violated precedence constraints, we can shift activities to left (past) which keeps the schedule more compact in comparison with Right shift rescheduling [4]. Last but not least, the proposed method can repair any number of precedence and resources conflicts while traditional schedule repair algorithms work with a single disruption. Hence our method is appropriate for repairing schedules where several disruptions are scattered in time and on resources, which is typical for manually modified/constructed schedules.

We implemented the proposed techniques within an interactive Gantt viewer (Figure 1). Thought that exist many interactive Gantt viewers we are not aware about anyone providing automated schedule repair. Hence the main focus of the paper is on the formal description of the repair techniques and on theoretical justification of their soundness and completeness.

All presented techniques exploit the feature of the problem that there are no deadlines so it is always possible to shift activity forward in time. Nevertheless, when repairing the precedence constraints we tried to exploit also free time before the activity, that is, to shift activities backward. The main motivation was to keep the schedule as compact as possible (not to increase makespan a lot). We proposed two alternative repair techniques with the tradeoff between the compactness of the schedule and the speed of the technique. The technique producing more compact schedules can probably be speeded-up by using additional information during computation. This is a topic of future work. The current technique for repairing resource conflicts shifts activities only forward so another open question is whether the ideas from precedence repairs (backward shifts) can be used there.

Acknowledgements The research is supported by the Czech Science Foundation under the contract no. 201/07/0205.

References

1. Abumaizar RJ, Svestka JA, Rescheduling job shops under random disruptions. International Journal of Production Research 35(7):2065–2082 (1997)
2. Barták R, Müller T, Rudová H, Minimal Perturbation Problem – A Formal View. Neural Network World 13(5): 501–511 (2003)
3. Błazewicz J, Lenstra JK, and Rinnooy Kan AHG. Scheduling projects to resource constraints: Classification and complexity. Discrete Applied Mathematics, 5:11–24 (1983)
4. Brandimarte P, Rigodanza M, Roero L, Conceptual modeling of an object oriented scheduling architecture based on the shifting bottleneck procedure. IIE Transactions 32(10):921–929 (2000)
5. El Sakkout H, Richards T, Wallace M, Minimal Perturbation in dynamic scheduling. Proceedings of the 13th European Conference on Artificial Intelligence (ECAI98). John Wiley & Sons (1998)
6. Kocjan W, Dynamic Scheduling – State of the Art Report. SICS Technical Report T2002:28. SICS (2002)
7. Müller T, Barták R, Rudová H, Iterative Forward Search Algorithm: Combining Local Search with Maintaining Arc Consistency and a Conflict-Based Statistics. LSCS'04 - International Workshop on Local Search Techniques in Constraint Satisfaction (2004)
8. Oddi A, Policella N, Cesta A, Smith SF, Boosting the Performance of Iterative Flattening Search. AI*IA 2007: Artificial Intelligence and Human-Oriented Computing, LNCS 4733, pp. 447–458, Springer Verlag (2007)
9. Subramaniam V, Raheja AS, mAOR: A heuristic-based reactive repair mechanism for job shop schedules. The International Journal of Advanced Manufacturing Technology 22: 669–680 (2003)