Incremental Propagation Rules for Precedence Graph with Optional Activities and Time Windows

Roman Barták*, Ondřej Čepek*†

*Charles University
Faculty of Mathematics and Physics
Malostranské náměstí 2/25, 118 00 Praha 1, Czech Republic
{roman.bartak, ondrej.cepek}@mff.cuni.cz

†Institute of Finance and Administration
Estonská 500, 101 00 Praha 10, Czech Republic

Abstract

Constraint-based scheduling is a powerful tool for solving real-life scheduling problems thanks to a natural integration of special solving algorithms encoded in global constraints. The filtering algorithms behind these constraints are based on propagation rules modelling some aspects of the problems, for example a unary resource. This paper describes new incremental propagation rules integrating a propagation of precedence relations and time windows for activities allocated to a unary resource. Moreover, the rules also cover so called optional activities that may or may not be present in the final schedule.

Introduction

Real-life scheduling problems usually include a variety of constraints so special scheduling algorithms (Brucker, 2001) describing a single aspect of the problem can hardly be applied to solve the problem completely. Constraint-based scheduling (Baptiste, Le Pape, Nuijten, 2001) provides a natural framework for modelling and solving real-life problems because it allows integration of different constraints. The above mentioned special scheduling algorithms can be often transformed into propagators for the constraints so the big effort put in developing these algorithms is capitalised in constraint-based scheduling.

Many filtering algorithms for specialised scheduling constraints have been developed in recent years (Baptiste, Le Pape, Nuijten, 2001). There exist algorithms based for example on edge-finding (Baptiste & Le Pape, 1996) or not-first/not-last (Torres & Lopez, 1997) techniques that restrict the time windows of the activities. Other algorithms are based on relative ordering of activities, for example filtering based on optimistic and pessimistic resource profiles (Cesta & Stella, 1997). Recently, as scheduling and planning technologies are coming together, filtering algorithms combining filtering based on relative ordering and time windows appeared. Detectable precedences by Vilím (2002) are one of the first attempts for such a combination. Laborie (2003) presents a similar rule called energy precedence constraint for reservoir-like resources.

Filtering algorithms for scheduling constraints typically assume that all the constrained activities will be included in the final schedule. This is not always true, for example assume that there are alternative processes to accomplish a job or alternative resources per activity. These alternatives are typically modelled using optional activities that may or may not be included in the final schedule depending on which process or resource is selected. The optional activity may still participate in the constraints but it should not influence other activities until it is known to be in the schedule. This could be realised by allowing the duration of the optional activity to be zero for time-windows based filtering like edge-finding (Baptiste, Le Pape, Nuijten, 2001). However, this makes filtering weaker and as shown in (Vilím, Barták, Čepek, 2004) a stronger and faster filtering can be achieved if optional activities are assumed in the filtering algorithm directly. The paper (Focacci, Laborie, Nuijten, 2000) proposed a global precedence graph where alternative resources correspond to paths in the graph, but the graph is used merely for cost-based filtering (optimization of makespan or setup times).

In this paper we address the problem of integrated filtering based on precedence relations and time windows. From the beginning we assume the existence of optional activities. A filtering algorithm for these so called detectable precedences with optional activities on a unary resource has been proposed in (Vilím, Barták, Čepek, 2004). This algorithm uses Θ-Λ-tree to achieve O(n.log n) time complexity and it is a monolithic algorithm (must be repeated completely if there is any change of domains). The same pruning can be achieved by the energy precedence constraint proposed by Laborie (2003) if it is applied to a unary resource (the energy precedence constraint is defined for reservoirs). However, the energy precedence constraint is not defined for optional activities and details of implementation are not given in the paper.

We propose a new set of propagation rules that keep a transitive closure of the precedence relations, deduce new precedence relations, and shrink the time windows of the activities. They may also deduce that some optional activity will not be present in the final schedule. There are two main differences from the algorithm proposed in (Vilím, Barták, Čepek, 2004). First, we use "light" data structures, namely domains of variables. Second, the new rules are incremental so they directly react to changes of particular domains rather than running a monolithic algorithm from scratch. Such rules are much easier for implementation and for integration to existing constraint solvers and the hope is their incremental nature will lead to a good practical efficiency. The implementation of the rules is currently being done so the paper reports a work in progress.

The paper is organised as follows. We first give more details on the problem to be solved. Then we describe the constraint services available for implementation of new constraints. In the main part of the paper, we describe a constraint-based representation of the precedence graph and we propose a set of propagation rules for the precedence graph. After that, we describe propagation rules for shrinking time windows by using information about precedence relations.

The Problem

In this paper we address the problem of modelling a unary resource where activities must be allocated in such a way that they do not overlap in time. We assume that there are time windows restricting the position of these activities. The time window [R,D] for an activity specifies that the activity cannot start before R (release time) and cannot finish after D (deadline). We assume the activity to be non-interruptible so the activity occupies the resource from its start till its completion, i.e. for a time interval whose length is equal to the given length of the activity. We also assume that that there are precedence constraints for the activities. The precedence constraint A«B specifies that activity A must not finish later than activity B starts. The precedence constraints describe a partial order between the activities. The goal of scheduling is to decide a total order that satisfies (extends) the partial order (this corresponds to the definition of a unary resource) in such a way that each activity is scheduled within its time window. Last but not least we allow some activities to be so called *optional*. It means that it is not known in advance whether such activities are allocated to the resource or not. If the optional activity is allocated to the resource, that is, it is included in the final resource schedule then we call this activity *valid*. If the activity is known not to be allocated to the resource then we call the activity *invalid*. In other cases, that is the activity is not decided to be or not to be allocated to the resource, we call the activity *undecided*. Optional activities are useful for modelling alternative resources for the activities (an optional activity is used for each alternative resource and exactly one optional activity becomes valid) or for modelling alternative processes to accomplish a job (each process may consist of a different set of activities).

Note that for the above defined problem of scheduling with time windows it is known that deciding about an existence of a feasible schedule is NP-hard in the strong sense (Garey & Johnson, 1979) even when no precedence relations or optional activities are considered, so there is a little hope even for a pseudo-polynomial solving algorithm. Hence using propagation rules and constraint satisfaction techniques is justified there.

Constraints and Constraint Services

Constraint satisfaction problem is defined as a triple (X,D,C), where X is a finite set of variables, D is a set of domains for these variables, each variable may have its own domain which is a finite set of values, and C is a set of constraints restricting possible combinations of the values assigned to variables (a constraint is a relation over the variables' domains). The task is to find a value for each variable from the corresponding domain in such a way that all the constraints are satisfied (Dechter, 2003).

There exist many constraint solvers that provide tools for solving constraint satisfaction problems, for example ILOG Solver, Mozart or the clpfd library of SICStus Prolog. These solvers are typically based on combination of domain filtering with depth-first search. Domain filtering is a process of removing values from the domains that do not satisfy some constraint. Each constraint has a filtering algorithm assigned to it that does this job for the constraint, and these algorithms communicate via the domains of the variables – if a filtering algorithm shrinks a domain of some variable, the algorithms for constraints that use this variable propagate the change to other variables until a fixed point is reached or until some domain becomes empty. Such a procedure is called a (generalised) arc consistency. When all domains are reduced to singletons then the solution is found. If some domain becomes empty then no solution exists. In all other cases the search procedure splits the space of possible assignments by adding a new constraint (for example by assigning a value to the variable) and the solution is being searched for in sub-spaces defined by the constraint and its negation (other branching schemes may also be applied).

The constraint solvers usually provide an interface for user-defined filtering algorithms so the users may extend the capabilities of the solvers by writing their own filtering algorithms (Schulte, 2002). This interface consists of two parts: triggers and propagators. The user should specify when the filtering algorithm is called – a *trigger*. This is typically a change of domain of some variable, for example when the lower bound of the domain is increased, the upper bound is decreased, or any element is deleted from the domain. The *propagator* then describes how this change is propagated to domains of other variables. The constraint solver provides procedures for access to domains of variables and for operations over the domains (membership, union, intersection, etc.). The output of the propagator is a proposal how to change domains of other variables in the constraint. The algorithm may also deduce that the constraint cannot be satisfied (fail) or that the constraint is entailed (exit). We will describe the propagation rules in such a way that they can be easily transformed into a filtering algorithm in the above sense. Each propagation rule will consist of a trigger describing when the rule is activated and a propagator describing how the domains of other variables are changed.

Rules for the Precedence Graph

As we mentioned above, precedence relations are defined among the activities. These precedence relations define a precedence graph which is an acyclic directed graph where nodes correspond to activities and there is an arc from A to B if A«B. Frequently, the scheduling algorithms need to know whether A must be before B in the schedule, that is whether there is a path from A to B in the precedence graph. It is possible to look for the path each time such a query occurs. However, if such queries occur frequently then it is more efficient to provide the answer immediately, that is, in time O(1). This can be achieved by keeping a transitive closure of the precedence graph.

Definition 1: We say that a precedence graph G is *transitively closed* if for any path from A to B in G there is also an arc from A to B in G.

Defining the transitive closure is more complicated when optional activities are assumed. In particular, if A«B and B«C and B is undecided then we cannot deduce that A«C simply because if B is removed – becomes invalid – then the path from A to C is lost. Therefore, we need to define transitive closure more carefully.

Definition 2: We say that a precedence graph G with optional activities is *transitively closed* if for any two arcs A to B and B to C such that B is a valid activity and A and C are either valid or undecided activities there is also an arc A to C in G.

It is easy to prove that if there is a path from A to B such that A and B are either valid or undecided and all inner nodes in the path are valid then there is also an arc from A to B in a transitively closed graph (by induction of the path length). Hence, if no optional activity is used (all activities are valid) then Definition 2 is identical to Definition 1.

In the next paragraphs we will propose a constraint model for the precedence graph and two propagation rules that maintain the transitive closure of the graph with optional activities. We index each activity by a unique number from the set 1,..,n, where n is the number of activities. For each activity we use a 0/1 variable Valid indicating whether the activity is valid (1) or invalid (0). If the activity is not known yet to be valid or invalid then the domain of Valid is {0,1}. The precedence graph is encoded in two sets attached to each activity. CanBeBefore is a set of indices of activities that can be before a given activity. CanBeAfter is a set of indices of activities that can be after the activity. If we add an arc between A and B (A«B) then we remove the index of A from CanBeAfter(B) and the index of B from CanBeBefore(A). For simplicity reasons we will write A instead of the index of A. Note that these sets can be easily implemented as finite domains of two variables so a special data structure is not necessary. For this implementation we propose to include value 0 in above two sets to ensure that the domain is not empty even if the activity is first or last (an empty domain in CSP indicates the non-existence of a solution). The value 0 is not assumed as an index of any activity in the propagation rules. To simplify description of propagation rules we define the following sets (not kept in memory but computed on demand):

MustBeAfter = CanBeAfter \ CanBeBefore
MustBeBefore = CanBeBefore \ CanBeAfter
Unknown = CanBeBefore ∩ CanBeAfter.

MustBeAfter and MustBeBefore are sets of activities that must be after respectively before the given activity. Unknown is a set of activities that are not yet known to be before or after the activity.

We initiate the precedence graph in the following way. First, the variables Valid, CanBeBefore, and CanBeAfter with their domains are created. Then the known precedence relations are added in the above-described way (domains of CanBeBefore and CanBeAfter are pruned). Finally, the Valid variables for the valid activities are set to 1 (activities that are known to be invalid from the beginning may be omitted from the graph) and the following propagation rule is fired when Valid(A) is set.

The propagation rule is invoked when the validity status of the activity is known. "Valid(A) is instantiated" is its trigger. The part after → is a propagator describing pruning of domains. "exit" means that the constraint represented by the propagation rule is entailed so the propagator is not further invoked (its invocation does not cause further domain pruning). We will use the same notation in all rules.

```
Valid(A) is instantiated →                                /1/
  if Valid(A) = 0 then
    for each B do /* disconnect A from B */
      CanBeBefore(B) ← CanBeBefore(B) \ {A}
      CanBeAfter(B) ← CanBeAfter(B) \ {A}
  else /* Valid(A)=1 */
    for each B∈MustBeBefore(A) do
      for each C∈MustBeAfter(A)\MustBeAfter(B) do
        /* new precedence B«C */
        CanBeAfter(C) ← CanBeAfter(C) \ {B}
        CanBeBefore(B) ← CanBeBefore(B) \ {C}
        if B∉CanBeBefore(C) then    // break the cycle
          post_constraint(Valid(B)=0 ∨ Valid(C)=0)
  exit
```

Observation: Note that rule /1/ maintains symmetry for all valid and undecided activities because the domains are pruned symmetrically in pairs. This symmetry can be defined as follows: if Valid(B)≠0 and Valid(C)≠0 then B∈CanBeBefore(C) if and only if C∈CanBeAfter(B). This moreover implies that B∈MustBeBefore(C) if and only if C∈MustBeAfter(B).

We shall show now, that if the entire precedence graph is known in advance (no arcs are added during the solving procedure), then rule /1/ is sufficient for keeping the (generalised) transitive closure according to Definition 2. To give a formal proof we need to define several notions more precisely.

Let J={0,1, … ,n} be the set of activities, where 0 is a dummy activity with the sole purpose to keep all sets CanBeAfter(i) and CanBeBefore(i) nonempty for all $1 \le i \le n$. Furthermore, let G=(J\{0},E) be the given precedence graph on the set of activities, and G^T=(J\{0},T) its (generalised) transitive closure (note that the previously used notation i«j does not distinguish between the arcs which are given as input and those deduced by transitivity). The formal definition of the set T can be now given as follows:

1. if (i,j)∈E then (i,j)∈T
2. if (i,j)∈T and (j,k)∈T and Valid(i)≠0 and Valid(j)=1 and Valid(k)≠0 then (i,k)∈T

Furthermore, the set T is not maintained as a list of pairs of activities. Instead, it is represented using the set variables CanBeAfter(i) and CanBeBefore(i), $1 \le i \le n$ in the following manner: (i,j)∈T if and only if i∉CanBeAfter(j) and j∉CanBeBefore(i). The incremental construction of the set T can be described as follows.

<u>Initialization</u>: for every i ∈ J\{0} set

- CanBeAfter(i) ← J\{i}
- CanBeBefore(i) ← J\{i}
- Valid(i) ← {0,1}

<u>Set-up</u>: for every arc (i,j)∈E set

- CanBeAfter(j) ← CanBeAfter(j)\{i}
- CanBeBefore(i) ← CanBeBefore(i)\{j}

<u>Propagation</u>: whenever a variable is made valid, call rule /1/

Clearly, T is empty after the initialization and T=E after the set-up. Now we are ready to state and prove formally that rule /1/ is sufficient for maintaining the set T on those activities which are already valid or still undecided.

Proposition 1: Let $i_0, i_1, \dots, i_m$ be a path in E such that Valid(i_j)=1 for all $1 \le j \le m-1$ and Valid(i_0)≠0 and Valid(i_m)≠0 (that is, the endpoints of the path are both either valid or undecided and all inner points of the path are valid). Then (i_0,i_m)∈T, that is i_0∉CanBeAfter(i_m) and i_m∉CanBeBefore(i_0).

Proof: We shall proceed by induction on m. The base case m=1 is trivially true after the set-up. For the induction step let us assume that the statement of the lemma holds for all paths (satisfying the assumptions of the lemma) of length at most m-1. Let $1 \le j \le m-1$ be an index such that Valid(i_j)←1 was set last among all inner points $i_1, \dots, i_{m-1}$ on the path. By the induction hypothesis we get

- i_0∉CanBeAfter(i_j) and i_j∉CanBeBefore(i_0) using the path $i_0, \dots, i_j$
- i_j∉CanBeAfter(i_m) and i_m∉CanBeBefore(i_j) using the path $i_j, \dots, i_m$

We shall distinguish two cases. If i_m∈MustBeAfter(i_0) (and thus by symmetry also i_0∈MustBeBefore(i_m)) then by definition i_m∉CanBeBefore(i_0) and i_0∉CanBeAfter(i_m) and so the claim is true trivially. Thus let us in the remainder of the proof assume that i_m∉MustBeAfter(i_0).

Now let us show that i_0∈CanBeBefore(i_j) must hold, which in turn (together with i_0∉CanBeAfter(i_j)) implies i_0∈MustBeBefore(i_j). Let us assume by contradiction that i_0∉CanBeBefore(i_j). However, at the time when both i_0∉CanBeAfter(i_j) and i_0∉CanBeBefore(i_j) became true, that is when the second of these conditions was made satisfied by rule /1/, rule /1/ must have posted the constraint (Valid(i_0)=0 ∨ Valid(i_j)=0) which contradicts the assumptions of the lemma. By a symmetric argument we can prove that i_m∈MustBeAfter(i_j). Thus when rule /1/ is triggered by setting Valid(i_j)←1 both i_0∈MustBeBefore(i_j) and i_m∈MustBeAfter(i_j) hold (and i_m∉MustBeAfter(i_0) is assumed), and therefore rule /1/ removes i_m from the set CanBeBefore(i_0) as well as i_0 from the set CanBeAfter(i_m), which finishes the proof.

From now on there will be no need to distinguish between the "original" arcs from E and the transitively deduced ones, so we will work solely with the set T. To simplify notation we shall switch back to the A«B notation (which is equivalent to (A,B) ∈ T).

In some situations arcs may be added to the precedence graph during the solving procedure, either by the user, by the scheduler, or by other filtering algorithms like the one described in the next section. The following rule updates the precedence graph to keep transitive closure when an arc is added to the precedence graph.

```
A«B is added →                                        /2/
  CanBeAfter(B) ← CanBeAfter(B) \ {A}
  CanBeBefore(A) ← CanBeBefore(A) \ {B}
  if A∉CanBeBefore(B) then    // break the cycle
    post_constraint(Valid(A)=0 ∨ Valid(B)=0)
  else
    if Valid(A)=1 then    // transitive closure
      for each C∈MustBeBefore(A)\MustBeBefore(B) do
        add C«B
    if Valid(B)=1 then    // transitive closure
      for each C∈MustBeAfter(B)\MustBeAfter(A) do
        add A«C
  exit
```

The rule /2/ does the following. If a new arc is added then the sets CanBeBefore and CanBeAfter are updated. If a cycle is detected then the cycle is broken in the same way as in rule /1/. The rest of the propagation rule ensures that if an arc is added and one of its endpoints is valid then other arcs are added recursively to keep a transitive closure. The following proposition shows that all necessary arcs are added by rule /2/.

Proposition 2: If the precedence graph is transitively closed and some arc is added then the propagation rule /2/ updates the precedence graph to be transitively closed again.

Proof: If an arc A«B is added and B is valid then according to the definition of transitive closure for each C such that B«C the arc A«C should be present in the precedence graph. The rule /2/ adds all these arcs. Symmetrically, if A is valid then for each C such that C«A all arcs C«B (where A«B) are added by the rule. Note also, that if the rule adds a new arc then this change in the precedence graph is propagated further so it may force adding other arcs. Hence all the necessary arcs are added. The rule adds only new arcs so the recursive calls to the rule must stop sometime.

Rules for Time Windows

An absolute position of the activity in time is frequently restricted by a *release time* and *deadline* that define a *time window* for processing the activity. The activity cannot start before the release time and it must be finished before the deadline. We assume the activity to be uninterruptible so it occupies the resource from its start till its completion. The processing time of activity A is constant, we denote it by p(A). The goal of time window filtering is to remove time points from the time window when the activity cannot be processed. Usually, only the lower and upper bounds of the time window change so we are speaking about shrinking the time window.

The standard constraint model for time allocation of the activity assumes two variables – start(A) and end(A) – describing when the activity A starts and completes. Initially, the domain for the variable start(A) is [release_time(A), deadline(A)-p(A)] and, similarly, the initial domain for the variable end(A) is [release_time(A)+p(A), deadline(A)]. If these two initial domains are empty then the activity is made invalid. We will use the following notation to describe bounds of the above domains:

est(A) = min(start(A))	earliest start time
lst(A) = max(start(A))	latest start time
ect(A) = min(end(A))	earliest completion time
lct(A) = max(end(A))	latest completion time

This notation can be extended in a natural way to sets of activities. Let Ω be a set of activities, then:

$$\mathrm{est}(\Omega) = \min\{\mathrm{est}(A), A\in\Omega\}$$
$$\mathrm{lst}(\Omega) = \max\{\mathrm{lst}(A), A\in\Omega\}$$
$$\mathrm{ect}(\Omega) = \min\{\mathrm{ect}(A), A\in\Omega\}$$
$$\mathrm{lct}(\Omega) = \max\{\mathrm{lct}(A), A\in\Omega\}$$
$$p(\Omega) = \sum\{p(A), A\in\Omega\}$$

During propagation, we will be increasing est and decreasing lct which corresponds to shrinking the time window for the activity. For simplicity reasons we use a formula est(A) ← X to describe a requested change of est(A) which actually means est(A) ← max(est(A), X). Similarly lct(A) ← X means lct(A) ← min(lct(A), X).

The time windows can be used to deduce a new precedence between activities. In particular, if est(A)+p(A)+p(B)>lct(B) then activity A cannot be processed before activity B and hence we can deduce B«A. This is called a *detectable precedence* in (Vilím, 2002). Vice versa, the precedence graph can be used to shrink time windows of the activities. In particular, we can compute the earliest completion time of the set of valid activities that must be processed before some activity A and the latest start time of the set of valid activities that must be processed after A. These two numbers define bounds of the time window for A. Formally:

$$\mathrm{est}(A) \leftarrow \max\{\mathrm{est}(\Omega)+p(\Omega) \mid \Omega\subseteq\{X|X«A \ \&\ \mathrm{Valid}(X)=1\}\}$$
$$\mathrm{lct}(A) \leftarrow \min\{\mathrm{lct}(\Omega)-p(\Omega) \mid \Omega\subseteq\{X|A«X \ \&\ \mathrm{Valid}(X)=1\}\}$$

The above two formulas are special cases of the *energy precedence constraint* (Laborie, 2003) for unary resources. Note also that the new bound for est(A) can be computed in *O*(*n*.log *n*) time, where *n* is the number of activities in $\Theta = \{X \mid X«A \ \&\ \mathrm{Valid}(X)=1\}$, rather that exploring all subsets $\Omega\subseteq\Theta$. The algorithm is based on the following observation: if Ω' is the set with the maximal $\mathrm{est}(\Omega')+p(\Omega')$ then $\Omega'\supseteq\{X \mid X\in\Theta \ \&\ \mathrm{est}(\Omega')\leq\mathrm{est}(X)\}$, otherwise adding such X to Ω' will increase $\mathrm{est}(\Omega')+p(\Omega')$. Consequently, it is enough to explore sets $\Omega_X = \{Y \mid Y\in\Theta \ \&\ \mathrm{est}(X)\leq\mathrm{est}(Y)\}$ for each $X\in\Theta$ which is done by the following algorithm (the new bound is computed in the variable *end*):

```
dur ← 0
end ← inf
for each Y ∈ {X | X«A & Valid(X)=1}
        in non-increasing order of est(Y) do
    dur ← dur + p(Y)
    end ← max(end, est(Y)+dur)
```

The bound for lct(A) can be computed in a symmetrical way in *O*(*n*.log *n*) time, where *n* is the number of activities in {X | A«X & Valid(X)=1}.

We now present two groups of propagation rules working with time windows and a precedence graph. The first group of rules realise the energy precedence constraint in an incremental way by reacting to changes in the precedence graph. The rules are invoked by making the activity valid /1a/ and by adding a new precedence relation /2a/. Because these rules have the same triggers as the rules for the precedence graph, they can be actually combined with them. Hence, we name the new rules using the number of the corresponding rule for the precedence graph.

The rules shrink the time windows using information about the precedence relations as described above. Only valid activities influence time windows of other (non invalid) activities. This corresponds to our requirement that optional activities that are not yet known to be valid should not influence other activities but they can be influenced. Notice also that if A«C, C«B, and C is valid then it is enough to explore possible increase of est(C) only. The reason is that if est(C) is really increased then the rule /3/ is invoked for C (see below) and the change is propagated directly to est(B). Similarly, only activities B such that there is no valid activity C in between B and A are explored for change of lct(B).

When activity A becomes valid and B is after A (A«B) or when A is valid and arc A«B is added then A can (newly) participate in sets Ω_X that are used to compute est of B (see above). Visibly, only sets containing A are of interest because only these sets can lead to change of est(B). The other sets Ω_X used to update est(B) have already been explored or will be explored when calling the rules for some valid activity in Ω_X. Moreover, all valid activities C such that C«A are used to compute est(A) so they can complete together no later than in est(A). Hence these activities do not influence directly est(B) (they influence it through changes of est(A)). Thus, we need to explore all subsets of valid activities X such that X«B and ¬X«A and these subsets contain A. Only these subsets can deduce a possible change of est(B). These are exactly the sets used in rules /1a/ and /2a/. A symmetrical analysis can be done for activities B before A. Note also that sets Ω' in rules /1a/ and /2a/ can be explored in the same way as we described for the energy precedence constraint above.

```
Valid(A) is instantiated →                               /1a/
  if Valid(A)=1 then
    for each B∈MustBeAfter(A) s.t.
          ¬∃C Valid(C)=1 & A«C & C«B do
      let Ω = {X | X∈MustBeBefore(B) &
                 X∈CanBeAfter(A) & Valid(X)=1}
      est(B) ← max{est(Ω'∪{A})+p(Ω')+p(A) | Ω'⊆Ω}
    for each B∈MustBeBefore(A) s.t.
          ¬∃C Valid(C)=1 & B«C & C«A do
      let Ω = {X | X∈MustBeAfter(B) &
                 X∈CanBeBefore(A) & Valid(X)=1}
      lct(B) ← min{lct(Ω'∪{A})-p(Ω')-p(A) | Ω'⊆Ω}
  exit
```

```
A«B is added →                                           /2a/
    if Valid(A)=1 & Valid(B)≠0 then
      let Ω = {X | X∈MustBeBefore(B) &
                 X∈CanBeAfter(A) & Valid(X)=1}
      est(B) ← max{est(Ω'∪{A})+p(Ω')+p(A) | Ω'⊆Ω}
    if Valid(B)=1 & Valid(A)≠0 then
      let Ω = {X | X∈MustBeAfter(A) &
                 X∈CanBeBefore(B) & Valid(X)=1}
      lct(A) ← min{lct(Ω'∪{B})-p(Ω')-p(B) | Ω'⊆Ω}
    exit
```

The second group of rules is triggered by shrinking the time window (/3/ for increased est and /4/ for decreased lct). The rules in this group can deduce that the activity is invalid, if it has an empty time window, and they can deduce a new detectable precedence. Moreover, if the activity is valid then the change of its time window is propagated to other activities whose relative position to a given activity is known (they are before or after the given activity). If est of valid activity A is increased then it may influence est of B such that A«B (note that B is either valid or undecided, because invalid activities are disconnected from the graph). This happens if and only if est(B) ≤ est(Ω_A)+p(Ω_A) (see above for the definition of Ω_A with respect to B). Notice that rule /3/ computes est(Ω_A)+p(Ω_A) to update est(B). Symmetrically, rule /4/ updates lct(B) for activities B such that B«A, if necessary. Hence, the propagation rules incrementally maintain the energy precedence constraint.

```
est(A) is increased →                                      /3/
  if Valid(A)=0 or est(A)+p(A) > lct(A) then
    Valid(A) ← 0
    exit
  else
    ect(A) ← est(A)+p(A)
    for each B∈Unknown(A) do
      if est(A)+p(A)+p(B) > lct(B) then
        B«A    /* detectable precedence */
    if Valid(A)=1 then
      for each B∈MustBeAfter(A) s.t.
                 ¬∃C Valid(C)=1 & A«C & C«B do
        est(B) ← est(A)+p(A)+
                 Σ{p(X) | X∈MustBeBefore(B) &
                          est(A)≤est(X) & Valid(X)=1}
```

```
lct(A) is decreased →                                      /4/
  if Valid(A)=0 or est(A)+p(A) > lct(A) then
    Valid(A) ← 0
    exit
  else
    lst(A) ← lct(A)-p(A)
    for each B∈Unknown(A) do
      if est(B)+p(B)+p(A) > lct(A) then
        A«B    /* detectable precedence */
    if Valid(A)=1 then
      for each B∈MustBeBefore(A) s.t.
                 ¬∃C Valid(C)=1 & B«C & C«A do
        lct(B) ← lct(A)-p(A)-
                 Σ{p(X) | X∈MustBeAfter(B) &
                          lct(X)≤lct(A) & Valid(X)=1}
```

Conclusions

The paper reports a work in progress on constraint models for the unary resource with precedence relations between the activities and time windows for the activities. Optional activities that may or may not be allocated to the resource are also assumed. We propose a set of propagation rules that keep a transitive closure of the precedence relations, deduce additional precedence constraints based on time windows, and shrink the time windows for the activities. These rules are intended to complement the existing filtering algorithms based on edge-finding etc. to further improve domain pruning. Our next steps include formal complexity analysis, detail comparison to existing propagation rules (edge finder, etc.), implementation of the proposed rules, and testing in real-life environment.

Acknowledgements

The research is supported by the Czech Science Foundation under the contract no. 201/04/1102. We would like to thank anonymous reviewers for comments on early draft.

References

Baptiste, P. and Le Pape, C. 1996. Edge-finding constraint propagation algorithms for disjunctive and cumulative scheduling, *Proceedings of the Fifteenth Workshop of the U.K. Planning Special Interest Group (PLANSIG).*

Baptiste P., Le Pape C., and Nuijten W. 2001. *Constraint-Based Scheduling: Applying Constraint Programming to Scheduling Problem*s, Kluwer Academic Publishers.

Brucker P. 2001. *Scheduling Algorithms*, Springer Verlag.

Cesta A. and Stella C. 1997. A Time and Resource Problem for Planning Architectures, *Recent Advances in AI Planning (ECP'97)*, LNAI 1348, Springer Verlag, 117-129.

Dechter R. 2003. *Constraint Processing*, Morgan Kaufmann.

Focacci F., Laborie P., and Nuijten W. 2000. Solving Scheduling Problems with Setup Times and Alternative Resources. *Proceedings of AIPS 2000.*

Garey M. R. and Johnson D. S. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H.Freeman and Company, San Francisco.

Laborie P. 2003. Algorithms for propagating resource constraints in AI planning and scheduling: Existing approaches and new results. *Artificial Intelligence*, 143, 151-188.

Schulte C. 2002. Programming Constraint Services, High-Level Programming of Standard and New Constraint Services, Springer Verlag.

Torres P. and Lopez P. 1999. On Not-First/Not-Last conditions in disjunctive scheduling, *European Journal of Operational Research*, 127, 332-343.

Vilím P. 2002. Batch Processing with Sequence Dependent Setup Times: New Results, *Proceedings of the 4th Workshop of Constraint Programming for Decision and Control, CPDC'02*, Gliwice, Poland.

Vilím P., Barták R., and Čepek O. 2004. Unary Resource Constraint with Optional Activities, *Principles and Practice of Constraint Programming (CP 2004)*, LNCS 3258, Springer Verlag, 62-76.