Double Precedence Graphs

Roman Barták*, Ondřej Čepek*†

*Charles University, Faculty of Mathematics and Physics
Malostranské nám. 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

Reasoning on precedence relations is crucial for many planning and scheduling systems. In this paper we propose a double precedence graph where direct precedence relations are kept in addition to traditional precedence relations. By the direct precedence between activities A and B we mean that A directly precedes B (no activity is between A and B). We also show how these direct precedence relations can be used in incremental filtering of time windows and in introducing sequence-dependent setup times between activities.

Introduction

Precedence relations play a crucial role in planning systems while time windows are more important for scheduling. As scheduling and planning technologies are coming together, filtering algorithms combining filtering based on precedence relations and time windows appeared in the context of constraint-based scheduling. 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. In (Barták & Čepek, 2005) we proposed 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. These rules achieve the same pruning as a monolithic algorithm proposed in (Vilím, Barták, Čepek, 2004) and as the energy precedence constraint proposed by Laborie (2003) if it is applied to a unary resource (the energy precedence constraint is defined for reservoirs) and optional activities are not used. In (Barták & Čepek, 2005) we focused on "implementation-friendly" design that uses light data structures (domains of variables in the sense of constraint satisfaction technology) and that is easy to integrate into existing constraint solvers. In this paper we further extend our work to cover sequence dependent setup times. We propose a so called double precedence graph where direct precedence relations are kept in addition to standard precedence relations. By the direct precedence between activities A and B we mean that A directly precedes B, in particular no activity can be scheduled between A and B. We will show how the propagation rules from (Barták & Čepek, 2005) can be modified to handle the double precedence graphs. We will also describe in detail how this additional information about direct precedence relations is used in modelling of setup times and in filtering of time windows.

The paper is organized as follows. First, we will describe in detail the scheduling problem that is behind our work. Then, we will propose a constraint model for sequence-dependent setup times which motivates the introduction of direct precedence relations. The main part of the paper is devoted to the description of incremental propagation rules that maintain a transitive closure of the double precedence graph. We will conclude by showing that additional information about direct precedence relations can also be used in incremental propagation of time windows.

Motivation

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 each activity to be non-interruptible so the activity occupies the resource from its start till its completion, that is, for a time interval whose length is equal to the given length of the activity. We also assume 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. Moreover, we assume so called *sequence-dependent setup times* between the activities. For each pair of activities A and B there is a setup time $T_{A,B}$ meaning that if A is allocated directly before B then there must be a gap of at least $T_{A,B}$ time units between the completion of A and the start of B. We do not assume other restrictions on setup times. For example, setup times can be asymmetric ($T_{A,B} \neq T_{B,A}$) and a triangular inequality does not need to hold between the setup times (a triangular inequality between the setup times says that $T_{A,B} \leq T_{A,C}+T_{C,B}$). Last but not least we allow some

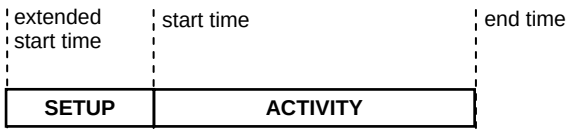
activities to be so called *optional*. It means that it is not known in advance whether such activities are allocated to the resource or not. If the optional activity is allocated to the resource, that is, it is included in the final resource schedule then we call this activity *valid*. If the activity 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, setup times, and optional activities are considered. Hence there is a little hope even for a pseudo-polynomial solving algorithm and therefore using propagation rules and constraint satisfaction techniques is justified there.

Modelling Sequence-Dependent Setup Times

Sequence-dependent setup times appear frequently in real-life scheduling problems with complex resources. Setup times describe time necessary to setup a machine when switching from one item to a different item, for example to change a mould in the injection machine when changing the shape of a product. Sequence dependence typically means that switching from A to B may be different from switching B to A (we also call it asymmetry). For example, setup time to switch from transparent items to black items is different from switching black items to transparent ones. Usually, a triangular inequality holds between the setup times, that is $T_{A,B} \leq T_{A,C} + T_{C,B}$, where $T_{A,B}$ is a setup time for going from A to B. Scheduling models for setup times typically assume this triangular inequality (Vilím & Barták, 2002) and we are not aware about any model of setup times where the triangular inequality is not assumed. Nevertheless, there exist resources where this triangular inequality does not hold. For example, it might be faster to produce some intermediate product C than to switch directly from A to B. In this paper, we focus our attention to general sequence dependence setup times where neither symmetry of setup times nor triangular inequality is requested.

Typically, setup time is assumed to be an empty gap between two consecutive activities. We propose to include the setup time in the duration of the second activity. Basically, it means that duration of each activity will consists of its real duration plus a setup time. Clearly because of time windows we still need to keep the original start time of the activity but we add the extended start time to model the start time including the setup. The extended start time can now participate in non-overlapping constraints like edge-finding without modifying these constraints.



The difference between the start time and the extended start time equals exactly to the setup time for a given activity (if it is the first activity, we can use a startup time there). The open question is how to find out the setup time. If we know the directly preceding activity, we can look for the setup time in the list of all setup times for a given activity. If there are several candidates for the direct predecessor then we can compute the minimal and the maximal setup time. Basically, we can post a binary constraint between the variable describing a direct predecessor and the variable describing the setup time. The relation behind this constraint is extensionally defined – it is a list of setup times for the activity where index of each element corresponds to identification of the possible predecessor.

The above model of setup times has many advantages. First, the model does not require any restriction on setup times like the triangular inequality. Second, it is easy to implement. We will show later how the precedence graph can be extended to a double precedence graph that provides necessary information to propagate setup times. Last but least, it is not necessary to modify other filtering algorithms, like edge finding (Baptiste & Le Pape, 1996) or not-first/not-last rules (Torres & Lopez, 1999), to work with setup times, provided that these algorithms can accommodate variable duration of activities. Currently, we see only one important drawback. Domain filtering with the proposed model might be weaker than other models that assume some specific features of setup times. However, this is an obvious trade-off between efficiency and generality.

From the point of view of precedence graphs, we need additional information to be deducible from the precedence graph – the direct predecessor of each activity. We will show in the next section, how this information can be incrementally maintained.

Double Precedence Graph and Its Maintenance

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. Let A«B and B«C and B be undecided. In such a case, it is not possible to deduce that A«C 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 (activities are valid) then Definition 2 is identical to Definition 1.

In (Barták & Čepek, 2005) we proposed a constraint model for the precedence graph and two propagation rules that maintain the transitive closure of the graph with optional activities. In this paper, we extend this model by adding information about possible direct predecessors and, symmetrically, possible direct successors to the transitively closed precedence graph.

Definition 3: We say that A can *directly precede* B if both A and B are either valid or undecided activities, B is not before A (¬B«A) and there is no valid activity C such that A«C and C«B (relation « is assumed to be from the transitive closure of the precedence graph with optional activities).

The relation of direct precedence introduces a new type of arc, say $«_d$, in the precedence graph and hence we are speaking about the *double precedence graph*. There is one significant difference between the arcs of type « and the arcs of type $«_d$. While the arcs « are added into the graph as scheduling proceeds, the arcs $«_d$ are removed from the graph. In the final schedule there is exactly one arc of type $«_d$ going into each valid activity (with the exception of the very first activity in the schedule) and one arc of type $«_d$ going from each valid activity (with the exception of the very last activity in the schedule). In the following paragraphs, we revise the constraint model from (Barták & Čepek, 2005) by adding information about the direct precedence relations.

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 a CSP indicates the non-existence of a solution). The value 0 is not assumed as an index of any activity in the propagation rules. Usually, CSPs are solved by removing inconsistent values from the domains, this is called domain filtering. Our propagation rules do exactly the same job – inconsistent values are removed from the above sets. To simplify description of the propagation rules we define for every activity A the following sets:

$$\text{MustBeAfter}(A) = \text{CanBeAfter}(A) \setminus \text{CanBeBefore}(A)$$
$$\text{MustBeBefore}(A) = \text{CanBeBefore}(A) \setminus \text{CanBeAfter}(A)$$
$$\text{Unknown}(A) = \text{CanBeBefore}(A) \cap \text{CanBeAfter}(A).$$

MustBeAfter(A) and MustBeBefore(A) are sets of those activities that must be after and before the given activity A respectively. Unknown(A) is a set of activities that are not yet known to be before or after activity A. These sets can be stored in memory and incrementally maintained whenever the sets CanBeBefore or CanBeAfter are pruned or these sets can be computed on demand.

In the subsequent complexity analysis, we will assume that the set operations membership and deletion require time O(1), which can be realised for example by using a bitmap representation of the sets. Note that this assumption also holds for sets MustBeAfter, MustBeBefore, and Unknown even if they are computed on demand (in the propagation rules, we will only check membership in these sets).

To model direct precedence relations and hence a double precedence graph, we add two sets to each activity: CanBeRightBefore and CanBeRightAfter containing initially values 0,..,n with the same meaning as above. The contents of these two new sets are defined according to Definition 3 as follows:

$$A \in \text{CanBeRightBefore}(B) \equiv$$
$$A \in \text{CanBeBefore}(B) \wedge$$
$$\neg\exists C\ \text{valid}(C)=1 \wedge C \in \text{MustBeAfter}(A) \wedge C \in \text{MustBeBefore}(B)$$
$$A \in \text{CanBeRightAfter}(B) \equiv$$
$$A \in \text{CanBeAfter}(B) \wedge$$
$$\neg\exists C\ \text{valid}(C) =1 \wedge C \in \text{MustBeBefore}(A) \wedge C \in \text{MustBeAfter}(B)$$

Clearly, $A «_d B \Leftrightarrow A \in \text{CanBeRightBefore}(B) \Leftrightarrow B \in \text{CanBeRightAfter}(A)$. Note that if the transitive closure of the precedence graph is kept then it is easy to maintain the above two sets. In particular, each time an activity is removed from CanBeBefore, the same activity is removed from CanBeRightBefore (similarly for CanBeRightAfter). Moreover, if an arc A«B is added to re-establish the transitive closure (because there exists a valid activity C such that A«C and C«B) then B is removed from CanBeRightAfter(A) and A is removed from CanBeRightBefore(B).

We initiate the double precedence graph in the following way. First, the variables Valid(A), CanBeBefore(A), CanBeAfter(A), CanBeRightAfter(A), and CanBeRightBefore(A) with their domains are created for every activity A. Then the known precedence relations are added in the above-described way (domains of CanBeBefore(A), CanBeRightBefore(A), CanBeAfter(A), and CanBeRightAfter(A) are pruned). Finally, the Valid(A) variable for every valid activity A

is set to 1 (activities that are known to be invalid from the beginning may be omitted from the graph or their Valid variables are set to 0).

Propagation rule /1/ is invoked when the validity status of the activity becomes 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}
      CanBeRightBefore(B) ← CanBeRightBefore(B) \ {A}
      CanBeRightAfter(B) ← CanBeRightAfter(B) \ {A}
  else                  // Valid(A)=1
    for each B∈MustBeBefore(A) do
      for each C∈MustBeAfter(A) do
        CanBeRightAfter(B) ← CanBeRightAfter(B) \ {C}
        CanBeRightBefore(C) ← CanBeRightBefore(C) \ {B}
        if C∉MustBeAfter(B) then    // new precedence B«C
          CanBeAfter(C) ← CanBeAfter(C) \ {B}
          CanBeBefore(B) ← CanBeBefore(B) \ {C}
          CanBeRightAfter(C) ← CanBeRightAfter(C) \ {B}
          CanBeRightBefore(B) ← CanBeRightBefore(B) \ {C}
          if C∉CanBeAfter(B) 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). Finally, the same deduction implies that B∈CanBeRightBefore(C) if and only if C∈CanBeRightAfter(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≤i≤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, 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≤i≤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.

Initialization: for every i ∈ J\{0} set
- CanBeAfter(i) ← J\{i}
- CanBeBefore(i) ← J\{i}
- Valid(i) ← {0,1}

Set-up: for every arc (i,j)∈E set
- CanBeAfter(j) ← CanBeAfter(j)\{i}
- CanBeBefore(i) ← CanBeBefore(i)\{j}

Propagation: whenever an activity 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.

Proposition 1: Let $i_0, i_1, \dots, i_m$ be a path in E such that Valid(i_j)=1 for all 1≤j≤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≤j≤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.

Q.E.D.

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).

Proposition 2: The worst-case time complexity of the propagation rule /1/ (instantiation of the Valid variable) including all possible recursive calls is $O(n^2)$, where n is a number of activities.

Proof: If an activity is made invalid then it is removed from the sets CanBeBefore, CanBeAfter, CanBeRightBefore, and CanBeRightAfter of all other activities which takes the total time O(n).

If activity A becomes valid then some new arcs may be added to the graph. The maximal number of such arcs is $\Theta(n^2)$ – when $\Omega(n)$ of the activities must be before A and $\Omega(n)$ of the activities must be after A. It may happen that some other activities (at most O(n)) become invalid to break cycles. However, we already know that the time complexity of making an activity invalid is O(n). Together, the worst-case time complexity to make an activity valid is $O(n^2)$.

Q.E.D.

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 (Barták & Čepek, 2005). The following rule updates the double precedence graph to keep transitive closure when an arc of type « is added to the double precedence graph.

```
A«B is added →                                                    /2/
  if A∈MustBeBefore(B) then exit
  CanBeAfter(B) ← CanBeAfter(B) \ {A}
  CanBeRightAfter(B) ← CanBeRightAfter(B) \ {A}
  CanBeBefore(A) ← CanBeBefore(A) \ {B}
  CanBeRightBefore(A) ← CanBeRightBefore(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) do
        CanBeRightAfter(C) ← CanBeRightAfter(C) \ {B}
        CanBeRightBefore(B) ← CanBeRightBefore(B) \ {C}
        if C∉MustBeBefore(B) then      // new precedence
            enqueue_for_propagation(add C«B)
    if Valid(B)=1 then    // transitive closure
      for each C∈MustBeAfter(B) do
        CanBeRightAfter(A) ← CanBeRightAfter(A) \ {C}
        CanBeRightBefore(C) ← CanBeRightBefore(C) \ {A}
        if C∉MustBeAfter(A) then      // new precedence
            enqueue_for_propagation(add A«C)
  exit
```

The rule /2/ does the following. If a new arc A«B is added then the sets CanBeBefore(A), CanBeAfter(B), CanBeRightBefore(A), and CanBeRightAfter(B) 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 one of endpoints of the added arc is valid then other arcs are added recursively to keep a transitive closure. If such an arc C«B is added then the arc $C«_dB$ between the same nodes is removed because there cannot be a direct precedence between B and C (there is A between B and C). The following proposition shows that all necessary arcs are added by rule /2/.

Proposition 3: If the precedence graph G is transitively closed and arc A«B is added to G then the propagation rule /2/ updates the precedence graph G to be transitively closed again.

Proof: Assume that arc A«B is added into G at a moment when arc B«C is already present in G. Moreover assume that Valid(A)≠0, Valid(B)=1, and Valid(C)≠0. We want to show that A«C is in G after rule /2/ is fired by the addition of A«B. The presence of arc B«C implies that C∈MustBeAfter(B) (and by symmetry also B∈MustBeBefore(C)). There are two possibilities. Either C∉MustBeAfter(A) in which case rule /2/ adds the arc A«C into G, or C∈MustBeAfter(A) (and by symmetry also A∈MustBeBefore(C)) which means that arc A«C was already present in G when arc A«B was added.

The case when arc A«B is added into G at a moment when arc C«A is already present in G and Valid(C)≠0, Valid(A)=1, Valid(B)≠0 holds can be handled similarly.

Thus when an arc is added into G, all paths of length two which include this new arc are either already spanned by a transitive arc, or the transitive arc is added by rule /2/. In the latter case this may invoke adding more and more arcs. However, this process is obviously finite (cannot cycle) as an arc is added into G only if it is not present in G, and if an arc is removed form G (breaking the cycle), it can never be added back as one of its endpoins becomes invalid (and thus is permanently disconnected from G).

Therefore, it is easy to see, that when the process of recursive arc additions terminates, the graph G is transitively closed. Indeed, for every path of length two in G one of the arcs is added later than the other, and we have already seen that at a moment of such an addition the transitive arc is either already on G or is added by rule /2/ in the next step.

Q.E.D.

Proposition 4: The worst-case time complexity of the propagation rule /2/ (adding a new arc) including all recursive calls to rules /1/ and /2/ is $O(n^3)$, where n is a number of activities.

Proof: If arc A«B is added and B must also be before A then one of the activities A or B will become invalid which takes time O(n) (see Proof of Proposition 2). If both A and B are undecided then the rule prunes sets CanBeAfter(B), CanBeRightAfter(B), CanBeBefore(A), and CanBeRightBefore(A) and exits without further propagation. If A is valid and B is undecided (or vice versa) then all predecessors of A are connected to B. There are at most O(n) such predecessors and the new arcs are added by recursive invocation of rule /2/. The recursion stops at this level because every predecessor X of a valid predecessor C of A is also a predecessor of A (due to transitive closure) and hence the arc X«B has already been enqueued for propagation when addition of A«B was processed. Moreover, any duplicate copy of the same arc in the queue will be processes in time O(1) (see the first line of rule /2/). The "worst" situation happens when both A and B are valid. Then all predecessors of A are recursively connected to all successors of B. There

are at most $O(n^2)$ such connections and processing each connection takes time O(n), so the worst-case time complexity is $O(n^3)$.

Q.E.D.

Proposition 5: The propagation rules /1/ and /2/ maintain correctly the sets CanBeRightBefore and CanBeRightAfter.

Proof: We will prove the proposition for the set CanBeRightBefore only, the set CanBeRightAfter is maintained symmetrically. At the beginning, the set CanBeRightBefore(B) contains all activities but B which is all right, because all activities are undecided. Each time A is deleted from CanBeBefore(B), A is also deleted from CanBeRightBefore(B) in both rules /1/ and /2/. If any C becomes valid, A∈MustBeBefore(C), and B∈MustBeAfter(C) then A is deleted from CanBeRightBefore(B) in rule /1/. If a new arc A«C is added, C is valid, and B∈MustBeAfter(C) then A is deleted from CanBeRightBefore(B) in rule /2/. Similarly, if a new arc C«B is added, C is valid, and A∈MustBeBefore(C) then A is deleted from CanBeRightBefore(B) in rule /2/. According to Definition 3 these are the only ways how the direct precedence relation can be influenced.

Q.E.D.

Propagating Time Windows

Our primary motivation for introducing the double precedence graph was modelling sequence-dependent setup times. However, the direct precedence relations between activities can also be used for propagating time windows. In (Barták & Čepek, 2005), we proposed a combination of the precedence graph with time windows propagation. In particular, we showed how time windows can be used to deduce new so called detectable precedences and how the precedence graph can be used to tighten lower and upper bounds of time windows. Moreover, if a lower bound for some time window is increased then this information can be incrementally propagated to lower bounds of time windows of subsequent activities. The following code from (Barták & Čepek, 2005) shows how increase of the earliest start time (est) of some valid activity A is propagated to the subsequent valid or undecided activities B.

for each B∈MustBeAfter(A) & ¬∃C Valid(C)=1 & A«C & C«B **do**
  est(B) ← est(A)+p(A)+
    ∑{p(X) | X∈MustBeBefore(B) & est(A)≤est(X) &
      Valid(X)=1}

Notice that only activities B such that ¬∃C Valid(C)=1 & A«C & C«B are influenced. These activities B are exactly the current direct successors of A according to Definition 3. Hence the code can be rewritten using a double precedence graph as follows:

for each B∈MustBeAfter(A)∩CanBeRightAfter(A) **do**
  est(B) ← est(A)+p(A)+
    ∑{p(X) | X∈MustBeBefore(B) & est(A)≤est(X) &
      Valid(X)=1}

Note also that if the earliest start time of some valid activity B that is a direct successor of A is increased then this increase is propagated to direct successors of B and so on. Hence, every successor of A can eventually be influenced. A similar incremental propagation can be realized for the latest completion time of activities.

Conclusions

The paper reports a work in progress on incremental filtering algorithms for unary resources with time windows, precedence relations, setup times, and optional activities. In particular, we focused on extending the precedence graph to provide information about direct precedence relations between the activities. We proposed a double precedence graph where this information is incrementally maintained and hence available in time O(1). We showed how direct precedence relations can be used in modelling sequence-dependent setup times and in propagation of time windows.

Acknowledgements

The research is supported by the Czech Science Foundation under the contract no. 201/04/1102.

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)*.

Barták R. and Čepek O. 2005. Incremental Propagation Rules for A Precedence Graph with Optional Activities and Time Windows. In *Proceedings of The 2nd Multidisciplinary International Conference on Scheduling: Theory and Applications (MISTA 2005)*, New York.

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.

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)*, Springer Verlag, 62-76.

Vilím P. and Barták R. 2002. Filtering Algorithms for Batch Processing with Sequence Dependent Setup Times. In *Proceedings of The Sixth International Conference on Artificial Intelligence Planning and Scheduling (AIPS 2002)*, AAAI Press, Toulouse, 312-320.