Incremental Maintenance of Double Precedence Graphs: A Constraint-Based Approach

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 additionally to traditional precedence relations (A can directly precede B if no activity must be allocated between A and B). Such precedence relations are motivated by solving problems like modelling sequence-dependent setup times or describing state transitions. We describe a constraint-based model for double precedence graphs and present incremental propagation rules for maintaining the transitive closure of the graph.

Introduction

As planning and scheduling technologies are coming together, the role of common solving techniques, like handling precedence relations, is increasing. In planning, the precedence is typically a consequence of a causal link between the activities. In scheduling, the precedence is derived from a sequence of activities in a job or it is forced by ordering of activities in a unary resource. Precedence relations can be deduced from time windows describing when the activities can be processed (Vilím 2002) and vice versa, precedence relations can be used to shrink the time windows (Laborie 2003) or (Cesta and Stella 1997). These approaches require information about predecessors and successors of each activity which is usually kept in a *precedence graph*. If there are frequent queries about predecessors and successors then it more efficient to keep a *transitive closure* of the precedence graph rather than to compute this information on demand. Note also that transitive closure simplifies detection of inconsistency of precedence relations. If there are activities A and B such that there are arcs from A to B and from B to A in a transitively closed precedence graph then the set of precedence relations is inconsistent. In the paper we propose a new constraint model of the precedence graph that incrementally updates its transitive closure.

In some problems, information about direct predecessors or successors is also necessary. Let us assume sequence dependent setup times in unary resources where the gap (setup time) between two successive activities depends on both activities. We can include such sequence dependent setup time in the duration of the second activity provided that we know the direct predecessor of the activity. In the paper we propose an extension of precedence graphs called *double precedence graphs* that keep information about direct predecessors and successors of each activity.

Finally, when solving some problems, at some time we may not know whether a particular activity appears in the precedence graph or not. Assume a problem with alternative resources, each resource having its own precedence graph. We may not know to which resource the activity belongs but we would like to reason about the activity. For example, if the activity causes inconsistency in some precedence graph then we know that the activity cannot be allocated to the corresponding resource. To model such situations, we propose to extend the double precedence graph by so called *optional activities*. At the beginning, such activities are undecided and they do not influence other activities in the graph but they can be influenced. When the activity is known to belong to the precedence graph then it becomes valid. If the activity is known not to belong to the graph then it becomes invalid. Invalidity of the activity can be deduced automatically, if the activity causes inconsistency, or it is decided by the solver (when the activity is allocated to another resource).

To summarise the contributions of this paper – we introduce optional activities and direct precedences into precedence graphs and we present incremental propagation rules that keep a transitive closure of this double precedence graph with optional activities. An important aspect of our proposal is the fact that we use sets to describe the precedence graph. This simplifies implementation of the proposed rules as propagators in constraint satisfaction packages (Dechter 2003) – the sets can be easily represented as domains of variables. Consequently, the proposed rules can be integrated with other constraint reasoning techniques, for example those from (Barták and Čepek 2005) or (Laborie 2003).

Double Precedence Graphs

The precedence relations among activities define a *precedence graph* that is an acyclic directed graph where nodes correspond to activities and there is an arc from A to B if A«B (A must be before B).

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 the 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 by induction of the path length 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. Hence, if no optional activity is used (all activities are valid) then Definition 2 is identical to Definition 1.

As we argued in Introduction, for some problems it is important to keep also the direct precedence relations between the activities.

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 (the relation « is 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 problem solving proceeds, the arcs «$_d$ are typically removed from the graph. Moreover, if parallel activities are not assumed then in the solution 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).

Constraint Model

As mentioned in the Introduction we propose to realise a reasoning on precedence relations using a constraint satisfaction technology. Our model achieves global consistency and provides full information about precedence relations and direct precedence relations to all other constraints. This model also fully integrates reasoning on 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 undecided 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. Because adding new arcs is realised by removing values from above two sets, these sets can be easily implemented as finite domains of two variables (in a CSP it is only possible to remove values from domains as reasoning proceeds). To simplify description of the propagation rules we also 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.

To model direct precedence relations we add two sets to each activity: CanBeRightBefore and CanBeRightAfter containing initially values 1,..,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 $\in$ CanBeRightBefore(B) $\Leftrightarrow$ B $\in$ 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).

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 described sets. Moreover, we guarantee global consistency, that is, every value that remains in the domain after filtering can be part of some solution (proofs are omitted due to space limits).

We initiate the double precedence graph in the following way. First, the variables CanBeBefore(A), CanBeAfter(A), CanBeRightBefore(A), CanBeRightAfter(A), and Valid(A) with their domains are created for every activity A. Then the known precedence relations are added by pruning the domains of CanBeBefore(A), CanBeAfter(A), CanBeRightBefore(A), and CanBeRightAfter(A) as described above. Note, that because all activities are still undecided at this stage, domain change is not propagated to other variables. 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
            enqueue_for_propagation(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).

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. It is possible to prove 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 (we omit the proof due to space limits). However, in some situations arcs may be added to the precedence graph during the solving procedure, either by the user, by the scheduler/planner, or by other filtering algorithms like the one described in (Barták and 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
    enqueue_for_propagation(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). Note finally that the propagators for new arcs are evoked after the propagator of the current rule finishes. Hence, new arcs are enqueued for later propagation. It may happen that the same arc is present more times in the propagation queue. To prevent running the propagation rule more times for the same arc, we check at the beginning whether the rule has already been invoked and in such a case, the propagator is stopped immediately. Again, it is possible to prove that 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. 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.

Explicit Direct Precedence Relations. So far, we assumed that the direct precedence relations are derived only from existing "normal" precedence relations. It is possible to prove that in such a case the propagation rules /1/ and /2/ maintain correctly the sets CanBeRightBefore and CanBeRightAfter and this propagation is complete, that is, if $A \in$ CanBeRightBefore(B) then there exists a scenario in which A is scheduled right before B. In some problems there may be also explicit direct precedence relations. For example, if there is not enough time for setup between A and B then we can deduce that A cannot directly precede B ($\neg$ A «$_d$ B). We will now present a rule that propagates information about such explicit direct precedence relations.

When a direct precedence A «$_d$ B is forbidden then A is deleted from the set CanBeRightBefore(B) and B is deleted from the set CanBeRightAfter(A). According to Definition 3 we can deduce that either A is after B (B « A) or there must be a valid activity C between A and B. Actually, there must be some C before B and right after A and some D after A and right before B (C=D may happen):

$$\text{CanBeRightAfter}(A) \cap \text{CanBeBefore}(B)=\emptyset \Rightarrow B\text{«}A$$

$$\text{CanBeAfter}(A) \cap \text{CanBeRightBefore}(B)=\emptyset \Rightarrow B\text{«}A.$$

The above implications can be used to deduce that B must be before A, if there is no activity that may appear between A and B. Vice versa, if A must be before B and both A and B are valid then we can actively look for activities between A and B. The following rule realizes this deduction.

```
CanBeRightAfter(A) or CanBeAfter(A) or CanBeBefore(A) or
CanBeRightBefore(B) or CanBeBefore(B) or CanBeAfter(B) is
changed, or Valid(A) or Valid(B) is instantiated →          /3/
  if Valid(A)=0 or Valid(B)=0 or A∈MustBeAfter(B) then exit
  if CanBeRightAfter(A)∩CanBeBefore(B)=∅
    or CanBeAfter(A)∩CanBeRightBefore(B)=∅ then
      enqueue_for_propagation(add B«A)
      exit
  if A∈MustBeBefore(B) & Valid(A)=1 & Valid(B)=1 then
      if {C}=CanBeRightAfter(A)∩CanBeBefore(B) then
        // C is the only possible direct successor of A
        enqueue_for_propagation(add A«C)
        enqueue_for_propagation(add C«B)
        Valid(C) = 1
        exit
      if {C}= CanBeAfter(A)∩CanBeRightBefore(B) then
        // C is the only possible direct predecessor of B
        enqueue_for_propagation(add A«C)
        enqueue_for_propagation(add C«B)
        Valid(C) = 1
        exit
```

The propagation rule /3/ is activated only when a direct precedence relation is discarded from the double precedence graph due to "external" reasons. Specifically, this rule is not active if sets CanBeRightBefore and CanBeRightAfter are changed within rules /1/ and /2/. This is because, rules /1/ and /2/ remove the direct precedence due to Definition 3 so no further propagation can be achieved by rule /3/. The following propagation rule /4/ removes the direct precedence relation from the graph and activates rule /3/.

```
A«dB is deleted →                                           /4/
  CanBeRightAfter(A) ← CanBeRightAfter(A) \ {B}
  CanBeRightBefore(B) ← CanBeRightBefore(B) \ {A}
  activate rule /3/ for A and B
  exit
```

There are several differences between rules /1/ and /2/ and rule /3/. The rules /1/ and /2/ are called only once and they deduce all changes in the double precedence graph. The rule /3/ can be evoked more times until it ensures validity of Definition 3, that is, either B is after A or there is a valid activity C between A and B. The open question is whether this rule deduces all changes in the precedence graph or whether the propagation can be further strengthened.

Conclusions

The paper presents a constraint-based approach to model double precedence graph where both standard and direct precedence relations are represented and optional activities are assumed. We proposed incremental propagation rules that keep a transitive closure of the precedence graph and achieve global consistency. We also proposed a propagation rule for explicit direct precedences that are useful to model problems like sequence-dependent setup times. Last but not least, the proposed model can be easily integrated with other constraint reasoning techniques like reasoning on time windows. Hence, we believe that this approach can accelerate exploitation of constraint satisfaction technology in solving problems that include both planning and scheduling components.

References

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.

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.

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

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