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

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 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 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 filtering algorithms 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). However, we are not aware of any algorithm that actively combines information about precedence relations and time windows. Detectable precedences by Vilím (2002) are probably the first attempt for such a combination, but a transitive closure is not assumed there.

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. Using optional activities in the precedence-based filtering is more complicated because the transitive closure should not assume such activities until they are known to be in the final schedule. We are not aware of any precedence-based filtering that uses optional activities.

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. 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. It should be stressed here that the rules are designed in such a way that they can be easily converted into a filtering algorithm that can be integrated as a new constraint into existing constraint satisfaction packages (as opposed to implementing everything from scratch). 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 and 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). 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*. Otherwise the activity is *invalid*. 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 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 (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 a 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. 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 not known to be valid 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. 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 there is also an arc A to C in G. Note that if no optional activity is used then the above defined notion of transitive closure becomes identical to the classical one.

In the next paragraphs we will propose a constraint model for the precedence graph and a set of propagation rules that maintain the transitive closure of the graph with optional activities. We index each activity by a number from the set 1,..,n, where n is the number of activities. For each activity we use three variables to describe its position in the graph and validity: CanBeBefore, CanBeAfter, and Valid. 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, and Valid is a 0/1 variable 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 variable is {0,1}. There is the following reason for this representation: adding an arc to the precedence graph is realised via pruning domains of two variables. In particular, 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. We use the following initial domains for CanBeBefore and CanBeAfter variables for activity A: {0,...,n}-{A}. We use the value 0 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. We can define the following symbolic variables (whose value is not kept in memory but computed upon 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 construct the precedence graph incrementally. First, the variables with their domains are created. Then the known precedence relations are added in the above-described way. 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).

The first propagation rule disconnects the activity from the precedence graph when the activity becomes invalid. "Valid(A)=0" is a trigger of this rule saying that value 0 has been assigned to variable Valid(A). The part after → is the propagator describing pruning of domains. For example, "A∉CanBeBefore(B)" means that value A is removed from the domain of variable CanBeBefore(B). "exit" means that the constraint represented by the propagation rule is entailed. We will use the same notation in all rules.

```
Valid(A)=0 →                                   /1/
  for each B do
    A∉CanBeBefore(B)
    A∉CanBeAfter(B)
  exit
```

The second propagation rule computes the transitive closure when the activity becomes valid.

```
Valid(A)=1 →                                   /2/
  for each B∈MustBeBefore(A) do
    for each C∈MustBeAfter(A) do
      B∉CanBeAfter(C)
      if B∉CanBeBefore(C) then  // break the cycle
        post_constraint(Valid(B)=0 ∨ Valid(C)=0)
      C∉CanBeBefore(B)
      if C∉CanBeAfter(B) then   // break the cycle
        post_constraint(Valid(B)=0 ∨ Valid(C)=0)
  exit
```

Notice that rule /2/ can detect a cycle in the precedence graph. Cycles are allowed in the precedence graph with optional activities only if one of the activities in the cycle is invalid which breaks the cycle. When a cycle is detected we post a constraint ensuring that the above condition will be satisfied for the cycle. By using the constraint, the decision about which of the activities is invalid is actively postponed. Notice also that if the variable domains are maintained consistently, the condition B∉CanBeBefore(C) is valid if and only if the condition C∉CanBeAfter(B) is valid, and hence the last two lines in rule /2/ are in fact redundant and can be removed.

It can be shown, that if the entire precedence graph is known in advance (no arcs are added during the solving procedure), then rule /2/ is sufficient for keeping the (generalised) transitive closure (an arc spans every path in which all inner vertices are valid).

However, 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 two rules /3/ and /4/ ensure that transitive closure is updated when an arc (say from A to B) is added to the graph, which causes A to be removed from CanBeAfter(B), and B to be removed from CanBeBefore(A). It should be stressed here that rules /3/ and /4/ are triggered only when a new arc is added, not when the domains are pruned by rule /2/. Again, rules /3/ and /4/ can detect a cycle in the precedence graph. This

situation is treated similarly as in rule /2/ (and when both rules /3/ and /4/ are used after an edge is added it is sufficient to test for the cycle just in one of them). Notice also that domain pruning occurs only if at least one of the endpoints of the added arc is valid (at the time of arc addition).

```
A∉CanBeAfter(B) →                                          /3/
  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
      C∉CanBeAfter(B)
      B∉CanBeBefore(C)

B∉CanBeBefore(A) →                                         /4/
  if B∉CanBeAfter(A) then     // break the cycle
     post_constraint(Valid(A)=0 ∨ Valid(B)=0)
  else if Valid(B)=1 then     // transitive closure
  for each C∈MustBeAfter(B) do
    C∉CanBeBefore(A)
    A∉CanBeAfter(C)
```

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:

```
est(Ω) = min{est(A), A∈Ω}
lst(Ω) = max{lst(A), A∈Ω}
ect(Ω) = min{ect(A), A∈Ω}
lct(Ω) = max{lct(A), A∈Ω}
```

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).

There are two groups of propagation rules working with time windows. The first group is triggered by shrinking the time window (/5/ for increased est and /6/ for decreased ltc). The rules in this group can deduce a new precedence relation using the time windows. This precedence relation is called a detectable precedence in (Vilím, 2002). 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).

```
est(A)↑ →                                                  /5/
  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
          // A cannot be before B
          A∉CanBeBefore(B)
          B∉CanBeAfter(A)
  if Valid(A)=1 then
       for each B∈MustBeAfter(A) do
          est(B) ← est(A)+p(A)+
∑{p(X)|X∈MustBeBefore(B) & X∈CanBeAfter(A) &
est(A)≤est(X) & Valid(X)=1}

lct(A)↓ →                                                  /6/
  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
             // B cannot be before A
             B∉CanBeBefore(A)
             A∉CanBeAfter(B)
       if Valid(A)=1 then
          for each B∈MustBeBefore(A) do
             lct(B) ← lct(A)-p(A)-
  ∑{p(X)|X∈MustBeAfter(B) & X∈CanBeBefore(A)
  & lct(X)≤lct(A) & Valid(X)=1}
```

The second group of rules reacts to changes in the precedence graph. In particular these rules are evoked by adding a new precedence relation (/3a/ and /4a/) or by making the activity valid (/2a/). Because these rules have the same triggers as the rules for the precedence graph, they can be actually combined with them. Hence, we index the new rules using the number of the corresponding rule for the precedence graph.

The new rules shrink the time windows using information about the precedence relations. Basically, if

we know that a set of activities Ω must be processed before some activity A and the activities from Ω cannot overlap (unary resource) then we can compute the earliest start time of A as the maximal completion time of activities in Ω. In fact, we can take any subset Ω' of Ω and apply the above principle to Ω'. This principle is already used in rules like edge-finding. Note finally, that these rules are applied only when the activity is valid. This corresponds to our requirement that optional activities that are not yet known to be valid should not influence other activities. As soon as the activity becomes valid, it should influence all other activities even if these activities are not yet know to be valid (/2a/).

Valid(A)=1 → /2a/
  for each B∈MustBeAfter(A) 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) 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}

A∉CanBeAfter(B) → /3a/
  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}

A∉CanBeBefore(B) → /4a/
  if Valid(A)=1 & Valid(B)≠0 then
    let Ω = {X | X∈MustBeAfter(B) &
        X∈CanBeBefore(A) & Valid(X)=1}
    lct(B) ← min{lct(Ω'∪{A})-p(Ω')-p(A) | Ω'⊆Ω}

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 proofs of soundness and completeness, detail comparison to existing propagation rules (edge finder, etc.), implementation of the proposed rules, and testing in real-life environment.

Remark

This is a slightly revised and extended version of the paper submitted to MISTA 2005.

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.

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.

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.