Incremental Filtering Algorithms for Precedence and Dependency Constraints

Roman Barták, Ondřej Čepek
Charles University in Prague, Faculty of Mathematics and Physics
Malostranské nám. 2/25, 118 00 Praha 1, Czech Republic
E-mail: roman.bartak@mff.cuni.cz, ondrej.cepek@mff.cuni.cz

Abstract

Precedence constraints play a crucial role in planning and scheduling problems. Many real-life problems also include dependency constraints expressing logical relations between the activities – for example, an activity requires presence of another activity in the plan. For such problems a typical objective is a maximization of the number of activities satisfying the precedence and dependency constraints. In the paper we propose new incremental filtering rules integrating propagation through both precedence and dependency constraints. We also propose a new filtering rule using the information about the requested number of activities in the plan. We demonstrate efficiency of the proposed rules on the log-based reconciliation problems and min-cutset problems.

1. Introduction

Planning and scheduling belong among the most successful application areas of constraint satisfaction. Solving these problems depends on efficient handling of temporal and resource constraints. The simplest but popular form of a temporal constraint is a precedence relation expressing that one activity must appear before another activity in the plan. In addition to precedence relations, many problems also include dependency constraints between the activities. This is typical for planning problems where existence of activity in the plan depends on presence of other activities in the plan. Similar constraints appear in oversubscribed problems where the task is to schedule the maximal number of activities and inclusion of an activity in the schedule may require presence of other activities in the schedule [6]. Such problems can be modelled using optional activities; the system then decides about validity or invalidity of optional activities respecting all the constraints.

In this paper we focus on modelling precedence constraints using a precedence graph and on integrating reasoning on dependency constraints in this model. In particular, we propose a new constraint-based model of the precedence graph with optional activities and we design new filtering rules for incremental maintenance of transitive closure for such precedence graphs. In the filtering we also use information about dependency constraints. This is, we believe, the first time when filtering through precedence and dependency constraints is realised in an integrated way. We also propose new objective-based filtering for these problems. This filtering uses information about the requested number of valid activities in the final plan.

The paper is organized as follows. We will first introduce the problem more formally and survey the existing solving approaches. Then we will describe the filtering rules for maintaining a transitive closure of the precedence graph with optional activities. We will also show their theoretical time complexity and prove their soundness. After that, we will describe the propagation rule doing filtering based on requested number of valid activities. We will conclude the paper with experimental comparison of our approach with the existing model.

2. Problem description and related works

In this paper we address the problem of modelling precedence constraints between the activities in over-subscribed problems. The *precedence constraint* A « B specifies that activity A must be before activity B in the schedule. To model over-subscribed problems, we assume *optional activities*. An optional activity has one of the following three statuses. If the activity is not yet known to be or not to be included in the schedule then it is called *undecided*. If the activity is included in the schedule then it is called *valid*. If the activity is known not to be included in the schedule then it is called *invalid*. We also assume dependency constraints between the activities. The *dependency constraint* A => B specifies that if activity A is valid then activity B must be valid as well. In other words, if activity A is included in the schedule then activity B must be included as well. This is one of the dependency constraints proposed in the general model for manufacturing scheduling [6]. The scheduling task is to decide about (in)validity of the undecided activities and to find a sequence of valid activities satisfying the

precedence and dependency constraints. Usually, the problem is formulated as an optimization problem, where the task is to find a feasible solution in the above sense that maximizes the number of valid activities.

Though our motivation is mainly in the area of scheduling, the above problem is also known as a log-based reconciliation problem in databases. The straightforward constraint model for this problem has been proposed in [2]. The model uses *n* integer variables $p_1,\ldots, p_n$ giving the position of activities in the schedule (*n* is the number of activities). The initial domain of these variables is 1,…,*n*. There are also *n* Boolean (0/1) variables $a_1,\ldots, a_n$ describing whether the activity is valid (1) or invalid (0). The precedence constraint between activities *i* and *j* is then described using the formula:

$(a_i \wedge a_j) \Rightarrow (p_i < p_j)$ or equivalently $(a_i * a_j * p_i < p_j)$.

The dependency constraint between activities *i* and *j* can be formulated as:

$$a_i \Rightarrow a_j.$$

The solver uses standard constraint propagation over above constraints combined with enumeration of the Boolean variables a_i's. The paper [2] also proves that the log-based reconciliation problem is NP-hard – if there are no dependency constraints then the problem reduces to the problem of finding the smallest cutset in a directed graph (that is, the smallest set of vertices whose remove makes the input graph acyclic) [4].

In [3] an improvement of the above precedence constraint has been proposed using the reasoning on graph properties. Namely a global cutset constraint has been proposed that uses graph contraction techniques to infer some simple Boolean constraints. Still, this model assumes the dependency constraints separately; in particular the constraints are modelled in the above implication form.

The paper [5] also studies the log-based reconciliation problem, but rather than proposing a new filtering algorithm, a decomposition technique is used. The technique is again motivated by the minimal cutset problem and the dependency constraints are handled separately. Moreover, as opposed to the above described models, the technique from [5] is incomplete – meaning that it does not guarantee optimality.

Our approach is different from the above techniques by integrating reasoning on both precedence and dependency constraints. We cannot use the contraction techniques from [3], because our aim is to eventually use the designed filtering algorithm in a scheduler where the precedence graph is used by other constraints like the constraint that integrates reasoning on precedence relations and time windows [1].

3. Filtering rules for precedence and dependency constraints

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. If access to all predecessors and successors of a given activity is frequently requested, like in [1], then it is more efficient to keep a transitive closure of the graph where this information is available in time O(1), rather than to look for predecessors/successors on demand. We propose the following definition of transitive closure of the precedence graph with optional activities.

Definition 1: 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 on the path length). Hence, if no optional activity is used (all activities are valid) then Definition 1 corresponds to a standard definition of the transitive closure.

We propose to realise reasoning on precedence relations using constraint satisfaction technology. This allows integration of our model with other constraint reasoning techniques, namely the one proposed in [1]. This integration requires the model to provide full information about precedence relations to all other constraints. 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 – not yet known 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(A) is a set of indices of activities that can be before activity A. CanBeAfter(A) is a set of indices of activities that can be after activity A. For simplicity reasons we will write A instead of the index of A. To simplify description of the propagation rules we also define for every activity A the following derived 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 (Figure 1).

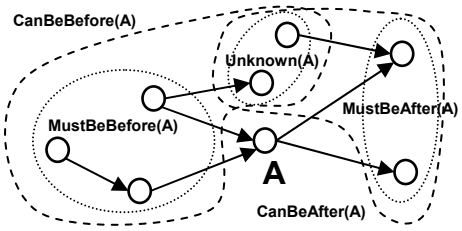

Figure 1. Representation of the precedence graph

Note on representation. The main reason for using sets to model the precedence graph is their possible representation as domains of variables in constraint satisfaction packages. Recall that domains of variables can only shrink as problem solving proceeds. The sets in our model are also shrinking as new arcs « are added to the precedence graph. Hence a special data structure is not necessary to describe the graph in constraint satisfaction packages. Moreover, these packages usually provide tools to manipulate the domains, for example membership and deletion operations. In the subsequent complexity analysis, we will assume that these operations require time O(1), which can be realised for example by using a bitmap representation of the sets. Note finally, that empty domain implies inconsistency that may be a problem for the very first and very last activity which has no predecessors and successors respectively. To solve the problem we can simply leave activity A in both sets CanBeAfter(A) and CanBeBefore(A). Then no domain of CanBeBefore and CanBeAfter will ever be empty but we can detect inconsistency via the empty domain of Valid variables.

The goal of propagation rules is to remove inconsistent values from the above described sets – this is called domain filtering in constraint satisfaction. First, we will focus on making a transitive closure of the precedence graph according to Definition 1. Note that the transitive closure of the precedence graph also simplifies detection of inconsistency of the graph. The precedence graph is inconsistent if there is a cycle of valid activities. In a transitively closed graph, each such cycle can be detected by finding two valid activities such that A « B and B « A. Our propagation rules prevent cycles by making invalid the last undecided activity in each cycle. This propagation is realised by using an exclusion constraint. As soon as there is a cycle A « B and B « A detected, the following exclusion constraint can be posted:

$$\text{Valid}(A) = 0 \vee \text{Valid}(B) = 0.$$

This constraint ensures that each cycle is broken by making at least one activity in the cycle invalid. Instead of posting the constraint directly to the constraint solver, we propose keeping the set Ex of exclusions. The above exclusion constraint is modelled as a set $\{A,B\} \in Ex$. Now, the propagation of exclusions is realised explicitly – if activity A becomes valid then all activities C such that $\{A,C\} \in Ex$ are made invalid (see rule /1/ below).

In addition to precedence constraints, there are also dependency constraints in the problem. The dependency A => B can be easily described using the constraint:

$$(\text{Valid}(A) = 1) \Rightarrow (\text{Valid}(B) = 1).$$

Similarly to exclusions, we propose to keep the set Dep of dependencies instead of posting the above constraints, and to realise the propagation of dependencies explicitly. In particular, if activity A becomes valid then all activities C such that $(A{=>}C) \in Dep$ are made valid. Reversely, if activity A becomes invalid then all activities C such that $(C{=>}A) \in Dep$ are made invalid (see rule /1/ below).

Keeping exclusions and dependencies explicitly has the advantage of stronger filtering. In particular, if exclusion {A,B} is to be added to Ex and there is a dependency $(A{=>}B) \in Dep$ then we can make activity A invalid because A must be invalid in any solution satisfying the above exclusion and dependency constraints (the exclusion is resolved so it can be removed from Ex). Moreover, if {A,B} is added to Ex and there is an activity C such that $(C{=>}A) \in Dep$ and $(C{=>}B) \in Dep$ then we can make activity C invalid. Again, C must be invalid in any solution satisfying the above exclusion and dependency constraints.

The above reasoning is realised by the following propagation rule that 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 C s.t. (C=>A)∈Dep do Valid(C) ← 0
   Ex := Ex \ {{A,X} | X is an activity}
 else  // Valid(A)=1
   for each C s.t. (A=>C)∈Dep do Valid(C) ← 1
   for each C s.t. {A,C}∈Ex do Valid(C) ← 0
   for each B∈MustBeBefore(A) s.t. Valid(B)≠0 do
     for each C∈MustBeAfter(A)\MustBeAfter(B)
                                 s.t. Valid(C)≠0 do
       CanBeAfter(C) ← CanBeAfter(C) \ {B}
       CanBeBefore(B) ← CanBeBefore(B) \ {C}
       if C∉CanBeAfter(B) then // break cycle
         if (C=>B)∈Dep then Valid(C) ← 0
         else if (B=>C)∈Dep then Valid(B) ← 0
            else
              Ex ← Ex ∪ {{B,C}}
              for each X s.t. (X=>B)∈Dep and
                                 (X=>C)∈Dep do
                 Valid(X) ← 0
exit
```

Note that rule /1/ maintains symmetry of sets modelling the precedence graph for all valid and undecided activities because the domains are pruned symmetrically in pairs. 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 transitive closure according to Definition 1.

Proposition 1: Let A_0, A_1, … , A_m be a path in the precedence graph such that Valid(A_j)=1 for all $1 \le j \le m-1$ and Valid(A_0)≠0 and Valid(A_m)≠0 (that is, the endpoints of the path are not invalid and all inner points of the path are valid). Then $A_0 \ll A_m$, that is, $A_0 \notin$ CanBeAfter(A_m) and $A_m \notin$ CanBeBefore(A_0).

Proof: We shall proceed by induction on m. The base case m=1 is trivially true after initialisation (we assume that for every arc (X,Y) in the precedence graph X is removed from CanBeBefore(Y) and Y is removed from CanBeAfter(X) in the initialisation phase). 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(A_j) ← 1 was set last among all inner points A_1, … , A_{m-1} on the path. By the induction hypothesis we get

- $A_0 \notin$ CanBeAfter(A_j) and $A_j \notin$ CanBeBefore(A_0) using the path A_0, … , A_j
- $A_j \notin$ CanBeAfter(A_m) and $A_m \notin$ CanBeBefore(A_j) using the path A_j, … , A_m

We shall distinguish two cases. If $A_m \in$ MustBeAfter(A_0) (and by symmetry also $A_0 \in$ MustBeBefore(A_m)) then by the definition (of the MustBeBefore sets) we get $A_m \notin$ CanBeBefore(A_0) and $A_0 \notin$ CanBeAfter(A_m) and so the claim is true trivially. Thus let us in the remainder of the proof assume that $A_m \notin$ MustBeAfter(A_0).

Now let us show that $A_0 \in$ CanBeBefore(A_j) must hold, which in turn (together with $A_0 \notin$ CanBeAfter(A_j)) implies $A_0 \in$ MustBeBefore(A_j). Let us assume by contradiction that $A_0 \notin$ CanBeBefore(A_j). However, at the time when both $A_0 \notin$ CanBeAfter(A_j) and $A_0 \notin$ CanBeBefore(A_j) became true, that is, when the second of these conditions was made satisfied by rule /1/, rule /1/ must have done one the following things

- in case of a dependency constraint between A_0 and A_j, make one of these activities invalid
- in case of no dependency between A_0 and A_j, add the pair (A_0,A_j) into the set Ex of exclusions.

The latter case moreover implies that at the moment when A_j is made valid A_0 is made invalid and hence both cases contradict the assumptions of the lemma.

By a symmetric argument we can prove that $A_m \in$ MustBeAfter(A_j). Thus when rule /1/ is triggered by setting Valid(A_j)←1 both $A_0 \in$ MustBeBefore(A_j) and $A_m \in$ MustBeAfter(A_j) hold (and $A_m \notin$ MustBeAfter(A_0) is assumed), and therefore rule /1/ removes A_m from the set CanBeBefore(A_0) as well as A_0 from the set CanBeAfter(A_m), which finishes the proof.

Q.E.D.

Proposition 2: If implemented properly, the worst-case time complexity of the propagation rule /1/ including all possible recursive calls is $O(n^3)$, where n is a number of activities.

Proof: If an activity A is made invalid then it is necessary to find all the activities it is dependent on. This can be done in O(n) if the dependency graph as well as its transposed graph (where edges are reversed) is represented by adjacency lists, or if it is represented by an adjacency matrix (one matrix is then sufficient as it is easy to read out both predecessors and successors of A). Also the removal of all exclusion pairs that include A can be done in O(n) if the exclusion pairs are kept in memory as a symmetric n x n binary matrix. The recursive calls that make other activities invalid thus take O(n) per activity and at most n activities can be made invalid, so the total time for all the recursive calls is $O(n^2)$.

If activity A becomes valid then the detection of dependencies and exclusions (not counting the recursive calls) can be handled in O(n) as above. The recursive calls that make activities invalid take O(n) per activity (as proved above), which gives a total $O(n^2)$ for all such activities. The recursive calls that make activities valid take $O(n^2)$ per activity (as will be proved below), which gives a total $O(n^3)$ for all such activities.

In the two nested loops where new arcs may be added to the graph up to $\Theta(n^2)$ pairs B,C may be inspected for activity A, so this inspection (deciding for which pairs B,C an arc should be added) can take up to $\Theta(n^2)$ for each activity A. This gives the $O(n^2)$ bound used above for each recursive call that makes an activity valid.

It is important to note, that only $O(n^2)$ arcs can be added to the graph during all recursive calls, so the part of the code inside the two nested loops is executed $O(n^2)$ times over all recursive calls (using this bound individually for each activity A which is made valid would yield an overall $O(n^4)$ time bound). The part of the code inside the two nested loops (excluding the recursive calls) takes O(n) time (because of the for loop, all other tests can be performed in O(1) time). Thus we get a total $O(n^3)$ bound for all executions of the code inside the two nested loops (excluding the recursive calls) and a total $O(n^2)$ bound for all recursive calls that make activities invalid.

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/planner, or by other filtering

algorithms like in [1]. The following rule /2/ updates the precedence graph to keep transitive closure when an arc is added to the precedence graph. We can also use the same rule for the initialisation of precedence graph – the known arcs are added using this rule rather than added by explicit changes of sets CanBeBefore and CanBeAfter.

```
A«B is added →                                          /2/
  if A∈MustBeBefore(B) then exit
  CanBeAfter(B) ← CanBeAfter(B) \ {A}
  CanBeBefore(A) ← CanBeBefore(A) \ {B}
  if A∉CanBeBefore(B) then   // break the cycle
    if (A=>B)∈Dep then Valid(A) ← 0
    else if (B=>A)∈Dep then Valid(B) ← 0
      else
        Ex ← Ex ∪ {{A,B}}
        for each X s.t. (X=>A)∈Dep and
                        (X=>B)∈Dep do
           Valid(X) ← 0
  else    // transitive closure
    for each C∈MustBeBefore(A)\MustBeBefore(B) do
      if Valid(A)=1 or
               (C=>A)∈Dep or (B=>A)∈Dep then
        add C«B
    for each C∈MustBeAfter(B)\MustBeAfter(A) do
      if Valid(B)=1 or
               (C=>B)∈Dep or (A=>B)∈Dep then
        add A«C
exit
```

The rule /2/ does the following. If a new arc A«B is added then we first check whether the arc is not already present in the graph. If it is a new arc then the corresponding sets are updated and a possible cycle is detected (we use the same reasoning as in rule /1/). Finally, if any end point of the arcs is valid, then necessary arcs are added to update the transitive closure according to Definition 1. Moreover, we can add more arcs using information about dependencies – this is useful for earlier detection of possible cycles. Assume that arc A«B has been added. If (B=>A) ∈ Dep then all predecessors of A can be connected to B like in the case when A is valid. This is sound because if B becomes valid then A must be valid as well and such arcs will be added anyway and if B becomes invalid then any arc related to B is irrelevant. For the same reason, if there is any predecessor C of A such that (C=>A) ∈ Dep then C can be connected to B. The same reasoning can be applied to successors of B. Note that the propagators for new arcs are evoked after the propagator of the current rule finishes. The following proposition shows that all necessary arcs are added by rule /2/.

Proposition 3: If the precedence graph G is transitively closed (in the sense specified by Definition 1) and arc A « B is added to G then 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)). Now 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 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 with a valid midpoint 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 no arc is ever removed from G. More on the time complexity of arc additions follows in Proposition 4.

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 with a valid midpoint one of the arcs on the path 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 in 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: Every recursive call to rule /1/ is making some activity invalid, so following the arguments from the proof of Proposition 2, we get that the total time needed to process all such calls is $O(n^2)$. The rest of the code, excluding the recursive calls to itself (to rule /2/), can be executed in O(n) time. To see this it is enough to realize that each test for dependency or exclusion can be handled in O(1) time (if the dependency graph and exclusion pairs are stored using a matrix representation as in the proof of Proposition 2) and therefore each of the three "for each" loops can be handled in O(n) time. Because only $O(n^2)$ arcs can be added over all recursive calls the total $O(n^3)$ time bound follows.

Q.E.D.

4. Objective-based filtering rule

As we mentioned in the introduction, a typical objective in problems with optional activities is a maximization of the number of valid activities. Such objective can be converted into the following constraint:

$$Obj = \Sigma_A \, Valid(A)$$

where the task is to maximize the value of variable Obj. This constraint can be realized as it stands, that is, as the sum of variables Valid. In this section, we will present a filtering rule realizing stronger propagation through this constraint. Namely, the rule can deduce better bounds for variable Obj and the rule can also deduce values of some not-yet decided Valid variables.

The proposed filtering rule is based on ideas of constructive disjunction. If activity A is still undecided, we will explore both alternatives, namely Valid(A) = 1 and Valid(A) = 0, to find out their influence on variable Obj and vice versa. Recall, that variables Valid participate in dependency and exclusion constraints and these constraints are explicitly available via sets Dep and Ex. We will use these constraints to estimate bounds of variable Obj. In particular, if activity A becomes valid (Valid(A) = 1) then all undecided activities B such that (A=>B) ∈ Dep must also become valid and, similarly, all undecided activities C such that {A,C} ∈ Ex must become invalid. Symmetrically, if activity A becomes invalid (Valid(A) = 0) then all undecided activities B such that (B=>A) ∈ Dep must also become invalid. Using this deduction and taking into account the numbers of known valid and invalid activities we can estimate bounds for variable Obj. These computed bounds are then used to define better bounds for Obj and vice versa, by comparing the computed bounds with the current bounds of Obj, we can deduce that one of the alternatives is not viable and hence the remaining alternative is forced (unless, both alternatives are not viable and then a failure is detected). For example, if the computed lower bound of Obj for Valid(A) = 1 is greater than the current upper bound of Obj then it is not possible to assign value 1 to Valid(A).

The following filtering rule /3/ realises the above described reasoning (N is a number of activities there). Note, that the filtering rule is not idempotent, that is, the rule is expected to be called again if it proposes a change to any Valid variable or a change to Obj variable. An idempotent version of the rule would be possible but then the rule should integrate propagation rule /1/ and the code would become more complicated (while the pruning power would be the same).

It may seem that the filtering power of rule /3/ can be further strengthen by the following deduction. Irrespectively of assigning 0 or 1 to Valid(A), the activities from the set { C : Valid(C)={0,1} ∧ {A,C}∈Ex ∧ (A=>C)∈Dep } must become invalid and hence their Valid variables can be set to 0. This is surely true but notice that exclusion {X,Y} is added to set Ex by rules /1/ and /2/ only if neither (X=>Y) nor (Y=>X) are elements of Dep. If this is ensured for any exclusion {X,Y} then the above mentioned set will always be empty and hence the deduction based on this set is useless.

```
bounds of Obj changed or
any Valid(X) instantiated →                              /3/
  NumValid ← |{X : Valid(X)=1}|
  NumInvalid ← |{X : Valid(X)=0}|
  MinObj ← lb(Obj)  // current lower bound
  MaxObj ← ub(Obj)  // current upper bound
  LB ← max( MinObj, NumValid )
  UB ← min( MaxObj, N - NumInvalid )
  for each A s.t. Valid(A)={0,1} do
    ValidLB ← 1+ NumValid +
         |{C : Valid(C)={0,1} ∧ (A=>C)∈Dep }|
    ValidUB ← N - NumInvalid -
         |{C : Valid(C)={0,1} ∧ {A,C}∈Ex }|
    InvalidLB ← NumValid
    InvalidUB ← N - 1 - NumInvalid -
         |{C : (C=>A)∈Dep }|
    if ValidLB ≤ MaxObj & ValidUB ≥ MinObj then
      if InvalidLB ≤ MaxObj & InvalidUB ≥ MinObj
      then
        LB ← max( LB, min(ValidLB,InvalidLB) )
        UB ← min( UB, max(ValidUB,InvalidUB) )
      else
        Valid(A) ← 1
        LB ← max( LB, ValidLB )
        UB ← min( UB, ValidUB) )
    else
     if InvalidLB ≤ MaxObj & InvalidUB ≥ MinObj
      then
        Valid(A) ← 0
        LB ← max( LB, InvalidLB )
        UB ← min( UB, InvalidUB) )
      else fail
  end for
  lb(Obj) ← LB
  ub(Obj) ← UB
  if NumValid + NumInvalid = N then exit
```

5. Experimental results

To evaluate the practical applicability of the proposed filtering rules, we did some preliminary experiments with log-based reconciliation problems and min-cutset problems. The proposed filtering rules were implemented in SICStus Prolog 3.12.3 using the standard interface for the definition of global constraints. The experiments run under Windows XP Professional on 1.1 GHz Pentium-M processor with 1280 MB RAM.

5.1. Log-based reconciliation problems

Though our original motivation to introduce dependency constraints into a precedence graph is in scheduling, log-based reconciliation problems fit perfectly our problem specification where precedence and dependency constraints are combined. We took the problem set from [3] and we compared our approach with the constraint model proposed in [2]. Unfortunately implementation of the cutset global constraint proposed in [3] was not available to us so we have no direct

comparison of runtimes yet. Nevertheless, for two problems, where neither approach found (proved) an optimal solution, our technique improved significantly the lower bound of the objective function. Table 1 presents the results for the CLP model (Original) from [2] and our approach (Precedence). We compare both runtime (RT – measured in milliseconds) and the number of backtracks (BT) to find an optimal solution. We used a limit of 50 minutes to cut the search and we report the best solution found within this time limit (recall that the task is to maximize the number of valid activities).

Table 1. Log-based reconciliation benchmarks from [3].

Bench	Original			Precedence		
	Best	RT	BT	Best	RT	BT
r100v1	98	141	16	98	438	1
r100v2	77	250	85	77	125	3
r100v3	95	156	49	95	313	7
r100v4	100	31	1	100	360	1
r100v5	52	16	3	52	62	5
r200v1	65	63	13	65	78	5
r200v2	191	74657	8015	191	3313	42
r500v1	198	219	3	198	407	5
r500v2	498	1265	32	498	2547	2
r800v1	770	-	-	780	-	-
r800v2	318	3828	327	318	984	10
r1000v1	389	672	3	389	1266	5
r1000v2	935	-	-	957	-	-

We have found most of the problems quite easy; frequently the first found solution was the optimal solution. The runtime of our approach for these problems is slightly longer than in the original model; this is due to overhead for building more complex data structures. Nevertheless, the table clearly demonstrates that our approach requires significantly less backtracks to find the solution so the filtering power of the proposed propagation rules pays off there. The table also demonstrates that as soon as the problems are becoming harder, the difference between our approach and the original model is more significant (see problems r200v2 and r800v2). For two problems, r800v1 and r1000v2, neither approach was able to find/prove an optimal solution within the fifty minutes limit. Nevertheless, our propagation rules lead to much better lower bound. The lower bounds for these problems reported in [3] are 771 for r800v1 and 943 for r1000v2, so we also improved the best lower bounds reported there.

To support the above claim that our approach is prevailing over the original model for harder problems, we did a second set of experiments using pseudo-real log-based reconciliation problems proposed in [5]. These problems have a structure typical for real-life problems so the results are more interesting from the practical point than using completely random problems. Table 2 shows the specification of problems used in our experiment – this specification is identical to problems used in [5], though we generated own problems because the problems from [5] were not available. The table also shows the best solutions obtained in our experiments.

Table 2. Pseudo-real log-based reconciliation problems.

Bench	Act	Prec	Dep	Original best	Precedence best
p50-3	150	162	175	146	146
p50-4	200	229	211	193	193
p50-5	250	290	346	244	244
p50-6	300	375	377	288	290
p50-7	350	451	468	333	333
p50-8	400	527	593	376	378
p50-9	450	630	680	404	406

We again compared the CLP model proposed in [2] with our filtering rules. We used the time limit of four hours (14 400 000 milliseconds) to cut search, Table 2 reports the best solution found within this time limit. Starting with p50-6, the original model was not able to find/prove the optimal solution within the time limit while our technique found and proved optimal solutions for all the problems. Figure 2 shows the comparison of runtimes and the number of backtracks for both approaches (we use a logarithmic scale). Our approach requires more than an order of magnitude less backtracks to find the solution and it also requires much less time.

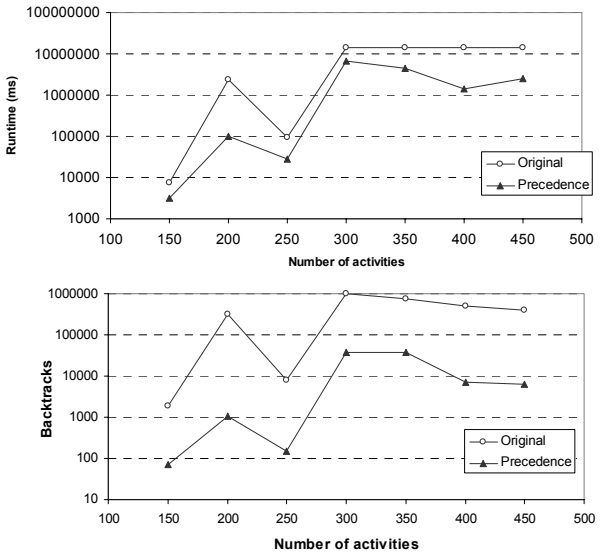

Figure 2. Computation results on pseudo-real log-based reconciliation problems

5.2. Min-cutset problems

We believe that using a precedence graph is better than using absolute positioning in a sequence for modelling problems with precedence relations. Though our approach is proposed for problems with both precedence and dependency constraints, we also demonstrate superiority of the precedence graph over absolute positioning on a well known min-cutset problem. The min-cutset problem consists of precedence relations only and the task is to find the largest set of vertices such that the sub-graph induced by these vertices does not contain any cycle (or symmetrically to find the smallest set of vertices such that all cycles are broken if these vertices are removed from the graph). This problem is known to be NP-hard [4].

We use the data set from [7] to compare our approach based on the precedence graph with the CLP model from [2] based on absolute positioning in the sequence of activities. All the problems in the data set consist of 50 activities while the number of precedence constraints varies. Figure 3 shows the comparison of runtimes and the number of backtracks for both approaches (we use a logarithmic scale). Again our approach requires more than an order of magnitude less backtracks and less runtime to find the optimal solution. In fact, with the exception of problems with 50 and 100 precedence constraints, the original CLP model was not able to find the optimal solution (or to prove optimality) within the time limit of 50 minutes while our approach (Precedence) found and proved optimal solutions. Note finally, that concerning the runtime we cannot compete with the GRASP heuristic proposed in [7], but this was not our original ambition as we tackle different problems. Moreover, opposite to the GRASP approach our technique is complete and, indeed, for some problems we have found better solutions than reported in [7].

6. Conclusions

In the paper we proposed new incremental filtering rules integrating reasoning on precedence and dependency constraints in the context of constraint satisfaction. We experimentally demonstrated that our approach is prevailing over the existing model on log-based reconciliation problems and min-cutset problems. Though we focused on a particular form of dependencies, we believe that our approach is extendable to other dependency constraints, for example, those in [6] where existence of some activity forces removal of another activity etc. Moreover, with the exception of cost-based filtering, our model can be extended to open precedence graphs where the number of activities is not known in advance and new activities are added to the precedence graph as the solving proceeds.

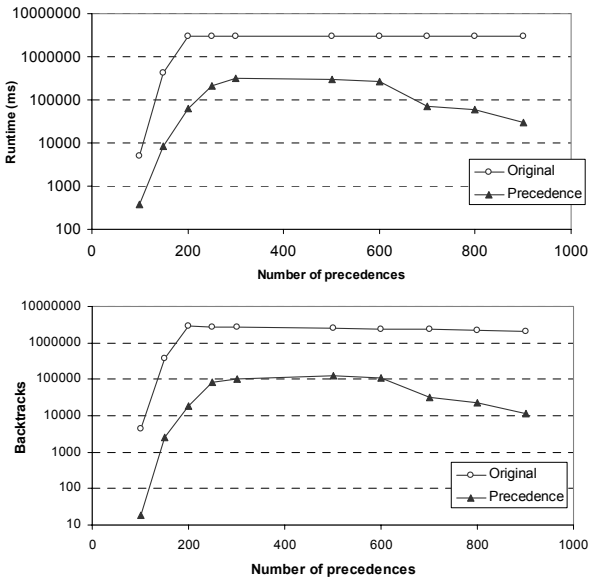

Figure 3. Computation results on min-cutset problems

7. Acknowledgements

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

8. References

[1] R.Barták, "Incremental Propagation of Time Windows on Disjunctive Resources", *Proceedings of the Nineteenth International Florida Artificial Intelligence Research Society Conference*, AAAI Press, 2006, pp. 25-30.

[2] F. Fages, "CLP versus LS on log-based reconciliation problems for nomadic applications", *ERCIM CompulogNet Workshop on Constraints*, Praha, 2001.

[3] F. Fages, A. Lal, "A Global Constraint for Cutset Problems", *Proceedings of Fifth International Workshop on Integration of AI and OR techniques in constraint programming for combinatorial optimization problems (CPAIOR'03)*, Montreal, 2003.

[4] M. R. Garey, D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman & Comp., San Francisco, 1979.

[5] Y. Hamadi, "Cycle-cut decomposition and log-based reconciliation", *ICAPS Workshop on Connecting Planning Theory with Practice*, Whistler, 2004, pp. 30-35

[6] W. Nuijten, T. Bousonville, F. Focacci, D. Godard, C. Le Pape, "MaScLib: Problem description and test bed design", 2003. http://www2.ilog.com/masclib

[7] P.M. Pardalos, T. Qian, M.G. Resende, "A greedy randomized adaptive search procedure for the feedback vertex set problem" *Journal of Combinatorial Optimization*, 2:399-412, 1999.