Sweep Synchronization as a Global Propagation Mechanism

Nicolas Beldiceanu*, Mats Carlsson*, and Sven Thiel+

*SICS, Lägerhyddsvägen 18, SE-75237 Uppsala, Sweden
{nicolas,matsc}@sics.se
+MPI für Informatik, Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany
sthiel@mpi-sb.mpg.de

January 23 2003
SICS Technical Report T2003:02
ISRN: SICS-T–2003:02-SE
ISSN: 1100-3154

Abstract. This paper presents a new generic filtering algorithm that simultaneously considers n conjunctions of constraints as well as those constraints mentioning some variables Y_k of the pairs X,Y_k ($1 \leq k \leq n$) occurring in these conjunctions. The main benefit of this new technique comes from the fact that, for adjusting the bounds of a variable X according to n conjunctions, we do not perform n sweeps in an independent way but rather synchronize them. We then specializes this technique to the non-overlapping rectangles constraint where we consider the case where several rectangles of height one have the same X coordinate for their origin as well as the same length. For this specific constraint we come up with an incremental bipartite matching algorithm which is triggered while we sweep over the time axis. We illustrate the usefulness of this new pruning method on a timetabling problem, where each task can't be interrupted and requires the simultaneous availability of n distinct persons. Each person has its own periods of unavailability and can only perform one task at a time.

Keywords: Constraint Programming, Non-overlapping, Sweep, Timetabling.

Sweep Synchronization as a Global Propagation Mechanism

Nicolas Beldiceanu*, Mats Carlsson*, and Sven Thiel+

*SICS, Lägerhyddsvägen 18, SE-75237 Uppsala, Sweden
{nicolas,matsc}@sics.se
+MPI für Informatik, Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany
sthiel@mpi-sb.mpg.de

Abstract. This paper presents a new generic filtering algorithm that simultaneously considers n conjunctions of constraints as well as those constraints mentioning some variables Y_k of the pairs X,Y_k ($1 \leq k \leq n$) occurring in these conjunctions. The main benefit of this new technique comes from the fact that, for adjusting the bounds of a variable X according to n conjunctions, we do not perform n sweeps in an independent way but rather synchronize them. We then specializes this technique to the non-overlapping rectangles constraint where we consider the case where several rectangles of height one have the same X coordinate for their origin as well as the same length. For this specific constraint we come up with an incremental bipartite matching algorithm which is triggered while we sweep over the time axis. We illustrate the usefulness of this new pruning method on a timetabling problem, where each task can't be interrupted and requires the simultaneous availability of n distinct persons. Each person has its own periods of unavailability and can only perform one task at a time.

1 Introduction

It has been quoted in [10] that one of the important challenges for constraint programming, is to "identify innovative ways to combine constraints to produce efficient pruning techniques". The purpose of this paper is to present an instance of such a constraint pattern and its corresponding filtering algorithm. In [2] we have introduced a generic sweep algorithm for adjusting the minimum or the maximum value of a domain variable[1] X according to a conjunction of constraints where all constraints mention X as well as another domain variable Y. The main contribution of this paper is to present a generalization of the previous algorithm to the following constraint pattern. We now consider n conjunctions $\mathcal{C}_k$ $(1 \leq k \leq n)$ of constraints such that all the constraints of each conjunction $\mathcal{C}_k$ mention a given pair of domain

[1] A domain variable is a variable that ranges over a finite set of integers; min(V) and max(V) respectively denote the minimum and maximum values of variable V, while dom(V) and sizedom(V) respectively designates the set of possible values of variable V and the number of elements of that set.

variables X, Y_k [2]. In addition we also take into account those additional constraints mentioning some of the variables Y_k of the previous pairs of variables X, Y_k. $\mathcal{S}$ denotes the conjunction of these additional constraints throughout this paper, while m_k designates the number of constraints of $\mathcal{C}_k$. The main benefit of this new technique is to obtain more pruning by replacing n independent sweeps by one single process where we coordinate the different sweeps.

The new filtering algorithm is based on an idea which is widely used in computational geometry and which is called sweep [4]. Consider the illustrative example given in Fig. 1 where we have five constraints and their projection on two given variables, and assume that we want to find out the smallest value of X. By trying out $X = 0$, $X = 1$, $X = 2$ and $X = 3$ we conclude that $X = 3$ is the first possible value. The new synchronized sweep algorithm performs this search efficiently; See Sect. 3 for details on this particular example.

In dimension 2, a plane sweep algorithm solves a problem by moving a vertical line from left to right along the abscissa. It uses the following data structures:

- a data structure called the *sweep-line status*, which contains some information related to the current position Δ of the sweep-line,
- a data structure named the *event point series*, which holds the events to process, ordered in increasing order wrt. the abscissa.

The algorithm initializes the sweep-line status for the initial value of Δ. Then the sweep-line jumps from event to event; each event is handled, updating the sweep-line status. In our context, the sweep-line scans the values of the domain variable X that we want to prune. The sweep-line status contains for each value-variable pair val - Y_k $(1 \le k \le n)$ such that $val \in \mathrm{dom}(Y_k)$ the fact that both assigning Δ to X and val to Y_k is compatible or not with the conjunction of constraints $\mathcal{C}_k$ (i.e. the restricted domain of Y_k according to the fact that $X = \Delta$). If, for some value of Δ, the restricted domains of $Y_1,..,Y_n$ are incompatible with some constraint of $\mathcal{S}$, then we will remove Δ from $\mathrm{dom}(X)$. The synchronized sweep filtering algorithm will try to adjust the minimum[3] value of X wrt. a set of conjunctions $\mathcal{C}_k$ $(1 \le k \le n)$ of constraints by moving a sweep-line from the minimum value of X to its maximum value. In our case, the events to process correspond to the starts and the ends of forbidden regions wrt. the constraints occurring in $\mathcal{C}_k$ $(1 \le k \le n)$ as well as the pairs of variables X, Y_k. Throughout this paper, we use the notation $\left(R_x^- .. R_x^+, R_y^- .. R_y^+\right)$ to denote for an ordered pair R of intervals its lower and upper bounds.

A prerequisite for using this technique is to have a polynomial algorithm for checking a necessary condition for the satisfiability of the conjunction of constraints $\mathcal{S}$. A weak way of achieving this is to check the satisfiability of each constraint of $\mathcal{S}$ independently. This is what is done in the example of Sect. 3. A stronger way is to check for a global necessary condition for the satisfiability of $\mathcal{S}$. In this later case, this

---

[2] A constraint mentioning X, Y_{k_1}, Y_{k_2} $(k_1 \neq k_2)$ should be included in more that one conjunction.

[3] It can also be used in order to adjust the maximum value, or to prune completely the domain of a variable.

essentially means that the constraints of $\mathcal{S}$ have a specific structure of which one takes advantage for deriving a necessary condition, which can be evaluated efficiently. This is what is done for the timetabling problem presented in Sect. 4.

The next section recalls the notion of forbidden regions, which is a way to represent constraints that is suited for the algorithms of this paper. Sect. 3 describes the synchronized sweep algorithm itself. Finally Sect. 4 presents its specialization to the non-overlapping rectangles constraint where each rectangle has a height of one as well as the same length, and where in addition we take into account the fact that some rectangles have the same X coordinate for their origin. For this purpose we come up with an incremental bipartite matching algorithm adapted to the fact that, while we sweep over the time axis, we hide and restore some nodes and all their attached edges.

2 Constraint Description

We call an ordered pair R of intervals a *forbidden region of the constraint* CTR *wrt. the variables* X and Y_k if: $\forall x \in R_x^-..R_x^+$, $\forall y \in R_y^-..R_y^+$: $CTR(V_1,..,V_k)$ with the assignment $X = x$ and $Y_k = y$ has no solution, no matter which values are taken by the other variables of constraint $CTR(V_1,..,V_k)$. Fig. 1 of Sect. 3 gives several concrete examples of constraints and their respective forbidden regions.

The synchronized sweep algorithm computes the forbidden regions on request, in a lazy evaluation fashion. The forbidden regions of each constraint CTR mentioning a pair of variables $(X,Y_k)\ (1 \le k \le n)$ are gradually generated as a set of forbidden rectangles $R_{k1},...,R_{kl}$ such that:

- $R_{k1} \cup \cdots \cup R_{kl}$ represents all forbidden regions of constraint CTR wrt. variables X and Y_k,
- the forbidden rectangles $R_{k1},...,R_{kl}$ do not pairwise intersect,
- $R_{k1},...,R_{kl}$ are sorted by ascending start position on the X axis.

GETNEXTFORBIDDENREGIONS$(X,Y_k,CTR,previous)$ is used for gradually getting the forbidden regions for each triple $(X,Y_k,CTR)\ (1 \le k \le n)$ that we want to be processed by the synchronized sweep algorithm. It generates all the forbidden regions R_{CTR} of constraint CTR such that $R_{CTR_x}^- = next_{CTR}$ and $R_{CTR_y}^+ \ge \min(Y_k) \wedge R_{CTR_y}^- \le \max(Y_k)$, where $previous$ is the position of the previous start event of constraint CTR and $next_{CTR}$ is the smallest value greater than $previous$ such that there exists such a forbidden region R_{CTR} of CTR.

3 A Synchronized Sweep Algorithm

We describe the new synchronized sweep algorithm which can coordinate several sweep-lines in order to achieve more pruning. Since all the different sweep-lines (i.e.

one sweep-line for each conjunction of constraints) move along the same abscissa (i.e. the domain of *X*), we actually merge them within one single sweep. As this algorithm is generic because its takes various functions as input parameters we will not analyze its worst-case complexity since we would not get any sharp result which has some significance in practice. But we will make the worst-case analysis of an instance of this algorithm described in the next section.

As usual for sweep algorithms the principal task is to come up with an incremental way of handling the modifications of the sweep-line status. In fact, our new synchronized sweep algorithm is similar to the one introduced in [2], except regarding the organization of the sweep-line status.

Data Structures. As is the case for most sweep algorithms, the new synchronized sweep algorithm uses one data structure for recording the sweep-line status and another data structure for storing the event points. For the current position Δ of the sweep-line, the *sweep-line status* contains for each possible value val of Y_k $(1 \le k \le n)$ the number $nforbid_k[val]$ of forbidden regions that currently intersect the sweep-line at the point of coordinates Δ, val. In addition we have for each variable Y_k $(1 \le k \le n)$ a counter $CountZeros_k$ that gives the number of values $val \in \mathrm{dom}(Y_k)$ for which $nforbid_k[val] = 0$. Finally a last counter $CountContradictions$ indicates the number of variables Y_k $(1 \le k \le n)$ for which $CountZeros_k = 0$ ($CountZeros_k$ is equal to 0 when the domain of Y_k is empty according to the hypothesis that $X = \Delta$). The *event point series*, denoted Q_{event}, contains the start and end+1 on the X axis, of the forbidden regions that intersect the sweep-line of the constraints of $\mathcal{C}_k$ $(1 \le k \le n)$ wrt. variables X and Y_k. These events are sorted in increasing order and recorded in a heap.

Synchronization Primitives.

Since we don't want to assume what sort of constraints are present in $\mathcal{S}$ or what kind of algorithm is used for checking whether the conjunction of constraints of $\mathcal{S}$ may be feasible, we employ the following set of synchronization[4] primitives in order to make our synchronized sweep algorithm generic:

- INITSYNCHRONIZATION$(Y_1..Y_n, y_1..y_n)$ tells that we want to synchronize according to variables $Y_1..Y_n$, and in addition performs the same job as ISSYNCHRONIZED,
- ISSYNCHRONIZED$(y_1..y_n)$ returns `FALSE` if the conjunction of constraints of $\mathcal{S}$ is for sure false; returns `TRUE` if the conjunction of constraints of $\mathcal{S}$ holds or if we can't find out whether it holds or not[5]; returns also in $y_1..y_n$ predicted values for $Y_1..Y_n$ (i.e. values which may satisfy the constraints of $\mathcal{S}$); when we enter the procedure, $y_1..y_n$ contains the values predicted for $Y_1..Y_n$ by the last call to ISSYNCHRONIZED or to INITSYNCHRONIZATION,

---

[4] The term *synchronization* denotes the fact that, for a given position Δ of the sweep line, we want to check whether the current domains of $Y_1,..,Y_n$ are compatible with the conjunction of the constraints of $\mathcal{S}$.

[5] Since for certain type of constraints, checking whether there exists a solution or not is NP-hard, one would only use a necessary condition which can be tested in polynomial time; this explains why we took this definition for the result returned by ISSYNCHRONIZED.

– TELLNEWEVENTTOSYNCHRONIZATION$(kind, k, first, last)$ if *kind*=start (respectively end), notifies the synchronization process that, for the current position of the sweep-line, there is a new start (respectively end) event for variable Y_k for all values between *first* and *last*.

Principle of the Algorithm. In order to check if $X = \Delta$ is feasible wrt. each conjunction of constraints $\mathcal{C}_k$ $(1 \le k \le n)$, the sweep-line status records for each value *val* of Y_k the number $nforbid_k[val]$ of forbidden regions intersecting the point of coordinates Δ, val. If, for $X = \Delta$, $\exists k \in 1..n$ such that $\forall val \in \mathrm{dom}(Y_k)$: $nforbid_k[val] > 0$, the sweep-line will move to the right to the next event to handle; in addition, if this is not the case, the synchronization process (i.e. the primitive ISSYNCHRONIZED) checks the constraints mentioning variables $Y_1,..,Y_n$ according to the restricted domains of variables $Y_1,..,Y_n$: since we make the hypothesis that $X = \Delta$, the domain of Y_k $(1 \le k \le n)$ consists only of those values *val* for which $nforbid_k[val] = 0$. If we find a contradiction the sweep-line will also move to the right to the next event.

Table 1. Status of the sweep-line at each stage of the algorithm. For each position Δ $(0 \le \Delta \le 3)$ of the sweep-line, it gives for each pair $\mathcal{C}_k, val$ $(1 \le k \le 2,\ 0 \le val \le 4)$ the corresponding number of forbidden regions $nforbid_k[val]$.

	Δ = 0		Δ = 1		Δ = 2		Δ = 3	
	$\mathcal{C}_1$	$\mathcal{C}_2$	$\mathcal{C}_1$	$\mathcal{C}_2$	$\mathcal{C}_1$	$\mathcal{C}_2$	$\mathcal{C}_1$	$\mathcal{C}_2$
4	0	2	0	1	1	1	1	1
3	0	1	1	1	1	1	2	1
2	1	1	1	0	2	0	1	1
1	1	0	2	0	1	0	1	0
0	2	0	1	0	1	0	0	0

Before going more into the detail of the algorithm, let us first illustrate how it works on a concrete example. Assume that we want to find out the minimum value of variable X such that the conjunction of the five constraints $\texttt{alldifferent}(\{X, Y_1, W\})$, $|X - Y_1| > Z$, $X + 2 \cdot Y_2 \le S$, $X + 1 \le T \vee T + 1 \le X \vee Y_2 + 1 \le U \vee U + 4 \le Y_2$, $|Y_1 - Y_2| < 2$ that are given in Fig. 1 holds. In this example, we have the following two conjunctions of constraints: conjunction $\mathcal{C}_1$ consists of those constraints mentioning variables X and Y_1 (i.e. $\texttt{alldifferent}(\{X, Y_1, W\})$ and $|X - Y_1| > Z$), while conjunction $\mathcal{C}_2$ contains those constraints mentioning X and Y_2 (i.e. $X + 2 \cdot Y_2 \le S$ and $X + 1 \le T \vee T + 1 \le X \vee Y_2 + 1 \le U \vee U + 4 \le Y_2$). Finally the set $\mathcal{S}$ of synchronization constraints contains the last constraint $|Y_1 - Y_2| < 2$. Table 1 shows the content of the sweep-line status for different values of Δ. More precisely, it gives for each value of Δ and for each possible value of Y_1 and Y_2 the respective number of forbidden regions of $\mathcal{C}_1$ and $\mathcal{C}_2$. Let us now explain how we find out the possibly smallest feasible value of X :

– If $X = 0$ the domain of Y_1 and Y_2 are respectively restricted to values 3,4 and 0,1; since according to these restrictions the synchronization constraint $|Y_1 - Y_2| < 2$ can't hold (i.e. as one can check on the rightmost part of Fig. 1, the restricted domain of Y_1 and Y_2 is included in a forbidden region of $|Y_1 - Y_2| < 2$), 0 is not a feasible value for X .

– If $X = 1$ we get also a contradiction for a similar reason as when $X = 0$.

- If $X = 2$ the domain of Y_1 is empty and so, 2 is not a feasible value for X .
- Finally, if $X = 3$ the domain of Y_1 and Y_2 are respectively restricted to values 0 and 0,1; since according to these restrictions there exists at least one solution for the synchronization constraint $|Y_1 - Y_2| < 2$ (i.e. $Y_1 = 0, Y_2 = 0$), the sweep-line stops and the minimum value of X is adjusted to value 3.

The Main Procedure. The procedure FINDMINIMUM (see Algorithm 1) implements the synchronized sweep algorithm. It adjusts the minimum value of a variable X wrt. a set of conjunctions of constraints $\mathcal{C}_1,..,\mathcal{C}_n$ and a set $\mathcal{S}$ of synchronization constraints, such that for each conjunction $\mathcal{C}_k$ $(1 \le k \le n)$, all constraints mention the pair of variables X, Y_k . The main parts of FINDMINIMUM are given below.

- Lines 1-8 initialize the event queue to the start and end events associated to the leftmost forbidden regions of each constraint present in one of the conjunctions of constraints $\mathcal{C}_1,..,\mathcal{C}_n$. Note that, for a constraint of a conjunction $\mathcal{C}_k$, we only insert events that are within $\min(X)..\max(X)$ and $\min(Y_k)..\max(Y_k)$.
- Lines 9-10 check if we can avoid sweeping over the domain of X . This is actually the case if the following two conditions simultaneously hold:
  (1) No event was inserted into the event queue.
  (2) The current restrictions of the domain variables $Y_1,..,Y_n$ is compatible with the synchronization constraints of $\mathcal{S}$.
- Line 12 initializes to 0 each array $nforbid_k[\min(Y_k),..,\max(Y_k)]$ of the sweep-line status which is associated to the conjunction $\mathcal{C}_k$ $(1 \le k \le n)$. Line 13 sets $nforbid_k[val]$ to 1, for those values val not in $\mathrm{dom}(Y_k)$. These values will not be considered any more, since no corresponding end event will be added.
- Lines 15-16 extract from the event queue all events associated to the current position Δ of the sweep-line and update the sweep-line status. Afterwards line 17 checks whether there exists some feasible solution for $X = \Delta$. If this is the case, line 18 exits with a success.
- Line 19 reports a failure since a complete sweep over the full domain of variable X was done without finding any position Δ without contradiction.

Input: a domain variable X that occurs in all the constraints of *n* conjunctions $\mathcal{C}_1,..,\mathcal{C}_n$ of constraints, and for each conjunction $\mathcal{C}_i$ $(1 \le i \le n)$ a domain variable Y_i which occurs in all the constraints of $\mathcal{C}_i$. Finally a set $\mathcal{S}$ of additional constraints which all mention some variables Y_i $(1 \le i \le n)$ and values $\hat{y}_1 .. \hat{y}_n$ which are the predicted values for variables $Y_1..Y_n$ by the last call to ISSYNCHRONIZED .

Output: An indication that no solution exists or an indication that a solution exists; values $\hat{x}$, $\hat{y}_1 .. \hat{y}_n$.

Ensure: If a solution may exist then $\hat{x}$ is the smallest value of X such that:

(1) $\forall i \in 1..n : \hat{y}_i \in \mathrm{dom}(Y_i)$,

(2) $\forall i \in 1..n : (\hat{x}, \hat{y}_i)$ does not belong to any forbidden region of any constraint of the conjunction $\mathcal{C}_i$ wrt. variables X and Y_i ,

(3) $Y_1 = \hat{y}_1 .. Y_n = \hat{y}_n$ is an assignment that may be compatible with the additional constraints which all mention variables Y_i $(1 \le i \le n)$.

1: $Q_{event} \leftarrow$an empty event queue, $CountContradictions \leftarrow 0$.
2: **if** INITSYNCHRONIZATION$(Y_1..Y_n, \hat{y}_1..\hat{y}_n) = 0$ **then return** (**false**, $\hat{x}$, $\hat{y}_1 .. \hat{y}_n$).
3: **for** each conjunction of constraints $\mathcal{C}_k$ $(1 \le k \le n)$ **do**
4: $CountZeros_k \leftarrow \text{sizedom}(Y_k)$.
5: **for** each constraint CTR_i $(1 \le i \le m_k)$ of $\mathcal{C}_k$ **do**
6: **for** each forbidden region $R_{CTR_i} \in$ GETNEXTFORBIDDENREGIONS$(X, Y_k, CTR_i, \min(X)-1)$ **do**
7: Insert $\max(R_{CTR_i}{}_x^-, \min(X))$ into Q_{event} as a start event associated to $\mathcal{C}_k$.
8: **if** $R_{CTR_i}{}_x^+ + 1 \le \max(X)$ **then** Insert $R_{CTR_i}{}_x^+ + 1$ into Q_{event} as an end event of $\mathcal{C}_k$.
9: **if** (Q_{event} is empty **or** the leftmost position of any event of Q_{event} is greater than $\min(X)$)
10: **and** ISSYNCHRONIZED$(\hat{y}_1..\hat{y}_n)$ **then** $\hat{x} \leftarrow \min(X)$, **return** (**true**, $\hat{x}$, $\hat{y}_1 .. \hat{y}_n$).
11: **for** each conjunction of constraints $\mathcal{C}_k$ $(1 \le k \le n)$ **do**
12: $nforbid_k \leftarrow$array ranging over $\min(Y_k)..\max(Y_k)$ initialized to 0.
13: $nforbid_k[val] \leftarrow 1$, **for** $val \in \min(Y_k).\max(Y_k) \setminus \text{dom}(Y_k)$[6].
14: **while** Q_{event} is not empty **do**
15: $\Delta \leftarrow$ the leftmost position of any event of Q_{event} .
16: **for** each event E at position Δ of Q_{event} **do** HANDLEEVENT(E) .
17: **if** $CountContradictions = 0$ **and** ISSYNCHRONIZED$(\hat{y}_1..\hat{y}_n)$ **then**
18: $\hat{x} \leftarrow \Delta$, **return** (**true**, $\hat{x}$, $\hat{y}_1 .. \hat{y}_n$).
19: **return** (**false**, $\hat{x}$, $\hat{y}_1 .. \hat{y}_n$).

Algorithm 1: FINDMINIMUM$(\mathcal{C}_1,..,\mathcal{C}_n, \mathcal{S}, X, Y_1,..,Y_n)$

Handling Start and End Events. Depending on whether we have a start or an end event E that comes from a forbidden region R_E associated with a conjunction $\mathcal{C}_k$ we add 1 or -1 to $nforbid_k[i]$ ($l \le i \le u$), where l and u are respectively the start and the end on the Y_k axis of the forbidden region R_E . In addition, when we have a start event which removes a value i for the first time from Y_k (i.e. $nforbid_k[i] = 0$), we notify the synchronization process that value i was removed from Y_k . Conversely, when we have an end event which restores a value i of a variable Y_k (i.e. $nforbid_k[i] = 1$) we also inform the synchronization process of this fact. When E was the last start event in Q_{event} of a given constraint CTR , we search for the next events of CTR and insert them into the event queue Q_{event} .

[6] In Alg. 1, $A \setminus B$ denotes the set difference between A and B .

1: Extract E from Q_{event} and get the corresponding forbidden region R_E, constraint CTR and conjunction $\mathcal{C}_k$.
2: **if** E is a start event **then**
3: $kind \leftarrow$ start, $trigger \leftarrow 0$, $inc \leftarrow 1$, $dec \leftarrow -1$.
4: **if** Q_{event} does not contain any start event associated to constraint CTR and conjunction $\mathcal{C}_k$ **then**
5: $previous\_x_E \leftarrow \underline{R_E}_x$.
6: **for** each forbidden region $R_{CTR_i} \in$ GETNEXTFORBIDDENREGIONS$\left(X, Y, CTR, previous_x_E\right)$ **do**
7: Insert $\underline{R_{CTR_i}}_x$ into Q_{event} as a start event associated to $\mathcal{C}_k$.
8: **if** $\overline{R_{CTR_i}}_x + 1 \leq \max(X)$ **then** Insert $\overline{R_{CTR_i}}_x + 1$ into Q_{event} as an end event of $\mathcal{C}_k$.
9: **else** $kind \leftarrow$ end, $trigger \leftarrow 1$, $inc \leftarrow -1$, $dec \leftarrow 1$.
10: $prev\_nforbid \leftarrow -1$, $l \leftarrow \max\left(\underline{R_E}_y, \min\left(Y_k\right)\right)$, $u \leftarrow \min\left(\overline{R_E}_y, \max\left(Y_k\right)\right)$.
11: **for** each $i \in l..u$ **do**
12: $cur\_nforbid \leftarrow nforbid_k[i]$.
13: **if** $cur\_nforbid = trigger$ **then**
14: **if** $prev\_nforbid \neq trigger$ **then** $first \leftarrow i$.
15: $last \leftarrow i$, $CountZeros_k \leftarrow CountZeros_k + dec$.
16: **if** $CountZeros_k = trigger$ **then** $CountContradictions \leftarrow CountContradictions + inc$.
17: **else**
18: **if** $prev\_nforbid = trigger$ **then** TELLNEWEVENTTOSYNCHRONIZATION$\left(kind, k, first, last\right)$.
19: Add inc to $nforbid_k[i]$, $prev\_nforbid \leftarrow cur\_nforbid$.
20: **if** $prev\_nforbid = trigger$ **then** TELLNEWEVENTTOSYNCHRONIZATION$\left(kind, k, first, last\right)$.

Algorithm 2: HANDLEEVENT(E)

We consider again the example presented just after Table 1 and illustrate the updates of the sweep-line status performed by HANDLEEVENT as well as the synchronization process. Fig. 1 shows 5 constraints and their respective forbidden regions (shaded) wrt. two given variables and their domains. The statement *Var* in *min..max*, where *min* and *max* are two integers such that *min* is less than or equal to *max*, creates a domain variable *Var* for which the initial domain is made up from all values between *min* and *max* inclusive. The first constraint requires X, Y_1 and W be pairwise distinct, while the last four constraints correspond to arithmetic and disjunctive constraints. Fig. 2 illustrates the updates of the sweep-line status for adjusting the minimum value of X according to the conjunction of constraints given in Fig. 1: We show for each position Δ of the sweep-line the corresponding status. It consists of the two "profiles" $nforbid_1[0..4]$ and $nforbid_2[0..4]$ which respectively give for each value v of Y_1 and Y_2 the number of forbidden regions that currently intersect the sweep-line at the point of coordinate Δ, v. Then we have the three counters $CountZeros_1$, $CountZeros_2$ and $CountContradictions$ which were described in the "Data Structures" section. The synchronization process is described by giving the forbidden regions of the synchronization constraint $|Y_1 - Y_2| < 2$ as described in Fig. 1. In addition we show with a circle the pairs of values that are still compatible with the current position of the sweep-line. We have to move the sweep-line when either all values of Y_1 or of Y_2 are forbidden when $X = \Delta$ (for instance, when $X = 2$ all values of Y_1 are forbidden and therefore $CountZeros_1$ is equal to zero), or that all the pairs of values,

which are still compatible with, $X = \Delta$ are forbidden for the synchronization constraint $|Y_1 - Y_2| < 2$ (this is for instance the case when $X = 0$ or when $X = 1$).

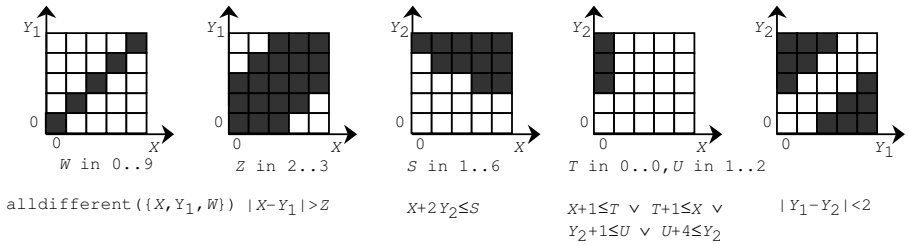

Fig. 1. Examples of forbidden regions. X in 0..4, Y_1 in 0..4, Y_2 in 0..4.

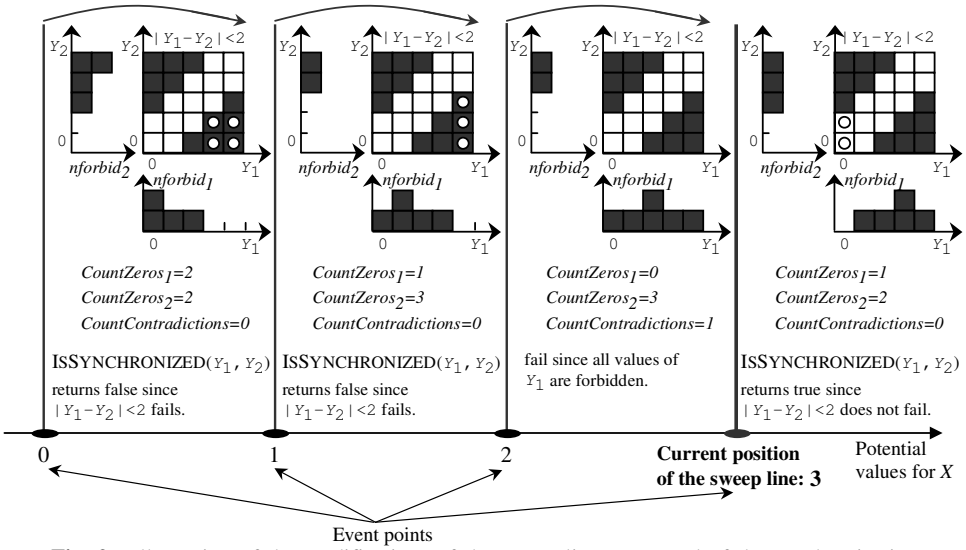

Fig. 2. Illustration of the modifications of the sweep-line status and of the synchronization process.

4 Application to a Timetabling Constraint

We first describe the timetabling constraint we consider. Finally, we enlighten its relation to sweep synchronization and show how to adapt the previous sweep algorithm in order to derive an incremental filtering algorithm.

Definition of the Timetabling Constraint. The goal is to schedule *ntask* tasks. The *t*-th task takes place at time $Start_t$ and lasts $duration_t$ time units. It involves $nperson_t$ different persons, and the *k*-th person must be drawn from a certain group $group_{t,k}$. Unavailability periods of a person can be modeled by fixed tasks, which prevent other

tasks from being scheduled during these periods. The t-th task $(1 \le t \le ntask)$ is defined by the following attributes:

- $Start_t$ is a domain variable which represents the start of the task,
- $duration_t$ is a strictly positive integer which stands for the duration,
- $Person_{t,1},\ldots,Person_{t,nperson_t}$ are domain variables representing the persons that are actually assigned to the task. Initially we have $\mathrm{dom}(Person_{t,k}) = group_{t,k}$.

The constraint holds for an assignment of the involved domain variables if the following conditions are fulfilled for $t = 1,\ldots,ntask$:

1. For each $o = 1,\ldots,ntask$ with $o \neq t$, the tasks do not overlap or no person is assigned to both tasks:
$$Start_t + duration_t \le Start_o \vee Start_o + duration_o \le Start_t \vee Person_{t,k} \neq Person_{o,h}, \quad (1)$$
where $1 \le k \le nperson_t$ and $1 \le h \le nperson_o$.
2. The persons assigned to task t are pairwise different: $Person_{t,k} \neq Person_{t,h}$ for $1 \le k < h \le nperson_t$.

This timetabling constraint is situated between the *cumulative* constraint [3] and the non-overlapping rectangles constraint [2] in the following sense:

- First assume that, for each task i, we replace variables $Person_{i,1},..,Person_{i,nperson_i}$ by a fixed height $nperson_i$ which tells, how many persons task i requires during each instant of its execution. Then the constraint "at each instant we should not exceed the maximum number of available persons" (i.e. the *cumulative* constraint) is a necessary condition for the timetabling constraint.
- Assume now that, for each task i, we replace variables $Person_{i,1},..,Person_{i,nperson_i}$ by a fixed height $nperson_i$ as well as by a domain variable $Person_i$ in order to associate to each task a rectangle. The coordinates of the lowest leftmost corner are $Start_i$ and $Person_i$, while the sizes are $duration_i$ and $nperson_i$. Each solution of the constraint "no two rectangles should overlap" (i.e. the non-overlapping rectangles constraint) is a solution for this timetabling problem where, for each task i, $Person_{i,1},..,Person_{i,nperson_i}$ take consecutives values.

Currently the standard way to model this constraint is to use for every part of each task a rectangle $Rec_{t,k}$ with origin $(Start_t, Person_{t,k})$, width $duration_t$ and height 1. For any two different parts $(t,k) \neq (o,h)$ we have $\mathrm{non\_overlapping}(Rec_{t,k}, Rec_{o,h})$. This gives exactly the same set of conditions as above. However, this formulation does not make use of the fact that all rectangles, which come from the same tasks, have the same start. In order to get more propagation one can also state a *cumulative* constraint as was previously explained. Even so, obvious propagation is missing, especially when each person has his own unavailability periods. For instance, assume we have to choose two persons from a group of persons and set up a meeting during four consecutive periods between these two selected persons, but because of the unavailability, all the persons of the group don't have 4 consecutive free slots in common. The standard model will not capture infeasibility and, for this reason, one

should let the algorithm associated to the non-overlapping rectangles constraint take advantage of the fact that several rectangles have the same origin.

Relation to Sweep Synchronization. Assume that we want to adjust the minimum of the domain variable $Start_t$ of a task $t\,(1 \le t \le ntask)$. We show how to partition the constraints, which mention parts of the tasks, so that the sweep synchronization framework can be applied. We have the following conjunctions of constraints, which are considered by the sweep: $\mathcal{C}_1,..,\mathcal{C}_{nperson_t}$, where $\mathcal{C}_k$ contains all the constraints of the form $\text{non\_overlapping}(Rec_{t,k}, Rec_{o,h})$ (see Condition (1)) with $o \neq t$ and $1 \le h \le nperson_o$. For the previous non-overlapping constraint, there can be at most one non-empty forbidden region $R_{t,k,o,h} = \left(r_x^- .. r_x^+, r_y^- .. r_y^+\right)$ wrt. $\left(Start_t, Person_{t,k}\right)$, where:

$$r_x^- = \max\left(Start_o\right) - duration_t + 1 \quad r_x^+ = \min\left(Start_o\right) + duration_o - 1$$
$$r_y^- = \max\left(Person_{o,h}\right) \qquad\qquad r_y^+ = \min\left(Person_{o,h}\right)$$

Finally the set $\mathcal{S}$ of synchronization constraints contains only the constraint $\text{alldifferent}\left(Person_{t,1},\ldots,P_{t,nperson_t}\right)$.

4.1 A Graph-Theoretic View of the *alldifferent* Constraint

Since we only consider one task t, we introduce the following abbreviations: $n = nperson_t$ and $P_i = Person_{t,i}$ for $i = 1,\ldots,n$. It is well known that the constraint $\text{alldifferent}\left(P_1,\ldots,P_n\right)$ can be modeled as a matching problem in a bipartite graph G [9]. On the left side we have a node p_i for every person variable P_i and on the right hand side we have a node v_j for every value that occurs in some domain of the *P*-variables. We draw an edge between p_i and v_j iff the value j is contained in the domain of P_i. Then there is a one-to-one correspondence between the *p*-perfect matchings of G and the variable assignments satisfying the constraint. A matching M is a set of edges such that no two edges are incident to same node. It is *p-perfect* iff every node p_i is incident to an edge in M.

In our case the graph G is not static. As we sweep along the time axis in order to find possible starting times for the task, some persons may become unavailable and some other persons may become available again. Translated to a graph theoretic point of view this means that some nodes on the *v*-side (and all incident edges) may be hidden from the graph and later be restored again.

So the problem is to maintain a dynamic bipartite graph G which supports the operations $hide\left(v_j\right)$ and $restore\left(v_j\right)$ and to answer queries of the form "Does G contain a *p*-perfect matching?" efficiently.

One additional problem is that we cannot afford to represent the bipartite graph explicitly by adjacency lists, for the graphs tend to be dense, and hence the space requirement and the construction time might be $\Theta\left(n \cdot l\right)$, where l denotes the number of *v*-nodes. Thus we want to work with the implicit representation of the graph.

4.2 Integrating the Algorithm in the Sweep Synchronization Framework

We give an overview how we support the synchronization primitives:

- INITSYNCHRONIZATION$(P_1,\ldots,P_n,a_1,\ldots,a_n)$: We compute an initial p-perfect matching M in G. If no such matching exists we report failure. Otherwise the value assignment corresponding to M is returned in $a_1,\ldots,a_n$.
- ISSYNCHRONIZED$(a_1,\ldots,a_n)$: This function returns `TRUE` iff G contains a p-perfect matching. And the corresponding assignment is stored in $a_1,\ldots,a_n$.
- TELLNEWEVENTTOSYNCHRONIZATION$(kind,k,first,last)$: If $kind$ indicates a start event we call $hide(v_j)$ for $j = first,\ldots,last$. In case of an end event the respective nodes are restored.

4.3 The Matching Algorithm

4.3.1 The Static Case

Our matching algorithm is a variant of the algorithm by Ford and Fulkerson [6]. Our description is based on [8, Chapter 7.6]. First we will present the algorithm for the static case and then show how to modify it for the dynamic setting described above. We need some definitions. A node x of G is called *matched* with respect to some matching M if it is incident to some edge in M; otherwise x is called *free*. A simple path p in G is called *alternating* if the edges in p are alternately in M and not in M. An alternating path a from a free node x to a different free node y is called *augmenting*. This stems from the fact that $M' = M \oplus a = (M \setminus a) \cup (a \setminus M)$ is a new matching where all the previous nodes and in addition the nodes x and y are matched. Suppose that p_i is a free node on the *P*-side with respect to some matching M and that M_p is some p-perfect matching. Then $M_p \oplus M$ contains one path starting in p_i and this path is augmenting with respect to M. This implies the correctness of the following algorithm:

```
1: M ← some matching M_0 (maybe empty).
2: for all free nodes p_i on the P-side do
3:     if there is an augmenting path a wrt. to M starting in p_i then Augment M by a.
4:     else Abort with "no p-perfect matching exists".
5: Return "M is p-perfect".
```

Algorithm 3: The Ford and Fulkerson algorithm

In order to look for an augmenting path we use breadth first search. Algorithm 4 grows a tree T of alternating paths starting from a free root p_r. We use a queue Q to store all p-nodes in T which have to be explored and we mark every v-node which is reached during the search. As long as Q is not empty, we pop the first node p from Q and examine all its incident edges $\{p,v\}$. If v has been reached we do nothing, otherwise we mark v and store the information that p is the father of v in T. In case that v is matched, we grow our tree by the matching mate p' of v, which amounts to appending p' to Q. Note that we do not have to store any father

information here because the father of a *p*-node in T is always its mate. If the node v is free however, the tree path from v to p_r is an augmenting path, which can be traced with the aid of the father information. In this case we augment the matching and terminate the search.

The running time of the BFS-algorithm is $O(m)$ where m is the number of edges of G. Algorithm 3 makes one call to BFS for every free node until it finds a node, which cannot be matched, or it has computed a *p*-perfect matching. Let c denote the cardinality of a maximum cardinality matching in G and c_0 denote the cardinality of M_0. Then the total worst case running time is $O(n+(c+1-c_0)\cdot m)$ which is bounded by $O(n\cdot m)$. We want to mention that there are implementations of the basic scheme of Ford and Fulkerson which achieve a running time of $O(\sqrt{n}\cdot m)$ by using an other strategy to find augmenting paths, see for example [7] or [1]. However, these algorithms are designed for computing maximum cardinality matchings, and if G contains a large but no *p*-perfect matching then these algorithms may take a long time to discover this. Another reason for choosing the BFS-strategy is that it can be adapted easily to the dynamic setting, which we will discuss below.

```
1: Q ← [p_r]. Mark all v-nodes as not reached.
2: while Q is not empty do
3:     Extract the first node p from Q.
4:     for all edges {p,v} incident to p do
5:         if v has not been reached then
6:             Mark v as reached and father(v) ←p.
7:             if v is matched then Append the matching mate p' of v to Q.
8:                              else Augment the matching with the aid of the father-array.
9:                                   Return "p_r is matched now".
10: Return "p_r cannot be matched".
```

Algorithm 4: BFS algorithm for finding augmenting paths

We want to point out, that there is no need to store the graph explicitly. In the algorithms only the adjacency lists of *p*-nodes are scanned. This amounts to a scan of a domain of a *P*-variable[7]. When a *v*-node is considered, we are only interested in its matching mate, we never have to know all adjacent nodes.

4.3.2 The Dynamic Case

Data Structures. We describe some data structures which are used in the dynamic setting. For storing the set of hidden *v*-nodes, we use a boolean array $Hidden[1..l]$.

The matching is represented by two arrays $VMate[1..n]$ and $PMate[1..l]$. We maintain the following invariants: $VMate[i]=0 \vee PMate[VMate[i]]=i$ and $PMate[j]=0 \vee VMate[PMate[j]]=j$. The matching M given by these arrays is defined as follows: $M=\{\{p_i,v_j\} \mid VMate[i]=j\neq 0 \wedge Hidden[j]=\texttt{FALSE}\}$. We explain this definition. A value of 0 indicates that the respective node is free. Otherwise the value indicates the matching mate, but if the *v*-node of a matching edge is hidden, this edge is by

[7] Note that the complexity bounds from above only hold if the scanning time is linear in the size of the domain. This is actually the case for the domain variables representation in SICStus Prolog.

definition also hidden from M. One could also set the values in both arrays to 0 when a *v*-node is hidden, but keeping this information may allow us to augment the matching more easily when the *v*-node is restored later.

Finally, we describe the data structures needed to construct the tree T with root p_r in the BFS-algorithm. We use an array $Mark[1..l]$ which stores for every node v_j (even the hidden ones) one of the following three values:

- *InTree*: v_j belongs to T and there is alternating path from p_r to v_j.
- *OnBorder*: v_j is currently hidden from the graph, and hence it does not belong to T. But there is a hidden edge which connects a node p_i in T with v_j. Note that there will be an augmenting path from p_r to v_j as soon as v_j is restored.
- *Outside*: v_j does not belong to T and there is no edge (not even a hidden one) which connects v_j to some *p*-node in T.

We also have an array $Father[1..l]$ which stores for every node v_j that is not marked *Outside* its p-father in the tree. Algorithm 5 shows how to modify Algorithm 4 such that this information is computed.

```
1: Q ← [p_r]. Mark[j] ← Outside, for j∈1..l.
2: while Q is not empty do
3:     Extract the first node p_i from Q.
4:     for each j∈dom(P_i) do
5:         if Mark[j] =Outside then
6:             Father[j] ← i.
7:             if Hidden[j]=FALSE then
8:                 if PMate[j]=k≠0 then Mark[j] ←InTree. Append p_k to Q.
9:                 else Augment the matching with the aid of the Father and PMate arrays.
10:                    Return "p_r is matched now".
11:            else Mark[j] ← OnBorder.
12: Return "p_r cannot be matched".
```

Algorithm 5: Modified BFS algorithm

The Operations. Suppose that the algorithm above is executed, and it is not able to match p_r. For the dynamic setting we want to derive a condition which is easy to check and which must necessarily hold if p_r can be matched again after a sequence of hide and restore operations.

So let us consider the next time when the graph contains an augmenting path a that starts in p_r and ends in a free node v_f. As we will see later, the hide and restore operations do not change the arrays $PMate$ and $VMate$ (but they change the matching, because they update the array $Hidden$). This implies that not all *v*-nodes on a can be marked *InTree*, otherwise a would already have existed when T was constructed. Let $e=\{p_i,v_j\}$ be the first edge on a such that v_j is not marked *InTree*. All nodes on the prefix of a from p_r to p_i belong to T, and hence e has been scanned by the BFS-algorithm, but v_j was hidden at that time. So, v_j is marked *OnBorder*. This proves that the graph cannot contain a p-perfect matching until the first *v*-node marked *OnBorder* is restored.

The implementation of the operations $hide(v_j)$ and $restore(v_j)$ is straightforward. We simply update the flag $Hidden[j]$. Only if we have a restore operation for a node marked *OnBorder*, we have to do some extra work: we declare our marks as *invalid*, and if $PMate[j] = 0$, we store v_j as *source* for constructing an augmenting path.

We now discuss the function ISSYNCHRONIZED. We distinguish two cases:

- The previous call of the function has found a p-perfect matching: we simply call Algorithm 3. This will either extend the current matching to a perfect matching or it will construct a tree and valid array $Mark$.
- The previous call has not found a perfect matching: if the array $Mark$ is valid, we can immediately report that there is no p-perfect matching and terminate the call. Otherwise, we check whether we have a *source* v_j for constructing an augmenting path. If this is the case and the root p_r of the tree T is still free, then we can find an augmenting path. We start in v_j and trace the path with the array $Father$ and $VMate$ until we hit the first free p-node (this is not necessarily p_r itself, but it may be a descendant p_i of p_r in T; the node p_i has become a free node, because his former v-mate has been hidden). We finish the function by calling Algorithm 3.

The worst case running time of the function ISSYNCHRONIZED is the same as for Algorithm 3. But this is a very pessimistic estimation; in some cases we are able to avoid running the matching algorithm at all. We want to mention one detail, which we have glossed over so far. In order to implement Algorithm 3 efficiently we maintain a list F of free nodes. This list must be updated in hide and restore operations, but it is easy to see that this requires only constant time.

Finally, we will describe the function INITSYNCHRONIZATION. Recall that we have to compute an initial p-perfect matching M_i, if one exists. We want to use Algorithm 3, but we do not want to start with an empty matching M_0. Some constraint solvers offer a primitive to query the smallest value in the domain of a variable P_i that is greater than a value j. This leads to the following heuristic. The value for P_1 is just $j_1 = \min(P_1)$. The value for P_2 is the smallest value j_2 in $\mathrm{dom}(P_2)$ which is greater than j_1, if it exists; otherwise p_2 remains a free node. Continuing this way, we obtain a matching M_0, which we can use as a starting point for Algorithm 3.

4.4 Improving the Best Case Running Time

In our algorithms we use some arrays which are indexed by resource numbers and which have to be initialized at the beginning of the algorithms. These arrays may be quite large, and hence the initialization time may be very long. But it may very well be that the algorithm accesses only a small fraction of the array so that the initialization time may dominate the running time in practice. In the sequel we will discuss a technique which allows us to reinitialize an array in constant while preserving constant random access time.

In order to represent an array $Orig[1..l]$ we need three arrays $Pos[1..l]$, $Values[1..l]$, $ValidIdxAtPos[1..l]$ and an integer value $UsedValues$. The idea is to simulate an access

to $Orig[1..l]$ by some indirect addressing scheme, which is described in Algorithm 6. The improvement is that it is only required to set $UsedValues$ to 0 for reinitializing the array $Orig[1..l]$.

1: **if** $Pos[j]>UsedValues$ **or** $ValidIdxAtPos[Pos[j]]\neq j$ **then**
2: Increment $UsedValues$. $Pos[j] \leftarrow UsedValues$. $ValidIdxAtPos[Pos[j]] \leftarrow j$.
3: Initialize $Values[Pos[j]]$ to its default value.
4: Return $Values[Pos[j]]$.

Algorithm 6: Simulating the access of entry j

4.5 Implementation and Experimental Results

The synchronized sweep algorithm with both matching algorithms, static and dynamic, were implemented as an extension of the filtering algorithm for the non-overlapping rectangles constraint described in [2]. The first time a non-overlapping rectangles constraint is posted it checks if all the rectangle heights are equal to one. If this is the case, the synchronized sweep algorithm will be used for adjusting the minimum and maximum value of the X coordinates of the origin of the rectangles. For the Y coordinates, we keep the original sweep algorithm of [2].

Benchmark Description In order to get some insight of the practical behavior of the synchronized sweep algorithm, we generate the following problem patterns.

Table 2. Parameters of the three patterns.

Pattern (instance)	#resources	#meetings	meeting size	#unavailability	(x-coordinate, *y*-coordinate, length)
I(n)	n	$2.n-1$	$n-1$	$2.n^2$	$((i-1)\cdot 2\cdot n+j,\ j,\ 1)\quad i,j\in 1..n$ $((i-1)\cdot 2\cdot n+n+j,\ n+1-j,\ 1)\quad i,j\in 1..n$
II(n)	n	n	n	n^2	$((i-1)\cdot 2\cdot n+j,\ j,\ n)\quad i,j\in 1..n$
III(n)	$9+n$	$\approx 18+2\cdot n$	$\approx (7+n)/2$	$\approx (9+n)^2/1.5$	

The last column gives for patterns I and II, the x and y coordinates and the length of the periods of unavailability. For the three patterns, the x-coordinates of the rectangles to place have to be greater than 1, while all the y-coordinates are between 1 and the number of resources. The third pattern consists of random instances with a fixed density of the use of the available space. The fourth column gives the average number of persons in a meeting. Fig. 3 gives a graphical representation of patterns I and II for the case where n=3. The dark rectangles correspond to unavailability periods, while the grey rectangles represent a person in a meeting.

Pattern I Pattern II

Fig. 3. Pattern I and II for n=3.

Analysis Benchmarks were run on a 550 Mhz Pentium-II processor with 128MB physical memory under Microsoft Windows 2000 Professional and a version of SICStus Prolog compiled with Microsoft Visual C++ version 6.0 in optimized mode. On the previous patterns, we have compared the standard filtering algorithm for the non-overlapping rectangles constraint described in [2] with two variants of the sweep-synchronization algorithm presented in the second part of this paper. The first variant uses the static version of the matching algorithm while the second variant

utilizes the dynamic version. Table 3 gives the number of backtracks and the time[8] in msec after searching for the first solution for all approaches. We make the following observations. Using sweep-synchronization, even with the static version of the matching algorithm, can bring an improvement of several orders of magnitude over the standard non-overlapping rectangles constraint. As shown by pattern II, this can also be the case when we have to adjust the origin of one single meeting according to several unavailability periods. Using the dynamic version of the matching algorithm versus the static version improves the time only when within one single sweep, the sweep-line moves according to a large number of forbidden regions. This was typically the case of the second pattern where we perform one big sweep over all unavailability periods. Finally we mention that, using sweep-synchronization does not lead to a significant increase of memory consumption compared to the standard non-overlapping rectangles constraint.

Table 3. Number of backtracks and time (msec) for the first solution.

Pattern (instance)	#rectangles	Non-overlapping rectangles constraint		Sweep-synchronization (static matching)		Sweep-synchronization (dynamic matching)	
I(4)	50	66	80	0	50	0	40
I(5)	82	528	671	0	81	0	80
I(6)	122	4440	7761	0	160	0	151
I(7)	170	40320	99313	0	280	0	280
II(10)	110	-	time out	0	40	0	40
II(25)	650	-	time out	0	220	0	131
II(50)	2550	-	time out	0	2193	0	861
II(100)	10100	-	time out	0	30334	0	9886
III(1)	136	1070	2414	0	371	0	350
III(2)	182	9896	24855	1	741	1	731
III(3)	232	6003	22612	0	1442	0	1341
III(4)	254	102494	412924	2	1483	2	1372
III(5)	269	22811	94356	5	2073	5	1952
III(6)	328	149144	817746	3	3224	3	3054
III(7)	379	268624	1726252	3	5348	3	5047
III(8)	407	-	time out	3	5408	3	5078
III(9)	489	-	time out	5	9544	5	8953
III(10)	512	-	time out	0	9814	0	9243

5 Conclusion

The first part of this paper introduces a new sweep algorithm in order to handle a specific constraint pattern. Through the utilization of forbidden regions and the introduction of synchronization primitives this algorithm was made generic. As a concrete example the last part of this paper applies this new sweep algorithm to a practical timetabling problem which requires the concept of synchronization in order to reason globally about the unavailability periods of different persons. In this example the synchronization constraint is an *alldifferent* constraint. For checking it efficiently, we have come up with a specific incremental matching algorithm on a bipartite graph. This algorithm takes advantage of the kind of modifications occurring on the bipartite graph while we sweep over the time axis. This new method was implemented within the SICStus finite domain solver by extending the filtering algorithm associated to the non-overlapping rectangles constraint. Experimental results show that, for the previous timetabling problem, this can lead to an

[8] A time limit of one hour was given for each instance.

improvement of several orders of magnitude over the standard non-overlapping rectangles constraint. Other useful concrete examples would be the case of the non-overlapping rectangles constraint where the origins of several rectangles have the same coordinate on a given axis. In this case one would have to replace the bipartite matching algorithm by an algorithm for checking a necessary condition for the fact that a set of segments do not pairwise overlap. For this purpose one could for instance specialize Jackson's preemptive schedule from Carlier and Pinson [5] to a similar type of incremental algorithm to the one we propose for checking the *alldifferent* constraint.

References

1. Alt, H., Blum, N., Mehlhorn, K., Paul, M.: Computing a maximum cardinality matching in a bipartite graph in time $O\left(n^{1.5}\sqrt{m/\log n}\right)$. *Inform.Processing Letters*, 37(4):237–240, (1991).
2. Beldiceanu, N., Carlsson, M.: Sweep as a Generic Pruning Technique Applied to the Non-Overlapping Rectangles Constraint. In *Proc. of the 7th CP*, 377-391, Paphos, (2001).
3. Beldiceanu, N., Aggoun, A.: Extending CHIP in order to solve complex scheduling and placement problems. *Mathl. Comput. Modelling* 17 (7):57–73 (1993).
4. de Berg, M., van Kreveld, M., Overmars, M., Schwarzkopf, O.: *Computational Geometry – Algorithms and Applications*. Springer, (1997).
5. Carlier, J., Pinson, E.: A Practical Use of Jackson's Preemptive Schedule for Solving the Job-Shop Problem. *Annals of Operations Research* 26:269–287, (1990).
6. Ford, L.R., Fulkerson, D.R.:*Flows in Networks*. Princeton Univ.Press, Princeton, NJ, (1963).
7. Hopcroft, J.E., Karp, R.M.: An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, (1973).
8. Mehlhorn, K., Näher, S.: *LEDA: a platform for combinatorial and geometric computing*. Cambridge University Press, Cambridge, (1999).
9. Régin, J.C.: A filtering algorithm for constraints of difference in CSPs. *Proc. 12th Conf. American Assoc. Artificial Intelligence*, volume 1, 362–367, (1994).
10. Van Hentenryck, P.: Constraint Programming for Combinatorial Search Problems. In *Constraints 2(1).* Kluwer Academic Publishers, 99–101, (1997).