Sweep as a Generic Pruning Technique Applied to the Non-Overlapping Rectangles Constraint

Nicolas Beldiceanu and Mats Carlsson

SICS, Lägerhyddsv. 18, SE-752 37 UPPSALA, Sweden
{nicolas,matsc}@sics.se

May 28, 2001

SICS Technical Report T2001:13

ISRN: SICS-T–2001/13-SE

ISSN: 1100-3154

Abstract. We first present a generic pruning technique which aggregates several constraints sharing some variables. The method is derived from an idea called *sweep* which is extensively used in computational geometry. A first benefit of this technique comes from the fact that it can be applied on several families of global constraints. A second main advantage is that it does not lead to any memory consumption problem since it only requires temporary memory that can be reclaimed after each invocation of the method.
We then specialize this technique to the non-overlapping rectangles constraint, describe several optimizations, and give an empirical evaluation based on six sets of test instances of different pattern.
Keywords: Constraint Programming, Global Constraint, Sweep, Non-Overlapping.

1 Introduction

The main contribution of this paper is to present a generic pruning technique for finite domain constraint solving[1]. As a second contribution, we specialize the technique to the non-overlapping rectangles constraints and evaluate its performance. Finally, we identify and evaluate four optimizations which should be applicable to many global constraints.

The technique is based on an idea which is widely used in computational geometry and which is called *sweep* [8, pp. 10–11]. Consider e.g. Fig. 1 which shows 5 constraints and their projections on two given variables X and Y. Assume that we want to find the smallest value of X so that the conjunction of the 5 constraints is feasible for some Y. By trying $X = 0, \ldots, 4$, we conclude

[1] A *domain variable* is a variable that ranges over a finite set of integers; $\min(X), \max(X)$ and $\mathrm{dom}(X)$ denote respectively the minimum value, the maximum value, and the set of possible values for X.

that $X = 4$ is the only feasible value. The sweep algorithm performs this search efficiently; see Sect. 3.2 for the details on this particular example.

In dimension 2, a plane sweep algorithm solves a problem by moving a vertical line from left to right[2]. The algorithm uses the two 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. A common application of the sweep algorithm is to solve the segments intersection problem [8, p. 278], with a time complexity that depends both on the number of segments and on the number of segment intersections.

In our case, the sweep-line scans the values of a domain variable X that we want to prune, and the sweep-line status contains a set of constraints that have to hold for $X = \Delta$. The generic pruning technique, which we call *value sweep pruning*, accumulates the values to be currently removed from the domain of a variable Y which is different from X. If, for some value of Δ, all values of Y have to be removed, then we will prune Δ from $\mathrm{dom}(X)$. The method is based on the aggregation of several constraints that have two variables in common. Let:

- X and Y be two distinct domain variables,
- $C_1(V_{11}, \ldots, V_{1n_1}), \ldots, C_m(V_{m1}, \ldots, V_{mn_m})$ be a set of m constraints such that $\forall i \in 1..m : \{X, Y\} \subseteq \{V_{i1}, \ldots, V_{in_i}\}$ (i.e. all constraints mention both variables X and Y).

The value sweep pruning algorithm will try to adjust the minimum[3] value of X wrt. the conjunction of the previous 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 ends of forbidden 2-dimensional regions wrt. constraints $C_1, \ldots, C_m$ and variables X and Y.

In this paper, we use the notation $(F_x^- .. F_x^+, F_y^- .. F_y^+)$ to denote an ordered pair F of intervals and their lower and upper bounds. $\mathrm{rand}(S)$ denotes a random integer in the set S.

The next section presents the notion of forbidden regions, which is a way to represent constraints that is suited for the value sweep algorithm. Sect. 3 describes the value sweep pruning algorithm and gives its worst case complexity. Sect. 4 presents the specialization of this algorithm to the non-overlapping rectangles constraint, as well as several optimizations. Sect. 5 provides an empirical evaluation of six different variants of the algorithm according to several typical test patterns.

[2] In general, a plane sweep algorithm does not require neither the sweep-line to be vertical nor moving it from left to right.

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

2 Forbidden Regions

We call F a *forbidden region of the constraint* C_i *wrt. the variables* X *and* Y if: $\forall x \in F_x^- .. F_x^+, y \in F_y^- .. F_y^+ : C_i(V_{i1}, \ldots, V_{in_i})$ with the assignment $X = x$ and $Y = y$ has no solution. We say that $X = a$ *is feasible wrt.* $C_1, \ldots, C_m$ if $a \in \mathrm{dom}(X) \wedge \exists b \in \mathrm{dom}(Y)$ such that (a, b) is not in any forbidden region of $C_1, \ldots, C_m$ wrt. X and Y.

Fig. 1 shows 5 constraints and their respective forbidden regions (shaded) wrt. two given variables X and Y and their domains. The first constraint requires that X, Y and R be pairwise distinct. Constraints (B,C) are usual arithmetic constraints. Constraint (D) can be interpreted as requiring that two rectangles of respective origins (X, Y) and (T, U) and sizes $(2, 4)$ and $(3, 2)$ do not overlap. Finally, constraint (E) is a parity constraint of the sum of X and Y.

The value sweep pruning algorithm computes the forbidden regions on request, in a lazy evaluation fashion. The algorithm generates the forbidden regions of each constraint C_i gradually as a set of rectangles $R_{i1}, \ldots, R_{in}$ such that:

- $R_{i1} \cup \cdots \cup R_{in}$ represents all forbidden regions of constraint C_i wrt. variables X and Y.
  Note that we do not request the rectangles to form a partition of the forbidden space. This is because it sometime allows generating fewer rectangles: more than 3 rectangles are necessary to cover the forbidden regions of example B of Fig. 1 if we would ask for a partition.
- $R_{i1}, \ldots, R_{in}$ are sorted by ascending start position on the X axis.

This will be handled by providing the following three functions[4] for each triple (X, Y, C_i) that we want to be used by the value sweep algorithm:

- get_first_forbidden_regions(X, Y, C_i), whose value is all the forbidden regions R_{C_i} of C_i such that:

$$\begin{cases} R_{C_i}{}_x^- \leq \mathit{first}_{C_i} \leq R_{C_i}{}_x^+ \\ R_{C_i}{}_y^+ \geq \min(Y) \wedge R_{C_i}{}_y^- \leq \max(Y) \end{cases}$$

  where first_{C_i} is the smallest value $\in \min(X)..\max(X)$ such that there exists such a forbidden region R_{C_i} of C_i.
- get_next_forbidden_regions(X, Y, C_i, x_i'), whose value is all the forbidden regions R_{C_i} of C_i such that:

$$\begin{cases} R_{C_i}{}_x^- = \mathit{next}_{C_i} \\ R_{C_i}{}_y^+ \geq \min(Y) \wedge R_{C_i}{}_y^- \leq \max(Y) \end{cases}$$

  where x_i' is the position of the previous start event of C_i and next_{C_i} is the smallest value $> x_i'$ such that there exists such a forbidden region R_{C_i} of C_i.

[4] Two analogous functions get_last_forbidden_region and get_prev_forbidden_regions are also provided for the case where the sweep-line moves from the maximum to the minimum value.

- check_if_in_forbidden_regions(X, Y, x, y, C_i), which is true iff given values $x \in \text{dom}(X)$ and $y \in \text{dom}(Y)$ belong to a forbidden region of constraint C_i.
- max_ysize_forbidden_regions(X, Y, C_i), whose value is an upper bound of the quantity $\max_{x \in \text{dom}(X)} |\mathsf{Forbid}(Y, x)|$, where $\mathsf{Forbid}(Y, x)$ is the set of values y of variable Y such that the constraint $C_i(V_{i1}, \ldots, V_{in_i})$ with the assignment $X = x$ and $Y = y$ has no solution.

If we consider constraint (E) of Fig. 1 (i.e. $X + Y \mod 2 = 0$), and we assume $X \in 0..2$ and $Y \in 1..3$, then a complete scan of X would produce the following sequence of calls:

- get_first_forbidden_regions$(X, Y, X + Y \mod 2 = 0)$ returns regions $(0..0, 1..1)$ and $(0..0, 3..3)$.
- get_next_forbidden_regions$(X, Y, X+Y \mod 2 = 0, 0)$ returns region $(1..1, 2..2)$.
- get_next_forbidden_regions$(X, Y, X+Y \mod 2 = 0, 1)$ returns regions $(2..2, 1..1)$ and $(2..2, 3..3)$.

A call to max_ysize_forbidden_regions$(X, Y, X + Y \mod 2 = 0)$ would return 2, as for any value of $X \in 0..2$ we have at most two forbidden values for $Y \in 1..3$. We now show how to use the function max_ysize_forbidden_regions(X, Y, C_i) in order to get a condition, which tells us when the algorithm can for sure not bring anything. This condition can be used to avoid useless work. Let:

- X and Y be two distinct domain variables,
- $C_1(V_{11}, \ldots, V_{1n_1}), \ldots, C_m(V_{m1}, \ldots, V_{mn_m})$ be a set of m constraints such that $\forall i \in 1..m : \{X, Y\} \subseteq \{V_{i1}, \ldots, V_{in_i}\}$ (i.e. all constraints mention both variables X and Y).

If $\sum_{i \in 1..m}$ max_ysize_forbidden_regions$(X, Y, C_i) < |\text{dom}(Y)|$ then value sweep pruning is not useful since all values of Y cannot be completely covered by the different forbidden regions.

3 The Value Sweep Pruning Algorithm

3.1 Data Structures

The algorithm uses the following data structures:

The sweep-line status. Denoted P_{status}, this contains the current possible values for variable Y wrt. $X = \Delta$. More precisely, P_{status} can be viewed as an array which records for each possible value of Y the number of forbidden regions that currently intersect the sweep-line. The basic operations required on this data structure, and their worst-case complexity in a reasonable implementation, are shown in Table 1. An (a, b)-tree [7] based data structure can provide the array operations with complexity as in Table 1.

The event point series. Denoted Q_{event}, this contains the start and the end+1 (+1 since the end is still forbidden whereas end+1 is not), on the X axis, of those forbidden regions of the constraints $C_1, \ldots, C_m$ wrt. variables X and Y that intersect the sweep line. These *start events* and *end events* are sorted in increasing order and recorded in a queue. The basic operations required, and their complexity e.g. in a heap, are also shown in Table 1. The existence check can be implemented in $O(1)$ time with a reference counter. This last operation is the trigger which is used in order to gradually enqueue the start and end events associated to the forbidden regions of a given constraint C_i when a start event associated to constraint C_i is removed from the queue Q_{event}.

3.2 Principle of the Algorithm

In order to check if $X = \varDelta$ is feasible wrt. $C_1, \ldots, C_m$, the sweep-line status records all forbidden regions that intersect the sweep-line. If, for $X = \varDelta$, $\forall i \in \mathrm{dom}(Y) : P_{status}[i] > 0$, $\varDelta$ will move to the right.

Before going more into the detail of the sweep algorithm, let us illustrate how it works on a concrete example. Assume that we want to find out the minimum value of variable X wrt. the conjunction of the 5 constraints that were given in Fig. 1. Fig. 2 shows the contents of P_{status} for different values of $\varDelta$. The smallest feasible value of X is 4, since this is the first point where P_{status} contains an element with value 0. We now present the main procedure.

3.3 The Main Procedure

The procedure FindMinimum implements the value sweep pruning algorithm for adjusting the minimum value of a variable wrt. a set of constraints. It can readily be transformed into an analogous procedure FindMaximum for adjusting the maximum value. The main parts of FindMinimum correspond to:

lines 1–9: Initialize the event queue to the start and end events associated to the left-most forbidden regions of each constraint. Note that we only insert events that are effectively within $\min(X)..\max(X)$ and $\min(Y)..\max(Y)$. If no such events are found or if no event intersects $\min(X)$, we exit from the procedure.

lines 10–11: Initialize P_{status} to 0 for the values that belong to $\mathrm{dom}(Y)$ and to 1 otherwise[5]. These last values will be forbidden forever, since no corresponding end event was inserted in the event queue.

lines 13–18: Extract from the event queue all events associated to $\varDelta$ and handle them. Afterwards, check whether there exists some feasible solution for $X = \varDelta$ and, if so, exit from the procedure.

line 19: Fail since the sweep-line did a complete scan of the domain of variable X without finding any solution.

Holes in the domain of variable X are handled in the same way as constraints $C_1, \ldots, C_m$: an additional constraint which, for each interval of consecutive

[5] $P \setminus Q$ denotes the set difference between P and Q.

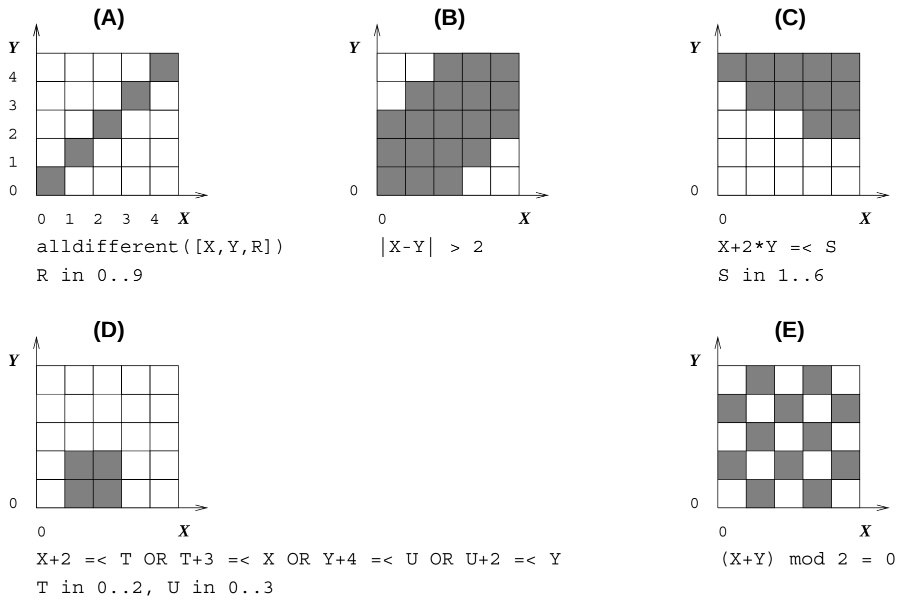

Fig. 1. Examples of forbidden regions. X in 0..4, Y in 0..4.

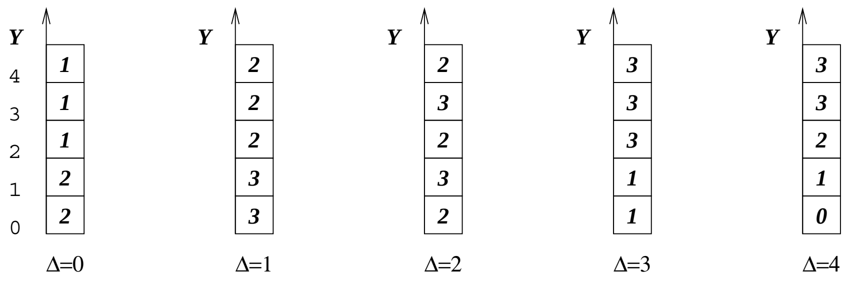

Fig. 2. Status of the sweep-line at each stage of the algorithm. Values denote the number of forbidden regions per Y position.

```
Input: A set of constraints $C_1, \ldots, C_m$ and two domain variables $X$ and $Y$ present in
each constraint.
Output: An indication as to whether a solution exists, and values $\hat{x}, \hat{y}$.
Ensure: Either $\hat{x}$ is the smallest value of $X$ such that $\hat{y} \in \mathrm{dom}(Y)$ and $(\hat{x}, \hat{y})$ does
    not belong to any forbidden region of $C_1, \ldots, C_m$ wrt. variables $X$ and $Y$, or no
    solution exists.
 1: $Q_{event} \leftarrow$ an empty event queue
 2: for all constraint $C_i (1 \le i \le m)$ do
 3:   for all forbidden region $R_{C_i} \in$ get_first_forbidden_regions$(X, Y, C_i)$ do
 4:     Insert $\max({R_{C_i}}_x^-, \min(X))$ into $Q_{event}$ as a start event
 5:     if ${R_{C_i}}_x^+ + 1 \le \max(X)$ then
 6:       Insert ${R_{C_i}}_x^+ + 1$ into $Q_{event}$ as an end event
 7: if $Q_{event}$ is empty
    or the leftmost position of any event of $Q_{event}$ is greater than $\min(X)$ then
 8:   $\hat{x} \leftarrow \min(X)$, $\hat{y} \leftarrow \mathrm{rand}(\mathrm{dom}(Y))$
 9:   return (true, $\hat{x}, \hat{y}$)
10: $P_{status} \leftarrow$ an array ranging over $\min(Y)..\max(Y)$ with all zero elements
11: $P_{status}[i] \leftarrow 1$ for $i \in \min(Y)..\max(Y) \setminus \mathrm{dom}(Y)$
12: while $Q_{event}$ is not empty do
13:   $\Delta \leftarrow$ the leftmost position of any event of $Q_{event}$
14:   for all event $E$ at $\Delta$ of $Q_{event}$ do
15:     HandleEvent$(E)$
16:   if $P_{status}[i] = 0$ for some $i$ then
17:     $\hat{x} \leftarrow \Delta$, $\hat{y} \leftarrow$ a random $i$ such that $P_{status}[i] = 0$
18:     return (true, $\hat{x}, \hat{y}$)
19: return (false, 0, 0)
```

Algorithm 1: FindMinimum$(C_1, \ldots, C_m, X, Y)$

removed values, generates start and end event. The next procedure, HandleEvent, specifies how to handle start and end event points.

3.4 Handling Start and End Events

Depending on whether we have a start or an end event E, we add 1 or -1 to $P_{status}[i], l \leq i \leq u$, where l and u are respectively the start and the end on the Y axis of the forbidden region that is associated to the event E. When E was the last start event of a given constraint C_E, we search for the next events of C_E and insert them in the event queue Q_{event}.

3.5 Discussion

Before running again the complete algorithm, we check first whether the previously returned solution $(\hat{x}, \hat{y})$ is still feasible[6].

The motivation to assign a random value to $\hat{y}$ comes from the fact that, if we use the algorithm for pruning several variables, we don't want to get the same feasible solution for several variables, since a single future assignment could invalidate this feasible solution. This would result in reevaluating again the algorithm for several variables.

If we use the algorithm for doing a complete pruning, then each call to the algorithm will lead to a complete sweep over the domain of the variable we want to prune. However, if we use the algorithm for adjusting the minimum or maximum value then we will have a complete sweep on each branch of the search tree. This is because the sweep process will be stopped each time we find a first feasible solution.

Let f denote the total number of forbidden regions intersecting the initial domain of the variables X, Y under consideration, and m the number of constraints. For a complete sweep, Table 1 indicates the number of times each operation is performed, and its total worst case cost, assuming a reasonable implementation. Hence, the overall complexity of the algorithm is $O(m + f \log f)$. Note that, if the problem is loose, $m \gg f$.

From a deductive point of view, the value sweep pruning algorithm is similar to the work done by du Verdier [4]. However the main difference is that the set of forbidden regions associated to each pair of variables (X, Y) was stored explicitly in a quadtree or octtree [9] for which one needs to restore the previous state on backtracking. With the sweep, one can reclaim the data structures Q_{event} and P_{status} after each invocation of the method. Value sweep pruning can also be seen as a specific form of shaving [6], where the main difference is that value sweep pruning does not try out each value one by one.

4 Value Sweep for Non-Overlapping Rectangles

Assume that we want to implement a constraint NonOverlapping$(P_1, \ldots, P_m)$ over a set of rectangles, which should hold if no two rectangles $P_i, P_j, i \neq j$

[6] For this check, we use the check_if_in_forbidden_regions function.

```
1: Extract $E$ from $Q_{event}$
2: Get the corresponding forbidden region $R_E$ and constraint $C_E$
3: Let $l = \max(R_{E}{}_y^-, \min(Y)), u = \min(R_{E}{}_y^+, \max(Y))$
4: if $E$ is a start event then
5:     Add 1 to $P_{status}[i], l \leq i \leq u$
6:     if $Q_{event}$ does not contain any start event associated to constraint $C_E$ then
7:         $x'_E \leftarrow R_{E}{}_x^-$
8:         for all forbidden region $R_{C_i} \in$ get_next_forbidden_regions$(X, Y, C_E, x'_E)$ do
9:             Insert $R_{C_E}{}_x^-$ into $Q_{event}$ as a start event
10:            if $R_{C_E}{}_x^+ + 1 \leq \max(X)$ then
11:                Insert $R_{C_E}{}_x^+ + 1$ into $Q_{event}$ as an end event
12: else
13:     Add -1 to $P_{status}[i], l \leq i \leq u$
```

Algorithm 2: HandleEvent(E)

Table 1. Maximum no. of calls and total cost per basic operation in a sweep of FindMinimum

Operation	**Max. times**	**Total cost** (O)
Initialize to empty the queue	1	1
Compute the first forbidden regions of C_i	m	$m + f$
Add an event to the queue	$2 \times f$	$2 \times f$
Extract the next event from the queue	$2 \times f$	$2 \times f \log f$
Check if there exists some start event associated to C_i	f	f
Initialize to zero a range of array elements	1	1
Add 1 or -1 to a range of array elements	$2 \times f$	$2 \times f \log f$
Check if there exists an array element with value 0	$2 \times f$	$2 \times f \log f$
Compute the index of a random array element with value 0	1	$\log f$

overlap. This constraint is a special case of the diffn constraint [2], and has been used to model a wide range of placement and scheduling problems [1]. It could be implemented by decomposition into a conjunction of $m(m-1)/2$ pairwise non-overlapping constraints:

$$\begin{array}{c} C_{ij}(\langle X_i, w_i, Y_i, h_i\rangle, \langle X_j, w_j, Y_j, h_j\rangle) \Leftrightarrow \\ X_i + w_i \leq X_j \vee X_j + w_j \leq X_i \vee Y_i + h_i \leq Y_j \vee Y_j + h_j \leq Y_i \end{array} \tag{1}$$

where we denote by the tuple $\langle X_i, w_i, Y_i, h_i\rangle$ a rectangle with origin coordinates (X_i, Y_i), width w_i and height h_i. Each pairwise constraint could in turn be implemented by *cardinality* or *constructive disjunction* [10]. This section shows how to instead specialize the value sweep scheme to the NonOverlapping constraint, thus avoiding decomposition.

Without loss of generality, we assume that w_i and h_i are fixed, and we only discuss how to adjust $\min(X_i)$.

4.1 The Basic Algorithm

It is straightforward to see that there can be at most one (non-empty) forbidden region $R_{ij} = (r_x^- .. r_x^+, r_y^- .. r_y^+)$ of C_{ij} wrt. (X_i, Y_i), where:

$$\begin{array}{ll} r_x^- = \max(X_j) - w_i + 1 & r_x^+ = \min(X_j) + w_j - 1 \\ r_y^- = \max(Y_j) - h_i + 1 & r_y^+ = \min(Y_j) + h_j - 1 \end{array} \tag{2}$$

Hence, we get the following definitions for the functions driving the algorithm:

$$\text{get\_first\_forbidden\_regions}(X_i, Y_i, C_{ij}) = \begin{cases} \{(r_x^- .. r_x^+, r_y^- .. r_y^+)\} & \text{if } r_x^- \leq r_x^+ \wedge r_y^- \leq r_y^+, \\ \emptyset & \text{otherwise} \end{cases}$$

$$\text{get\_next\_forbidden\_regions}(X_i, Y_i, C_{ij}, x_i') = \emptyset$$

$$\text{check\_if\_in\_forbidden\_regions}(X_i, Y_i, x, y, C_{ij}) = r_x^- \leq x \leq r_x^+ \wedge r_y^- \leq y \leq r_y^+$$

$$\text{max\_ysize\_forbidden\_regions}(X_i, Y_i, C_{ij}) = \begin{cases} r_y^+ - r_y^- + 1 & \text{if } r_x^- \leq r_x^+ \wedge r_y^- \leq r_y^+, \\ 0 & \text{otherwise} \end{cases}$$

where $r_x^-, r_x^+, r_y^-, r_y^+$ are defined in (2).

Given these definitions, we are now in a position to define Algorithm 3 which adjusts $\min(X_i)$ for each rectangle so that a feasible origin is found for each rectangle. We also maintain for each rectangle P_i the value $\text{witness}(X_i)$ to enable a quick check whether the origin point $(\min(X_i), \text{witness}(X_i))$ is feasible. From the complexity analysis of Sect. 3.5, we have that the worst case complexity of Algorithm 3 is $O(m^2 + m \times f \log f)$ where f is the average number of rectangles that could overlap with the domain of placement of a given rectangle P_i.

```
Input: A set of rectangles $P_1, \ldots, P_m$.
Output: The number of lower bounds that were adjusted, or $\infty$ if no solution exists.
Ensure: Either $(\min(X_i), \text{witness}(X_i))$ is a feasible pair of coordinates for $1 \leq i \leq m$,
    or no solution exists.
 1: $c \leftarrow 0$
 2: for all rectangle $P_i (1 \leq i \leq m)$ do
 3:   Let $S = \{C_{ij} : 1 \leq j \leq m \wedge i \neq j\}$
 4:   if $\exists C \in S$ : check_if_in_forbidden_regions$(X_i, Y_i, \min(X_i), \text{witness}(X_i), C)$ then
 5:     $(r, \hat{x}, \text{witness}(X_i)) \leftarrow$ FindMinimum$(S, X_i, Y_i)$
 6:     if $r =$ false then
 7:       return $\infty$
 8:     else if $\hat{x} \neq \min(X_i)$ then
 9:       $c \leftarrow c + 1$, $\min(X_i) \leftarrow \hat{x}$
10: return $c$
```

Algorithm 3: NonOverlapLeft$(P_1, \ldots, P_m)$

4.2 An Algorithm with a Shared Event Queue

The worst-case cost of NonOverlapLeft is dominated by the creation of the event queue, which is done from scratch for each successive call to FindMinimum. Hoping to reducing the complexity if $m \gg f$, we shall show how to instead create a single, *shared* event queue which is valid throughout the **for** loop.

Consider again $R_{ij} = (r_x^- .. r_x^+, r_y^- .. r_y^+)$ as defined by (2). We note that the only dependency of R_{ij} on P_i is r_x^- (r_y^-) which depends on w_i (h_i). Let $R_{\_j} = (\max(X_j)+1 .. \min(X_j)+w_j-1, \max(Y_j)+1 .. \min(Y_j)+h_j-1)$ denote a *relative forbidden region associated to* P_j.

We then define a modified Q_{event} data structure consisting of two arrays of relative forbidden regions associated to P_j for $1 \leq j \leq m$, ordered by ascending $\max(X_j)$ and $\min(X_j) + w_j$ respectively. To use the shared event queue, the FindMinimum procedure needs to be modified as follows:

- Lines 1–6 are replaced by a search for the smallest $\Delta \geq \min(X)$.
- The **while** loop in line 12 should terminate when $\Delta > \max(X)$ or when Q_{event} is empty.
- The code must ignore events linked to forbidden regions that are empty.
- The event extraction operation must be modified according to the new data structure, and relative forbidden regions must be translated to absolute ones according to w_i and h_i of the current rectangle P_i.

The NonOverlapLeft procedure must be modified accordingly. Before line 2, the shared event queue must be built (takes $(O(m \log m))$ time) and passed in each call to Algorithm 1. Thus compared to the worst-case complexity analysis in Sect. 4.1, we replace an $O(m^2)$ term by an $O(m \log m)$ term, an improvement especially is the problem is loose $(m \gg f)$.

4.3 A Filtering Algorithm

A simple filtering algorithm for NonOverlapping can be implemented as follows: Repeatedly call NonOverlapLeft (and similarly for the other three bounds) until failure or a fixpoint is reached. In the latter case, suspend if not all rectangles are fixed; succeed otherwise. The filtering algorithm should typically act as a coroutine which is resumed whenever one of the bounds is pruned by some other constraint. An implementation along these lines has been done for SICStus Prolog [3]. The implemented version provides optional extensions (variable width and height, wrap-around in either dimension, minimal margins between rectangles, global reasoning pruning), but these will not be discussed further.

4.4 Optimizations

Here, we will describe several optimizations which have been added to the basic filtering algorithm described above. The impact of these optimizations are empirically investigated in Sect. 5. Most of these optimizations are in fact generic to the family of value sweep pruning algorithms, and some could even be applied to most global constraints. Let $\mathcal{B}(P_i)$ denote the *bounding box of* P_i, i.e. the convex hull of all the feasible instances of a rectangle P_i and $\mathcal{C}(P_i)$ denote the *compulsory part [5] of* P_i, i.e. the intersection of all the feasible instances of a rectangle P_i:

$$\begin{array}{ll} \mathcal{B}(P_i)_x^- = \min(X_i) & \mathcal{C}(P_i)_x^- = \max(X_i) \\ \mathcal{B}(P_i)_x^+ = \max(X_i) + w_i - 1 & \mathcal{C}(P_i)_x^+ = \min(X_i) + w_i - 1 \\ \mathcal{B}(P_i)_y^- = \min(Y_i) & \mathcal{C}(P_i)_y^- = \max(Y_i) \\ \mathcal{B}(P_i)_y^+ = \max(Y_i) + h_i - 1 & \mathcal{C}(P_i)_y^+ = \min(Y_i) + h_i - 1 \end{array} \tag{3}$$

Sources and targets. Two properties are attached to each rectangle P_i: the *target* property, which is true if P_i can still be pruned or needs checking; and the *source* property, which is true of P_i can lead to some pruning.

The point is that substantially less work is needed for rectangles lacking one or both properties: the **for** loop of Algorithm 3 only needs to iterate over the *targets*; when building the event queue, only *sources* need to be considered.

Consider a typical placement problem, in which most of the time spent searching for solutions will be spent in the lower parts of the search tree, where most rectangles are already fixed. Thus few rectangles will have *target* properties, and rectangles that can no longer interact with the non-fixed ones will lack both properties.

Initially, all rectangles have both properties. As the search progresses, the transitions $\{source, target\} \Rightarrow \{source\} \Rightarrow \emptyset$ take place.[7]

The first transition takes place whenever a rectangle is ground and has been checked (end of the **for** loop in Algorithm 3). The second type of transition is done when a fixpoint is reached by means of the following linear algorithm:

[7] On backtracking, the converse transitions take place.

1. Compute the bounding box $\mathcal{B}$ of all *targets*.
2. For each *source* P_i, if the bounding box of P_i is disjoint from $\mathcal{B}$, then remove its *source* property.

Initial check of compulsory parts. A necessary condition for $\mathsf{NonOverlapping}(P_1, \dots, P_m)$ is that the compulsory parts of P_i be pairwise disjoint. The following sweep algorithm verifies the necessary condition in $O(m \log m)$ time, and as a side effect, removes the *target* property from all ground rectangles. Thus it provides a quick initial test and avoids doing useless work later in the filtering algorithm:

1. Form a Q_{event} with start (end) events corresponding to $\mathcal{C}(P_i)_x^-$ ($\mathcal{C}(P_i)_x^+ + 1$) for $1 \leq i \leq m$ with non-empty $\mathcal{C}(P_i)$.
2. Let P_{status} record for each Y value the number of compulsory parts that currently intersect the sweep-line.
3. If after processing all events at Δ some element of P_{status} is greater than 1, the check fails.
4. When Q_{event} is empty, remove the *target* property from all ground P_i.

Domination. We say that *rectangle P_i dominates rectangle P_j* if the following relation holds between P_i and P_j for all $a \in \text{dom}(X_i)$:

$$\begin{array}{c}\textbf{if } X_i = a \text{ is feasible wrt. all constraints on } P_i \\ \textbf{then } X_j = a \text{ is also feasible wrt. all constraints on } P_j\end{array} \tag{4}$$

The point is to avoid useless work in line 4 of Algorithm 3. We have come up with a domination check which runs in $O(1)$ time and finds many instances of domination. Roughly, throughout the **for** loop, we maintain a “most dominating rectangle” P_{dom} among the P_i for which the test in line 4 is found false. In line 4, we first check if P_{dom} dominates P_i, in which case we can ignore P_i in the loop. Similarly for the other three sweep directions.

Incrementality. When the filtering algorithm is resumed, typically very few (usually one) rectangles have been pruned by some other constraint since the last time the algorithm suspended. We would like to avoid running a complete check of all rectangles vs. all rectangles, and instead focus on the subset of rectangles that could be affected by the external events. This idea is captured by the following steps, and is valid if we still are on the same branch of the search tree as at the previous call to the filtering algorithm.

1. Compute the bounding box $\mathcal{B}$ of the *targets* that were pruned since the last time. This takes $O(m)$ time.
2. In Algorithm 3 and in the initial check, ignore any rectangles that do not intersect $\mathcal{B}$, but if Algorithm 3 adjusts some bound, $\mathcal{B}$ must be updated to include the newly pruned rectangle.

5 Performance Evaluation

Wanting to measure the speed rather than the pruning power of the sweep algorithm, and the speedups of the optimizations, we generated six sets of problem instances, each consisting of three instances of m rectangles, $m \in \{100, 200, 400\}$; see Table 2. The sets were selected to represent typical usages of the constraint. E.g., Set 2 is a loose problem; Set 3 and 4 use rectangles of different sizes; in Set 5, the rectangles are all the same; Set 6 is 95% ground: it was computed by taking a solved instance of Set 4 and resetting the origin variables of 5% of the rectangles to their initial domains.

Table 2. Rectangle P_i for the different sets

	$\min(X_i)$	$\max(X_i)$	w_i	$\min(Y_i)$	$\max(Y_i)$	h_i
Set 1	1	10000	rand(1..20)	1	$101 - h_i$	rand(1..20)
Set 2	rand(1..200)	10000	rand(1..20)	rand(1..90)	$101 - h_i$	rand(1..20)
Set 3	1	10000	i	1	$1.05\sqrt{\sum_{j=1}^{m} j^2} - h_i$	i
Set 4	1	10000	$w_i^{(4)}$	1	$1.05\sqrt{\sum_{j=1}^{m} w_i * h_i} - h_i$	$h_i^{(4)}$
Set 5	1	10000	1000	1	10000	1000

$$\text{where } (w_i^{(4)}, h_i^{(4)}) = \begin{cases} ((m+3-i)/2, (m+1+i)/2), & \text{for odd i} \\ ((m+2+i)/2, (m+2-i)/2), & \text{otherwise} \end{cases}$$

Each of the 18 instances was run by setting up the constraint and fixing the origins of each $P_i, 1 \leq i \leq m$, to its lower bound. Each instance was run six times with different parameters controlling the algorithm (see Sect. 4.4):

s The sweep algorithm with shared event queue.
sp The sweep algorithm plus *sources* and *targets*.
sc The sweep algorithm plus the initial check.
sd The sweep algorithm plus domination.
si The sweep algorithm plus incrementality.
s∗ All optimizations switched on.

Fig. 3 summarizes the benchmark results. There is one graph per set, each with six plots comparing the different settings. Each legend is ordered by decreasing runtime, in milliseconds. The benchmarks were run in SICStus Prolog compiled with `gcc -O2` version 2.95.2 on a 248 MHz UltraSPARC-II processor, running Solaris 7. The results tell us the following:

– Set 4 was the most difficult instance, while Set 6 was the fastest to solve by at least an order of magnitude.
– The *sources* and *targets* was by far the most effective optimization. Incrementality was also generally effective. Both can be generalized to a large class of global constraints.

- Domination alone was not effective. We conjecture that it does contribute to the performance of **s**∗, at least on Set 5.
- The initial check optimization was not effective on any of the problem sets. We applied it each time the filtering algorithm was resumed. If used more judiciously, it might prove effective in some cases.
- There is a synergetic effect when several optimizations are combined.

Finally, we have compared the sweep (**s**∗) algorithm with implementations of the same constraint based on decomposition, cardinality and constructive disjunction as well as with diffn [2] in CHIP V5. The results for 100 rectangles are shown in Table 3. For cardinality, runtimes became prohibitive for larger instances.

6 Conclusion

We have presented a value sweep pruning algorithm which performs global constraint propagation by aggregating several constraints that share two variables. This method is quite general and can be applied on a wide range of constraints. The usual way to handle finite domain constraints is to accumulate forbidden one-dimensional regions in the domain of the variables of the problem. However, this is inefficient for constraints that do not initially have any one-dimensional forbidden regions since they have to be handled in a generate-and-test way (i.e. forbidden values appear only after fixing some variables). Value sweep pruning is an alternative which allows to accumulate forbidden regions much earlier in time. A key point is that we do not represent explicitly all forbidden regions but rather compute them lazily in order to perform specific pruning.

The main weak point of the algorithm is in line 2 of Algorithm 1. We would like to efficiently filter out the constraints C_i that do not generate any forbidden regions wrt. the variables X and Y under consideration.

We have shown how the value sweep algorithm can be used in a filtering algorithm for the non-overlapping rectangles constraint, first by simple specialization, and then by a modified sweep algorithm that uses a shared event queue corresponding to relative forbidden regions. Again, the weak point is in the search for relevant, non-empty forbidden regions in the event queue. Some combination of interval and range trees [8] could be appropriate.

We have described four optimizations to the filtering algorithm. The algorithm and the optimizations have been implemented, and a performance evaluation and some indication to their generality are given. The evaluation shows an improvement by several orders of magnitude overs implementations based on decomposition into binary constraints.

Acknowledgements

The research reported herein was supported by NUTEK (the Swedish National Board for Industrial and Technical Development). The idea of a shared event queue is due in part to Sven Thiel.

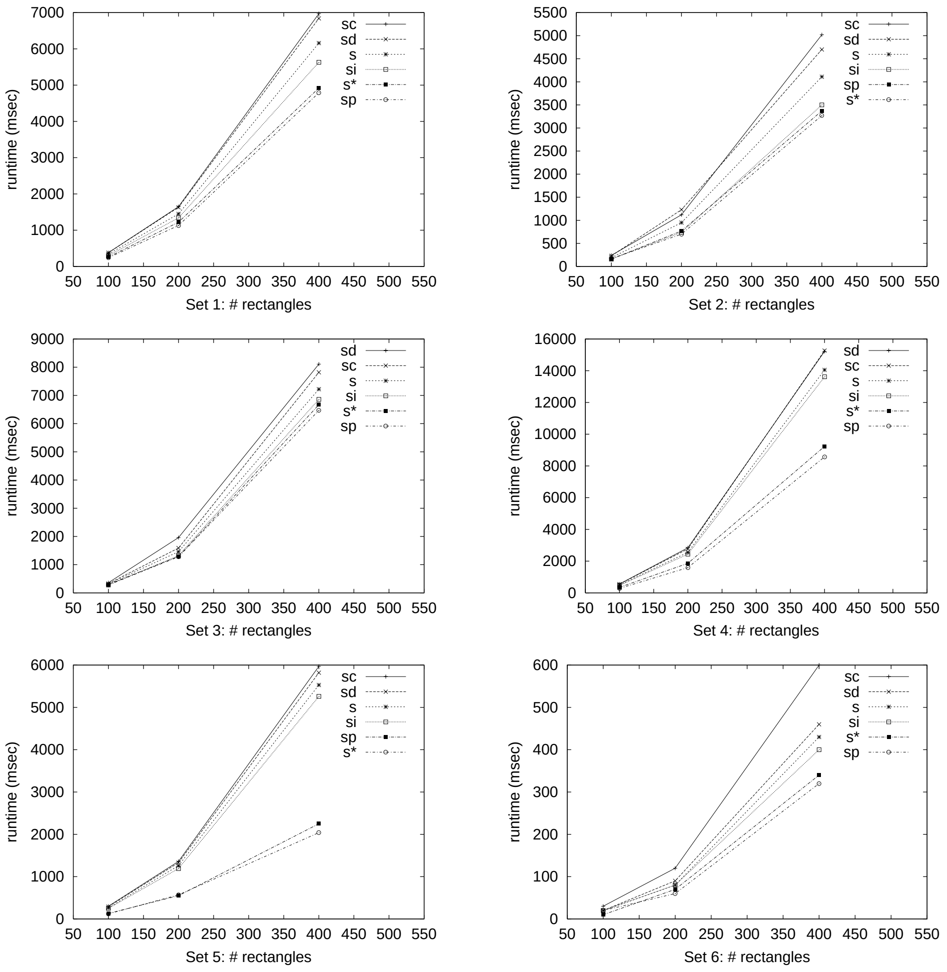

Fig. 3. Benchmark results

Table 3. Runtime (msec) for 100 rectangles

	Set 1	**Set 2**	**Set 3**	**Set 4**	**Set 5**	**Set 6**
card	113830	5110	508150	382870	9751490	1940
cd	5300	210	44190	16330	590890	10
diffn	600	140	690	1030	520	10
sweep	260	170	300	350	120	10

References

1. A. Aggoun and N. Beldiceanu. Extending CHIP in order to solve complex scheduling and placement problems. *Mathl. Comput. Modelling*, 17(7):57–73, 1993.
2. N. Beldiceanu and E. Contejean. Introducing global constraints in CHIP. *Mathl. Comput. Modelling*, 20(12):97–123, 1994.
3. M. Carlsson, G. Ottosson, and B. Carlson. An open-ended finite domain constraint solver. In H. Glaser, P. Hartel, and H. Kucken, editors, *Programming Languages: Implementations, Logics, and Programming*, volume 1292 of *Lecture Notes in Computer Science*, pages 191–206. Springer-Verlag, 1997.
4. F.R. du Verdier. *Résolution de problèmes d'aménagement spatial fondée sur la satisfaction de contraintes. Validation sur l'implantation d'équipements électroniques hyperfréquences.* PhD thesis, Université Claude Bernard-Lyon I, July 1992. In French.
5. A. Lahrichi. Scheduling: the notions of hump, compulsory parts and their use in cumulative problems. *C. R. Acad. Sci., Paris*, 1982.
6. P. Martin and D.B. Shmoys. A new approach to computing optimal schedules for the job-shop scheduling problem. In *Proc. of the 5th International IPCO Conference*, pages 389–403, 1996.
7. K. Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching.* EATCS Monographs. Springer-Verlag, Berlin, 1984.
8. F.P. Preparata and M.I. Shamos. *Computational Geometry. An Introduction.* Springer-Verlag, 1985.
9. H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, 1989.
10. Pascal Van Hentenryck, Vijay Saraswat, and Yves Deville. Design, implementation and evaluation of the constraint language cc(FD). In A. Podelski, editor, *Constraints: Basics and Trends*, volume 910 of *Lecture Notes in Computer Science.* Springer-Verlag, 1995.