Filtering Algorithms for Tabular Constraints

Roman Barták[*]

Charles University, Faculty of Mathematics and Physics
Malostranské námì stí 2/25, 118 00 Praha 1, Czech Republic
bartak@kti.mff.cuni.cz

Abstract. Constraint satisfaction technology supports all constraint types, i.e. arbitrary relations can be expressed, theoretically. However, in practice constraint packages support only some constraints, namely arithmetical, logical, and special purpose global constraints. If a general relation constraint needs to be expressed, the user is forced to represent it as a system of supported constraints or to design a special filtering algorithm for it. In this paper we describe filtering algorithms for general binary constraint whose domain can be expressed as a table of compatible pairs of values. Such constraints may appear in practical applications where the final user (rather than the developer) specifies the constraint domain using a simple user interface, e.g., using a table.

1 Introduction

Constraint programming (CP) deals with the problems stated as a set of variables with domains attached to them and as a set of constraints restricting the combinations of values that can be assigned to the variables [8]. The task is to find a consistent valuation of the variables, i.e., a valuation satisfying all the constraints. In theory, the constraint is an arbitrary relation among the variables. However, in practice, constraints are usually arithmetic (e.g. comparison) or logical relations (e.g. implication) with well defined semantic. Such restriction has a practical motivation - it is possible to exploit semantic of the constraint to design more efficient filtering algorithms that remove inconsistent values from the variables' domains. For example, if the constraint is a linear relation then it is possible to propagate bounds of the domain rather than to check all the values in the domain. This significantly speeds up filtering and therefore such methods are used for other constraints as well even if they do not (sometimes) remove all incompatibilities from the domains. Because such filtering algorithms are fine-tuned in the constraint satisfaction packages it is highly recommended simplifying the constraints expressed as general relations into a more mathematical way. This is usually possible, if the developer knows the constraints in advance, but it could be very hard, if the end-user states the constraint in the form of a table directly in the system. Then, having some form of a general relation constraint is highly desirable to simplify development of the system.

[*] Supported by the Grant Agency of the Czech Republic under the contracts 201/99/D057 and 201/01/0942.

Our work on filtering algorithms for tabular constraints is motivated by practical problems where important real-life constraints are expressed in the form of tables. In particular, this work is motivated by complex planning and scheduling problems where the user states relations over objects like activities or resources. In complex environments, there could appear pretty complicated relations among the activities expressing, for example, transitions between the activities allocated to the same resource (like changing a colour of the produced item). Typically, the activities are grouped in such a way that transitions between arbitrary two activities within the group are allowed but a special set-up/transition activity is required for the transition between the activities of different groups. The most natural way to express such relation is using a table describing the allowed transitions (see Figure 1).

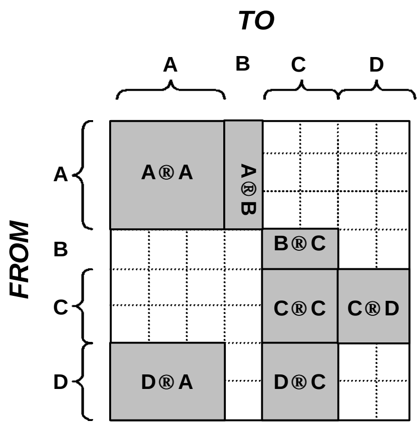


Fig. 1. A transition constraint expressed as a table of compatible transitions (shadow regions).

This table could be automatically converted into a mathematical formulation but such automatic conversion is usually very complicated or even impossible. Designing a special filtering algorithm is another possibility to tackle the problem of general relation constraints. We chose the second option because it seems to be more applicable in real-life problems thanks to its generality. Moreover, we can design the filtering algorithm in such a way that it exploits a typical structure of the constraint domain and thus filtering is more efficient when such a structured constraint appears. In particular, we observed that the transition constraints, that we model using the tabular constraint, have a rectangular structure, i.e., the constraint domain consists of several (possible overlapping) rectangles of compatible pairs. This information can be used to design a special filtering algorithm that is more efficient if the constraint domain is rectangular and that is still capable to do filtering on arbitrary other constraint domain.

In this paper, we describe the filtering algorithms for binary tabular constraints. We restrict to binary constraints because this type of a general constraint can be easily described using a table[1]. First, we introduce some geometric notions describing the structure of the constraint domain and we sketch the environment to which our

[1] General constraints of arity greater than three cannot be easily inputted as a table.

filtering algorithm fits. Then we describe one of the straightforward filtering algorithms and we propose its extension to work more efficiently with rectangular structured constraints. The main part of the paper is dedicated to the description of a sweep technique for filtering of tabular constraints. We conclude with practical examples of tabular constraints and with comparison to existing techniques.

2 Preliminaries

Constraint programming is a framework for declarative problem solving by stating constraints over the problem variables and then finding a value for each variable from the domain attached to the variable in such a way that all the constraints are satisfied. In general, the *constraint* is an arbitrary relation restricting the possible combinations of values for constrained variables. The *constraint domain* is a set of tuples satisfying the constraint, e.g., {(0,2), (1,1), (2,0)} for the constraint X+Y=2 and variable domains containing non-negative integers. We say that the constraint has a *simple rectangular structure*, if $C(Xs) = \times_{X \in Xs} C(Xs)\downarrow X$, where Xs is a set of constrained variables, C(Xs) is a constraint domain, and $C(Xs)\downarrow X$ is a projection of the constraint domain to the variable X (see Figure 2). For example, the above constraint X+Y=2 does not have a simple rectangular structure because the projection to both variables is {0,1,2} so the Cartesian product is larger than the constraint domain. The notion of a rectangular structure is derived from the structure of the constraint domain for binary variables (as explained above we restrict ourselves to binary tabular constraints because they seem enough for practical applications).

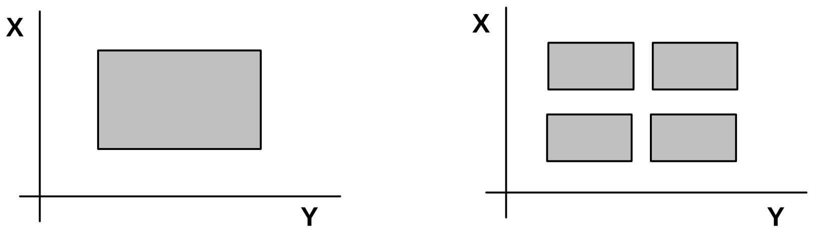


Fig. 2. Binary constraints with simple rectangular structure (shadow rectangles represent a constraint domain).

If the constraint has a simple rectangular structure then it is easy to satisfy this constraint: domains of the involved variables are restricted to respective projections and then, arbitrary combination of values from the reduced domains satisfies the constraint. Unfortunately, the structure of the constraint domain is usually more complicated, still we observed that the tabular constraints inputted by the users have a structure that can be decomposed into rectangles (see Figure 1). The *rectangle* R is a subset of the constraint domain such that R = X'×Y' where X' and Y' are intervals of values in projections of the constraint domain to respective variables. Arbitrary constraint domain can be represented as a union of rectangles, then we are speaking about the *rectilinear rectangular covering*. Note that in general we do not require these rectangles to be disjoint. The rectilinear rectangular covering is not unique for a given constraint domain (see Figure 3), however we prefer a smaller covering (a smaller number of rectangles) because complexity of our filtering algorithm depends on the size of the covering. The algorithms for finding a (minimal) rectilinear rectangular covering are out of scope of this paper; their description can be find in [5,7].

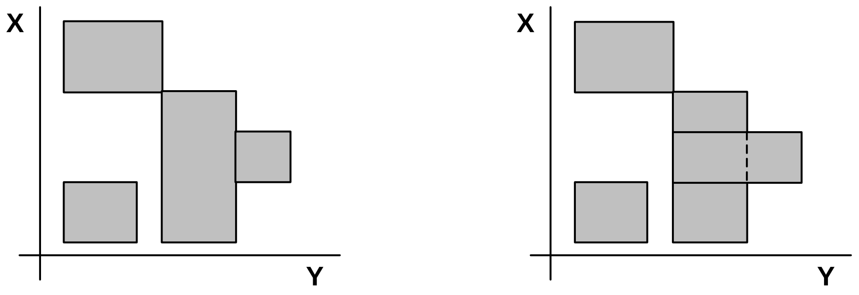


Fig. 3. Different rectilinear rectangular coverings for a single tabular constraint.

Constraint propagation (domain filtering, consistency technique) is one of the most important techniques used in constraint programming [8]. The goal of constraint propagation is to remove values from variables' domains that cannot be part of any solution satisfying all the constraints. There exist several notions of consistency typically differentiating in the number of removed inconsistent values. Among the consistency techniques arc-consistency plays the most important role thanks to its good ratio between the number of removed inconsistencies and complexity. We say that the constraint is *(hyper) arc-consistent*[2] if every value of every variable in the constraint is part of some solution of the constraint. For example, the constraint X+Y=2, where both the variables X and Y have domain {0,1,2}, is arc-consistent. Removing the values from domains to achieve some consistency is called *domain filtering*. The notion of consistency can be naturally extended to a problem consisting of several constraints. We say that a constraint satisfaction problem is arc-consistent if all the constraints are arc-consistent.

To make the problem arc-consistent, it is not enough to make every constraint consistent just once because deletion of the value by one constraint may evoke deletion of another value in another constraint that was consistent before the deletion. For example, if the value 0 is removed from the domain of the variable X by the constraint X>0 then we need to remove the value 2 from the variable Y to make the constraint X+Y=2 consistent again. Thus, we need to repeat revisions of the constraints until any domain is pruned. Perhaps the most widely used arc-consistency algorithm is AC-3 that re-revises only the constraints affected by the deletion. The

[2] Usually, the notion of arc-consistency is used for binary constraints only. For constraints of higher arity, the notions of hyper arc-consistency or generalised arc-consistency are used. For simplicity reasons we will use the term arc-consistency or simply consistency only.

advantage of this algorithm is that it can be used for arbitrary constraints, in particular no prerequisite for data structures is required. To integrate a new constraint into the AC-3 framework, the constraint must provide a REVISE procedure that does domain filtering (makes the constraint consistent) and it must define the trigger when the filtering should be evoked (e.g. when a domain of any involved variable is changed).

```
procedure AC-3(Constraints)
   Q ← Constraints
   while Q nonempty do
       C ← select_and _delete(Q)
       Change ← REVISE(C)
       Q ← Q ∪ affected_constraints(Constraints,Change)
   end do
end AC-3
```

Fig. 4. AC-3 algorithm for enforcing arc consistency.

3 GR Filtering Algorithms

In [1] a straightforward filtering algorithm GR (general relation) for tabular constraints was proposed. This algorithm enforces arc-consistency of the tabular constraint and it can be easily integrated to the standard AC-3 algorithm via the REVISE procedure. The filtering is evoked every time a domain of any involved variable is changed. We present here a simple extension of this algorithm that stops filtering when the constraint restricted to the actual domains of involved variables has a simple rectangular structure. Then further filtering is not necessary because all the combinations of the values are allowed by the constraint.

The GR algorithm uses a thread representation of the constraint domain. One of the variables is chosen as the leading variable and the second variable is called dependent. The constraint is represented as an (ordered) set of values of the leading variable and to each value a set of compatible values from the dependent variable is attached. Naturally, this extensional representation can be used only when it is finite. In particular, we need the domain of the leading variable to be finite and the set of compatible values of the dependent variable to be finitely representable (for details see [1]). Nevertheless, in practice this is not a strong restriction, because we are modelling tabular constraints and such restrictions are naturally included in the tabular input. In particular, the number of rows in the table is finite so each row can represent a value of the leading variable. Moreover, we input the set of compatible values to every row so we already have a finite representation, e.g. using a list of intervals. Note that it does not necessarily imply that the domain of the dependent variable must be finite as well - we can use intervals with minus/plus infinity (inf/sup) to describe the domain with infinite number of values, e.g. 8..sup represents all the values greater or equal to 8. Figure 5 shows how a tabular form is converted to the thread representation with the leading variable X and the dependent variable Y.

Notice that if there is no compatible value for the value of the leading variable then this value is simply omitted from the representation.

Tabular constraint:

X	**Y**	
1	2..20, 30..50	
2	-	*No compatible value*
3	inf..sup	*No restriction on Y*
4	10..50	

is represented as the list [1-[2..20,30..50], 3-[inf..sup], 4-[10..50]].

Fig. 5. Conversion of the tabular form of the constraint to the thread representation.

Each time a domain of the leading or dependent variable is pruned, the GR algorithm is evoked. This algorithm goes through the constraint domain representation and checks the compatible values with respect to the current domain of the variables. Note that the algorithm exploits the non-symmetrical representation of the constraint. In particular, it tests membership of the value *x* of the leading variable in the domain first (row 9) before testing non-emptiness of the intersection of the domain of the dependent variable Y with the values compatible with *x* (rows 10-11). This is faster than doing it in the reverse order (in case $x \notin$ domain(X)). The algorithm builds also a new domain for both the leading and dependent variable. We add a simple extension to the original algorithm that tests whether the constraint domain after reduction to the current domains of the variables has a simple rectangular structure (rows 14-20). If it has a simple rectangular structure then the constraint is entailed and the filtering algorithm is not called after future deletions of values from the domains.

In [1], several versions of the GR algorithm are studied with respect to time and space complexity. These algorithms change the domain of the constraint to respect the current domains of the constrained variables. Our extension for testing simple rectangular structure of the constraint domain can be used in these algorithms as well.

Proposition 1: The GR algorithm is sound and complete, i.e., it makes the constraint arc-consistent (only consistent values are kept in the domains and no consistent value is removed).

Proof: The value x is included in the new domain for the leading variable (row 12) if and only if it is part of the original domain (row 9), it is part of the projection of the constraint to the leading variable (row 8) and there is a compatible value for x in the domain of the dependent variable (rows 10-11). Visibly, new domain of the dependent variable contains only values from the original domain (row 10), that are compatible with any value of the leading variable (rows 8-9). The constraint is entailed if for every value of the leading variable the sets of compatible values of the dependent variable are unique. ❑

```
1  procedure GR(Constraint,X,Y)
2     NewDomainOfX ← empty
3     NewDomainOfY ← empty
4     ConstraintDomain ← domain(Constraint)
5     Entailed ← true
6     LastProjectionOfY ← empty
7     while non_empty(ConstraintDomain) do
8        (x-DY) ← select_and_delete(ConstraintDomain)
9        if x∈domain(X) then
10          CompatibleY ← intersection(domain(Y),DY)
11          if non_empty(CompatibleY) then
12             NewDomainOfX ← union(NewDomainOfX, {x})
13             NewDomainOfY ← union(NewDomainOfY, CompatibleY)
14             if Entailed then
15                if empty(LastProjectionOfY) then
16                   LastProjectionOfY ← CompatibleY
17                else
18                   Entailed ← (LastProjectionOfY == CompatibleY)
19                end if
20             end if
21          end if
22       end if
23    end while
24    X in NewDomainOfX
25    Y in NewDomainOfY
26 end GR
```

Fig. 6. The GR filtering algorithm.

Proposition 2: Time complexity of the GR algorithm is O(dx*dy), where dx and dy are sizes of domains of the leading variable and the dependent variable respectively.

> *Proof*: The while cycle in the algorithm is repeated O(dx) times and the most complex step in this cycle is computing intersection and union of the domains (rows 10,13) which has complexity O(dy). So together the complexity is O(dx*dy). ❑

4 Sweep Filtering Algorithm

The GR algorithm uses a straightforward representation of the constraint domain but this thread representation does not allow exploiting the structure of the domain to full extent. In fact, we can only use a simple rectangular structure of the domain there. If the structure is more complex, like Figure 1 shows, then the GR algorithm is less efficient because it must check all pairs of values. Therefore, we propose to represent the constraint domain as a set of rectangles that forms the rectilinear rectangular covering of the constraint domain. Then, the filtering algorithm can explore only the rectangles instead of working with threads (pairs (value, compatible values) used by the GR algorithm).

The proposed filtering algorithm is based on a technique called *sweep* that is widely used in computational geometry and that was first applied to domain filtering in [3]. The sweep algorithm moves a vertical line (called a *sweep line*) along the horizontal axis from left to right and each time it encounters or leaves an object (this is called an *event*), it triggers some event handler according to the event type. Thus the algorithms sweeps the plane, hence its name. In case of domain filtering, there are four types of events used by the sweep algorithm:

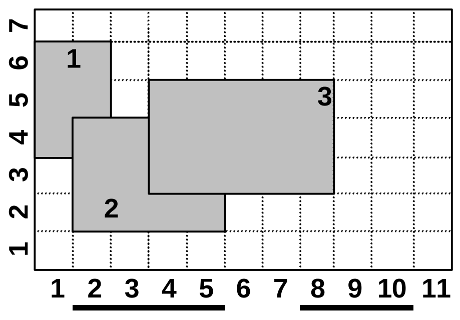
rect_start(PosX,NumR,IntY) - indicates the left border (PosX) of the rectangle identified by NumR with the vertical projection IntY,

rect_end(PosX,NumR) - indicates the right border (PosX) of the rectangle identified by NumR,

x_start(PosX) - indicates the start of some coherent interval within the current domain of the leading variable,

x_end(PosX) - indicates the end of some coherent interval within the current domain of the leading variable.

The list of events can be generated in advance from the constraint domain and the current domain of the leading variable. We call such a list an *event point series.* The events in the event point series are ordered increasingly according to the x-axis position of the event (PosX). Moreover, we require the start events to precede the end events with the same x-axis position. This is necessary for the algorithm to capture "one-point" overlaps between the objects. Figure 7 shows an example of the event point series for the constraint domain consisting of three overlapping rectangles and the domain of the leading variable consisting of two intervals (2..5 and 8..10).



Event point series:
rect_start(1,1,4..6),
rect_start(2,2,2..4)
x_start(2),
rect_end(2,1),
rect_start(4,3,3..5),
x_end(5),
rect_end(5,2),
x_start(8),
rect_end(8,3),
x_end(10)

Fig. 7. Construction of the event point series for the constraint domain.

During the computation, the SP (sweep pruning) algorithm keeps some global data structures that describe the status of computation:

InDomain - indicates whether the sweep line is within the domain of the leading variable, i.e., in between x_start and x_end events corresponding to a single coherent interval,

ActiveRects - describes the set of rectangles that are crossed by the sweep line, i.e., the rectangles where the rect_start event has been processed and the corresponding rect_end has not been reached yet.

Also, the SP algorithm incrementally builds new domains for the leading variable (*ListOfX*) and the dependent variable (*NewDomainOfY*). While *NewDomainOfX* is constructed using a set operations, *ListOfX* keeps a list of "border" points of intervals in the new domain of the leading variable (in the reverse order) that is then converted to the domain (rows 36-41).

```
27  procedure SP(Constraint,X,Y)
28     EventPointSeries ← make_event_point_series(Constraint,X)
29     NewDomainOfY, ActiveRects, ListOfX ← empty
30     DY ← domain(Y)
31     InDomain ← false
32     while non_empty(EventPointSeries) do
33        Event ← select_and_delete_first(EventPointSeries)
34        process_event(Event,DY,ActiveRects,InDomain,
                         ListOfX,NewDomainOfY)
35     end while
36     NewDomainOfX ← empty
37     while non_empty(ListOfX) do
38        Max ← select_and_delete_last(ListOfX)
39        Min ← select_and_delete_last(ListOfX)
40        NewDomainOfX ← union(Min..Max,NewDomainOfX)
41     end while
42     X in NewDomainOfX
43     Y in intersection(NewDomainOfY,DY)
44  end SP
```

Fig. 8. The SP filtering algorithm.

```
EVENT - ACTION
rect_start(PosX,NumR,IntY)
  45  if non_empty(intersection(IntY,DY)) then
  46     if InDomain then
  47       NewDomainOfY ← union(IntY,NewDomainOfY)
  48    if empty(ActiveRects) then
  49     ListOfX ← PosX : ListOfX
  50       end if
  51   end if
  52   ActiveRects ← r(NumR,IntY) : ActiveRects
  53  end if
rect_end(PosXx,NumR)
  54  if find_and_delete(r(NumR,_),ActiveRects) then
  55     if InDomain && empty(ActiveRects) then
  56       ListOfX ← PosX : ListOfX
  57    end if
  58  end if
x_start(PosX)
  59  InDomain ← true
  60  if non_empty(ActiveRects) then
  61    ListOfX ← PosX : ListOfX
  62    foreach r(NumR,IntY) in ActiveRects do
  63      NewDomainOfY ← union(NewDomainOfY,IntY)
  64    end foreach
  65  end if
x_end(PosX)
  66  InDomain ← false
  67  if non_empty(ActiveRects) then
  68    ListOfX ← PosX : ListOfX
  69  end if
```

Fig. 9. Event processing for the SP filtering algorithm.

The SP algorithm is more or less self-explanatory. Notice that only the rectangles having a non-empty projection to the domain of the dependent variable are processed (rows 45, 54), let us call these rectangles relevant. If the sweep line enters the relevant rectangle (rec_start event) and it is within the domain of the leading variable X (row 46), then the projection of the rectangle to Y-axis is added to the new domain of Y (row 47). If it is the first rectangle that has non-empty intersection with the current interval of X (row 48) then the start of the new interval is added to the new domain of X (row 49). When entering the relevant rectangle we make this rectangle active by memorising it in the ActiveRects structure (row 52). If we leave the last rectangle (rect_end event) that is active (row 55) then the end of the new interval is added to the new domain of X (row 56). If we enter a new interval within the domain of X (x_start event) and there is any active rectangle (row 60) then the new start of the new domain of X is created (row 61). Also, the new domain of Y is extended by projections of active rectangles to Y-axis (row 63). If we leave some interval within the domain of X (x_end event) and there is still some active rectangle then a new end of the interval is added to the new domain of X (row 68).

Figure 10 shows a run of the SP algorithm when exploring the constraint domain from Figure 7. Contents of basic data structures after processing an event is displayed there.

EVENT	ListOfX	InDomain	ActiveRects	NewDomainOfY
`rect_start(1,1,4..6)`	empty	false	r(1,4..6)	empty
`rect_start(2,2,2..4)`	empty	false	r(1,4..6)	empty
`x_start(2)`	2	true	r(1,4..6)	4..6
`rect_end(2,1)`	2,2	true	empty	4..6
`rect_start(4,3,3..5)`	4,2,2	true	r(3,3..5)	3..6
`x_end(5)`	5,4,2,2	false	r(3,3..5)	3..6
`rect_end(5,2)`	5,4,2,2	false	r(3,3..5)	3..6
`x_start(8)`	8,5,4,2,2	true	r(3,3..5)	3..6
`rect_end(8,3)`	8,8,5,4,2,2	true	empty	3..6
`x_end(10)`	8,8,5,4,2,2	false	empty	3..6

Fig. 10. Run of the SP filtering algorithm for DY=5..10 and the constraint domain from Fig. 7.

Proposition 1: The SP algorithm is sound and complete, i.e., it makes the constraint arc-consistent (only consistent values are kept in the domains and no consistent value is removed).

Proof: The new domain of the leading variable X is constructed from the intervals provided by the SP algorithm. Each such interval is a sub-interval of the original domain of X and it is also a subset of the projection of the relevant rectangles to X-axis. Thus every value in this interval has a support in Y. Moreover, if there is a value *x* from the domain of X that has support *y* in the domain of Y then there must be a rectangle containing the pair (*x*,*y*) and this rectangle becomes active sometime during the computation.

The new domain of the dependent variable Y is a union of projections of the active rectangles whose projection to X-axis has a non-empty intersection with any interval of the domain of X. In particular, all the values in the new domain of Y have a support in the domain of X. Among them, only the values belonging to the original domain of Y are selected (row 43). ❏

Proposition 2: Time complexity of the SP algorithm is O(dx + *r*(log *r* + dy)), where dx and dy are sizes of domains of the leading and dependent variable respectively (number of disjoint intervals) and *r* is a number of rectangles.

Proof: To achieve a good time complexity of the SP algorithm it is necessary to use a clever implementation of data structures. For example, instead of using the list of active rectangles (ActiveRects), it is better to use an array with the size *r* that indicates whether the rectangle is active or not. Then complexity of adding, finding and deleting an active rectangle (row 54) is O(1). Moreover, we can use a counter of active rectangles to find if there is any active rectangle, the time complexity of using this counter is O(1). Also to prevent repeated union of the same intervals IntY (row 63) when the rectangle expands through several intervals of the domain of X, it is possible to indicate in the above-mentioned array whether the rectangle has already been included. Moreover, instead of computing the union of the intervals IntY incrementally (row 63) with complexity *r***r* (insert sort technique), it is better to keep these intervals in a list and to produce the union at the end. Then the aggregated complexity of computing the new domain of Y is *r**log *r* (we sort at most *r* intervals).

Time complexity of processing the rect_start event is O(dy) that is the complexity of computing the intersection at row 45 (all other commands have complexity O(1)). Time complexity of processing the rect_end event is O(1) as discussed above. Both rect_start and rect_end events are evoked *r* times so the aggregated complexity is O(*r**dy). Time complexity of processing the x_start event is O(1) provided that the construction of the domain of Y is computed separately. Time complexity of processing the x_end event is O(1). Both x_start and x_end events are evoked dx times so the aggregated complexity is O(dx). Construction of new domain of X (rows 37-41) has complexity O(dx). Together the time complexity of single propagation step is O(dx + *r*(log *r* + dy)). ❑

5 Related Works

The idea of using semantic of the constraint to get more efficient filtering algorithms is not new and it is widely used especially in so called global constraints like all-different [6]. However, all these algorithms are designed for a particular constraint, i.e. for given semantic, while our filtering algorithm works with arbitrary binary constraint and exploits its rectangular structure. Thus, semantic of the constraint is not fixed and the algorithm works with arbitrary binary constraint.

Another example of using semantic information to do better filtering may be found in generic AC-5 algorithm by Van Hentenryck, Deville, and Teng [9] where using functional, anti-functional, and monotonic constraints improves efficiency of the arc-consistency algorithm. Our algorithm is less general and it is not motivated by arithmetic constraints. We rather took the real-life constraints from the area of planning and scheduling, analysed their typical structure (semantic) and then we proposed an algorithm to handle these constraints more efficiently.

Sweep technique to do domain filtering was proposed by Beldiceanu in [3]. His method aggregates several constraints sharing some variables so it removes more inconsistencies than using the constraints separately. Thus, the set of constraints behaves more like a special global constraint. We adapted the sweep technique to improve efficiency of the arc-consistency algorithm rather than to remove more values beyond arc-consistency.

Constraint engines embedded in programming languages often include tools for definition of general constraints, like *element* and *relation* predicates in SICStus Prolog [4]. The *relation* predicate in SICStus Prolog has behaviour necessary to model the tabular constraints. It uses the thread representation to describe the constraint domain, however unlike our GR algorithm the *relation* predicate is not able to handle infinite domains so the tabular constraint from Figure 5 cannot be modelled using the *relation* predicate. Moreover the *relation* predicate does not exploit the special structure of the constraint domain to improve efficiency. Using the GR algorithm, we improve the efficiency as far as ten times in comparison with the *relation* predicate simply because many user-defined tabular constraints have a simple rectangular structure or they soon converge to such a structure as the domains of constraints variables are pruned. The *element* constraint can be used to model tabular constraints as well, but we need two *element* constraints to model a general binary constraint. Nevertheless, there are two main drawbacks when the *element* predicate is used. First, it requires an extensional representation of the constraint domain, i.e., every pair of compatible values must be included. This becomes intractable soon when the number of compatible values is large. Of course, infinite domains cannot be represented this way. Our representation of the constraint domain is more compact and we can even work with infinite domains. The second disadvantage of the *element* predicate is that it performs only linear consistency when applied to tabular constraints. It means that some inconsistent pairs of values are not detected that could be a problem in some applications. Our algorithms perform full arc-consistency so they remove all incompatible pairs.

6 Discussion on Usage and Experiments

In Introduction we give a short motivation for tabular constraints and we provide more examples here. The basic motivation of our research is from integrated planning and scheduling problems. In these problems we need to introduce activities to do some task during the course of problem solving. To represent a space for the activity we use a slot model [2]. The basic attribute of the slot is Activity variable describing which activity can be filled in the slot. There are other attributes of the slot that are dependent on the value of the Activity variable, e.g., Start_time and Duration. Different activities may have different duration so we need to express the relation between Activity and Duration variables using a constraint. Unfortunately, the domain of this constraint is usually inputted as a table so it is hard to express such constraint in a mathematical way. Thus general binary constraint must be used. Note also, that duration is not necessary a single number but it could be an interval that expresses variability of duration per activity (e.g. we can choose different production rate of the machine when scheduling the activity). Thus, using the *element* constraint is not natural there and the *relation* constraint should be used instead. Similar analysis can be done for the relation between Activity and Start_time attributes that describes the time windows for the activity. However, now the situation is slightly different because there could be time windows with infinite intervals (see Figure 5). Unfortunately, the SICStus build-in predicate *relation* cannot be used there because it does not support infinite domains. This is the main reason why we started to design filtering algorithms for such tabular constraints. Moreover, we found that domains of tabular constraints had often a structure of rectangle(s) so we extended our first GR filtering algorithm to work more efficiently with rectangular constraint domains. The results were amazing: when compared to build-in *relation* predicate we get as ten times better performance even if our implementation seems pretty naive. The more sophisticated SP algorithm works even more efficiently when there are more rectangles in the constraint domain (see complexity analyses in Section 4) but we did not integrate this algorithm to our scheduling engine yet (so no experimental results are available). The problem is that this algorithm is very dependent on decomposition of the constraint domain to rectangles. Because there are several tabular constraints in the model, each constraint may require a different ordering of values in variable's domains to achieve a "nice" rectangular structure. So, one way of future research could be to investigate interference between several tabular constraints. We concentrate more on "globalisation" of the set of tabular constraints, i.e., instead of using a set of tabular constraints, we propose to use a global constraint that achieves better pruning. In our problem area, there are several binary constraints between the Activity variable and other variables so it seems more promising to integrate them into a single constraint rather than to improving efficiency of the filtering algorithm for binary constraints only. Note that contrary to the GR algorithm, the SP algorithm can be extended more naturally to n-ary constraints [5] so this could be the way to improve pruning.

7 Conclusions

In this paper we propose two algorithms for domain filtering of binary tabular constraints. Such constraints appear in real-life applications especially, when the user is allowed to enter the domain of the constraint. We concentrate on tabular constraints with rectangular structure of the constraint domain so our algorithms can exploit such a structure to improve efficiency. The GR algorithm may seem naive but its extension discovering the simple rectangular structure proved itself to be very powerful. We use this algorithm in a generic scheduling engine and it improves efficiency as far as ten times in comparison with the existing *relation* predicate. Moreover, we can handle infinite domains that are often used to describe less tight relations. The SP algorithm uses different technique to do pruning than GR algorithm and it has potential to grow to n-ary constraints so we are exploring this possibility now.

8 References

[1] Barták R.: A General Relation Constraint: An Implementation, in Proceedings of CP2000 Post-Workshop on Techniques for Implementing Constraint Programming Systems, Singapore (2000)

[2] Barták, R.: A Slot Representation of the Resource-Centric Models for Scheduling Problems, in Proceedings of ERCIM Working Group on Constraints/CompulogNet Area on "Constraint Programming" Workshop, Padova (2000)

[3] Beldiceanu N.: Sweep as a generic pruning technique, in Proceedings of CP2000 Post-Workshop on Techniques for Implementing Constraint Programming Systems, Singapore (2000)

[4] Carlsson M., Ottosson G., Carlsson B.: An Open-Ended Finite Domain Constraint Solver, in Proceedings Programming Languages: Implementations, Logics, and Programs (1997)

[5] Michalský, R.: Algorithms for Constraint Satisfaction, Master Thesis, Charles University, Prague (2001)

[6] Régin J.-Ch.: A filtering algorithm for constraints of difference in CSPs. Research Report LIRMM 93-068, LIRM, Université Montpellier, France (1993)

[7] Shearer J.B, Wu, S.Y., and Sahni S.: Covering Rectilinear Polygons by Rectangles, in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 9 (1990)

[8] Tsang E.: Foundations of Constraint Satisfaction, Academic Press, London (1993)

[9] Van Hentenryck P., Deville Z, and Teng. C.-M.: A generic arc-consistency algorithm and its specializations, in Artificial Intelligence 57, pp. 291-321 (1992)