Implementing Propagators for Tabular Constraints

Roman Barták, Roman Mecl

Charles University in Prague, Faculty of Mathematics and Physics
Institute for Theoretical Computer Science
Malostranské nám. 2/25, 118 00 Praha 1, Czech Republic
{bartak,mecl}@kti.mff.cuni.cz

Abstract. Many real-life constraints describing relations between the problem variables have complex semantics and the constraint domain is defined using a table of compatible tuples rather than using a formula. In the paper we study the implementation of filtering algorithms (propagators) for such tabular constraints. We concentrate on implementation aspects of these algorithms so the proposed propagators can be naturally integrated into existing constraint satisfaction packages like SICStus Prolog.

Introduction

Many real-life problems can be naturally modeled as a constraint satisfaction problem (CSP), i.e. using variables with a set of possible values (domain) and constraints restricting the allowed combinations of values. Quite often, the semantics of the constraint is well defined via mathematical or logical formulas, e.g. comparison, implication etc. However, the intentional description of some constraints is rather complicated and it is much easier to describe them extensionally as a set of compatible tuples. The relation between the type of activity and its duration or the description of next activity in the transition scheme in scheduling applications [1,2] are typical examples of such constraints. The constraint domain is specified there as a table (Figure 1) rather than as a formula, thus we are speaking about tabular constraints.

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

Fig. 1. Example of a tabular constraint: a range of compatible values of Y is specified for each value of X.

In this paper we study the filtering algorithms (propagators) for binary constraints where the constraint domain is described as a table. We compare two filtering

* Research is supported by the Grant Agency of the Czech Republic under the contract no. 201/01/0942.

algorithms that use a compact representation of the constraint domain. Such compact representation turned out to be crucial for efficiency of the algorithms applied to real-problems with non-trivial domains (i.e., the size of the domain is in the range of thousands or millions of elements). Rather than providing a theoretical study of the algorithms we concentrate on the practical aspects of the implementation, which are usually omitted in research papers. Thus the algorithms are proposed in such a way that they can be easily integrated into mainstream constraint solvers.

The paper is organized as follows. First, we give a motivation for using tabular constraints and we survey existing approaches to model such constraints. Then we describe two filtering algorithms for tabular constraints. These algorithms extends our previous works on tabular constraints [3,4] by using a more compact representation of the constraint domain and better entailment detection. In some sense, both algorithms converge by sharing a sweep technique. We conclude the paper by empirical comparison of the algorithms using large-scale real-life scheduling problems.

Motivation

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 constraints over the objects like activities and resources. In complex environments, there could appear pretty complicated relations between the activities expressing, for example, transitions between the activities allocated to the same resource like changing a color of the produced item [1]. Typically, the activities are grouped in such a way that the 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 2).

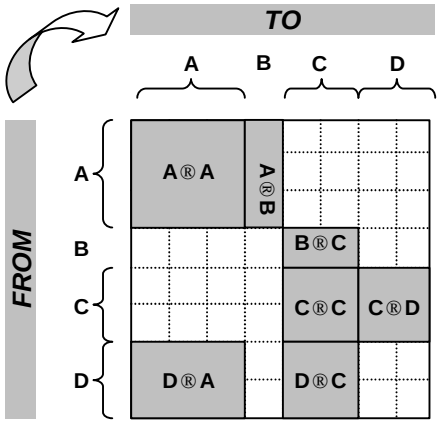


Fig. 2. A transition constraint expressed as a table of compatible transitions (shadow regions).

As we showed in [2] there are many other constraints of the above type in real-life planning and scheduling problems, e.g. description of time windows, duration, and cost of activity. These constraints can hardly be described using a mathematical formula because the user specifies the constraint domain (the set of compatible pairs) using a table. Therefore special filtering algorithms for such constraints are highly desirable. Efficiency of such filtering algorithms can be improved by exploiting information about the structure of the constraint domain. For example, we have noticed that the structure of many tabular constraints is rectangular, i.e., the constraint domain consists of several (possible overlapping) rectangles of compatible pairs (see Figure 2). 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.

Related Works

The existing constraint solvers already provide a mechanism to model tabular constraints without necessity to program a new filtering algorithm. For example, in SICStus Prolog [10], there is a `relation` constraint where the user can specify a binary constraint as a list of compatible pairs. In particular, for each value of the first (leading) variable, the user describes a range of compatible values of the second (dependent) variable. Thus the domain for the leading variable must be finite (till the version 3.8.7 the domain of the dependent variable must be finite as well).

The tabular constraint can also be modeled using a pair of `element` constraints. The input of the constraint domain is even less compact there. When the `element` constraints are used to model a tabular constraint then every pair of compatible values must be specified (the constraint domain is completely extensional). Thus, it is impossible again to represent infinite domains.

In [3] a straightforward filtering algorithm called general relation was proposed. This algorithm supports infinite domains for the dependent variable and it provides a mechanism to detect constraint entailment when the reduced constraint domain has a rectangular structure [2]. However, the general relation technique still uses a less compact representation identical to the `relation` constraint. Later in the paper we describe an extension of this algorithm that allows infinite domains for both leading and dependent variables and that is more time and memory efficient.

In [5], a new technique called sweep was proposed to explore constraint domains. This technique was applied to tabular constraints in [3]. The sweep filtering algorithm represents the constraint domain using a rectilinear rectangular covering (a set of possibly overlapping rectangles) so the representation is more compact. The complexity of the filtering algorithm depends on the number of rectangles rather than on the size of the constraint domain. However, this algorithm has no mechanism to detect constraint entailment. Later in the paper, we present an extension to this algorithm that includes a detector of constraint entailment and that uses a more compact representation of the constraint domain.

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 in such a way that all the constraints are satisfied. The value for a particular variable can be chosen only from the variable domain, i.e., from a set of possible values for the variable. *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 i.e. it is a subset of the Cartesian product of the variables' domains. For example, {(0,2), (1,1), (2,0)} is a domain of the constraint X+Y=2 where variables' domains consist of non-negative integers. If C is a constraint over the set of variables Xs then we denote a constraint domain C(Xs). We say that the constraint domain has a *rectangular structure*, if $C(Xs) = \times_{X \in Xs} C(Xs)\downarrow X$, where $C(Xs)\downarrow X$ is a projection of the constraint domain to the variable X (see Figure 3). For example, the above constraint X+Y=2 does not have a rectangular structure because the projection to both variables is {0,1,2} and the Cartesian product {0,1,2}×{0,1,2} is larger than the constraint domain. The notion of a rectangular structure is derived from the structure of the constraint domain for binary variables.

Assume that C(Xs) is a domain of the constraint C and D(X) is a domain of the variable X (a set of values). We call the intersection $C(Xs) \cap (\times_{X \in Xs} D(X))$ a *reduced domain* of the constraint. Note, that the reduced domain consists only of the tuples $(v_1,\ldots,v_n)$ such that $\forall i\ v_i \in D(X_i)$.

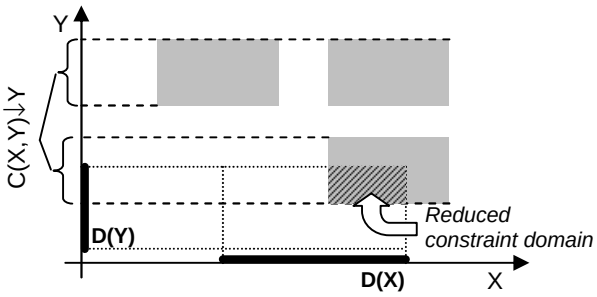


Fig. 3. Projection of the constraint domain (shadow rectangles) to variable Y and reducing the constraint domain.

Many constraint solvers are based on maintaining consistency of constraints during enumeration of variables. We say that the constraint is *consistent* (arc-consistent, hyper arc-consistent)[1] if every value of every variable participating in the constraint is part of some valuation satisfying the constraint. More precisely, every value of every variable participating in the constraint must be part of some tuple from the reduced constraint domain. For example, the constraint X+Y=2, where both the variables X and Y have domain {0,1,2}, is consistent while the constraint from Figure 3 is not

[1] 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 consistency there.

consistent. To make the constraint consistent we can reduce the domains of involved variables by projecting the reduced constraint domain to the variables:

$$\forall Y \in Xs\text{: } D(Y) \leftarrow (C(Xs) \cap (\times_{X \in Xs} D(X)))\downarrow Y.$$

The algorithm (procedure) that makes the constraint consistent is called a *propagator* [8]. More precisely, the propagator is a function that takes variables' domains as the input and that proposes a narrowing of these domains as the output. The propagator is *complete*, if it makes the constraint consistent, i.e., all locally incompatible values are removed. The propagator is *sound* if it does not remove any value that can be part of the solution. The propagator is *idempotent* if it reaches a fix point, i.e., the next application of the propagator to the narrowed domains does not narrow them more.

We say that the constraint satisfaction problem is (hyper) arc-consistent, if every constraint is consistent. It is not enough to make every constraint consistent by a single call to its (complete) propagator because the domain change might influence consistency of already consistent constraints. Thus the propagators are called repeatedly in a propagation loop until there is no domain change. In fact, the particular propagator is called only when domain of any variable involved in the constraint is changed (AC-8 algorithm). Many constraint systems allow a finer definition when the propagator is evoked via so called *suspensions*, for details see [7,8]. Nevertheless, the existing constraint solvers rarely go beyond the arc-consistency schema in the propagation loop.

When the domains of involved variables become singletons then it is not necessary to call the propagator again because it cannot narrow the domains anymore. Moreover, the propagator may be stopped sooner. Assume that the domain of X is {1,2,3} and the domain of Y is {5,6,7}. Then the propagator for the constraint X<Y deduces no domain narrowing. This is because every combination of values from the variables' domains satisfies the constraints - the constraint is entailed. We say that the constraint is *entailed* if the constraint is satisfied for any combination of the values from variables' domains. Visibly, the constraint is entailed if and only if the reduced constraint domain has a rectangular structure and the constraint is consistent.

The rest of the paper deals with propagators for general binary (tabular) constraints only. We expect the propagator to be evoked when the domain of any involved variable changes. Our goal is to design efficient, complete, and sound propagators.

Compact General Relation

The general relation (GR) constraint or more precisely the GR propagator was first described in [4]. This propagator uses a thread representation of the constraint domain where one variable is selected as the leading variable and the other variable is dependent. The constraint domain is represented as a list of pairs (*x*,*dy*), where *x* is a value of the leading variable and *dy* is a range of compatible values of the dependent variables (Figure 4). This is a natural representation of the constraints that are described using a table like in Figure 1. This representation requires a finite projection of the constraint domain to the leading variable and a finitely representable projection to the dependent variable (e.g. a finite number of intervals).

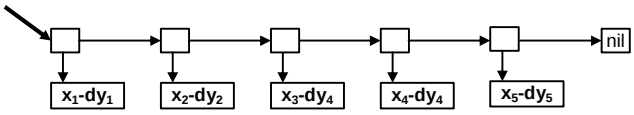


Fig. 4. Representation of the constraint domain by the GR propagator.

The filtering algorithm [4] simply explores the list and it tests whether x_i is part of the current domain of X and whether the intersection of dy_i with the current domain of Y is non-empty. In such a case x_i remains in the domain of X and $dy_i \cap D(Y)$ will be part of the narrowed domain of Y.

When using this algorithm with real-life constraints in a scheduling application [2], we have noticed that many dy_i are identical. Thus we can compact the domain representation which reduces memory consumption and it also speeds ups the filtering algorithm.

Domain Generator

Decomposition of the constraint domain to rectangles is used by sweep pruning algorithm proposed in [5] and it was applied to tabular constraint in [3]. We decided to go one step further and we decompose the domain into subsets with a rectangular structure rather than to individual rectangles. Note that such decomposition is more compact than a rectilinear rectangular covering because several rectangles can be represented in a single structure (see Figure 5). Moreover the decomposition algorithm is rather simply and efficient. We take the original description of the table using the list of pairs (x_i,dy_i) like in Figure 4 and we "compact" all the pairs with identical *dy* component. Formally, let $T = \{(x_i,dy_i) \mid i=1..n\}$ be the original table then we get a new table:

$$CT = \{(dx_i,dy_i) \mid dx_i = \{x \mid (x,dy_i) \in T\} \;\&\; j \neq k \Rightarrow dy_j \neq dy_k\}$$

We use a straightforward conversion algorithm with the time complexity $O(n.log\ n)$ where n is a number of the elements in the original table. This algorithm decomposes the constraint domain into a set of non-overlapping sub-domains with rectangular structure (like in Figure 5). Note that it is possible to design other conversion algorithms that produce different decompositions.

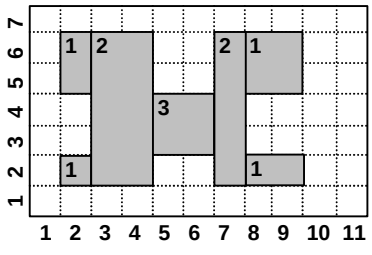


Fig. 5. A decomposition of the binary constraint domain into a set of non-overlapping sub-domains with a rectangular structure.

Filtering Algorithm

The filtering algorithm for the compacted GR relation mimics the behavior of the original filtering algorithm from [4]. There are just few changes to respect the new compact representation of the constraint domain. Figure 6 describes the final compact GR propagator. This algorithm incrementally constructs the projection of the reduced constraint domain to both variables. For each area DX-DY of the constraint domain, the algorithm checks whether it has a non-empty intersection with the reduced constraint domain (10-13); projections of this intersection become a part of the narrowed domains of the variables (17-18).

```
1  procedure GR(Constraint,X,Y)
2    NewDomainOfX ← empty
3    NewDomainOfY ← empty
4    ConstraintDomain ← domain(Constraint)
5    Entailed ← true
6    LastProjectionOfY ← empty
7    NewDomain ← empty
8    while non_empty(ConstraintDomain) do                    Domain filtering
9     (DX-DY) ← head(ConstraintDomain)
10    CompatibleX ← intersection(domain(X),DX)
11    if non_empty(CompatibleX) then
12     CompatibleY ← intersection(domain(Y),DY)
13     if non_empty(CompatibleY) then
14      if empty(NewDomain) then                             Domain shift
15       NewDomain ← ConstraintDomain
16      end if
17      NewDomainOfX ← union(NewDomainOfX,CompatibleX)
18      NewDomainOfY ← union(NewDomainOfY, CompatibleY)
19      if Entailed then                                     Entailment detector
20       if empty(LastProjectionOfY) then
21        LastProjectionOfY ← CompatibleY
22       else
23        Entailed ← LastProjectionOfY == CompatibleY)
24       end if
25      end if
26     end if
27    end if
28    ConstraintDomain ← tail(ConstraintDomain)
29   end while
30   X in NewDomainOfX
31   Y in NewDomainOfY
32   domain(Constraint) ← NewDomain
33 end GR
```

Fig. 6. The filtering algorithm of the compact GR propagator.

Time complexity of the compact GR propagator depends on the number of areas in the representation of a constraint domain. Each time the propagator is evoked, every such area is explored (8-29) and thus having a smaller number of areas in the domain representation is advantage. That is the reason why the compact GR propagator is more time and space efficient then the original GR propagator.

The proposed algorithm can keep only the areas that have a non-empty intersection with the reduced constraint domain. As we showed in [4] this significantly increases memory consumption so we decided just to shift the pointer to the start of the domain representation (14-16). This technique has minimal memory demands and it slightly decreases the number of areas to be explored when the propagator is evoked next time.

Usually, the research papers on filtering algorithms omit the implementation details like the detection of constraint entailment. We have found early detection of constraint entailment very important for the actual efficiency of the algorithm (see Experimental Results) and therefore we include the entailment detector in the code of the propagator (19-24).

Sweep Filtering Algorithm

The GR propagator uses a straightforward decomposition of the constraint domain to non-overlapping areas with a rectangular structure. Moreover, the projections of these areas to the leading variable are disjunctive (see Figure 5) which has no effect on the filtering algorithm but it simplifies detection of constraint entailment. In [9,11] a different decomposition of the constraint is proposed, in particular a decomposition into a set of rectangles covering the constraint domain (so called rectilinear rectangular covering). The filtering algorithm for such decomposition is based on a technique called sweep that is widely used in computational geometry and that was first applied to domain filtering in [5]. The sweep algorithm moves a vertical line (called a sweep line) along the horizontal axis (the leading variable) from left to right. Each time it encounters or leaves a rectangle (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 this paper, we propose a generalization of the filtering algorithm from [3]. It is based on observation that the sweep filtering algorithm can use more general objects than simple rectangles. The algorithm requires the object to have a rectangular structure and its projection to the leading variable to be an interval. We call such object a generalized rectangle (Figure 7).

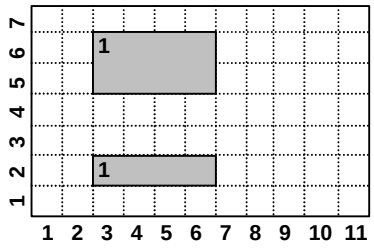


Fig. 7. Example of a generalized rectangle; it can be described using the term rect(3,6,[2,5..6]).

Domain Generator

Each constraint domain must be first decomposed into a set of generalized rectangles before it can be used by the sweep pruning algorithm. It is easy to get a sequence of non-overlapping generalized rectangles from the original table $T = \{(x_i,dy_i) \mid i=1..n\}$ describing the constraint domain. Simply, the neighboring pairs with the identical *dy* component are joined so we get a table:

$$CT_{sweep} = \{(min\_x_i, max\_x_i, dy_i) \mid \forall x\ min\_x_i \le x \le min\_x_i: (x,dy_i) \in T\}.$$

Notice the difference from the compact GR model: now the projection of objects in CT_{sweep} to the leading variable must be an interval and thus $|CT| \le |CT_{sweep}|$. Because the efficiency of the filtering algorithm depends on the number of generalized rectangles, we decided to generate a more compact decomposition from CT_{sweep}. The decomposition algorithm simply joins the neighboring parts of the rectangles. The idea is as follows: the algorithm takes the generalized rectangle and it tries to extend it to the largest possible *x*. Then this generalized rectangle is removed from the constraint domain and the process is repeated until the domain is empty (Figure 8).

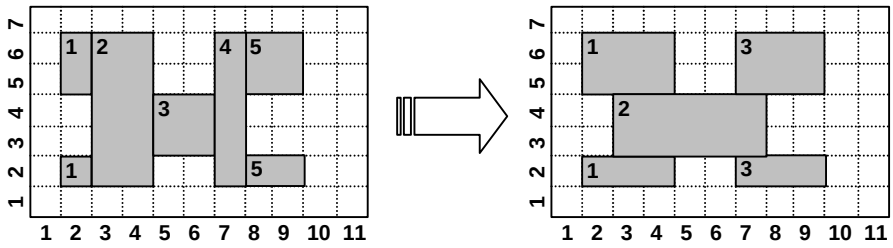


Fig. 8. Number of generalized rectangles can be decreased by using a different decomposition.

We present here a decomposition algorithm based on the sweep technique (Figure 9). This algorithm explores the (generalized) rectangles from CT_{sweep} from left to right and it tries to extend each rectangle to the right. To do this job, the algorithm keeps a set of active rectangles (*ActiveRects*), i.e., the rectangles that can still be extended, as well as an "active" projection of these rectangles to the dependent variable (*ActiveDy*). Note that there is an empty intersection of the projections of the active rectangles to the dependent variable. Thus if the active rectangle is closed then we can simply remove its projection from the "active" projection (line 12). Each time the algorithm takes a new rectangle, it tests whether the active rectangles can still be extended to this new rectangle (line 9). If the rectangle cannot be extended then it is removed from the set of active rectangles and it is put to the final decomposition - we call it closing the rectangle (12-13). After extending the active rectangles, the remaining part of the new rectangle (if any) will be included among the active rectangles (17-20). When all the rectangles are explored then the remaining active rectangles are closed (23-25).

In the worst case, the number of rectangles generated by this algorithm will be $|CT_{sweep}|$ but in many cases, the algorithm decreases the number of rectangles (see Figure 8). Note also that our decomposition generates non-overlapping rectangles.

```
1    procedure GenerateRectangles(D)
2     Rects ← empty
3     ActiveRects ← empty
4     ActiveDy ← empty
5     LastX ← inf
6     for each (Xmin..Xmax)-Dy in D (in increasing order of Xmin) do
7       TmpRects ← empty
8       for each r(RXmin,RDy) in ActiveRects do                    Rectangle
9         if RDy ⊆ Dy && LastX+1=Xmin then                         extension
10             TmpRects ← r(RXmin,RDy) : TmpRects
11        else
12             ActiveDy ← ActiveDy - RDy
13             Rects ← rect(RXmin,LastX,RDy) : Rects
14        end if
15      end for
16      ActiveRects ← TmpRects
17      if not empty_domain(Dy – ActiveDy) then
18        ActiveRects ← r(Xmin, Dy – ActiveDy) : ActiveRects       New
19        ActiveDy ← Dy                                            rectangle
20      end if
21      LastX ← Xmax
22    end for
23    for each r(Xmin,Dy) in ActiveRects do
24      Rects ← rect(Xmin,LastX,Dy) : Rects
25    end for
26   end GenerateRectangles
```

Fig. 9. The algorithm for domain decomposition

Rectangles	LastX	ActiveDy	ActiveRects	Rects
	inf	empty	empty	empty
(2..2)-[2,5..6]	2	[2,5..6]	r(2,[2,5..6])	empty
(3..4)-[2..6]	4	[2..6]	r(3,[3..4]), r(2,[2,5..6])	empty
(5..6)-[3..4]	6	[3..4]	r(3,[3..4])	rect(2,4,[2,5..6])
(7..7)-[2..6]	7	[2..6]	r(7,[2,5..6]), r(3,[3..4])	rect(2,4,[2,5..6])
(8..9)-[2,5..6]	9	[2,5..6]	r(7,[2,5..6])	rect(3,7,[3..4]), rect(2,4,[2,5..6])
				rect(7,9,[2,5..6]), rect(3,7,[3..4]), rect(2,4,[2,5..6])

Fig. 10. Run of GenerateRectangles with the constraint domain from Figure 8.

Filtering Algorithm

The filtering algorithm based on the sweep technique was proposed in [3,9] - using the generalized rectangles does not require any change of this algorithm. 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:

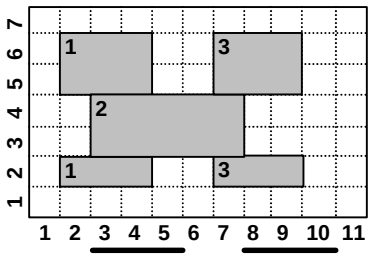
rect_start(PosX,NumR,IntY) - indicates the left border (*PosX*) of the rectangle identified by *NumR* with the vertical projection *IntY*,

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 11 shows an example of the event point series for the constraint domain consisting of three generalized rectangles and the domain of the leading variable consisting of two intervals (3..5 and 8..10).



Event point series:
rect_start(2,1,[2,5..6]),
rect_start(3,2,[3..4]),
x_start(3),
rect_end(4,1),
x_end(5),
rect_start(7,3,[2,5..6),
rect_end(7,2),
x_start(8),
rect_end(9,3),
x_end(10)

Fig. 11. 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.

```
1  procedure SP(Constraint,X,Y)
2    EventPointSeries ← make_event_point_series(Constraint,X)
3    ListOfDomY, ActiveRects, ListOfX ← empty
4    DY ← domain(Y)
5    InDomain ← false
6    while non_empty(EventPointSeries) do
7       Event ← select_and_delete_first(EventPointSeries)
8       process_event(Event,DY,ActiveRects,InDomain,ListOfX,ListOfDomY)
9    end while
10   NewDomainOfX ← empty
11   while non_empty(ListOfX) do
12      Max ← select_and_delete_last(ListOfX)
13      Min ← select_and_delete_last(ListOfX)
14      NewDomainOfX ← union(Min..Max,NewDomainOfX)
15   end while
16   X in NewDomainOfX
17   Y in intersection(union(ListOfDomY),DY)
18    Entailed ← all elements in ListOfDomY are identical
19 end SP
```

Fig. 12. The SP filtering algorithm.

EVENT - ACTION
rect_start(PosX,NumR,IntY) 20 if non_empty(intersection(IntY,DY)) then 21 if InDomain then 22 ListOfDomY ← IntY : ListOfDomY 23 if empty(ActiveRects) then 24 ListOfX ← PosX : ListOfX 25 end if 26 end if 27 ActiveRects ← r(NumR,IntY) : ActiveRects 28 end if
rect_end(PosXx,NumR) 29 if find_and_delete(r(NumR,_),ActiveRects) then 30 if InDomain && empty(ActiveRects) then 31 ListOfX ← PosX : ListOfX 32 end if 33 end if
x_start(PosX) 34 InDomain ← true 35 if non_empty(ActiveRects) then 36 ListOfX ← PosX : ListOfX 37 for each r(NumR,IntY) in ActiveRects do 38 ListOfDomY ← IntY : ListOfDomY 39 end for 40 end if
x_end(PosX) 41 InDomain ← false 42 if non_empty(ActiveRects) then 43 ListOfX ← PosX : ListOfX 44 end if

Fig. 13. Event processing for the SP filtering algorithm.

The SP algorithm is more or less self-explanatory (Figures 12, 13). Notice that only the rectangles having a non-empty projection to the domain of the dependent variable are processed (lines 20, 29); 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 (line 21), then the projection of the rectangle to y-axis is added to the new domain of Y (line 22). If it is the first rectangle that has non-empty intersection with the current interval of X (line 23) then the start of the new interval is added to the new domain of X (line 24). When entering the relevant rectangle we make this rectangle active by memorizing it in the *ActiveRects* structure (line 27). If we leave the last rectangle (*rect_end* event) that is active (line 30) then the end of the new interval is added to the new domain of X (line 31). If we enter a new interval within the domain of X (*x_start* event) and there is any active rectangle (line 35) then the new start of the new domain of X is created (line 36). Also, the new domain of Y is extended by projections of active rectangles to y-axis (37-39). 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 (line 43).

The algorithm incrementally builds new domains for the leading variable (*ListOfX*) and the dependent variable (*ListOfDomY*). *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 (10-15). *ListOfDomY* is a list of projections of the rectangles to the dependent variable. If all elements in this list are identical then the constraint is entailed (line 18). This detector is a simple improvement of the algorithm from [3] but it has a significant impact on the real-time efficiency of the propagator. Nevertheless, the detector is still not complete, i.e., it does not detect all constraint entailments.

EVENT	**ListOfX**	**InDom.**	**ActiveRects**	**NewDY**
`rect_start(2,1,[2,5..6])`	`empty`	`false`	`r(1,[2.5..6])`	`empty`
`rect_start(3,2,[3..4])`	`empty`	`false`	`r(2,[3..4])` `r(1,[2,5..6])`	`empty`
`x_start(3)`	`3`	`true`	`r(2,[3..4])` `r(1,[2,5..6])`	`[2,5..6]` `[3..4]`
`rect_end(4,1)`	`3`	`true`	`r(2,[3..4])`	`[2,5..6]` `[3..4]`
`x_end(5)`	`5,3`	`false`	`r(2,[3..4])`	`[2,5..6]` `[3..4]`
`rect_start(7,3,[2,5..6])`	`5,3`	`false`	`r(3,[2,5..6])` `r(2,[3..4]`	`[2,5..6]` `[3..4]`
`rect_end(7,2)`	`5,3`	`false`	`r(3,[2,5..6])`	`[2,5..6]` `[3..4]`
`x_start(8)`	`8,5,3`	`true`	`r(3,[2,5..6])`	`[2,5..6]` `[3..4]`
`rect_end(9,3)`	`9,8,5,3`	`true`	`empty`	`[2,5..6]` `[3..4]`
`x_end(10)`	`9,8,5,3`	`false`	`empty`	`[2,5..6]` `[3..4]`

Fig. 14. Run of the SP filtering algorithm for the constraint domain from Fig 11. The constraint is not entailed and the domains are narrowed to DX=[3..5,8..9], DY=[2..6].

Experimental Results

Time complexity of both filtering algorithms depends on the number of "rectangles" in the constraint domain; the complexity study can be found in [3,4]. We concentrate here on the real-life efficiency of the algorithms studied using several large-scale scheduling problems containing many tabular constraints. The algorithms are implemented in SICStus Prolog 3.8.7 [7,10] and the tests are run under Windows XP Professional on 1.7 GHz Mobile Pentium-M 4 with 768 MB RAM.

Table 1 summarizes the results on six tests problems. For each problem, we describe the number of tables inputted by the user and the total number of tabular constraints generated when solving the problem (some constraints share the same table). For each propagator we describe the total size of the tables, i.e., the total number of "rectangles" in all tables. This parameter indicates how good domain compaction the algorithm uses (the theoretical efficiency depends on this parameter). We also specify how many times the propagator is called. Naturally, this parameter cannot be included in the theoretical study of the individual propagators because it depends on the constraint "neighborhood". However, as the tests show early detection of constraint entailment influences the total running time. We compare both GR and SP propagators with and without detection of the constraint entailment.

Table 1. Comparison of GR and SP propagators using real-life scheduling problems (times are measured in seconds).

constraints		GR					SP				
tables	#	rect.	detect on		detect off		rect.	detect on		detect off	
			calls	time	calls	time		calls	time	calls	time
112	7568	117	7958	15	31051	15	356	7958	15	31051	16
135	12095	205	18253	95	242660	98	2112	30436	95	242660	107
244	15172	293	15198	27	51070	28	878	15983	27	51070	27.5
158	5742	367	16394	35	53405	36	6404	18782	42	53405	45
49	1993	151	3517	3	9263	3	1546	3553	4	9263	4,5
401	16985	455	18500	66	221586	72	3532	18500	67	221586	80

Before we start to analyze the results, it is necessary to highlight that the running times include complete solving of the problem so the actual running time of the propagator is just a fraction of this time. Therefore even a small improvement of the total running time actually means a significant improvement of the comparator.

The empirical study shows that even if the GR propagator is based on a rather straightforward idea it still outperforms the more advanced SP propagator. Time complexity of both comparators depends on the number of "rectangles" in the domain representation. As we can see, the decomposition for SP is never smaller then the decomposition for GR. That is because GR uses more general rectangles than SP so the results are not really surprising. Note also that both filtering algorithms are designed more or less independently on the actual decomposition so it is possible to improve the real-time efficiency by using more advanced decompositions leading to a smaller number of rectangles in the constraint domain.

The empirical study also confirms our claim that early detection of constraint entailment influences (in the positive way) actual running time. We can see that the number of calls to the propagator is significantly smaller when the entailment detector is on; also the running time is smaller. We can also see that the entailment detector for SP propagator is not complete so the propagator is called more times than for GR (where the entailment detector is complete).

Conclusions

The paper proposes and compares two approaches to domain filtering for binary tabular constraints. It also shows that some "implementation" details like detection of constraint entailment influences significantly real performance. We concentrate on real-life constraints rather than on artificial constraints over small domains.

The future research can go in the direction of designing better decompositions of the constraint domain and reducing the number of compatibility checks during filtering, e.g. using information about the cause of calling the propagator like in [6].

References

1. Barták, R.: Modelling Resource Transitions in Constraint-Based Scheduling. In W.I. Grosky, F. Plášil (eds.): Proceedings of SOFSEM 2002: Theory and Practice of Informatics, LNCS 2540, Springer Verlag (2002), pp. 186-194.
2. Barták, R.: Visopt ShopFloor: On the edge of planning and scheduling. In P. van Hentenryck (ed.) Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming (CP 2002), LNCS 2470, Springer Verlag (2002), pp. 587-602.
3. Barták, R.: Filtering Algorithms for Tabular Constraints. In Proceedings of Colloquium on Implementation of Constraint and Logic Programming Systems (CICLOPS), Paphos (2001), pp. 168-182.
4. Barták, R.: A General Relation Constraint: An Implementation. In Proceedings of CP2000 Post-Workshop on Techniques for Implementing Constraint Programming Systems (TRICS), Singapore (2000), pp. 30-40.
5. Beldiceanu N.: Sweep as a generic pruning technique. In Proceedings of CP2000 Workshop on Techniques for Implementing Constraint Programming Systems (TRICS), Singapore (2000), pp. 1-15.
6. Bessière Ch. and Régin J.-Ch.: Refining the Basic Constraint Propagation Algorithm. In Proceedings of JFPLC'2001 (2001).
7. Carlsson M., Ottosson G., Carlsson B.: An Open-Ended Finite Domain Constraint Solver. In Proceedings Programming Languages: Implementations, Logics, and Programs (1997)
8. Carlsson, M., and Schulte, Ch.: Finite-Domain Constraint Programming Systems. Tutorial at CP 2002.
9. Michalský, R.: Algorithms for Constraint Satisfaction, Master Thesis, Charles University, Prague (2001).
10. SICStus Prolog 3.8.7 User's Manual.
11. 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).