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. It means that 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 constraints that we call tabular constraints. In particular, we propose compact representations of extensionally defined binary constraints and we describe filtering algorithms for them. We concentrate on the 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) 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 and logical formulas like comparison or implication. However, the intentional description of some constraints is rather complicated and it is more convenient to describe them extensionally as a set of compatible tuples. A relation between the type of activity and its duration or the allowed transitions between the activities are typical examples of such constraints in the scheduling applications [1,2]. The constraint domain is specified there as a table of compatible tuples rather than as a formula, thus we are speaking about tabular constraints. Figure 1 shows an example of such a tabular constraint.

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.

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

In this paper we study the filtering algorithms (propagators) for binary constraints where the constraint domain is described as a table. We propose two new filtering algorithms that use a compact representation of the constraint domain. Such a compact representation turned out to be crucial for the efficiency of algorithms applied to real problems with non-trivial domains containing thousands or millions of values. Rather than providing a full theoretical study of the algorithms we concentrate on the practical aspects of the implementation, which are usually omitted in research papers. In particular, the algorithms are proposed in such a way that they can be easily integrated into existing 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 new filtering algorithms for tabular constraints. These algorithms extend our previous works on tabular constraints [3,4] by including better entailment detection and by using a more compact representation of the constraint domain. We also propose the algorithms for converting tables to the compact representation. We conclude the paper by an empirical study of the algorithms.

Motivation

Our work on filtering algorithms for tabular constraints is motivated by practical problems where important real-life constraints are defined 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 rather complicated relations between the activities expressing, for example, transitions between two 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 activity is required for the transition between two activities of different groups. The most natural way to express such a relation is using a table describing the allowed transitions (Figure 2).

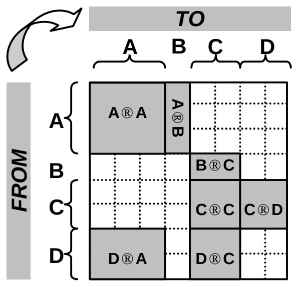


Fig. 2. A transition constraint expressed as a table of compatible transitions (shadow regions) between eight activities grouped into four groups A, B, C, and D.

As we showed in [2], there are many other constraints of the above type in real-life planning and scheduling problems, for example a description of the time windows, duration, and the cost of the activity. The users often define such constraints in the form of a table describing the set of compatible pairs. Therefore, we are speaking about *tabular constraints*. Because it is rather complicated to convert such a table into a mathematical formula defining a constraint with efficient propagation, it is more convenient to use special filtering algorithms handling the tabular constraints directly. Efficiency of such filtering algorithms can be improved by exploiting information about the typical structure of the constraint domain in a given application. For example, we have noticed that the structure of many above mentioned binary tabular constraints consists of several possible overlapping rectangles (see Figure 2) and the number of such rectangles is much smaller than the number of compatible pairs.

Related Works

Arc consistency (AC) or, in other words, propagation over binary constraints is studied for a long time and many AC algorithms have been proposed. Since 1986, we have the AC-4 algorithm [16] with an optimal worst-case time complexity. The average time complexity of this algorithm has been improved in its followers like AC-6 [6] and AC-7 [7]. All these algorithms are fine grained in the sense that they are working with individual pairs of compatible values – they use so called value based constraint graphs. Moreover, these algorithms need "heavy" data structures to minimize the number of consistency checks. These features complicate implementation and make the algorithms impractical due to a space complexity when large domains of variables are used. Therefore, an older but a simpler algorithm AC-3 [14] is used more widely in the constraint packages like SICStus Prolog, ECLiPSe, ILOG Solver, CHIP, Mozart etc. Actually, a variant of this algorithm that is called AC-8 [12] is used by these systems. AC-8 uses a list of variables with the changed domain instead of the list of constraints to be revised that is used by AC-3.

Recently, the new algorithms AC-3.1 [20] and AC-2000/2001 [8] based on the AC-3 schema have been proposed. AC-3.1 and AC-2001 achieve an optimal worst-case time complexity without using heavy data structures. However, they still require the data structures for individual values in the variables' domains which could complicate their usage for very large domains due to a space complexity.

In this paper, we concentrate on filtering algorithms for extensionally defined binary constraints over large domains. The proposed filtering algorithms are intended for existing constraint solvers so these algorithms must fit in the AC-3 (AC-8) schema. To achieve a good time and space complexity of the algorithms, we are exploiting the structure of the constraint domain. There exist several works about AC algorithms exploiting the structure of the constraint domain. For example, the generic AC-5 algorithm [19] achieves better time efficiency for functional, monotonic, and anti-functional constraints. The paper [11] describes a technique for converting the extensionally represented constraints into a set of simple constraints.

The existing constraint solvers usually provide a mechanism to model extensionally defined constraints without necessity to program a new filtering algorithm. For example, the `element` constraint is often included in the constraint solvers. *N* such `element` constraints can model arbitrary *N*-ary constraint. However, because every consistent tuple must be specified there, it is not possible to represent the constraints with infinite domains like the constraint from Figure 1.

In SICStus Prolog [17], there is a `relation` constraint where the user may specify a binary constraint as a list of compatible pairs similar to the table from Figure 1. In particular, for each value of the first (leading) variable, the user describes a range of compatible values of the second (dependent) variable. The range is a finite set of disjoint intervals. The domain for the leading variable must be finite and till the version 3.8.7 the range for the dependent variable must consist of finite intervals only.

The latest versions of SICStus Prolog (since 3.10.0) provide a generalization of the `relation` constraint. This new constraint called `case` allows compact modeling of arbitrary N-ary relations similar to our models. We compare empirically our filtering algorithms both to `relation` and `case` constraints later in the paper. Unfortunately, the filtering algorithms behind the `relation` and `case` constraints are not published which prevents a deeper comparison of the techniques.

In [4] 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 [3]. Later in the paper we describe an efficient extension to this algorithm that uses a more compact representation of the constraint domain.

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. 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 that is from a set of possible values for the variable. *Constraint* is an arbitrary relation restricting the possible combinations of values for the constrained variables. The *constraint domain* is a set of value tuples satisfying the constraint. 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 ordered set of variables Xs then we denote the 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 (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 domain structure of binary constraints where the constraint domain forms a rectangle with possible vertical and horizontal gaps.

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 $\forall i\ v_i \in D(X_i)$.

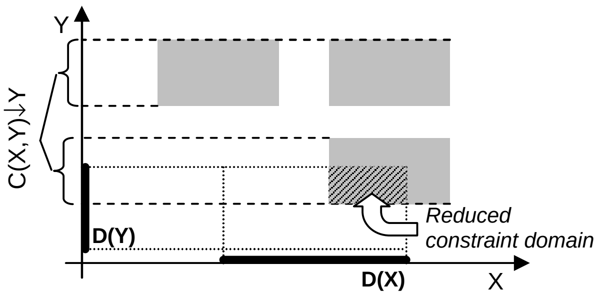


Fig. 3. Example of a constraint domain (shadow rectangles), its projection to the variable Y (C(X,Y)↓Y), and a reduced constraint domain (the striped rectangle).

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

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

The algorithm that makes the constraint consistent is called a *propagator* [10]. 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 that is 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 that is 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 the domain of any variable involved in the constraint is changed. Many constraint systems allow a finer definition when the propagator should be evoked via so called *suspensions*, for details see [9,10].

[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.

Nevertheless, the existing constraint solvers rarely go beyond the arc-consistency schema in the propagation loop.

When the domains of all variables in the constraint contain exactly one value then it is not necessary to call the propagator again because it will not narrow the domains anymore. However, the propagator may be stopped even 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 values from variables' domains. Visibly, the constraint is entailed if and only if the reduced constraint domain has a rectangular structure.

The rest of the paper deals with the propagators for extensionally defined binary constraints over totally ordered domains. We expect the propagator to be evoked when the domain of any variable involved in the constraint is changed. Our goal is to design efficient, complete, idempotent, 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 set 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 set of pairs (*x*,*dy*), where *x* is a value of the leading variable and *dy* is a set of compatible values of the dependent variables (Figure 4). The values of the leading variable are pair-wise different. 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, for example a finite set of disjoint intervals that we call a *range*. Notice that this representation also covers some infinite constraint domains.

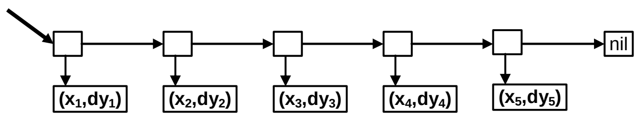


Fig. 4. Representation of the constraint domain by the GR propagator.

The filtering algorithm proposed in [4] simply explores the set representing the constraint domain and tests whether x_i is a 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 a part of the narrowed domain of Y.

When using the above 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 to reduce memory consumption and to speed-up the filtering algorithm.

Domain Generator

Let T = {(x_i,dy_i) | i=1..n} be a representation of the binary constraint domain where x_i are pair-wise different values of the leading variable and dy_i is a range of values of the dependent variable that are compatible with the value x_i. Such representation can be derived directly from the table defining the constraint. Because the time complexity of the above sketched GR propagator depends on the size of T, it could be beneficial to compact the representation and to upgrade the GR propagator for such a compact representation. In particular, it is possible to compact all pairs (x_i,dy_i) with identical dy_i component. Formally, for the original set T we get a new compacted set:

$$CT = \{(dx_i,dy_i) \ | \ dx_i = \{ x \mid (x,dy_i) \in T \} \ \& \ dx_i \neq \emptyset \}$$

We use a straightforward algorithm that converts T into CT with the time complexity O(*n.log n*), where *n* is a number of the elements in the set T. First, the algorithm orders lexicographically the set T according to dy_iThen, the pairs (dx_i,dy_i) with the identical component dy_i form continuous sub-sequences in the ordered set and thus it is easy to collect them in a linear time. Figure 5 shows an example of such a compact representation.

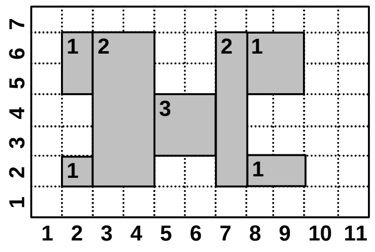


Fig. 5. A decomposition of the constraint domain (shadow rectangles) into a set of non-overlapping sub-domains with the rectangular structure.

Notice that the pair (*dx*,*dy*) in CT describes an area with a rectangular structure in the constraint domain and all these areas are pair-wise disjoint. Actually, the constraint domain is decomposed into a set of areas with a rectangular structure. This simplifies the filtering algorithm that can handle each such area independently as we will show in the next section. In general, the proposed filtering algorithm requires the areas to have a rectangular structure but it does not require them to be disjoint. Thus, we can see there a possibility to design other decompositions of the constraint domain that are perhaps even more compact.

On the other hand, the proposed filtering algorithm includes an entailment detector that requires the *dx* components to be disjoint (CT has this feature). Consequently, the areas are disjoint as well. Thus, if this particular entailment detector is used (and we will show later that it brings some speed-up) then CT is the optimal decomposition[2]. The open question is whether it is possible to design efficient entailment detectors that do not require the above feature.

[2] CT has the smallest number of rectangular areas such that their union equals to the constraint domain and their projections to the leading variable are disjoint.

Filtering Algorithm

The filtering algorithm for the compacted GR relation mimics the behavior of the original filtering algorithm from [4] that we described above. There are just few changes to respect the new compact representation of the constraint domain. Figure 6 describes the new compact GR propagator.

The compact GR propagator incrementally constructs the projection of the reduced constraint domain to both variables by exploring the areas in the compact representation of the constraint domain. For each area `(DX,DY)`, the propagator checks whether the area has a non-empty intersection with the reduced constraint domain (rows 10-13). Actually, the propagator constructs a *reduced area* `(CompatibleX,CompatibleY)` and the projections of this reduced area to both variables become parts of the narrowed domains of the variables (rows 17-18, 30-31).

```
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
9      (DX,DY) ← head(ConstraintDomain)                            filtering
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
15           NewDomain ← ConstraintDomain                          shift
16         end if
17         NewDomainOfX ← union(NewDomainOfX, CompatibleX)
18         NewDomainOfY ← union(NewDomainOfY, CompatibleY)
19         if Entailed then                                        Entailment
20           if empty(LastProjectionOfY) then                      detector
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.

The propagator might change the representation of the constraint domain to keep only the reduced constraint domain. This would help when the propagator is called next time because a smaller number of smaller areas will be explored which would speed-up the propagator. However, many constraint solvers including the solvers in Prolog keep the domains in memory after any change to allow fast recovery of the domain upon backtracking [10]. The paper [4] showed that it significantly increases memory consumption for a simple GR propagator that updates the constraint domain. So instead of keeping the reduced constraint domain, a technique called *domain shift* has been proposed in [4] to keep only a part of the reduced constraint domain. The set modeling the constraint domain is represented as a list there and domain shift means skipping the areas at the beginning of the list that are not part of the reduced constraint domain. This reduces a bit the size of the constraint domain (the number of areas to be explored when the propagator is called next time) while keeping low memory consumption. The compact GR propagator uses the same technique (rows 14-16, 32).

Last but not least, we have accompanied the compact GR propagator by an *entailment detector*. The entailment detector checks whether the reduced constraint domain has a rectangular structure. Because the projections of the areas to the leading variable are disjoint (this is a feature of CT), the entailment detection is done simply by comparing the projections of the non-empty reduced areas to the dependent variable (rows 19-24). If all these projections are identical then the constraint is entailed so it is not necessary to evoke the propagator again because it will not deduce any domain pruning. Note, that the research papers usually omit the implementation details like entailment detection. However, entailment detection may improve the time efficiency as we will show later.

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 (rows 8-29) and thus having a smaller number of areas in the domain representation is an advantage. That is the reason why the compact GR propagator is more time and space efficient then the original GR propagator.

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 disjoint (Figure 5) which has no effect on the filtering algorithm but it simplifies detection of constraint entailment. In [15] a different decomposition of the constraint is proposed, in particular the decomposition into a set of rectangles covering the constraint domain, so called rectilinear rectangular covering [18]. The filtering algorithm for such decomposition is based on the 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 axis of the leading variable. 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 algorithm 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 an object a *generalized rectangle* (Figure 7). In the next section, we will present a new algorithm constructing the domain representation with generalized rectangles from the original table modeling the constraint domain.

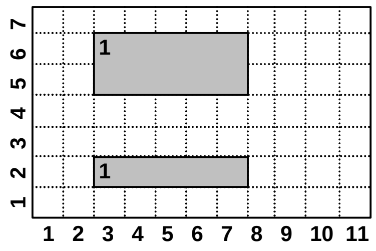


Fig. 7. Example of a generalized rectangle. This rectangle can be represented using the term `rect(3,7,[2,5..6])`.

Domain Generator

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

$$CT_{sweep} = \{(min_i .. max_i, dy_i) \mid min_i \le max_i \ \&\ \forall x\ min_i \le x \le max_i: (x,dy_i) \in T\}.$$

Notice the difference from the compact GR model; now the projection of an object in CT_{sweep} to the leading variable is 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).

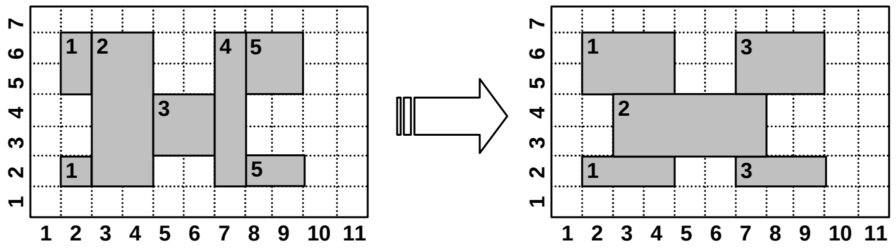


Fig. 8. The number of generalized rectangles covering the constraint domain can be decreased by using a different decomposition of the constraint domain.

We present here a decomposition algorithm based on the sweep technique (Figure 9). The set CT_{sweep} is ordered increasingly in the values *min_i*. Then, the decomposition algorithm explores the generalized rectangles from the ordered set CT_{sweep} and it tries to extend each rectangle to the right. To do this job, the algorithm keeps a set of the rectangles that can be extended, so called active rectangles (*ActiveRects*), as well as an "active" projection of these rectangles to the dependent variable (*ActiveDy*). The projections of the currently active rectangles to the dependent variable are disjoint. Thus, if an active rectangle is closed (see below) then we can simply remove its projection from the "active" projection (row 12). Each time the algorithm takes a new rectangle, it tests whether the active rectangles can still be extended to this new rectangle (row 9). If an active rectangle cannot be extended then it is removed from the set of active rectangles and it is put to the final decomposition (*Rects*) - we call it closing the rectangle (rows 12-13). After extending all the active rectangles, the remaining part of the new rectangle (if any) will be included among the active rectangles (rows 17-20). When all the rectangles are explored then the remaining active rectangles are closed (rows 23-25).

```
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 non_empty(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.

In the worst case, the number of rectangles generated by the above algorithm will be $|CT_{sweep}|$. However, the algorithm decreases the number of rectangles in many cases (see Figure 8). Note also that the presented decomposition algorithm generates non-overlapping rectangles. Figure 10 demonstrates the run of the algorithm.

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. Example run of `GenerateRectangles` with the constraint domain from Figure 8.

Filtering Algorithm

The filtering algorithm based on the sweep technique was proposed in [15, 3] and the same sweep pruning algorithm can be used for generalized rectangles without any modification. The sweep pruning (SP) algorithm moves a vertical line called a *sweep line* along the horizontal axis from left to right. Each time it encounters or leaves an object – this is called an *event* – it triggers a relevant event handler. 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*,

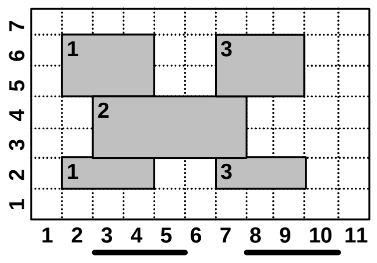
rect_end(PosX,NumR) - indicates the right border (*PosX*) of the rectangle identified by *NumR*,

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

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

The list of events can be generated in advance from the constraint domain and from 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-coordinate of the event (*PosX*). Moreover, we require the start events to precede the end events with the same x-coordinate. 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.

The SP algorithm incrementally builds the new domains for both variables by exploring the generalized rectangles (Figure 12). *ListOfX* keeps a list of bounds of the intervals in the new domain of the leading variable (in the reverse order). This list is then converted to the domain of the dependent variable (rows 10-15). *ListOfDomY* is a list of projections of the rectangles, which have non-empty intersection with the reduced constraint domain, to the dependent variable.



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. A constraint domain (left) and its corresponding event point series (right),

During the computation, the SP 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 that is between *x_start* and *x_end* events corresponding to a single continuous interval,

ActiveRects: describes the set of rectangles that are currently crossed by the sweep line that is the rectangles where the *rect_start* event has been processed and the corresponding *rect_end* has not been reached yet.

We have added a simple entailment detector (row 18) to the SP algorithm. If all the rectangle projections in *ListOfDomY* are identical then the constraint is entailed. Visibly, this entailment detector is not complete because it does not detect all constraint entailments. For example, assume the constraint domain from Figure 11, the domain of the leading variable to be {4,7}, and the domain of the dependent variable to be (3..5). Then the constraint is entailed but the algorithm does not detect it because the projections of the rectangles 1 and 2 are not identical.

```
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 filtering algorithm of the SP propagator.

The power behind the SP algorithm is hidden in the procedures for event processing (Figure 13). Notice that only the rectangles having a non-empty projection to the domain of the dependent variable are processed (rows 20, 29). Let us call these *rectangles relevant.*

If the sweep line enters a relevant rectangle (*rec_start* event) and it is within the domain of the leading variable X (row 21), then the projection of the rectangle to y-axis is added to the new domain of Y (row 22). If it is the first rectangle that has a non-empty intersection with the current interval of X (row 23) then the start of the new interval is added to *ListOfX* (row 24). When entering the relevant rectangle we make this rectangle active by memorizing it in the *ActiveRects* structure (row 27).

If we leave a rectangle (*rect_end* event) that is the last active rectangle and the sweep line is within the domain of X (row 30) then the end of a new interval is added to *ListOfX* (row 31).

If we enter a new interval within the domain of X (*x_start* event) and there is any active rectangle (row 35) then the start of a new interval is added to *ListOfX* (row 36). Also, the new domain of Y is extended by the projections of the active rectangles to y-axis (rows 37-39).

If we leave an interval within the domain of X (*x_end* event) and there is still some active rectangle then the end of a new interval is added to *ListOfX* (row 43).

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. The algorithms for event processing for the SP propagator.

Figure 14 gives an example of event processing. It describes how the data are changed after processing the events from the event point series.

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

Experiments and Comparison

We have compared the GR and SP propagators to existing `relation` and `case` constraints in SICStus Prolog 3.11.0 [9, 17]. The comparison was done using real-life scheduling problems solved by the Visopt ShopFloor system [2] and using a new artificial benchmark. The tests run under Windows XP Professional on 1.7 GHz Mobile Pentium-M 4 with 768 MB RAM and the running time is measured via the `statistics` predicate with the `walltime` parameter [17].

The GR and SP propagators are compared to `relation` and `case` constraints. The propagators behind these constraints maintain full arc consistency like the GR and SP propagators. The constraint domain in the `relation` constraint is described using a simple table T = $\{(x_i,dy_i) \mid i=1..n\}$, where x_i are pair-wise different values of the leading variable and dy_i is a range of values of the dependent variable compatible with the value x_i. Since the version 3.10.0 of SICStus Prolog, the `relation` constraint is implemented using a more general `case` constraint which, like our approach, allows more compact representation of the constraint domain. We use a table CT = $\{(dx_i,dy_i) \mid dx_i = \{x \mid (x,dy_i)\in \mathrm{T}\ \}\ \&\ dx_i \neq \emptyset\ \}$ to describe the constraint domain for the `case` constraint – the actual representation contains one more element in the list representing CT, for details of syntax see [17]. This is exactly the same table used by the GR propagator. Note finally, that the `case` constraint is implemented in C while our propagators are implemented in Prolog.

Real-life experiments in Visopt ShopFloor

Visopt ShopFloor [2] is a scheduling system where the user describes declaratively the resources, item flows, and the demands and the system generates a schedule of production. Because of its generic character, the system uses many tabular constraints to describe the real-life relations like the time windows and the resource state transitions [1]. Originally, the GR propagator was designed for this system to capture a typical structure of constraint domains that appear there.

We have selected five different problems based on real-life factories to demonstrate capabilities of the proposed propagators. These problems vary in the size and the structure of the factories – the actual data are confidential so it is not possible to publish them. Table 1 describes the size of the problems as a number of different constraint domains (tables), a number of tabular constraints using these domains, and an average size of the constraint domain representation. The size of the representation is measured as an average number of rectangles per table for GR and SP propagators and as an average length of the lists describing the domains in the `relation` and `case` constraints.

Table 1. The size of the test problems and constraint representations.

problem no.	tables	constraints		average representation size per table			
		total	per table	GR	SP	relation	case
1	401	16977	42	1.13	4.40	20.76	2.13
2	49	1921	39	3.08	15.78	39.57	4.08
3	158	5734	36	2.32	20.27	44.82	3.32
4	244	82804	339	1.20	1.80	3.60	2.10
5	112	7624	68	1.04	1.59	3.65	2.04

Notice, that many constraints share the same domain, for example more than three hundred constraints share the domain (in average) in the problem no 4. This domain sharing decreases memory consumption. Moreover, the domain generator for both GR and SP propagators runs once per table which decreases running time. Notice also that the domain representation is very compact for GR and SP propagators. The only exception is the domain representation for the SP propagator in problems 2 and 3. A compact representation further reduces the running time because the time complexity of GR and SP propagators depends on the number of rectangles in the representation [3,4]. As expected, the representation for the GR propagator is more compact than the representation for the SP propagator.

Table 2 compares the number of calls to GR and SP propagators with and without the entailment detector (this information is not available for the built-in `relation` and `case` constraints). When the entailment detector is not used (off) then the number of calls is identical for both GR and SP because this number is influenced by the environment, where the algorithm sits, not by the algorithm itself. When the entailment detector is used (on) then the number of calls is much smaller. Notice also that for the problems 2, 3, and 4, the number of calls to SP is larger than the number of calls to GR. This indicates that the entailment detector for SP is not complete.

Table 2. The number of calls to the propagators. The number in brackets indicates the number of calls relative to the number of calls when the entailment detector is used (on).

problem no.	GR		SP	
	on	off	on	off
1	18515	117328 (634%)	18515	117328 (634%)
2	3372	9062 (269%)	3436	9062 (264%)
3	16688	56413 (338%)	19072	56413 (296%)
4	82830	145654 (176%)	86265	145654 (169%)
5	8014	32301 (403%)	8014	32301 (403%)

Finally, Table 3 compares the running times of the algorithms. The average time of five runs for each problem is indicated in the table. Note also, that we compare a total running time to solve the problem including propagation in all constraints as well as search. Thus, the actual running time of the compared algorithms is just a fraction of the presented time. Still, only the compared algorithms are responsible for the difference in the running time so the relative time difference between the algorithms is higher than it might seem from Table 3. We decided for this test because it shows better what speed-up/slow-down one may expect in a complex system.

Table 3. The running time (in seconds) of the propagators. The numbers in brackets show time relative to the GR propagator with entailment detector (in percent).

problem no.	GR		SP		relation	case
	on	off	on	off		
1	82,0	81,7 (100%)	81,5 (99%)	88,8 (108%)	91,5 (112%)	89,0 (108%)
2	3,6	3,7 (102%)	4,6 (127%)	5,0 (138%)	4,2 (117%)	4,1 (114%)
3	48,6	53,8 (111%)	57,8 (119%)	66,6 (137%)	66,6 (137%)	62,2 (128%)
4	86,1	87,8 (102%)	88,4 (103%)	90,4 (105%)	96,2 (112%)	94,6 (110%)
5	26,5	26,3 (99%)	26,4 (100%)	27,7 (105%)	35,0 (132%)	37,9 (143%)

The GR propagator with entailment detector achieved the best results in the tests. It has the smallest running time for the problems 2, 3, and 4 and its running time is very close to the best results in tests 1 (SP on) and 5 (GR off).

The behavior of the SP propagator with entailment detector was less stable. In particular, it achieved much worse running time than GR in tests 2 and 3. However, recall that the representation of the constraint domain is less compact for SP than for GR (Table 1) and better domain generator may further improve time efficiency of SP. Moreover, the SP propagator uses simpler elementary operations than GR which is the reason why SP is better in some tests despite the fact that it has a larger domain representation than GR.

The tests also show that entailment detection pays off especially for the SP propagator where entailment detection brings almost no overheads. The entailment detector for GR is more complex and it adds more overhead to computation. That is the reason why the GR propagator without entailment detection may get slightly better results in some problems.

Notice finally, that the built-in `relation` and `case` constraints implemented in C achieved worse results than GR and SP (with the exception of problem 2 for SP) implemented in Prolog. The reason could be that GR is designed for tabular constraints which have almost rectangular structure. However, note that we use the same compact representation for the `case` constraint.

The real-life tests justified the choice of the GR propagator with entailment detector in the Visopt ShopFloor system. However, they did not uncover the general features of the new propagators like the relation between the compaction factor and the running time. Therefore, we proposed an artificial benchmark to test the comparators independently of a particular application.

Artificial benchmarks

Artificial benchmarks help to understand general features of the algorithms without a direct relation to a particular application. Random CSP [13] is a widely accepted set of benchmarks for testing constraint satisfaction algorithms. However, the disadvantage of Random CSP is that there is no direct relation between the parameters of the random problem, like density and tightness, and the structure of the constraint domain. Because time and space complexity of the propagators studied in this paper depends strongly on the structure of the constraint domain, we decided to design a new random benchmark where the number of rectangles in the constraint domain can be controlled by a parameter.

The basic idea of the proposed benchmark is to use a single binary constraint with a randomly generated domain. The size of domains for the variables is a parameter of the benchmark; we decided for the size 10 000 because we studied propagators for large domains. For each value of the leading variable, we randomly generate an interval of the compatible values for the dependent variable. The length of this interval is another parameter of the benchmark; we tested lengths from 1000 to 9000 with the step 1000. Note, that the length of the interval is identical for each value of the leading variable, only the position of the interval is generated randomly. Thus, the larger interval implies a smaller number of possible positions which further implies a smaller number of rectangles in the constraint domain. Figure 15 shows the relation between the length of the interval and the size of the domain representation – the number of rectangles. For each length we randomly generated ten constraint domains and we present the average results over these constraint domains.

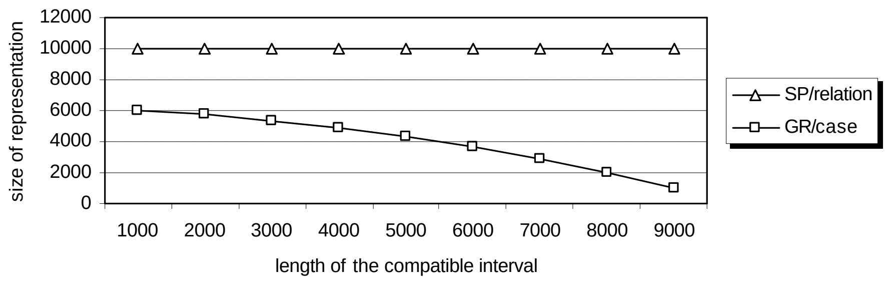


Fig. 15. A size of the domain representation for GR and SP propagators.

As we can see from Figure 15, the number of rectangles for the GR propagator is continuously decreasing with the increasing length of the interval. Unfortunately, the domain generator for the SP propagator does not compact the domain at all so we cannot expect good efficiency of the SP propagator in the following tests.

When the constraint domain is generated, the question is how to evoke the propagator. In the AC-3 (AC-8) schema, the propagator is evoked when the domain of any variable in the constraint is changed. So, we randomly prune the domains of the variables until one of these domains contains exactly one value. Then the constraint is entailed. The leading and dependent variables alternate in this process to suppress the role of the variable. By using this technique, we can measure the time spent only in the filtering algorithm. In particular, such benchmark abstracts from the complexity of the constraint satisfaction problem and the size of the search space is irrelevant here. Still, the question how to prune the variables' domains remains unanswered. We have tested three different schemas of domain pruning: domain splitting, arbitrary deletions, and shaving.

Domain splitting. In domain splitting, the variable's domain is randomly split into two parts and one of these parts is pruned. In particular, we generate a random value called a cutting point between the lower and upper bound of the variable domain and we randomly decide whether to cut the lower or upper part of the domain with respect to the cutting point. Figure 16 shows the running time (in milliseconds) as a function of the length of the compatible interval.

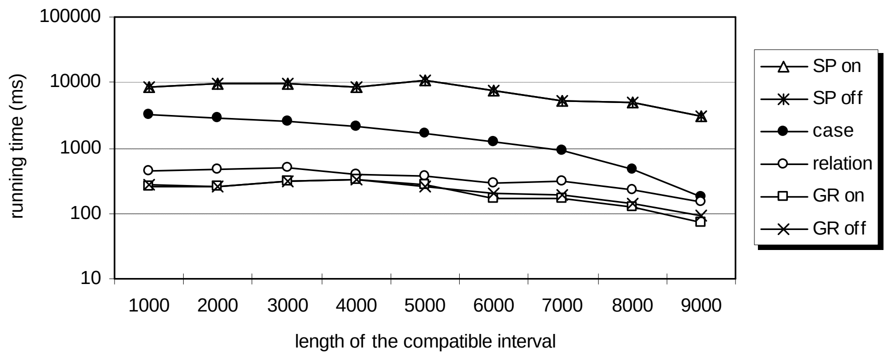


Fig. 16. The running time (a logarithmic scale in milliseconds) as a function of the length of the compatible interval for domain splitting.

For domain splitting, the GR propagator is the fastest propagator among the tested algorithms. It is about two times faster than the `relation` constraint (note, that we use a logarithmic scale in Figure 16). We can also see that entailment detection for GR pays off when the domain is more compacted. Surprisingly, the `case` constraint is not as good, even if it uses the same domain representation as the GR propagator. Nevertheless, the `case` constraint is becoming more efficient when the domain representation is more compact. The SP propagator does not behave well, probably because of the large domain representation.

Arbitrary deletions. In many constraint satisfaction problems, the values are deleted from all over the domain. To capture this situation, we randomly generate a given number of values in the variable's domain and, then, we remove these values together from the domain. The number of values for deletion is given in percent of the current size of the domain. We tested four scenarios with 5%, 10%, 20%, and 40% of values deleted together from the domain. Figure 17 shows the running time (in milliseconds) as a function of the length of the compatible interval for all these scenarios.

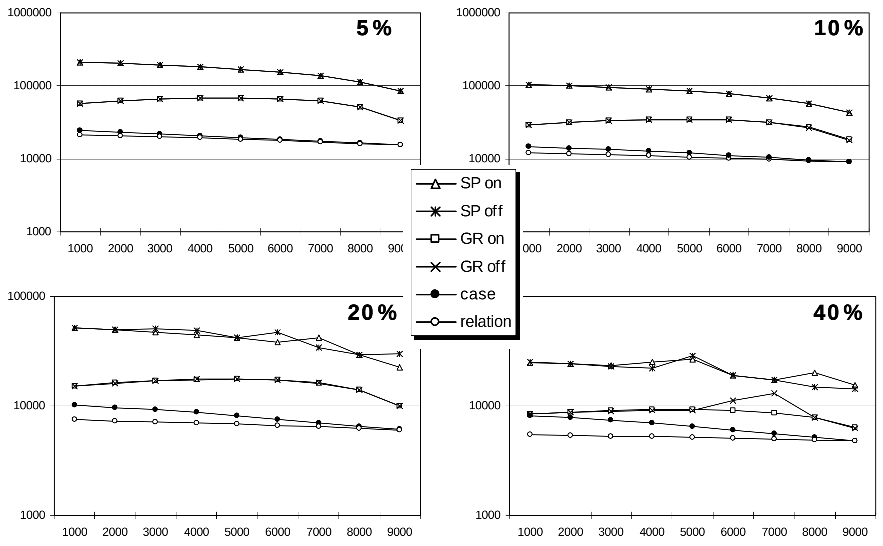


Fig. 17. The running time (a logarithmic scale in milliseconds) as a function of the length of the compatible interval for arbitrary deletions. A given percent of randomly selected values is deleted from the domain.

For arbitrary deletions, the `relation` constraint is the best followed by the `case` constraint. Again, for more compacted representation, the `case` constraint is closer to the `relation` constraint. The GR propagator did not behave very well in these tests. The reason could be that the compact representation of the GR propagator works better when intervals of values are pruned rather than when individual values are deleted from domains. Notice also that the more values are removed together, the closer the GR propagator is to the `relation` and `case` constraints. Entailment detection has almost no effect here.

Shaving. In some applications, like scheduling, the domains of variables are pruned in a specific way. In particular, upper or lower parts of the domains are pruned which we call shaving. This technique is close to domain splitting but we can now control better the number of deleted values. In particular, we shave a given percent of the domain, namely 5%, 10%, 20%, and 40%. The choice whether to shave upper or lower part of the domain is done randomly. Figure 18 shows the running time (in milliseconds) as a function of the length of the compatible interval.

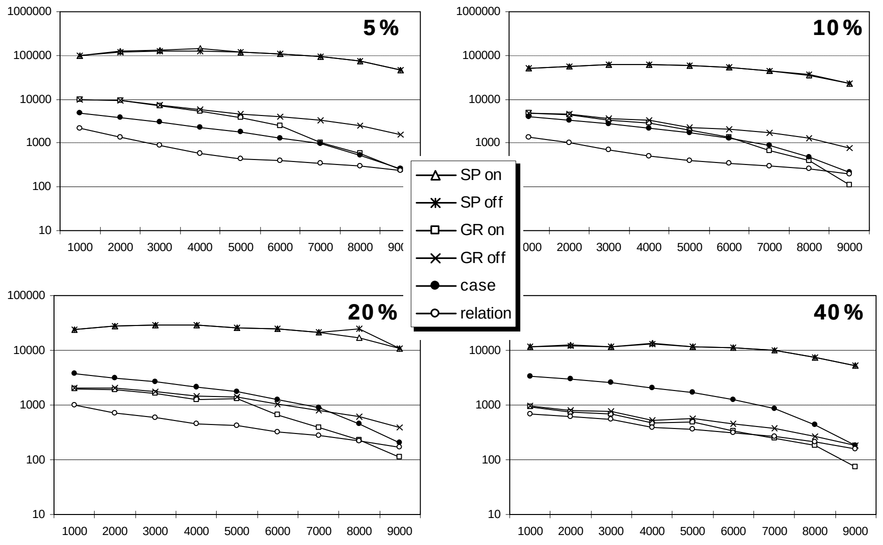


Fig. 18. The running time (a logarithmic scale in milliseconds) as a function of the length of the compatible interval for shaving. A given percent of values is shaved from the domain.

The power of the GR propagator is visible in the tests with shaving. Actually, the GR propagator outperforms the `relation` constraint when the constraint domain is more compacted and when more values are shaved. This trend is even stronger when we compare the GR propagator with the `case` constraint. Finally, notice that the entailment detector for GR also pays off significantly in these cases.

Conclusions

The paper proposed and compared two approaches to domain filtering for extensionally defined binary constraints, namely GR and SP propagators. Both approaches are based on the idea of compact representation of the constraint domain as a set of rectangles. They differ in the structure of these rectangles and in the way how this structure is explored during filtering. We presented the filtering algorithms as well as the algorithms for constructing a domain representation. We also described entailment detection mechanism and we showed that it improves real performance of the propagators. The experimental comparison showed that efficiency of the proposed propagators depends strongly on the size of domain representation. Thus, the future research may go in the direction of designing smaller decompositions of the constraint domain especially for the SP propagator. Also, both propagators explore the constraint domain completely after variable’s domain change which penalizes them when just a few values are deleted. It could be interesting to reduce the number of compatibility checks during filtering, for example using information about the cause of calling the propagator like in [8, 20].

References

1. Barták, R.: Modelling Resource Transitions in Constraint-Based Scheduling. In Grosky W.I., Plášil F. (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 Van Hentenryck P. (ed.): Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming, 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., Carlsson M.: Sweep as a generic pruning technique applied to the non-overlapping rectangles constraint. In Walsh T. (ed.): Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming, LNCS 2239, Springer Verlag (2001), pp. 377-391.
6. Bessière Ch.: Arc-consistency and arc-consistency again. Artificial Intelligence 65 (1994), pp. 179-190.
7. Bessière Ch., Freuder E.C., and Régin J.-Ch.: Using constraint metaknowledge to reduce arc consistency computation. Artificial Intelligence 107 (1999), pp. 125-148.
8. Bessière Ch. and Régin J.-Ch.: Refining the Basic Constraint Propagation Algorithm. In Proceedings of JFPLC‘2001 (2001).
9. Carlsson M., Ottosson G., Carlsson B.: An Open-Ended Finite Domain Constraint Solver. In Proceedings Programming Languages: Implementations, Logics, and Programs (1997).
10. Carlsson, M., and Schulte, Ch.: Finite-Domain Constraint Programming Systems. Tutorial at CP 2002.
11. Cheng K.C.K., Lee J.H.M., and Stuckey P.J.: Box constraint collections for adhoc constraints. In Rossi F. (ed.): Proceedings of the 9th International Conference on Principles and Practices of Constraint Programming, LNCS, Springer-Verlag (2003).
12. Chmeiss A., Jégou P.: Efficient Path-Consistency Propagation. International Journal of Artificial Intelligence Tools 2 (1998), pp. 121-142.
13. Gent I.P., MacIntyre E., Prosser P., Smith B.M., and Walsh T.: Random constraint satisfaction: Flaws and structure. Technical Report APES-08-1998, APES Research Group (1998).
14. Mackworth, A.K.: Consistency in Networks of Relations. Artificial Intelligence 8 (1977), pp. 99-118.
15. Michalský, R.: Algorithms for Constraint Satisfaction, Master Thesis, Charles University, Prague (2001).
16. Mohr R., Henderson T.C.: Arc and Path Consistency Revised. Artificial Intelligence 28 (1986), pp. 225-233.
17. SICStus Prolog 3.11.0 User's Manual, SICS (2003).
18. 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).
19. Van Hentenryck P., Deville Y., and Teng C.-M.: A generic arc-consistency algorithm and its specializations. Artificial Intelligence 57 (1992), pp. 291-321.
20. Zhang Y., Yap R.: Making AC-3 an Optimal Algorithm. In Proceedings of IJCAI-01 (2001), pp. 316-321.