Propagating Deletions in Tabular Constraints

Roman Barták*

Charles University, Faculty of Mathematics and Physics
Institute for Theoretical Computer Science
Malostranské nám. 2/25, Prague, Czech Republic
bartak@ktiml.mff.cuni.cz

Abstract. In the paper we propose a new filtering algorithm for extensionally defined binary constraints – so called tabular constraints. The algorithm combines a compact representation of the constraint domain with the principles of AC-3.1 and AC-2001 algorithms. We concentrate on the practical issues like covering large real-life constraints and integration to existing constraint solvers. The experimental results show a significant speed-up over the existing models of extensionally-defined constraints.

Introduction

Constraint propagation is intensively studied by researchers because of its importance for reducing the search space when solving hard combinatorial problems. Among the constraint propagation techniques, arc consistency (AC) is probably the most studied technique and many arc consistency algorithms have already been proposed. Despite the existence of AC algorithms with optimal worst-case time complexity, namely AC-4 and its improvements AC-6 and AC-7, a simple AC-3 is usually preferred in existing constraints solvers like ILOG Solver, CHIP, ECLiPSe, or SICStus Prolog. The reason is a good practical efficiency of AC-3 and an easier integration of various filtering algorithms for individual constraints including non-binary constraints into the AC-3 schema.

Recently, two new versions of AC-3 algorithm, AC-3.1 [8] and AC-2001 [3], have been independently proposed to achieve the optimal worst-case time complexity without complex data structures typical for AC-4, AC-6, and AC-7. These algorithms are still fine grained so they need to keep additional information about individual values. However, this information is not communicated between the constraints so the proposed techniques can be more easily integrated into existing constraint solvers based on the AC-3 schema.

We are not aware that the above-mentioned integration of AC-3.1 or AC-2001 to existing constraint solvers has already been done so our paper is probably the first description of such integration. Moreover, we do not cover just the implementation of the existing algorithm; the paper describes a new filtering algorithm for compactly represented constraints. In particular, we are trying to overcome the main difficulty of

* Supported by the Czech Science Foundation under the contract No. 201/04/1102.

AC-3.1 and AC-2001 which is still their memory consumption. Note also that while AC-3.1/AC-2001 keep information about supporters for individual values, our algorithm keeps the same information in the description of the constraint domain. Because the constraint domain can be seen as a symmetrical representation of value supporters, we can see our algorithm as an extension of the AC-3.1 and AC-2001 towards AC-7 [2].

The theoretical research, as described above, usually sees the constraint in a general way that is the constraint is an arbitrary relation between the variables. In practice, it means an ad-hoc representation of the constraint domains which is memory and time expensive. Currently, there exist two techniques how to overcome the above difficulties of the ad-hoc representations: the first technique converts the extensional representation into an intentional one, the second technique compacts the extensional representation.

The paper [6] is a recent representative of the first technique. The propagation rules are automatically generated and expressed as indexicals [5] which has the advantage of good memory efficiency if the semantics of the constraint is “clear”. The disadvantage is a non-trivial pre-processing step which cannot be often done during runtime due to implementation issues. Moreover, the decomposition of the original constraint cannot exploit the advantages of optimal AC-3.1 and AC-2001 algorithms.

The paper [1] represents the second technique of a compact extensional representation of ad-hoc constraints using a set of rectangles. The presented approach is efficient when the (binary) constraint domain can be decomposed into a small number of rectangles. However, the filtering algorithms presented in [1] are less efficient when only few values are pruned from domains. We further extend the work [1] by proposing a new filtering algorithm that propagates value deletions rather than computing value supports from scratch.

To summarize our contribution, we present a new view of optimal AC-3.1 and AC-2001 algorithms based on a compact representation of the constraint domain. Thus, it is not necessary to work with individual value pairs and the filtering of constraint domains decomposable into a relatively small number of rectangles can be even more efficient.

The paper is organized as follows. We first introduce some notions describing the extensionally defined constraints and propagators for these constraints. Then we present a compact representation of the extensionally defined constraint domain that is adapted from [1]. The new contribution is in Section 3 where a new filtering algorithm for such domains is introduced and its soundness and completeness is proved. Finally, we present the experimental evaluation of the proposed algorithm showing that the new algorithm is significantly more efficient than the former approach from [1].

Preliminaries

We survey here the terminology introduced in [1] to describe formally the constraints and their consistency. *Constraint* is a relation restricting possible combinations of values for the constraint variables. *Constraint domain* is a set of tuples satisfying the constraint. If C denotes the constraint and Xs is an ordered set of the variables constrained by C then C(Xs) denotes the constraint domain. For example, if C is a constraint X+Y=2 over non-negative integers, then C({X,Y}) = {(0,2),(1,1),2,0)} is its constraint domain. 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. Notice that the (binary) constraint domain has a rectangular structure if the domain forms a rectangle with possible vertical and horizontal strips of removed value pairs, hence the name rectangular structure.

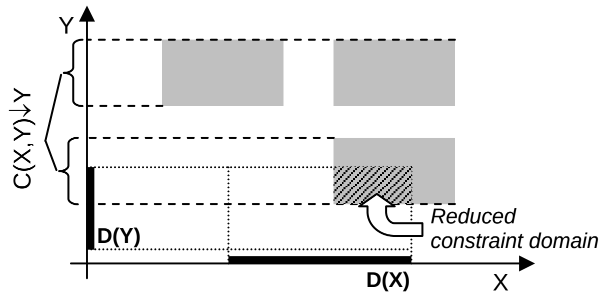

Fig. 1. A constraint domain (shadow rectangles), its projection to the variable Y (C({X,Y})↓Y), and a reduced constraint domain

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 (Figure 1). Note, that the reduced domain consists only of the tuples $(v_1,\ldots,v_n)$ such that $\forall i\ v_i \in D(X_i)$. We say that a constraint is *consistent* if every value of any variable constrained by C is a part of some tuple satisfying the constraint. Actually, the constraint is consistent in respect to the current domains of the constrained variables if the projections of the reduced domain to these variables are equal to the current domains of respective variables. Thus, it is possible to make the constraint consistent by projecting the reduced constraint domain to the constrained variables:

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

The algorithms attempting to make the constraint consistent by narrowing variables' domains are called *propagators*. 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 is a part of a tuple satisfying the constraint and consisting of values from the current variables' domains. 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.

When domain of any constraint variable is changed, the propagator is evoked by the constraint solver to make the constraint consistent or to check that the constraint is still consistent. By using this technique, derived from AC-3, it is possible to achieve a local consistency of the network of constraints (called generalized arc consistency). If the domains of the constraint variables become singleton then it is not necessary to call the propagator again. However, the propagator may be deactivated even sooner which improves the practical time efficiency of the solvers [1]. Assume that the domain of X is {1,2,3} and the domain of Y is {5,6,7}. Then a sound 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.

Compact Constraint Domains

When users specify a binary constraint domain, they usually use a table of compatible pairs. Typically, for a value of one variable called a *leading variable*, they specify a range of compatible values of the other variable called a *dependent variable*. Range is a finite set of disjoint intervals, for example {1..5, 8..15, 30..sup}. Such a table can be formally described as a set $T=\{(x_i,dy_i) \mid i=1..n\}$, where x_i are pair-wise different values of the leading variable and dy_i is a range of values of the dependent variable that are compatible with the value x_i. Paper [1] proposes a compact representation of the table T based on the observation that the ranges dy_i are often identical in real-life problems. Formally, the compacted set is defined as follows:

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

We call dx_i the *x*-component of (dx_i,dy_i) in CT and, similarly, dy_i is the *y*-component. Note that it is easy to obtain CT from T by collecting all elements of T with the identical *y*-component into a single element of CT. Figure 2 shows an example of such a compacted form.

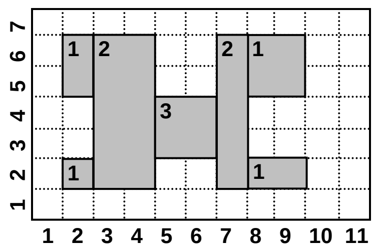

Fig. 2. Representation of the constraint domain using a set of non-overlapping areas with a rectangular structure

The set CT has some interesting features that can be exploited by the filtering algorithm. First, each element of CT describes an area with a rectangular structure. Hence we call the elements of CT *rectangles*. Second, the projections of these rectangles to the leading variable are pair-wise disjoint. Thus, we can see the original constraint as a disjunction of entailed constraints where the domains of these constraints are defined by the elements of CT.

Filtering Algorithm

The filtering algorithm proposed in [1] is basically a constructive disjunction of constraints with domains defined by the elements of CT. This algorithm, called GR (General Relation), computes the reduced constraint domain and its projection to both constrained variables. After any change of the variable domain, the algorithm does the above computation from scratch so it corresponds roughly to the REVISE procedure of AC-3. The main inefficiency behind this approach is that a lot work is done even if a small amount of values has been pruned. The paper [3] proposes a new approach, called AC-2000, based on idea of checking support just for the values that lost a support (a value compatible with a given value has been removed). We call this technique *propagation of deletions*. By using an additional data structure, it is possible to effectively check whether the value lost a support which leads to the worst-case time optimal algorithm called AC-2001. The same idea is independently presented in [8] under the name AC-3.1.

Based on the above observations we propose a new filtering algorithm for compactly represented extensional constraint domains. The algorithm propagates deletions by exploring only the values that lost some support. Instead of attaching a special data structure to each value, like AC-2001 and AC-3.1 do, we use the representation of the reduced constraint domain. Thus, in a single data structure we keep the supporters for both constrained variables so we can exploit the symmetry of the constraint in a similar sense like AC-7 [2] (if *a* supports *b* then *b* supports *a*).

The propagator is proposed in such a way that it can be easily integrated into existing constraint solvers, in particular we designed the algorithm for the `clpfd` library [4] of SICStus Prolog. We use a global data structure called a *state* [5] to pass information between the subsequent calls of the propagator. In particular, we keep the reduced constraint domain and the domains of both variables from the last call of the propagator in the state data structure of the propagator.

The algorithm is formally described in Figure 3 and Figure 4 illustrates its run. Before the propagator is evoked for the first time, the projections of the constraint domain CT to both variables are computed in the procedure **INIT**. These projections are assumed to be the initial domains of the constrained variables that are stored in the propagator's state. Then the propagator, realised in the function **FILTER**, is called explicitly to propagate the actual domains of the variables. We also expect the propagator to be called any time, when domain of any involved variable is changed. First, the propagator computes which values have been removed from the domain of the dependent variable – a set *DiffY* (line 2). Then, it checks which rectangles in CT are affected by this deletion (3-9). Actually, the values $y \in DiffY$ are removed from the *y*-components of the rectangles in CT (6). If the *y*-component of any rectangle becomes empty by this removal (7) then the *x*-component is removed from the domain of the leading variable X via *DelX* (10) and the rectangle is no more assumed to be an element of CT. This can be done because the *x*-components of the rectangles are disjoint so the values in the *x*-component of the removed rectangle lost their only support. A similar process is done for the leading variable (11-18). However, because the *y*-components of the rectangles are not necessarily disjoint so the values collected in *DelY* may still have another support in X, we cannot remove the values in *DelY* immediately. This additional support is looked for in the last loop (19-21). The values for which the support is not found there can be safely deleted (22). Notice also that the propagator computes the reduced domain of the constraint so this domain can be used in the subsequent calls. Actually, the structure CT keeps the updated information about the supporters.

Last but not least, the propagator is able to check the constraint entailment. If there is exactly one rectangle in CT (23) then the constraint is entailed. However, this entailment detector is not complete because CT may consist of more rectangles with identical *y*-components (see Figure 4). It is possible to extend the propagator to detect entailment completely but we think that it does not pay off here.

The above filtering algorithm can be further optimized during implementation. For example, the loop at lines 3-9 is processed only when $DiffY \neq \emptyset$. Similarly, the loop at lines 12-18 is processed only when $DiffX \neq \emptyset$. Finally, the loop at lines 19-21 can be safely exited when *DelY* becomes empty.

Theorem 1. *The proposed filtering algorithm is sound, complete, and idempotent.*

Proof: The proof is based on the observation that the propagator keeps the reduced constrained domain. If the propagator removes a value *a* from dom(X) then this value has no support in Y because the only rectangle containing pairs (*a*,*b*) for some *b* has been removed (7). Note that there is at most one such rectangle in CT for the value *a* because the *x*-components of the rectangles are disjoint. Similarly if *b* is deleted from dom(Y) then it lost a support in one rectangle that was deleted from the reduced constraint domain (16) and no support in another rectangle has been found (19-21). Hence, the propagator is sound.

The INIT procedure computes the projection of CT to both variables so at the beginning only locally consistent values are in the domains. Assume (for contradiction) that after finishing the propagator, there is an inconsistent value *a* in dom(X). Thus, there is no supporter of *a* in Y so there is no rectangle in CT containing a pair (*a*,*b*) for some *b*. Because originally the value *a* was locally consistent, the rectangle containing (*a*,*b*) must have been removed from CT during filtering. However, if the rectangle was removed then all values of its *x*-component have been removed as well (7). Similarly, if locally inconsistent value *b* remains in dom(Y) then there is no rectangle containing a pair (*a*,*b*) for some *a* in CT. The original rectangle containing (*a*,*b*) has been removed (16) and because there is no another rectangle in CT containing *b* in its *y*-coordinate (19-21), *b* has been removed as well (22). Hence, the propagator is complete.

If the repeated call to the algorithm narrows the domains then the newly removed values must be locally inconsistent due to soundness of the propagator. However, because the propagator is complete, all such values have already been removed. Hence the repeated call cannot narrow the domains and the propagator is idempotent.

❑

```
procedure INIT(X,Y, CT)
      DomX ← ∅
      DomY ← ∅
      for each (DX,DY) in CT do         // union the projections of all rectangles to X and Y
            DomX ← DomX ∪ DX
            DomY ← DomY ∪ DY
      end for
      dom(X) ← dom(X) ∩ DomX            // dom(X) is the actual domain of the variable X
      dom(Y) ← dom(Y) ∩ DomY            // dom(Y) is the actual domain of the variable Y
      call FILTER(X, Y, (DomX, DomY, CT))
end INIT

procedure FILTER(X,Y, State)
1     (OldDomX, OldDomY, CT) ← State
2     DiffY ← OldDomY – dom(Y)          // values deleted from Y since the last call to FILTER

3     DelX ← ∅
4     TmpCT ← ∅
5     for each (DX,DY) in CT do
6           RY ← DY – DiffY
7           if RY==∅ then DelX ← DelX ∪ DX     // values of X that lost support in Y
8           else TmpCT ← TmpCT ∪ {(DX,RY)}
9     end for
10    NewDomX ← dom(X) – DelX

11    DiffX ← OldDomX – dom(X) – DelX          // values deleted from X
12    DelY ← ∅
13    NewCT ← ∅
14    for each (DX,DY) in TmpCT do
15          RX ← DX – DiffX
16          if RX==∅ then DelY ← DelY ∪ DY     // values of Y that lost support in X
17          else NewCT ← NewCT ∪ {(RX,DY)}
18    end for

19    for each (DX,DY) in NewCT do             // try to find another support for DelY
20          DelY ← DelY – DY
21    end for

22    NewDomY ← dom(Y) – DelY
23    Entailed ← (|NewCT|==1)
24    State ← (NewDomX, NewDomY, NewCT)
25    dom(X) ← NewDomX
26    dom(Y) ← NewDomY
end FILTER
```

Fig. 3. Filtering algorithm GRA for propagating deletions

	dom(X)	***dom(Y)***	***CT***
after INIT	2..9	2..6	{ ({2,8,9},{2,5,6}), ({3,4,7},2..6), ({5,6},{3,4})}
deletion	2..6	5..6	
line 10	2..4	5..6	{({2,8,9},{5,6}), ({3,4,7},{5,6})}
line 22	2..4	5..6	{({2},{5,6}), ({3,4},{5,6})}

Fig. 4. Example of propagation for the constraint from Figure 2

Experiments And Discussion

We compare our algorithm with the original GR propagator with entailment detector from [1] and with the built-in `relation` and `case` constraints in SICStus Prolog. The GR propagator and our new filtering algorithm are implemented in Prolog and they both use an identical representation of the constraint domain – the set CT. The `relation` constraint is implemented by means of a more general `case` constraint which is implemented in C. We use the original table T to describe the domain for the `relation` constraint and the table CT to describe the domain for the `case` constraint. Note that both these constraints achieve the same so called domain consistency as our filtering algorithm and the GR propagator. Unfortunately, the filtering algorithms behind the `case` and `relation` constraints are not published so we can do just an empirical comparison without a deep analysis of the results. The tests run in SICStus Prolog 3.11.0 under Windows XP Professional on 1.7 GHz Mobile Pentium-M 4 with 768 MB RAM. The running time is measured in milliseconds via the `statistics` predicate with the `walltime` parameter [7].

To explore efficiency of the proposed algorithm, we use the set of abstract benchmarks proposed in [1]. The basic idea of these benchmarks is to apply domain pruning into a single randomly generated constraint until the domain of one of the constrained variables becomes singleton. The variables alternate in pruning to suppress the leading or dependent role of the variable.

The constraint domain is generated as follows. For each value of the leading variable an interval of compatible values of the dependent variables is generated. The length of this interval is identical for all the values and it is one of the parameters of the benchmark. Thus, only the position of the interval is introduced randomly. As Figure 5 shows, the size of the representation depends nicely on this parameter which is the main reason why we chose this approach rather than a completely random constraint domain. The other parameter of the benchmark is the size of the domain of the variables. We use the domain size 10 000 because we study the propagators for large domains. We tested all interval lengths between 1000 and 9000 with the step 1000 and for each length we generated ten problems. The presented results are average running times over these ten problems.

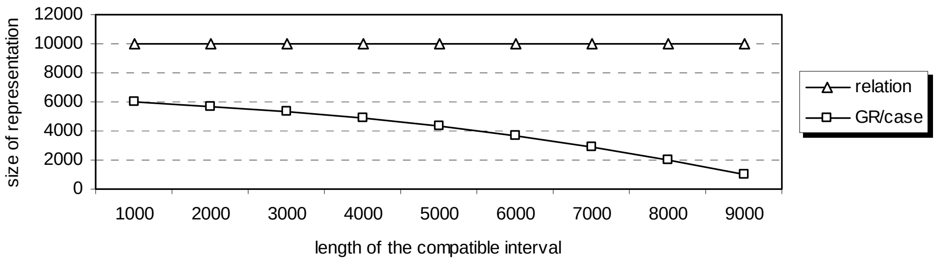

Fig. 5. Size of the constraint domain representation as a function of the length of interval used by the problem generator.

The experiments in [1] also showed that the efficiency of the propagator depends on the style of domain pruning and on the number of values deleted in a single pruning step. We present the comparison for two pruning styles, namely domain splitting and arbitrary deletions where the number of deleted values is chosen randomly. We also did some experiments where the number of deleted values is given as a parameter. These experiments confirmed our conclusions presented below.

Domain splitting

The domain splitting propagation style prunes the variable domain by splitting it into two parts and pruning one of them. This pruning style is used by some search procedures to decompose the search space. It is also known under the notion of shaving that is widely used in scheduling applications (during shaving, a part of the domain is deleted at the domain borders). In our experiments, we randomly generate a cutting point in between the current lower and upper bound of the domain and then we randomly prune the part above or below the cutting point. Figure 6 shows the running time of the propagators as a function of the interval length. We denote the new filtering algorithm as GRA (General Relation Advanced) there.

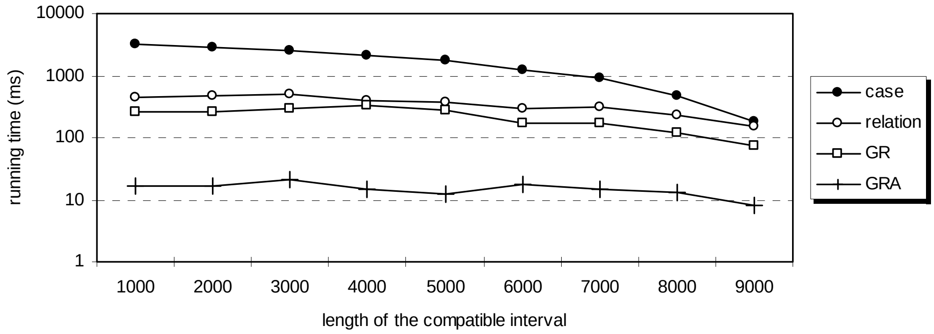

Fig. 6. The running time (a logarithmic scale in milliseconds) as a function of the interval length for domain splitting.

The running time of the GR propagator is comparable to the `relation` constraint. Actually, it is always slightly faster. Surprisingly, the `case` constraint is the worst propagator here despite the fact that it uses the same compact representation as the GR propagator. The hands-down winner here is our new filtering algorithm with a significant speed-up over all other propagators.

Our new filtering algorithm is designed to work best when a small number of values is deleted but the experiment shows that it works very well in average. We also did experiments where the size of the shaved area is given relatively to the size of the actual domain (in particular, 5%, 10%, 20%, 40%). While the GR propagator was faster than the `relation` and `case` constraints only when large portions of the domain were pruned together or when the domain was highly compacted [1], the new filtering algorithm was significantly faster in all the tests.

Arbitrary deletions

Probably the most typical pruning style in (random) problems is removing the values from all over the domain. To model such situation, we randomly select a random number of values from the variable domain and we prune all these values together. Figure 7 shows the comparison.

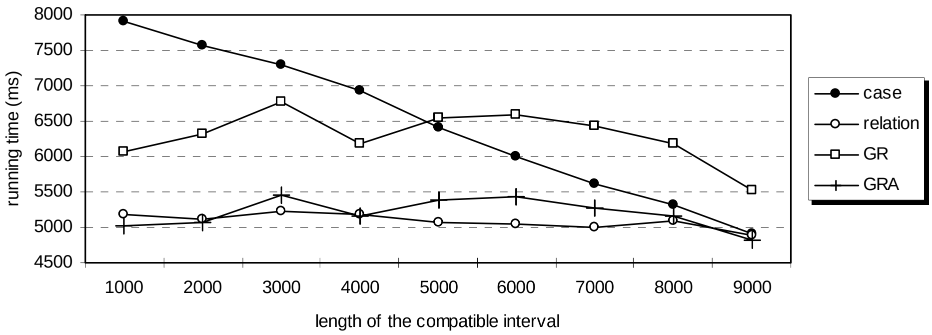

Fig. 7. The running time (in milliseconds) as a function of the interval length for arbitrary deletions.

The efficiency of the GR propagator is week in comparison to the `relation` constraint when random deletions are used. The hypothesis mentioned in [1] is that the reason is that the GR propagator is not designed for pruning individual values but for pruning large intervals of values. One of the motivations of our research was to confirm this hypothesis by redesigning the GR propagator to handle better deletions of individual values. As the above experiment showed, the runtime of the new filtering algorithm is now comparable to the `relation` constraint and it is faster than the `case` constraint (depending on the size of the constraint representation). The experiment also showed that the new filtering algorithm behaves similarly to the GR propagator (they have a similar efficiency curve in Figure 7) but the new filtering algorithm is about 20% faster. The experiments with a controlled number of deleted values (a given percent of variable domain is pruned, in particular 5%, 10%, 20%, 40%) showed that the gap between the GR propagator and the new propagator is enlarging when the number of deleted values is decreasing. Thus the original hypothesis seems correct. Moreover, we can see that taking care about the deleted values rather than looking for supporters of the values in domains pays-off even if large portions of the domain are pruned.

The presented results show that the new filtering algorithm is significantly better than the GR propagator in all the experiments. As expected, the speed-up is higher when a smaller number of values is deleted in a single step but there is a significant speed-up even in an average case. The new filtering algorithm is also much faster than the built-in `relation` and `case` constraints when domain splitting (shaving) is used and its speed is comparable to these constraints in the other experiment – arbitrary deletions. This is a quite good result if we take in account that the built-in constraints are implemented in a low-level C while our filtering algorithm is implemented in Prolog with no low-level optimisation. To confirm these good results, we did some preliminary experiments with random CSP where the constraint domain is made from random rectangles (to get a given tightness). In these experiments, our new filtering algorithm also beat all other tested propagators.

Conclusions

The paper presents a new filtering algorithm for compactly represented extensionally defined binary constraints. The algorithm is based on propagating deletions which makes it much more efficient than the existing GR propagator. Also the compact representation of the constraint domain makes the algorithm useful for very large domains, which differentiates the proposed approach from existing AC-3.1 and AC-2001 algorithms. Moreover, the algorithm can be naturally extended to n-ary constraints. Last but not least, rather than implementing the algorithm as a separate system, we developed the algorithm in such a way that it can be easily integrated into existing constraint solvers.

References

1. R. Barták and R. Mecl. Implementing Propagators for Tabular Constraints. In Recent Advances in Constraints. LNAI 3010, Springer-Verlag, 44-65, (2004).
2. Ch. Bessière, E.C. Freuder, and J.-Ch. Régin. Using constraint metaknowledge to reduce arc consistency computation. *Artificial Intelligence*, **107**, 125-148, (1999).
3. Ch. Bessière and J.-Ch. Régin. Refining the Basic Constraint Propagation Algorithm. In Proceedings of IJCAI-01, 309-315, (2001).
4. M. Carlsson, G. Ottosson, B. Carlsson. An Open-Ended Finite Domain Constraint Solver. In Proceedings Programming Languages: Implementations, Logics, and Programs, (1997).
5. M. Carlsson and Ch. Schulte. Finite-Domain Constraint Programming Systems. Tutorial at CP 2002, (2002).
   ftp://ftp.sics.se/pub/isl/papers/FD Systems.pdf
6. K.C.K. Cheng, J.H.M. Lee, and P.J. Stuckey. Box constraint collections for adhoc constraints. In F. Rossi (ed.): Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming, LNCS 2833, Springer-Verlag, (2003).
7. SICStus Prolog 3.11.0 User's Manual, SICS, (2003).
8. Y. Zhang and R. Yap. Making AC-3 an Optimal Algorithm. In Proceedings of IJCAI-01, 316-321, (2001).