Sparse-Sets for Domain Implementation

Vianney Le Clément de Saint-Marcq, Pierre Schaus, Christine Solnon, Christophe Lecoutre

Sparse-Sets for Domain Implementation

Vianney le Clément de Saint-Marcq[1,2,3], Pierre Schaus[1],
Christine Solnon[2,4], and Christophe Lecoutre[5]

[1] Université catholique de Louvain, ICTEAM institute,
Place Sainte-Barbe 2, 1348 Louvain-la-Neuve (Belgium)
{vianney.leclement,pierre.schaus}@uclouvain.be
[2] Université de Lyon, LIRIS, CNRS UMR5205, 69622 Villeurbanne (France)
christine.solnon@liris.cnrs.fr
[3] Université Lyon 1, F-69622 Villeurbanne (France)
[4] INSA Lyon, F-69622 Villeurbanne (France)
[5] Université Lille-Nord de France, Artois (France)
lecoutre@cril.fr

Abstract. This paper discusses the usage of sparse sets for integer domain implementation over traditional representations. A first benefit of sparse sets is that they are very cheap to trail and restore. A second key advantage introduced in this work is that sparse sets permit to get delta changes with a very limited cost, allowing efficient incremental propagation. Sparse sets can also be used to represent subset bound domains for set variables.

Keywords: Constraint Programming, Domains, Sparse-set, Incremental Filtering, Integer variable, Set variable, Subset bound

1 Introduction

The representation of variable domains is an important design choice when building a CP (Constraint Programming) solver. It can have a significant impact on the performances of the CP system. In this paper, we focus on finite domains and, without loss of generality, we assume that domains only contain positive integer values.

Possible domain representations are considered part of the CP folklore, thus not often clearly detailed or explained in the literature. Three popular representations of a finite domain $D \subseteq [1, N]$ are range sequences, bit vectors [15], and successor vectors [18]:

- The range sequence associated with D is the shortest sequence $\langle [a_1, b_1], \ldots, [a_k, b_k] \rangle$ such that D is covered, i.e., $D = \bigcup_{i=1}^{k} [a_i, b_i]$, and the ranges are ordered by their smallest elements, i.e., $a_i \leq a_{i+1}$ for $1 \leq i < k$. Clearly, a range sequence is unique, none of its ranges are empty, and $b_i + 1 < a_{i+1}$ for $1 \leq i < k$. Solvers using range sequences for domain implementation include Gecode [5].

- The bit vector associated with D is the string of N bits such that the i^{th} bit is 1 iff $i \in D$. Solvers using bit vectors for domain implementation include Choco [4].
- The successor vector [18] requires two vectors *pred* and *succ* allowing to access a successor of a value in the domain in constant time. This representation is further described in Section 1.5.1 of [8]. As for bit vector, it takes constant time to remove an arbitrary value in the domain and update the *pred* and *succ* vectors.

As reported in Table 1, those three representations have different space and time complexities. The space complexity of bit and successor vectors is linear with respect to N, whatever the number of values in D, whereas the space complexity of range sequences is linear with respect to the number of ranges k in D.[6]

Table 1. Complexity for a domain $D \subseteq [1, N]$ assuming a range sequence of length k implemented with a list. We assume that neither the bit vector nor the sparse set is augmented with explicit bounds. All rows except the last one are time complexities.

	Bit vect.	Successor vect.	Range seq.	Sparse Set
Restore Δ values	$\mathcal{O}(\Delta)$	$\Theta(\Delta)$	$\mathcal{O}(1)^*$	$\mathcal{O}(1)$
Value removal	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(k)$	$\mathcal{O}(1)$
Check value in D	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(k)$	$\mathcal{O}(1)$
Iterate	$\mathcal{O}(N)$	$\mathcal{O}(\lvert D\rvert)$	$\Theta(\lvert D\rvert)$	$\Theta(\lvert D\rvert)$
Iterate increasingly	$\mathcal{O}(N)$	$\Theta(\lvert D\rvert)$	$\Theta(\lvert D\rvert)$	$\mathcal{O}(sort(\lvert D\rvert))$
Get min/max	$\mathcal{O}(N)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\Theta(\lvert D\rvert)$
Space complexity	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(k)$	$\mathcal{O}(N)$

* Assuming an implementation with immutable linked lists, where a value removal builds a new list, possibly sharing a tail with the old list.

Therefore, range sequences may be more scalable than bit/successor vectors and successor for large sparse domains. Fortunately filtering algorithms rarely create holes in large domains. In such cases, algorithms updating the bounds of the domains are preferred (for instance, most filtering algorithms for scheduling constraints update bounds). Some CP systems maintain together an interval domain, which only stores the lower and upper bounds of D, and a bit vector

[6] Recently Pothitos and Stamatopoulos [11] have proposed a gap-based representation for domains, which may be viewed as the complementary of the range sequence representation: instead of memorizing all ranges (in a list or a binary tree), they propose to memorize all gaps between ranges in a binary tree (called a gap interval tree). The advantage is that the subtraction of a range of values is faster, as it affects only one tree node (i.e., it inserts or modifies only one node). However, the complexity of searching for, adding or deleting a value still depends on the number of gaps in the domain.

representation. In this case, the bit-vector is lazily created when a hole is created in the middle of D.

The three representations also induce different time complexities for classical domain operations. In particular, searching for, adding or deleting a value in a domain is done in constant time with bit and successor vectors, whereas the time complexity of these operations depends on the number of ranges when using range sequences: it is linear when using a list of ranges, and logarithmic when using a binary tree. As a counterpart, iterating over all values inside a domain D may be done in $\mathcal{O}(|D|)$ when using range sequences and successor vectors, whereas it is in $\mathcal{O}(N)$ when using bit vectors.[7]

The facility to restore domains when backtracking in the search tree is an important aspect to consider when choosing the domain representation. In this paper, we assume CP solvers based on trailing, the predominant approach.[8] A trail can be viewed as a stack containing the information to be restored upon backtracking to recover a previous state. In some cases (e.g., when storing the previous value of the entity being modified), only the first change per trailed entity within one search node must be restored upon backtracking [1]. We can rely on this observation to efficiently implement trail-based systems. A global time-stamp is incremented at the creation of each node of the search tree. Each trailed entity also contains an inner time-stamp corresponding to the last time it was added to the trail. We trail the entity value only if its time-stamp is outdated, i.e., it is not the same as the current global time-stamp. When backtracking to a parent node, each entry on the trail, up to the time-stamp of the parent node, is popped and restored.

While range sequences may be more scalable in memory for large domains with a few holes in the middle, it is also less trivial to restore them in a trail-based solver. Assuming a 32 bits system, a bit vector can be split into words of 32 bits. Each time a value is removed, the word corresponding to the flipped bit is trailed. Hence in the worst case, one could trail up to $N/32$ entries in a search node. A successor vector requires to trail each entry. Hence the number of trailed entities inside a search node is exactly equal to the number of removed values.

Contribution In this paper, we discuss an alternative domain representation based on sparse sets. This representation has the advantage that it only requires to trail at most one entity per domain for each node of the search tree and iterating over the values in a domain D is optimal (i.e., time complexity of $\Theta(|D|)$). As for the bit vectors or successor vectors, we can check in constant time if a value is present or not. Using sparse sets for representing domains is not new, as they are already used in some solvers since a few years.

What we believe is the most important contribution of the paper is the idea of instrumenting sparse set-based domain representation to provide at no

[7] In C, we may use the `ffs` function to get the position of the first bit set in the vector. The x86 CPU architecture implements such operation in hardware, but its use is however not guaranteed by the compiler.

[8] We refer to [14] for a comparison between trailing and copying mechanisms.

cost the delta changes to the constraints. This information is very important to achieve incremental filtering for some constraints. We also discuss how a single sparse set can be used to implement a set-variable domain with subset-bounds representation. To the best of our knowledge this can also be considered as a new contribution of the paper.

2 Sparse Sets for integer variable domains

Sparse sets were introduced by Briggs and Torczon in [2]. When using sparse sets, a domain D is represented by its size $size_D$ and two arrays dom_D and map_D. The dom_D array contains all values in the range $[1, N]$. The $size_D$ first elements of dom_D are considered to be part of D, the others have been removed (see Figure 1). The map_D array maps values to their position in the dom_D array.

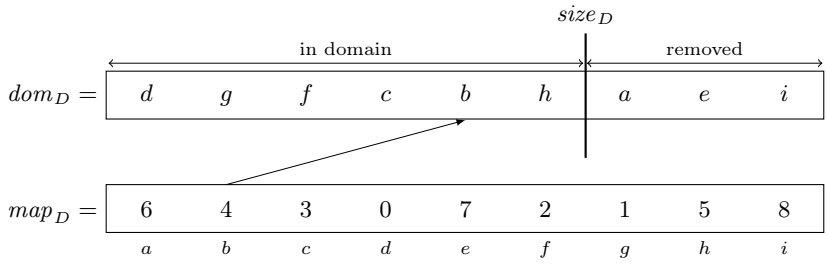

Fig. 1. Example representation of the domain $D = \{ b, c, d, f, g, h \}$, such that $size_D = 6$, when the initial domain is $\{ a, \ldots, i \}$. The $size_D$ first values in dom_D belong to the domain; the last values are those which have been removed. The map_D array maps values to their position in dom_D. For example, value b has index 4 in the dom_D array. In such representation, only the size needs to be kept in the trail.

For a domain D, the following invariants hold:

- $D = \{ dom_D[i] \mid 0 \leq i < size_D \}$
- $map_D[v] = i \Leftrightarrow dom_D[i] = v$
- The values in $dom_D[size_D \,..\, N-1]$ are not modified by any operation, where N is the size of the dom_D array, i.e., the size of the initial domain.

Thanks to the last invariant, the domain can be restored in constant time by setting the $size_D$ marker back to its previous position. Hence $size_D$ is the only entity to trail for the domain of dom_D.

To remove a value, we swap it with the last value of the domain (i.e., the value directly to the left of the $size_D$ marker), reduce $size_D$ by one and update the map_D array. Such operation is done in constant time, as shown in Figure 2.

Alternatively, we can restrict the domain to the intersection of itself and a set M. We first move all values of M which belong to the $size_D$ first elements

```
function Contains(D, v)                                          ▷ v ∈ D
    Return map_D[v] < size_D
end function

procedure Swap(D, i, j)          ▷ Swap elements at positions i and j in dom_D.
    dom_D[i], dom_D[j] ← dom_D[j], dom_D[i]
    map_D[dom_D[i]] ← i
    map_D[dom_D[j]] ← j
end procedure

procedure Bind(D, v)                                             ▷ D ← D ∩ { v }
    if map_D[v] ≥ size_D then
        size_D ← 0
    else
        Swap(D, map_D[v], 0)
        size_D ← 1
    end if
end procedure

procedure Remove(D, v)                                           ▷ D ← D \ { v }
    if map_D[v] < size_D then
        Swap(D, map_D[v], size_D − 1)
        size_D ← size_D − 1
    end if
end procedure

procedure ClearMarks(D)                                          ▷ M_D ← ∅
    mark_D ← 0
end procedure

procedure Mark(D, v)                                             ▷ M_D ← M_D ∪ (D ∩ { v })
    if map_D[v] < size_D ∧ map_D[v] >= mark_D then
        Swap(D, map_D[v], mark_D)
        mark_D ← mark_D + 1
    end if
end procedure

procedure Restrict(D)                                            ▷ D ← M_D
    size_D ← mark_D
end procedure
```

Fig. 2. Operations on the discrete representation of variables involve swapping values in the dom_D array. All procedures have an $\mathcal{O}(1)$ time complexity.

of dom_D, i.e., which are still in the domain, at the beginning of dom_D. Such operation is called MARK in Figure 2. The $mark_D$ counter keeps track of the marked values (see figure 3). We denote the set of marked values by M_D. Once all values are marked, we set $size_D$ to the size of the intersection, i.e., $mark_D$. The whole operation is done in $\mathcal{O}(|M|)$, with $|M|$ the size of M.

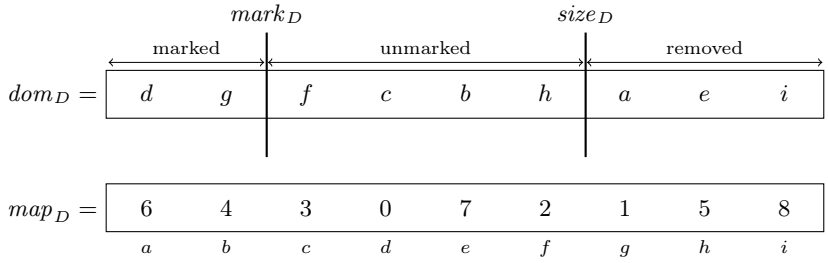

Fig. 3. Values can be marked in the discrete representation of a domain by moving the value to the beginning of the dom_D array and increasing the $mark_D$ marker. To restrict the domain to only the marked values, we only need to set $size_D$ to $mark_D$.

Operations on the bounds however are inefficient. This major drawback is due to the unsorted dom_D array. Searching for the minimum or maximum value requires the traversal of the whole domain. Increasing the lower bound or decreasing the upper bound involves removing every value between the old and new bound one by one.

Fortunately it is not frequent to have a propagator updating both the bounds of variable and removing values in the middle. OscaR [10] uses sparse set domain representation and also maintains bounds information: when one value is removed from the domain, if it happens to be the current minimum value, a new minimum value is searched. Castor [13] also maintains the bounds but update them lazily each time an iteration over the domain is achieved. The bounds are also trailed entities. It means that restoring the domain also involves restoring the bounds.

Very sparse domains: In the above representation, we use an array to implement map_D structure. For a very sparse initial domain this is not optimal since it requires a space proportional to the difference between the largest and smallest value. We can imagine using another implementation of map_D structure, for example with Hashmaps also allowing fast access based on keys consuming a memory proportional to the number of initial values in the domain.

3 Incremental propagation

For some constraints, it is much more efficient to realize an incremental filtering rather than filtering from scratch. Incremental filtering requires to know exactly

the domain changes (also called delta) since last call. This is generally achieved by maintaining some state inside the constraint. AC5-based solvers [18] provide this information by implementing a queue of events of single changes. In such approach, the information of every removed value arrives one by one to the propagator through a call to a `valRemoved(x,v)` method in charge of propagating the constraint given that `v` has been removed from domain of `x`. The approach has some drawbacks:

- It can be a source of inefficiency since one execution of a filtering algorithm is achieved for each removed value while it can (sometimes) be more efficient to treat them all at once.
- It is a potential source of implementation bugs. Indeed, the developer must be well aware that there can be a mismatch between the domains of the variables and the information received so far about removed values. For some filtering algorithm, it is important to consider not yet handled values in the event queue as part of the domain, since their removal has not yet been reflected in the internal state of the propagator.
- A propagator cannot choose a different filtering algorithm based on the size of the delta, since it has no access to the event queue. This is illustrated in the following example.

Example 1. Consider the constraint $x = y + c$ with c an integer constant. We assume a propagator achieving domain consistency filtering for this constraint based on AC5. When a value v is removed from $D(x)$, the value $v - c$ must be removed from $D(y)$, and vice-versa when a value is removed from $D(y)$. Assume now that the initial domains of x and y are very large, say $[0 .. 10000]$. If the propagation of other constraints leave only 10 values in the domain of x, the `valRemoved(x,v)` method will be called 9990 times. But if the propagator knows in advance that so many values have been removed, we could choose to scan the 10 remaining values in $D(x)$ instead of processing the 9990 removed values.

Advisors [7] is another technique used in Gecode enabling incremental propagation. In this approach the *advised* propagator receives a *log* of modification events used to update its state. As explained in Gecode's tutorial [16], the information provided concerns the bound changes but no accurate information is given about value removals. As for AC5, several bound notifications can happen before the actual propagate is executed. This technique is independent from the domain representation but comes with a cost for propagators using it.

We introduce a new technique based on sparse sets allowing propagators to have access to all delta changes at once without additional cost. The drawback is that our technique relies on the sparse set domain representation and is thus dependent on the domain representation.

The idea is the following: at the end of the propagation of a constraint, that constraint remembers the sizes of the domains. On the next propagation of the constraint, the values that have been removed from the domains will be between the saved old $size_D$ and the new $size_D$, as shown in Figure 4. By iterating over the values between the two markers, we can have access to the removed values

since the last execution of the propagator. When backtracking, the saved sizes must be reset.

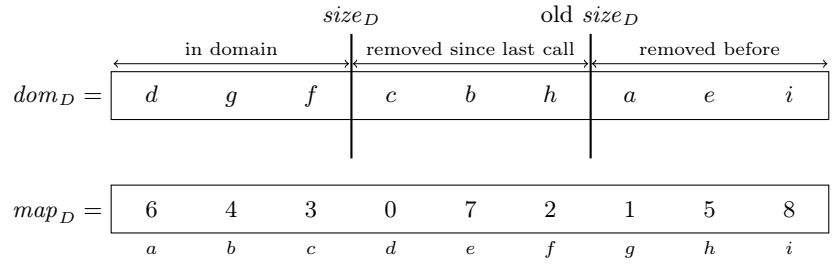

Fig. 4. If "old $size_D$" was the size of D on the last propagation of a constraint, the values c, b and h have been removed since then. To perform incremental propagation, the propagator only has to check the values for which c, b or h was a support. Note that the "old $size_D$" marker is a property of the propagator and is different for each propagator, while $size_D$ is a property of D.

This technique is used in Castor [13] and OscaR [10] solvers. An implementation skeleton of an OscaR propagator in Scala is given in Figure 5. As can be seen, one can specify a filtering procedure interested to have the delta changes of a specific variable `X`. Behind the scene it means that three values are updated after each execution of the filtering code: the old minimum, the old maximum and the previous size. Based on these three values and the sparse set representation, we can offer the query methods on the `delta` object. Each of these methods execute in constant time and iterating on the removed values in `X` takes $O(\Delta)$ time with Δ the number of removed values since last call to `propagate`.

Finally, note that successor vectors can also permit to get delta changes, provided that an additional stack is used to record deleted values. This is described in [8] and implemented in AbsCon. Similarly to sparse sets, at the end of the propagation of a constraint, it suffices to execute a cheap operation: storing the last deleted value of each domain.

4 Sparse Sets for integer set variable domains

The idea of subset bound representation for set variables was developed independently in [12, 6]. A set domain D is represented by two sets of integers $\underline{D}$ and $\overline{D}$, with $\underline{D} \subseteq \overline{D}$, representing set values $\{ v \mid \underline{D} \subseteq v \subseteq \overline{D} \}$. Thus $\underline{D}$ contains the required values that must be in the set, and $\overline{D}$ contains the possible values that can be in the set. This representation is usually complemented with an integer variable representing the cardinality of the set. The set variable domains are commonly represented (e.g., in Choco3 [4]) with two bit vectors, one for $\underline{D}$ and one for $\overline{D}$.

```
class MyConstraint(val X: CPVarInt) extends Constraint(X.s) {
    override def setup(l: CPPropagStrength): CPOutcome = {
        X.filterWhenDomainChanges { delta =>
            // ... filtering code ...
            delta.changed() // has the domain changed?
            delta.maxChanged() // has the max value changed?
            delta.minChanged() // has the min value changed?
            delta.size() // number of value removed
            delta.oldMin() // old minimum
            delta.oldMax() // old maximum
            delta.values() // iterator over the removed values
            Suspend
        }
        Success
    }
}
```

Fig. 5. Illustration of the functionality allowing to get the delta information on domain changes.

Sparse sets are a good alternative representation for set variable domains, offering the same advantages as for integer variables. The only difference is that two size values are needed as illustrated in Figure 6. These two size values are also trailed to restore the set variable domain on backtrack. For a domain D, the following invariants hold:

- $D = \{\, v \mid dom_D[0 \,..\, size_{\underline{D}} - 1] \subseteq v \subseteq dom_D[0 \,..\, size_{\overline{D}} - 1] \,\}$
- $map_D[v] = i \Leftrightarrow dom_D[i] = v$
- The values in $dom_D[0 \,..\, size_{\underline{D}} - 1]$ and $dom_D[size_{\overline{D}} \,..\, N - 1]$ are not modified by any operation, where N is the size of the dom_D array.

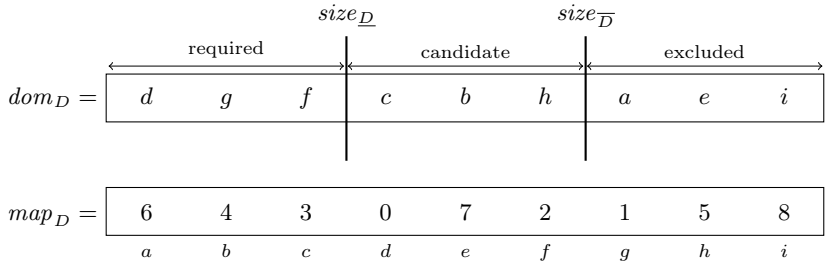

Fig. 6. Example representation of the domain $\underline{D} = \{\, d, g, f \,\}, \overline{D} = \{\, d, g, f, c, b, h \,\}$. The $size_{\underline{D}}$ first values belong to $\underline{D}$; the $size_{\overline{D}}$ first values belong to $\overline{D}$. In such representation, only $size_{\underline{D}}$ and $size_{\overline{D}}$ need to be kept in the trail.

Note that $size_{\underline{D}}$ can only increase since removing a required value should trigger a backtrack, and $size_{\overline{D}}$ can only decrease since no constraint can add a value in the possible set. The variable is bound to a single set value when $size_{\underline{D}} = size_{\overline{D}}$. This representation is also very convenient to remove every possible value in constant time when the upper bound on the cardinality of the set variable becomes equal to $size_{\underline{D}}$.

As for integer variables, one can offer to the propagators the delta changes on $\underline{D}$ and $\overline{D}$ by trailing the two old sizes without additional cost.

Set variables are implemented with sparse sets in OscaR [10].

4.1 Related Work

In many contexts different from domain implementation, using a sparse set looks the most effective solution to maintain state constraint systems. In this section, we illustrate our purpose with a possible use of sparse sets to manage the set of fixed and unfixed variables in constraint scopes.

For some kinds of (global) constraints, it is worthwhile handling separately the variables that are fixed (assigned) and those that are not. Indeed, when filtering such constraints, one typically needs to iterate over the unfixed variables involved in their scopes. For example, let us consider the statement given at Line 5 of Algorithm 5 (STR2) in [9]:

foreach variable $x \in scp(c) \mid x \notin past(P)$

Here, the loop iterates over all variables involved in a constraint c of a constraint network P. Specifically, we only need to consider the variables of the scope of c ($scp(c)$) that have not been assigned by the backtrack search algorithm (i.e., are not in $past(P)$). Depending on the implementation, with r and r' being respectively $|scp(c)|$ and $|scp(c) \setminus past(P)|$, this iteration is $O(r)$ or $O(r')$.

To guarantee $O(r')$, we can use a sparse set containing for each variable x of the constraint network the position of x in $scp(c)$. The position value is removed whenever a variable is fixed. As for the domains, this sparse set is made reversible by trailing its size.

One can also uses reversible sparse sets in STR2 [9] to efficiently maintain a set of valid tuples of a table constraint. Cheng and Yap [3] use sparse sets to efficiently maintain a set of MDDs (multi-valued decision diagrams).

5 Conclusion

We have shown in this paper how to use sparse sets for representing domains enabling constant time operations for searching for, adding or deleting a value in a domain. This sparse set based representation is already used in LAD [17], OscaR [10], Castor [13], AbsCon and Choco3 [4].[9] We introduced how to access delta changes in the domains when using spare-sets, important for incremental

[9] Graph domains implementation by Jean-Guillaume Fages.

propagation. We also explained subset bound implementation for set variables using sparse sets. As a future work we plan to compare the performances of incremental filtering based on sparse sets over AC5 based propagators.

Acknowledgments

The authors want to thank the anonymous reviewers for their supportive comments. Thanks also to Guido Tack for his explanations about advisors and to Jean-Guillaumes Fages for pointing us spare sets utilization inside Choco3 [4]. The first author is supported as a Research Assistant by the Belgian FNRS (National Fund for Scientific Research).

References

1. Aggoun, A., Beldiceanu, N.: Time stamps techniques for the trailed data in constraint logic programming systems. In: 8$^{\text{ème}}$ Séminaire Programmation en Logique – SPLT (1990)
2. Briggs, P., Torczon, L.: An efficient representation for sparse sets. ACM Letters on Programming Languages and Systems 2(1–4), 59–69 (1993)
3. Cheng, K., Yap, R.: An MDD-based generalized arc consistency algorithm for positive and negative table constraints and some global constraints. Constraints 15(2), 265–304 (2010)
4. Fages, J.G., Jussien, N., Lorca, X., Prud'homme, C.: Choco3: an open source java constraint programming library. Tech. rep. (2013)
5. Gecode Team: Gecode: Generic constraint development environment (2006), http://www.gecode.org
6. Gervet, C.: New structures of symbolic constraint objects: Sets and graphs. In: Third Workshop on Constraint Logic Programming – WCLP93 (1993)
7. Lagerkvist, M.Z., Schulte, C.: Advisors for incremental propagation. In: Bessière, C. (ed.) Thirteenth International Conference on Principles and Practice of Constraint Programming. Lecture Notes in Computer Science, vol. 4741, pp. 409–422. Springer-Verlag, Providence, RI, USA (2007)
8. Lecoutre, C.: Constraint Networks: Targeting Simplicity for Techniques and Algorithms. Wiley-ISTE (2009)
9. Lecoutre, C.: STR2: optimized simple tabular reduction for table constraints. Constraints 16(4), 341–371 (2011)
10. OscaR Team: OscaR: Scala in OR (2012), https://bitbucket.org/oscarlib/oscar
11. Pothitos, N., Stamatopoulos, P.: Flexible management of large-scale integer domains in CSPs. In: SETN 2010. Lecture Notes in Artificial Intelligence, vol. 6040, pp. 405–410. Springer (2010)
12. Puget, J.F.: Set constraints and cardinality operator: Application to symmetrical combinatorial problems. In: Third Workshop on Constraint Logic Programming – WCLP93 (1993)
13. le Clément de Saint-Marcq, V., Deville, Y., Solnon, C., Champin, P.A.: Castor: A constraint-based sparql engine with active filter processing. In: Simperl, E., Cimiano, P., Polleres, A., Corcho, O., Presutti, V. (eds.) The Semantic Web: Research and Applications. Lecture Notes in Computer Science, vol. 7295, pp. 391–405. Springer (2012)

14. Schulte, C.: Comparing trailing and copying for constraint programming. In: Proceedings of the Sixteenth International Conference on Logic Programming. pp. 275–289 (2000)
15. Schulte, C., Carlsson, M.: Finite domain constraint programming systems. Handbook of Constraint Programming pp. 495–526 (2006)
16. Schulte, C., Tack, G., Lagerkvist, M.Z.: Modeling and programming with Gecode (2013)
17. Solnon, C.: Alldifferent-based filtering for subgraph isomorphism. Artificial Intelligence 174(12–13), 850–864 (2010)
18. Van Hentenryck, P., Deville, Y., Teng, C.M.: A generic arc-consistency algorithm and its specializations. Artificial Intelligence 57(2), 291–321 (1992)