Constraints (2008) 13:206–226
DOI 10.1007/s10601-007-9027-7

A Reinforcement Learning Approach to Interval Constraint Propagation

Frédéric Goualard · Christophe Jermann

Published online: 10 January 2008

Abstract When solving systems of nonlinear equations with interval constraint methods, it has often been observed that many calls to contracting operators do not participate actively to the reduction of the search space. Attempts to statically select a subset of efficient contracting operators fail to offer reliable performance speed-ups. By embedding the *recency-weighted average* Reinforcement Learning method into a constraint propagation algorithm to dynamically learn the best operators, we show that it is possible to obtain robust algorithms with reliable performances on a range of sparse problems. Using a simple heuristic to compute initial weights, we also achieve significant performance speed-ups for dense problems.

Keywords Numerical constraints · Interval propagation · Reinforcement learning

1 Introduction

We consider the problem of finding tight enclosures for all the solutions of systems of nonlinear real equations:[1]

$$\begin{aligned} f_1(x_1, \ldots, x_n) &= 0 \\ &\ddots \\ f_n(x_1, \ldots, x_n) &= 0 \end{aligned} \tag{1}$$

[1] In the following, we will refer to System (1) even when considering systems in which all variables do not occur in all equations (*sparse* systems).

F. Goualard · C. Jermann (✉)
LINA, CNRS FRE 2729, University of Nantes, 2, rue de la Houssinière,
BP 92208 F-44322 Nantes cedex 3, France
e-mail: Christophe.Jermann@univ-nantes.fr

F. Goualard
e-mail: Frederic.Goualard@univ-nantes.fr

A successful approach associates interval domains to all variables and uses *interval arithmetic* [20] in a combination of *contracting operators*—to tighten the domains of the variables while retaining all solutions—and of an *exploration algorithm* that recursively splits domains.

An effective implementation of contracting operators relies on interval first-order methods, which start from the initial domains and then solve each of the n unary equations (*projections*):

$$f_i(\boldsymbol{I_1}, \ldots, \boldsymbol{I_{i-1}}, x_i, \boldsymbol{I_{i+1}}, \ldots, \boldsymbol{I_n}) = 0 \tag{2}$$

in turn, where $\boldsymbol{I_j}$ is the current interval domain for x_j. This process is iterated over all f_is until a fixed-point is reached (or, until the domains computed for the variables do not change too much).

For the linear case, the speed of convergence towards a solution depends heavily on the initial order of equations and variables, which defines *the transversal*, that is, the set of n projections (f_i, x_i) considered in Eq. 2. A classical result states that equations and variables should be initially reordered so as to make the corresponding coefficient matrix strictly diagonal dominant [23].

For the nonlinear case, it has been observed that nonlinear first-order methods are equally sensitive to the initial order of equations and variables that defines the n projections considered. However, to our knowledge, there is no sure-fire static method to select projections that ensures prompt convergence. What is more, it appears [8, 9, 11] that selecting more than n projections may sometime speed the solving process up.

The bc3 algorithm [4] studied in this paper associates the good principles of first-order methods with a smart propagation algorithm devised by Mackworth [19] to ensure consistency in a network of relations. It also uses clever numerical methods to reduce the computational burden of solving projections.

Contrary to standard first-order methods, bc3 considers all n^2 possible projections instead of only n of them, thereby avoiding a bad choice of a transversal. For sparse problems (those for which some variables do not occur in all equations), this is a reasonable strategy, since the number p of unary equations to solve is of the order of n. On the other hand, for large dense problems (i.e., $n^2 \gg n$ and $p \approx n^2$), the number of univariate equations to solve makes this approach computationally too expensive. In addition, depending on the problems, many projections may never lead to any tightening of the domains of the variables, for reasons that are not clearly understood yet. To make things worse, there may exist subsets of efficient projections, but only transiently at some point in the computation process. Therefore, there is no point in trying to statically select a subset of projections to consider in bc3.

In this paper, we embed the *recency-weighted average* [26] Reinforcement Learning method into bc3 to dynamically select the most efficient projections (that is, the ones leading to the maximum tightening of variables' domains). We experimentally show that the resulting algorithm outperforms bc3 for problems where no static transversal exists. We also present an heuristic to initialize weights associated to projections that leads to significant speed-ups with respect to bc3 when considering large dense problems with a static transversal.

In order to be reasonably self-content, we sketch the principles of interval constraint algorithms and we show the limits and weaknesses of bc3 in Section 2; We present in Section 3 how to add the *recency-weighted average* (rwa) Reinforcement

Learning method to bc3 to address its shortcomings, and we describe the resulting algorithm, after having discussed how to fix the various parameters arising from the use of rwa. Our new algorithm is compared with bc3 on a set of standard problems in Section 4. Lastly, we compare our approach with previous related works in Section 5, and we outline directions for future researches in Section 6.

2 Interval Constraint Solving

Classical iterative numerical methods suffer from defects such as loss of solutions, absence of convergence, and convergence to unwanted attractors due to the use of but a very small subset of the real numbers on computers: floating-point numbers [13] (aka *floats*). At the end of the fifties, Moore [20] popularized the use of intervals to control the errors made while computing with floats.

Interval arithmetic replaces floating-point numbers by closed connected sets of the form $\boldsymbol{I} = [\underline{\boldsymbol{I}}, \overline{\boldsymbol{I}}] = \{a \in \mathbb{R} \mid \underline{\boldsymbol{I}} \leqslant a \leqslant \overline{\boldsymbol{I}}\}$ from *the set* $\mathbb{I}$ *of intervals*, where $\underline{\boldsymbol{I}}$ and $\overline{\boldsymbol{I}}$ are floating-point numbers. In addition, each n-ary real function ϕ with domain $\mathcal{D}_\phi$ is extended to an interval function Φ with domain $\mathcal{D}_\Phi$ in such a way that the *containment principle* is verified:

$$\forall A \in \mathcal{D}_\phi \, \forall \boldsymbol{I} \in \mathcal{D}_\Phi : \, A \in \boldsymbol{I} \implies \phi(A) \in \Phi(\boldsymbol{I})$$

Example 1 The *natural interval extensions* of addition and multiplication are defined by:

$$\boldsymbol{I_1} + \boldsymbol{I_2} = \left[\underline{\boldsymbol{I_1}} + \underline{\boldsymbol{I_2}}, \overline{\boldsymbol{I_1}} + \overline{\boldsymbol{I_2}}\right]$$

$$\boldsymbol{I_1} \times \boldsymbol{I_2} = \left[\min\left(\underline{\boldsymbol{I_1}}\,\underline{\boldsymbol{I_2}}, \underline{\boldsymbol{I_1}}\,\overline{\boldsymbol{I_2}}, \overline{\boldsymbol{I_1}}\,\underline{\boldsymbol{I_2}}, \overline{\boldsymbol{I_1}}\,\overline{\boldsymbol{I_2}}\right), \max\left(\underline{\boldsymbol{I_1}}\,\underline{\boldsymbol{I_2}}, \underline{\boldsymbol{I_1}}\,\overline{\boldsymbol{I_2}}, \overline{\boldsymbol{I_1}}\,\underline{\boldsymbol{I_2}}, \overline{\boldsymbol{I_1}}\,\overline{\boldsymbol{I_2}}\right)\right]$$

Then, given the real function $f(x, y) = x \times x + y$, we may define its natural interval extension by $\boldsymbol{f}(\boldsymbol{x}, \boldsymbol{y}) = \boldsymbol{x} \times \boldsymbol{x} + \boldsymbol{y}$, and we have that, e.g., $\boldsymbol{f}([2, 3], [-1, 5]) = [3, 14]$.

Implementations of interval arithmetic use outward rounding to enlarge the domains computed so as not to violate the containment principle, should some bounds be unrepresentable with floating-point numbers [12].

Many numerical methods have been extended to use interval arithmetic [22, 24]. Given the system of nonlinear equations (1) and initial domains $\boldsymbol{I_1}, \ldots, \boldsymbol{I_n}$ for the variables, these methods are usually embedded into a *branch-and-prune* algorithm BaP (see Algorithm 1) that manages a set of boxes of domains to tighten. Starting from the initial box $\boldsymbol{D} = \boldsymbol{I_1} \times \cdots \times \boldsymbol{I_n}$, BaP applies a numerical method "prune" to tighten the domains in $\boldsymbol{D}$ around the solutions of System (1). It then bisects the resulting box along one of its dimensions whose width is larger than some specified threshold ε. The BaP algorithm eventually returns a set of boxes whose largest dimension has a width smaller than ε and whose union contains all the solutions to Eq. 1. Note, however, that some boxes may eventually contain zero, one, or more than one solution.

Interval nonlinear Gauss-Seidel is a possible implementation for prune. It considers the *n unary projections*:

$$\begin{aligned} f_1^{(1)}(x_1, I_2, \ldots, I_n) &= 0 \\ &\ddots \\ f_n^{(n)}(I_1, \ldots, I_{n-1}, x_n) &= 0 \end{aligned} \tag{3}$$

and uses any unidimensional root-finding method to tighten the domain of each variable x_i in turn. Using a unidimensional Newton-Raphson root-finder leads to the *Gauss-Seidel-Newton method* [23], whose extension to intervals is the *Herbort-Ratz method* [11].

Algorithm 1 Branch-and-Prune algorithm

```
[BaP] in: F = (f_1, ..., f_n): R^n → R^n
      in: D_in ∈ I^n
      out: sol ⊂ I^n
begin
  % Set of boxes to tighten further
  boxset ← {D_in}
  % Set of solution boxes
  sol ← ∅
  while boxset ≠ ∅ do
    % Choice of a box to tighten according to
    % an implementation-defined policy (FIFO, LIFO, ...)
    D ← extract_box(boxset)
    D ← prune(F, D)
    % Is the box small enough to be considered a solution?
    if w(D) ⩽ ε then
      if D ≠ ∅ then
        sol ← sol ∪ {D}
      endif
    else
      boxset ← boxset ∪ split(D)
    endif
  endwhile
end
```

Let UN be the elementary step performed by one unidimensional Newton application to the projection $f_i^{(j)}$, where i and j may be different [23]. As soon as D is moderately large, it is very likely that each projection constraint will have many "solutions" that are not solutions of the original real system, and whose discarding slows down the computation. The Newton method will also fail to narrow down the domain of some x_i if there is more than one solution to the corresponding projection constraint for the current box D, thereby demanding more splitting in BaP. Achieving the right balance between the amount of work required by the prune method and the number of splitting performed overall is the key to maximum efficiency of BaP. In this very situation, experimental evidences show that trying

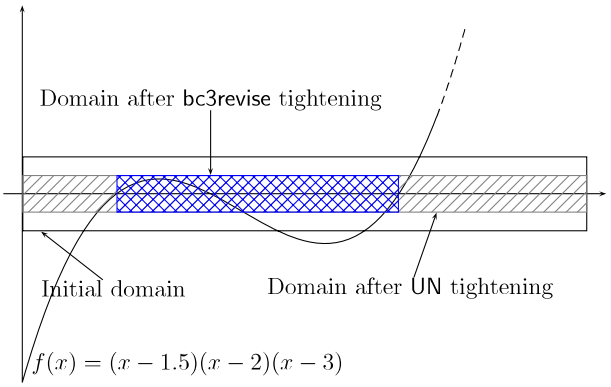

Fig. 1 Comparison of UN and bc3revise

harder to narrow down the domain of $\boldsymbol{x_i}$ pays off [4]. A way to do it is to ensure that the canonical intervals $[\underline{\boldsymbol{I}_j}, \underline{\boldsymbol{I}_j}^+]$ and $[\overline{\boldsymbol{I}_j}^-, \overline{\boldsymbol{I}_j}]$, whose bounds are two consecutive floating-point-numbers, are solutions of $\boldsymbol{f_i}^{(j)}(\boldsymbol{I_1}, \ldots, \boldsymbol{I_{j-1}}, \boldsymbol{x_j}, \boldsymbol{I_{j+1}}, \ldots, \boldsymbol{I_n}) = 0$. Algorithm bc3revise [4] ensures such a property (called *box consistency of* $\boldsymbol{x_j}$ *w.r.t. the constraint* $\boldsymbol{f_i} = 0$ *and* $\boldsymbol{D}$) for a projection $\boldsymbol{f_i}^{(j)}$. A simple method to implement it combines a dichotomic process with Newton-Raphson steps to isolate the leftmost and rightmost solutions included in $\boldsymbol{D}$ of each projection constraint.

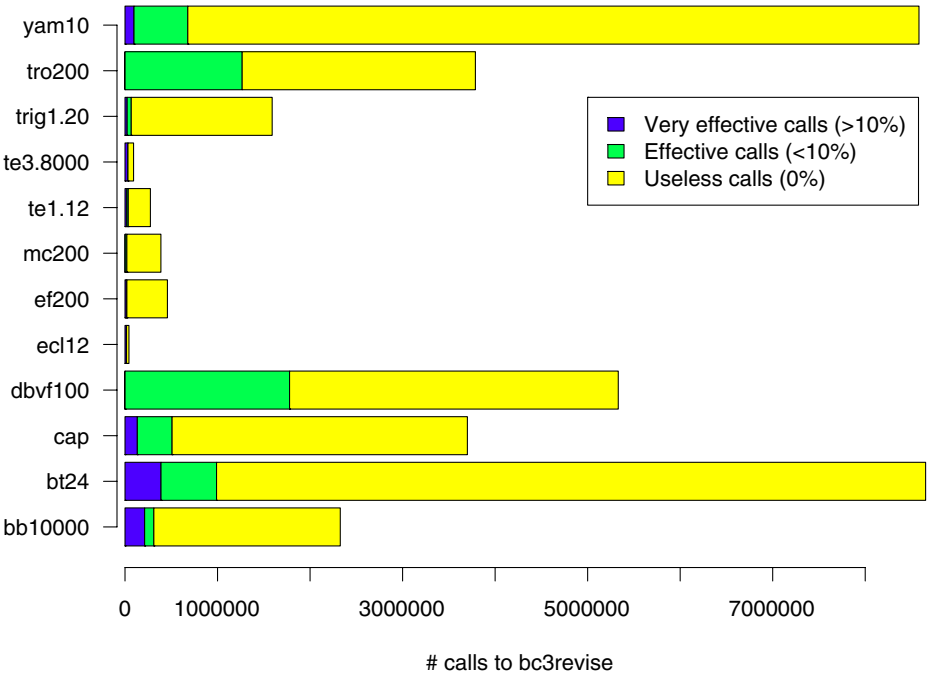

Fig. 2 Effectiveness of bc3

Example 2 Consider the constraint $f(x) = (x - 1.5)(x - 2)(x - 3) = 0$ and the domain $\boldsymbol{I} = [1, 4]$ for x (See Fig. 1). The UN method leaves $\boldsymbol{I}$ unchanged because the derivative of f over the initial domain contains 0 while bc3revise narrows down $\boldsymbol{I}$ to $\boldsymbol{I}' = [1.5, 3]$, which is the smallest interval included in $\boldsymbol{I}$ that contains all the solutions to the interval constraint $\boldsymbol{f}(\boldsymbol{x}) = 0$.

Interval constraint methods [3] combine interval arithmetic—to reliably solve a system of real equations without loss of solutions—and smart propagation algorithms [19], to take advantage of its possible sparsity (some variables may not occur in all constraints).

Algorithm bc3 [4] (see Algorithm 2) is such a method, which relies on the pruning operator bc3revise to tighten domains. It is akin to a free-steering generalized nonlinear Gauss-Seidel method with a twist [7]: as shown in Algorithm 2, the set of projections on which bc3revise is applied contains all the possible projections from the equation system, and not n of them only.

Algorithm bc3, or one of its variations, is often used as a basis to reliably solve nonlinear constraint systems, though its use of the at most n^2 projections of a system of n equations on n variables makes it a bad choice for large dense problems due to the overwhelming number of projections it then has to consider. It is also sensitive to a problem that plagues other interval constraint algorithms, whereby many calls of the contracting operators lead to no reduction of the domains at all. Figure 2 shows this situation for bc3 on twelve standard test problems to be presented in Section 4: calls to bc3revise are separated into three categories (*very effective calls* leading to a reduction of domain size by more than 10%, *effective calls* leading to a reduction of domain size by less than 10%, and *useless calls* leading to no reduction at all). As we can see, the majority of the work performed is essentially useless for almost all problems.

Algorithm 2 The bc3 algorithm

[bc3] in: at most n^2 projections $T = \{(f_i, x_j) \mid i, j \in \{1, \ldots, n\}\}$
in/out: box of domains $\boldsymbol{D} = \boldsymbol{I}_1 \times \cdots \times \boldsymbol{I}_n$

```
begin
  S ← T
  while S ≠ ∅ and D ≠ ∅ do
    (f_i, x_j) ← Choose a projection in S
    D' ← bc3revise(f_i, x_j, D)
    if I'_j ⊊ I_j then      % The domain of x_j has been narrowed down
      if I'_j ≠ ∅ then
        % We add to S all projections that rely on the domain of x_j
        S ← S ∪ {(f_β, x_γ) ∈ T | x_j occurs in f_β}
      endif
      D ← D'
    endif
    S ← S \ {(f_i, x_j)}
  endwhile
end
```

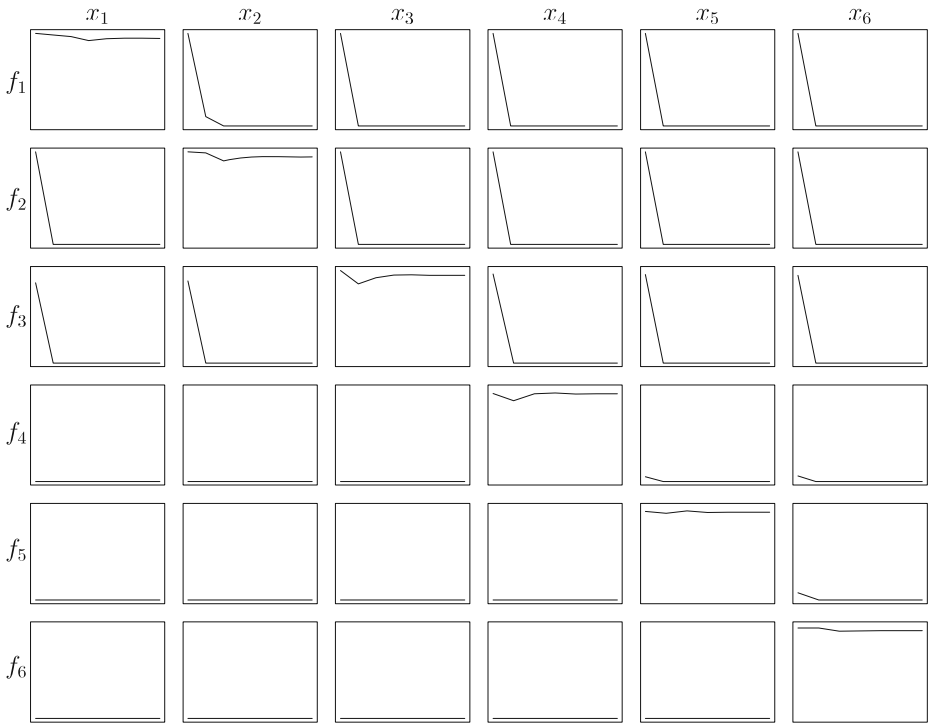

Fig. 3 History of narrowing effectiveness per projection for *Moré-Cosnard-6*; in ordinate, the efficiency in domain reduction percentage; in abscissa, calls to the corresponding projection

Since no fail-safe efficient strategy exists for choosing the right contracting operator (line 3 of Algorithm 2) at the right time, the standard implementation uses a queue to represent S (the contracting operators are applied in the order they are inserted).

These inefficiencies may have two different non-exclusive causes: either some of the at most n^2 projections never lead to any reduction, and therefore only clutter the propagation queue; or the effectiveness of projections varies widely during the solving process and may oscillate from nothing to good.

In the first case, optimizing bc3 boils down to statically identifying the best projections and using only these ones; in the second case, we have to keep all n^2 projections and find a means to consider at any time during the solving process only those projections with good tightening potential.

As the following examples show, it appears that, depending on the problem considered, both situations may arise. Consider the *Moré-Cosnard problem* [21] of dimension 6: Fig. 3 shows the history of effectiveness (in ordinate, percentage of reduction obtained in the range $[0, 1]$) of each projection when solving it with bc3 (abscissa goes from the first use of (f_i, x_j) to its last use).[2] One may easily see that

[2]Note that, for *Moré-Cosnard* as well as for the next example *Trigexp1*, each projection is used almost as many times as the others in an implementation of bc3 in which S is managed as a queue.

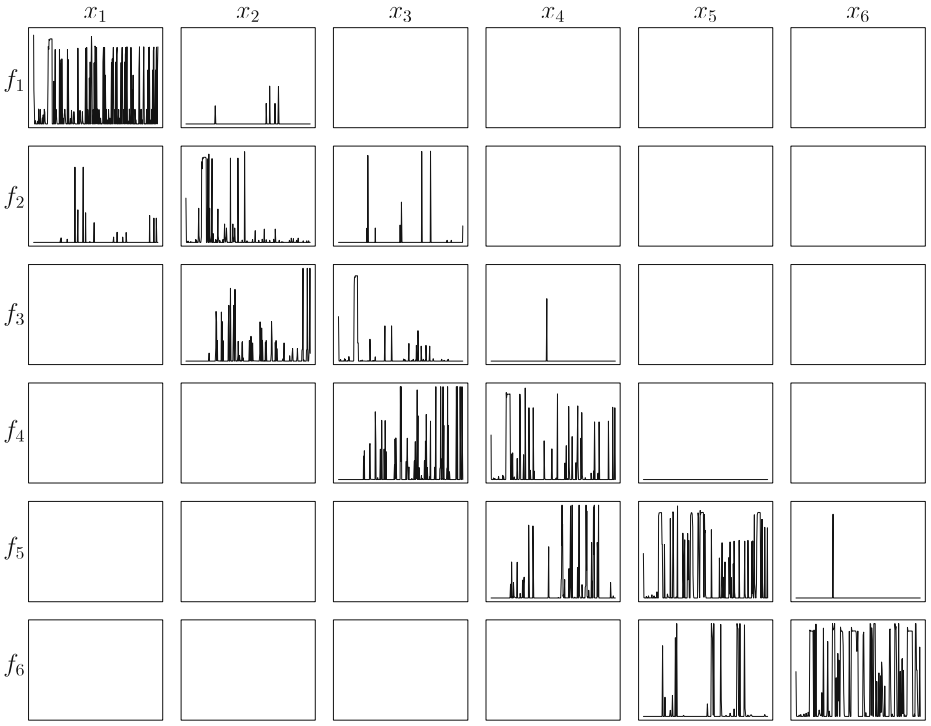

Fig. 4 History of narrowing effectiveness per projection for *Trigexp1* ($n = 6$)

the only useful projections are the ones on the diagonal. As a side note, we may also remark that some projection (e.g., (f_1, x_2), (f_2, x_3), ...) perform well the first time they are used, and then consistently badly afterwards. This does not bode well for identifying statically which are the best projections to retain.

On the other hand, consider the sparse problem *Trigexp1* [18] for $n = 6$: The history of effectiveness given in Fig. 4 shows that there are more than n useful projections (for example, (f_5, x_4) and (f_4, x_3) should probably both be retained). Furthermore the effectiveness of each projection varies widely during the solving process, and some projections that are not very good in the beginning become good or average afterwards (e.g., (f_5, x_4)), while some projections that are very good in the beginning become quite bad after some time (e.g., (f_2, x_2)).

These examples should convince us that we have to keep all projections for consideration in bc3, and that we must resort to some dynamic selection scheme to apply bc3revise only on those projections that offer the best narrowing potential at some point in the solving process.

3 Speeding-up Solving Through Reinforcement Learning

Reinforcement learning [26] is a sub-area of machine learning considering unsupervised agents that iteratively refine their strategy for choosing actions in an uncertain environment so as to maximize a long-term reward. Agents refine their knowledge of the environment by observing the effect of the most recently chosen actions. Hence,

they have to achieve the optimal trade-off between exploration—testing the different actions at hand—and exploitation—performing the actions that have the greatest potential reward. The problem is compounded in an ever-evolving environment, that is when the probability of a reward for an action may vary.

A standard problem considered in reinforcement learning is the *multi-armed bandit problem* [26]: given k slot machines with payoff probabilities unknown to the player and some time horizon, find the sequence of levers to pull in order to maximize the gains. In this problem, the action chosen is represented by the number associated with the lever to pull, the reward is the gain obtained by pulling the lever chosen, and the long-term objective is to maximize the cumulated rewards at the time horizon. The *non stationary* variant of the problem involves slot machines with varying payoff probabilities [1, 2].

A close look at our problem allows us to draw an analogy between the selection process of projections in bc3 and the non stationary multi-armed bandit problem: in our context, the k levers are the at most n^2 projections, and their payoff is the relative domain reduction[3] their use in bc3revise leads to. We use the relative domain reduction instead of the absolute one as a measure of efficiency in order not to favor too much the projections used early when the domains of variables are large to the detriment of projections applied on smaller domains.

No time horizon is given in bc3. However, by maximizing the sum of relative domain reductions, we expect both to avoid applying bc3revise on projections that do not lead to any reduction, and to reduce the overall number of calls to bc3revise, thereby accelerating the computation of solutions.[4]

3.1 Adaptation of Reinforcement Learning to bc3

A difficulty of the adaptation of the reinforcement learning approach to our problem of selecting the best projections is that, especially in big or dense problems, the number of projections among which to choose is so large that a lot of time can be spent exploring alternatives. Consequently, we have retained the *recency-weighted average* (rwa) [26, Chapter 2, Section 6] as a reasonable reinforcement learning method for our purpose, it being more exploitation-oriented than most other methods.

Algorithm rwa is a standard method to solve non stationary reinforcement learning problems. It associates to each possible action a weight that measures its interest. This weight is a weighted average of all past rewards, hence the name. At each iteration, the action with the highest weight is chosen, its reward observed, and its weight updated accordingly. This method adopts a pure exploitation strategy since alternative choices are never explored.

In our context, using rwa means associating a weight $W^{(ij)}$ with each projection (f_i, x_j); the set S in bc3 (see Algorithm 2) is replaced by a priority queue (heaviest weights available first). Line 3 is then replaced by the extraction of the projection with heaviest weight. Let $r^{(ij)}$ be the relative reduction obtained on Line 4. The

[3]The relative reduction is defined by $(\mathsf{w}(\boldsymbol{I}_j^b) - \mathsf{w}(\boldsymbol{I}_j^a))/\mathsf{w}(\boldsymbol{I}_j^b)$ where $\mathsf{w}(\boldsymbol{I}_j^b)$ (resp. $\mathsf{w}(\boldsymbol{I}_j^a)$) is the width of the domain of x_j before (resp. after) applying bc3revise on (f_i, x_j).

[4]Once again, in interval constraint programming, a *solution* is a Cartesian product of domains whose largest width is smaller than a predefined threshold, and for which it is not possible to prove that it does *not* contain any point satisfying the system.

Algorithm 3 Box Consistency with Reinforcement Learning (bcrl)

[bcrl] in: $T = \{(f_i, x_j, W^{(ij)}) \mid i, j \in \{1, \ldots, n\}\}$
in/out: box of domains $\boldsymbol{D} = \boldsymbol{I}_1 \times \cdots \times \boldsymbol{I}_n$
begin
1 forall $j \in \{1, \ldots, n\}$ do
2 $Q_j \leftarrow \{(f_i, x_j, W^{(ij)}) \in T \mid i \in \{1, \ldots, n\}\}$
3 done
4 *% Looping on the projections in all queues*
5 while $\boldsymbol{D} \neq \emptyset$ and $\exists j \in \{1, \ldots, n\}$ s.t. $Q_j \neq \emptyset$ do
6 *% Adding the n heaviest projections from different Q_js into queue S*
7 $S \leftarrow \bigcup_{j=1}^{n} \{\text{pop}(Q_j)\}$
8 forall $(f_i, x_j, W^{(ij)}) \in S$ do *% Considering at most n heaviest projections*
9 $\boldsymbol{D}' \leftarrow$ bc3revise$(f_i, x_j, \boldsymbol{D})$
10 $r^{(ij)} \leftarrow (\mathsf{w}(\boldsymbol{I_j}) - \mathsf{w}(\boldsymbol{I}'_j))/\mathsf{w}(\boldsymbol{I_j})$ *% Computing the relative reduction*
11 $W^{(ij)} \leftarrow W^{(ij)} + \alpha\left(r^{(ij)} - W^{(ij)}\right)$
12 if $\boldsymbol{I}'_j \subsetneq \boldsymbol{I_j}$ then
13 if $\boldsymbol{I}'_j \neq \emptyset$ then
14 forall $k \in \{1, \ldots, n\}$ do
15 *% Adding projections to reconsider in respective queues*
16 $Q_k \leftarrow Q_k \cup \{(f_\beta, x_k, W^{(\beta k)}) \in T \mid x_j$ occurs in $f_\beta, \beta \in \{1, \ldots, n\}\}$
17 done
18 endif
19 $\boldsymbol{D} \leftarrow \boldsymbol{D}'$
20 endif
21 done
22 endwhile
end

weight $W_{k+1}^{(ij)}$ that takes into account the k past payoffs and the most recent one $r_{k+1}^{(ij)}$ is obtained with the formula:

$$W_{k+1}^{(ij)} = W_k^{(ij)} + \alpha\left(r_{k+1}^{(ij)} - W_k^{(ij)}\right) \tag{4}$$

where α is a constant parameter between 0 and 1 that monitors the importance granted to the past payoffs w.r.t. the current one.

Using one priority queue S for all the projections, as done in bc3, is a suboptimal strategy here in that it may create propagation cycles leading to overall slow convergence phenomena [17]. As an illustration, consider two projections p_1 and p_2 such that when p_1 is applied, p_2 is inserted in S and conversely. It could well be the case that p_1 and p_2 are applied cyclically with enough success so that the other projections are not considered. Such a phenomenon is in general counterproductive in the long term: even though p_1 and p_2 produce good relative reductions, they do not reduce the domains of all the variables and do not consider all the constraints.

To avoid this, we use one priority queue Q_j per variable x_j; when it needs to be reconsidered, the projection (f_i, x_j) is always pushed in Q_j. The resulting algorithm bcrl is presented in Algorithm 3. Algorithm bcrl contains an inner loop over at most n projections on n different variables (less than n projections if some queues are

temporarily empty) in addition to the while loop to reach a fixed-point, which was already present in bc3.

Note: In order to rigorously validate our results and to assess the impact of the various choices we made, we have tested using one propagation queue per variable with bc3 as well. For all the problems considered here, the computational time required is essentially similar to the one required by bc3 with one propagation queue only (see Fig. 5).

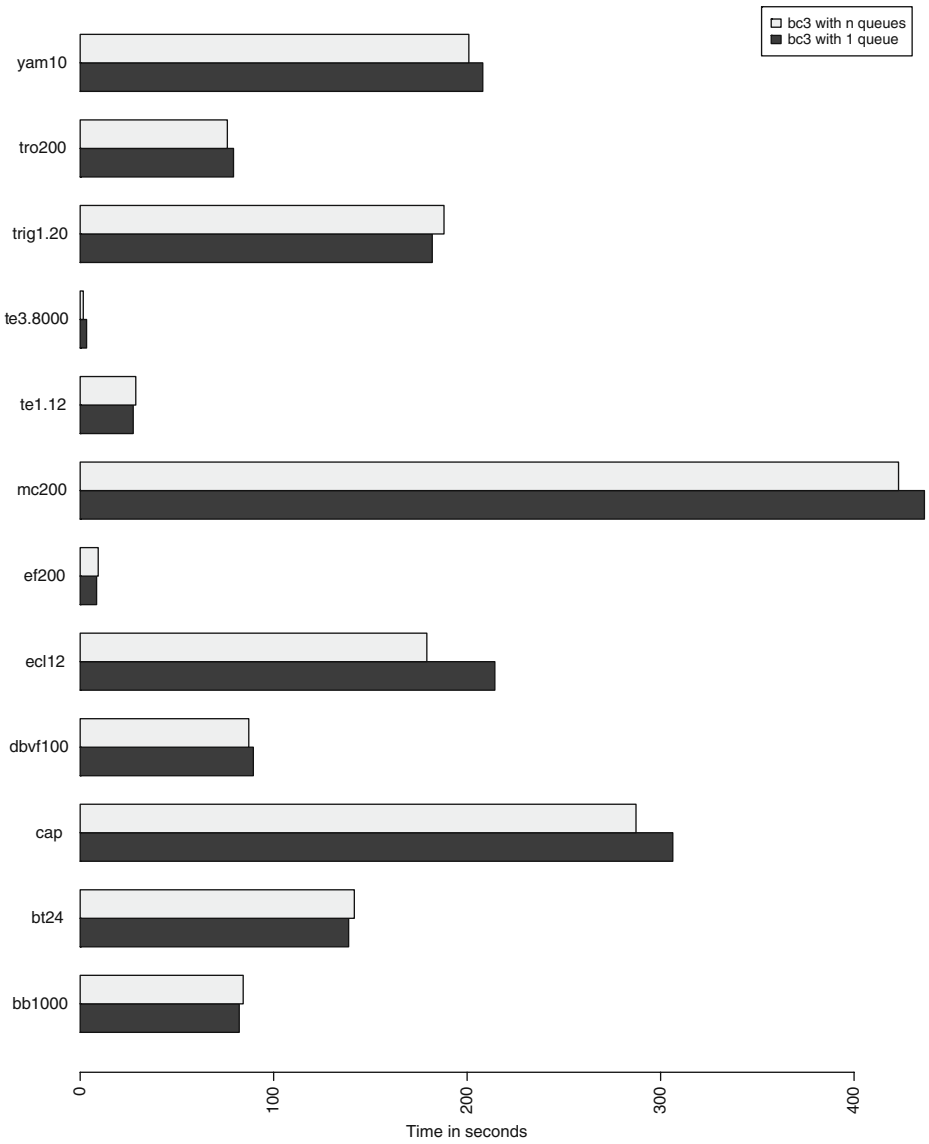

Fig. 5 Comparing bc3 with only one queue and with one queue per variable

3.2 Setting up bcrl

In order to obtain a fully defined algorithm for bcrl, we need to set two interdependent parameters: the value of α and the value of the initial weights.

According to Eq. 4, $W^{(ij)}$ is a weighted average of the past payoffs and of the initial weight $W_0^{(ij)}$:

$$W_k^{(ij)} = (1-\alpha)^k W_0^{(ij)} + \sum_{l=1}^{k} \alpha(1-\alpha)^{(k-l)} r_l^{(ij)} \tag{5}$$

Consequently, for a small α (e.g., $\alpha = 0.1$), the weight $\alpha(1-\alpha)^{(k-l)}$ of the payoffs will decrease only slightly with their age, with the exception of the initial "payoff", whose weight remains important. By contrast, with a large α (e.g., $\alpha = 0.9$), the weights of the payoffs decrease fast with their age, with the most recent payoff being much favored.

With a large α, a projection may see its weight plunge the first time it performs badly, while the aftermath of such an event would be dampened with a small α by the cumulative effect of its past history. On the other hand, the use of a small α requires extra care when initializing the weights W_0. In any case, a consequence of the weight update formula (5) is that the initial weight W_0 may be an important bias of the rwa method.

Without further information, we first decide to initialize all weights to 1, giving equal importance to all projections. Table 1 shows the impact of α on computation time for these initial weights (Algorithm bcrl-1). Both 0.1 and 0.9 seem good contenders for the choice as default values.

Figure 6 presents a comparison of the number of effective and useless calls for bc3 and bcrl-1 for standard test problems to be described in Section 4. For all problems, the lowest bar corresponds to bc3 while the topmosts correspond, from bottom to top, to bcrl-1 for $\alpha = 0.1$ and $\alpha = 0.9$.

Table 1 Incidence of α on computation times for bcrl-1

Problem / α	0.1	0.3	0.5	0.7	0.9
bb10000	93	68	63	56	**49**
bt24	132	130	152	139	**117**
cap	132	124	125	117	**115**
dbvf100	**44**	**44**	**44**	**44**	**44**
ecl12	291	285	284	**281**	283
ef200	**8**	**8**	**8**	**8**	9
mc200	**674**	749	920	1195	1331
te1.12	**25**	26	26	28	26
te3.8000	**1**	**1**	**1**	**1**	**1**
trig1.20	**82**	**82**	83	85	**82**
tro200	**45**	46	**45**	**45**	**45**
yam10	**46**	47	47	47	47

Times in seconds on an Intel Pentium IV at 3.8 GHz (rounded to the nearest sec.).
Boldfaced time: best time for a benchmark.

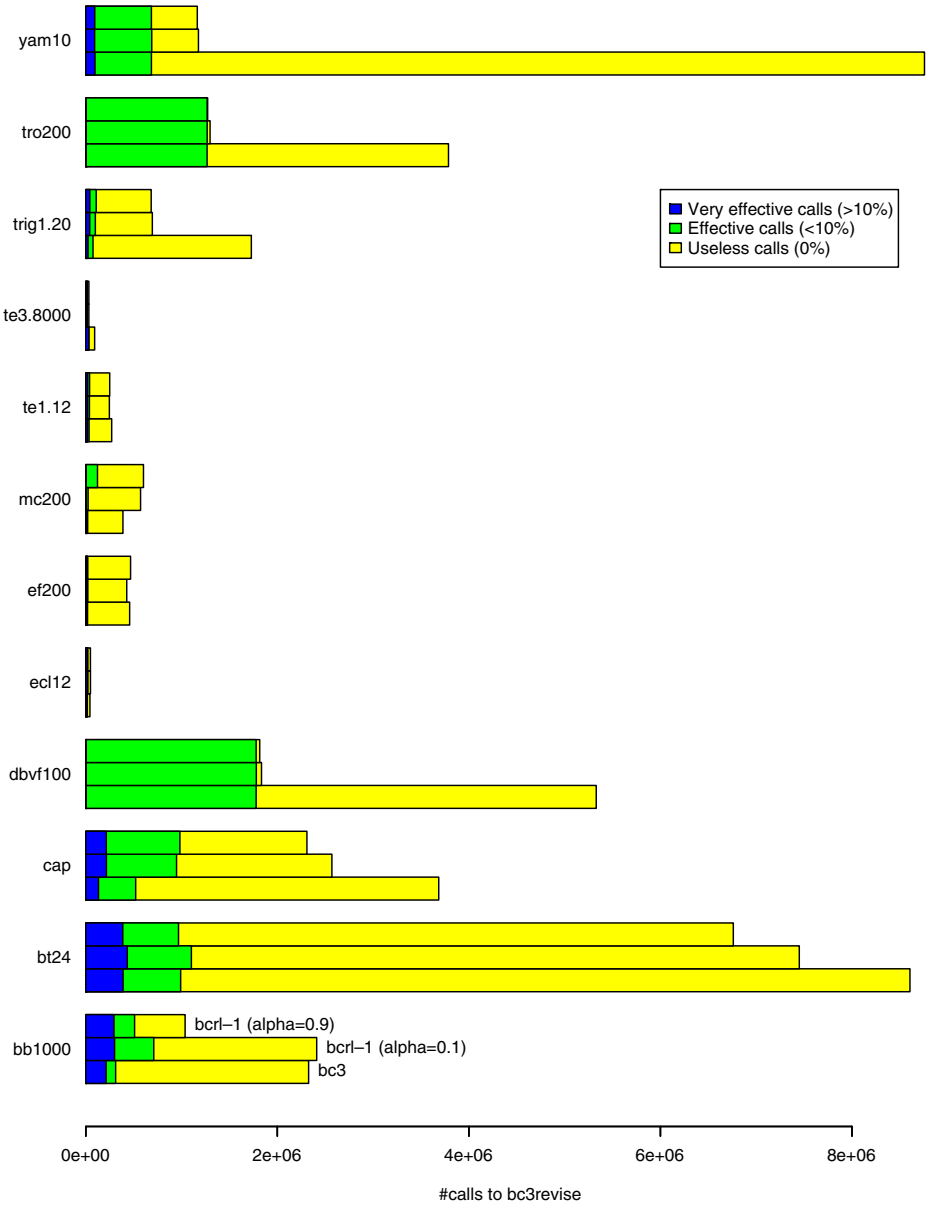

Fig. 6 Effectiveness of bcrl-1 ($\alpha = 0.1$ and $\alpha = 0.9$) vs. bc3

Overall, bcrl-1 reduces the total number of calls to bc3revise (and therefore, the solving time—see Fig. 8, Page 13) for most problems. On closer look, it appears that bcrl-1 requires more calls than bc3 on problem *mc200*, though the number of effective calls is also increased. For this dense problem, the number of projections is large (200^2), and bcrl-1 requires a long time to discover that it possesses a static transversal (see Page 6) because all projections have the same weight initially, and

Table 2 Incidence of α on computation times for bcrl-j

Problem / α	0.1	0.3	0.5	0.7	0.9
bb10000	**34**	**34**	**34**	36	47
bt24	155	147	158	144	**121**
cap	131	126	119	116	**115**
dbvf100	44	44	**43**	**43**	44
ecl12	278	270	275	269	**268**
ef200	**8**	**8**	**8**	**8**	9
mc200	**39**	**39**	**39**	**39**	**39**
te1.12	**25**	**25**	26	26	26
te3.8000	6	**5**	**5**	**5**	**5**
trig1.20	88	**85**	86	87	86
tro200	**45**	**45**	**45**	**45**	**45**
yam10	**46**	**46**	**46**	**46**	47

Times in seconds on an Intel Pentium IV at 3.8 GHz (rounded to the nearest sec.).
Boldfaced time: best time for a benchmark.

are then all considered in turn at least once at the beginning. This effect is worsened with $\alpha = 0.9$ by a perverse side-effect of it selecting effective projections more often than with $\alpha = 0.1$: out of 602199 calls to bc3revise, 122002 (20.25 %) lead to some insubstantial reduction (less than 10 %); by contrast, using $\alpha = 0.1$ yields 571999 calls to bc3revise (only 5 % less than with $\alpha = 0.9$), out of which only 24033 (4 %) lead to some reduction less than 10 %. Each successful call to bc3revise leads to testing whether to include in the propagation queues the projections that depend on the variable reduced (see Line 16 in Algorithm bcrl, Page 8). For a large dense problem such as *mc200*, this process takes a lot of time because there are $200^2 - 1$ projections to consider each time. A solution to this problem is to introduce a so-called *improvement factor* γ and to forbid propagation (that is, to bypass Line 17 in bcrl) whenever the reduction achieved by a call to bc3revise is smaller than γ %. When using an improvement factor of 10 %, bcrl-1 with $\alpha = 0.9$ becomes twice as fast as with $\alpha = 0.1$ on *mc200*. In addition, setting *mc200* aside, the choice of $\alpha = 0.9$ leads to better performances overall than $\alpha = 0.1$.

As a consequence, we decide to favor the reactivity offered by $\alpha = 0.9$, and we choose it as the default value in the rest of this paper.

3.3 Enhancing bcrl with an Initial Guess

In order to achieve good performances even for large dense problems with static transversals, we have to use some information at the beginning of the solving process to preset the weights to favor some projections over others. If our initialization heuristic is good, we expect that the best projections will be used more often than the others from the very start.

The heuristic chosen works as follows: we compute the interval Jacobian $\boldsymbol{J}$ of the system for the initial box and set $W^{(ij)}$ to the sum of the mignitude[5] of $\boldsymbol{J}_{ij}$ normalized

[5] $\mathsf{mig}\, \boldsymbol{I} = \min\{|a| \mid a \in \boldsymbol{I}\}$.

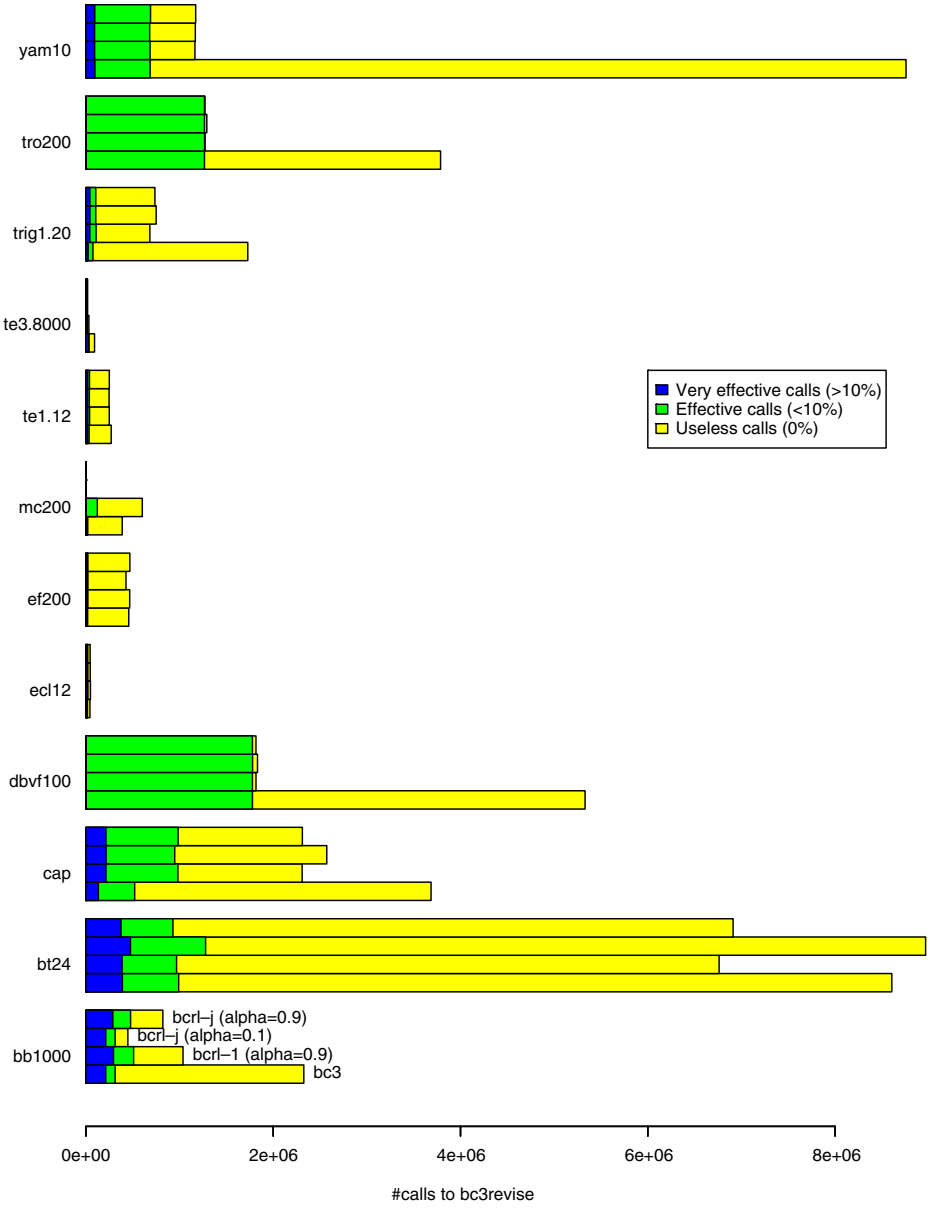

Fig. 7 Effectiveness of bc3 vs. bcrl-j ($\alpha = 0.1$ and $\alpha = 0.9$)

to the range [0, 0.5] and of the magnitude[6] of $\boldsymbol{J}_{ij}$ normalized to the range [0, 0.5]. The weight thus lies in the range [0, 1].

[6] mag $\boldsymbol{I} = \max\{|a| \mid a \in \boldsymbol{I}\}$.

Intuitively, the interval Jacobian indicates the steepness of the projections, i.e. which projections are more likely to allow reducing the domain of their associated variable. The magnitude of an entry in the Jacobian gives a worst-case information on the steepness while the magnitude gives a best-case information. The heuristic chosen mixes both equally.

In the following, times reported always take into account the time spent in computing the initial weights.

The resulting instance of bcrl is called bcrl-j. Table 2 presents the incidence of α for bcrl-j. Note how bcrl-j seems less sensitive to the value of the parameter α than bcrl-1. Indeed, since rwa follows a pure exploitation strategy, enhancing it with a good initial guess yields a more focused exploitation. Since the heuristic is good on our test set, the influence of α is dampened. However, we think it is a better choice to keep $\alpha = 0.9$ as the default value for bcrl-j in case the heuristic would fail on some problem, or when the problem does not have a clear transversal.

Figure 7 graphically compares the number of calls to bc3revise required to solve our set of test problems. The lowest bar corresponds to bc3, the middle bar to bcrl-1, and the topmost bars to bcrl-j. As expected, bcrl-j performs consistently better than bc3 and bcrl-1. For Problem *mc200*, it achieves such a speed-up that it is not even visible on the chart. The reason is that the heuristic clearly identifies the transversal in this problem and initializes the weights accordingly, allowing bcrl-j to purely exploit this transversal. For problems without a clear transversal (e.g. *ef200*), the heuristic does not hinder proper learning.

4 Evaluating bcrl

In order to assess the quality of our new algorithms, we have selected a set of twelve standard nonlinear problems [14] of various sizes, various characteristics (quadratic constraints, polynomial constraints, non-polynomial constraints involving sines, cosines, logarithms and exponentials), and various sparsities (*dense problems*, in which all variables occur in all equations, and *sparse problems*, with only a small subset of the variables in each equation). The characteristics for all these problems

Table 3 Test problems

Name	Code	Size	Sparsity	Constraints
Broyden-banded	*bb10000*	10 000	0.07 %	Quadratic
Broyden tridiagonal	*bt24*	24	10.95 %	Quadratic
Caprasse	*cap*	4	100.00 %	Polynomial
Discrete Boundary Value Fun.	*dbvf100*	100	2.93 %	Polynomial
Extended Crag-Levy	*ecl12*	12	14.58 %	Non-polynomial
Extended Freudenstein	*ef200*	200	1.00 %	Polynomial
Moré-Cosnard	*mc200*	200	100.00 %	Polynomial
Trigexp 1	*te1.12*	12	23.61 %	Non-polynomial
Trigexp 3	*te3.8000*	8 000	0.04 %	Non-polynomial
Trigo1	*trig1.20*	20	100.00 %	Non-polynomial
Troesch	*tro200*	200	1.49 %	Non-polynomial
Yamamura	*yam10*	10	100.00 %	Polynomial

are synthesized in Table 3. The *Size* column indicates the number n of variables and equations (all the problems are square); the *Sparsity* column gives the *sparsity index* defined as the number p of possible projections divided by n^2 (p is equal to n^2 for dense problems such as *mc200* but may be much smaller for sparse problems). The initial domains for all test problems are those given on the COPRIN web page [14].

Figure 8 presents the computation times for the three algorithms considered. All the experiments were conducted on an Intel Pentium IV at 3.8 GHz with 2GB of

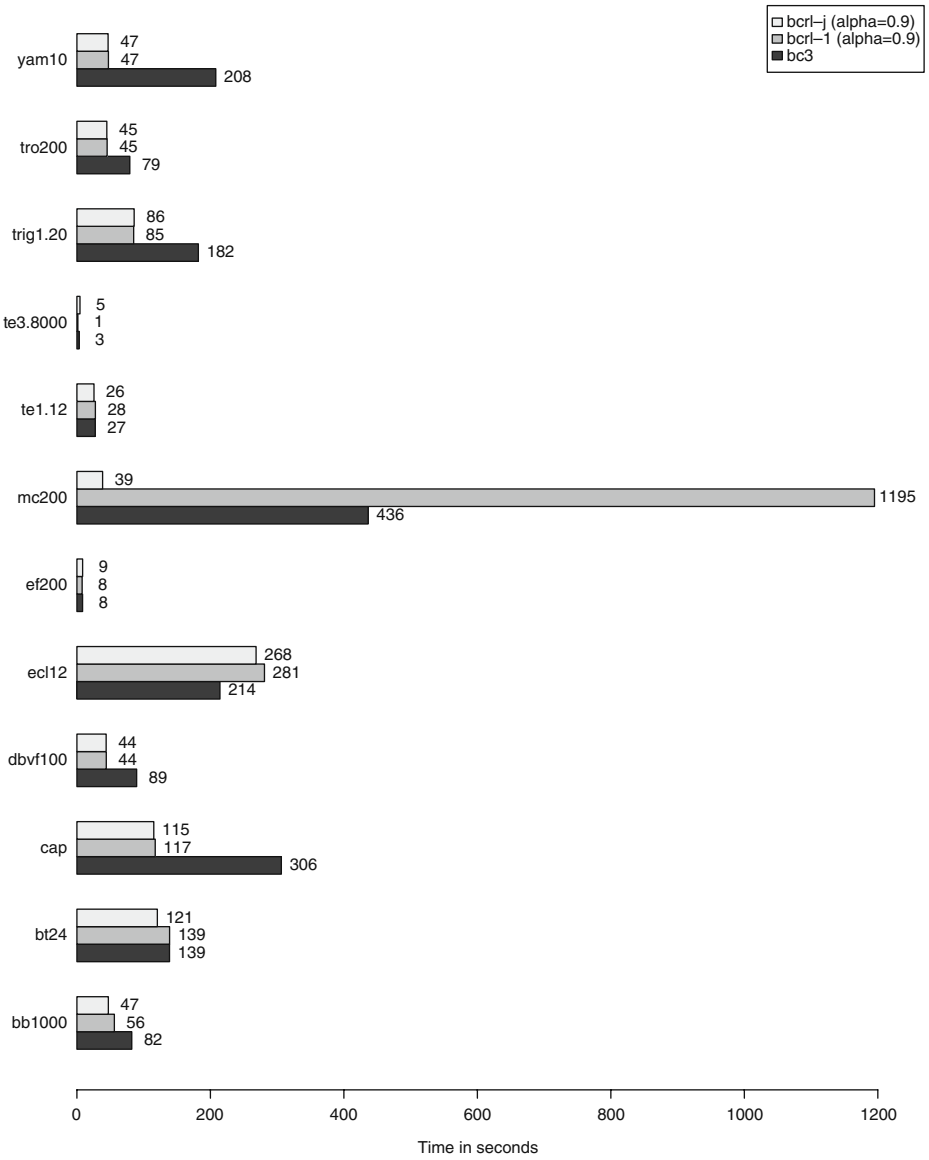

Fig. 8 Solving test problems with and without learning (times rounded to the nearest second)

RAM and a *Standard Unit Time* equal to 50.4 *s* (this is defined as the time required for performing 10^8 evaluations of the function Shekel 5 [5]). The constraint solving environment used is a C++ library written from scratch by the authors. The times reported correspond to the enclosing of all the solutions in boxes of domains whose largest dimension is smaller than 10^{-8}.

As noted previously, bc3 performs poorly on dense problems (e.g., *mc200*) due to the sheer number of projections to consider. The results for bcrl-1 are consistent with the observation made on Fig. 6: the method is often better than bc3 though it can perform poorly on dense problems with a clear static transversal (e.g., *mc200*), spending too much time in exploration. Though not always the fastest method, bcrl-j appears the most regular in its results since it always solves the problems in a time that is close to that of the best method.

5 Related Works

As said previously, there are already well established results for optimizing the resolution of systems of linear real equations with first-order methods that require strict diagonal dominance of the coefficient matrix [6, 23].

The literature dealing with nonlinear systems often revolves around methods that linearize them in order to exploit the results on linear systems: many papers consider an interval Newton-Gauss-Seidel method (aka Hansen-Sengupta's method [10]) that solves the linear system obtained from the local first-order expansion of nonlinear terms obtained with Newton's method by a preconditioned Gauss-Seidel method [15]. Accordingly, a lot of work is then devoted to finding the best preconditioners [16].

Sotiropoulos (sot) et al. [25] have an original approach in this respect: they select a transversal for a polynomial system at the beginning of the computation by looking at the syntactic structure of the equations (variables with the largest degree in the system, for example), and by using numerical considerations only to break ties. In their paper, the static transversal thus obtained is then used in an interval Newton-Gauss-Seidel algorithm.

Another approach uses a Gauss-Seidel-Newton method as presented in the introduction: Herbort and Ratz [11] (hr) compute the Jacobian $\boldsymbol{J}$ of the equation system w.r.t. the initial box $\boldsymbol{D}$, and they select projections according to whether the corresponding entry in the Jacobian straddles zero or not. Their method is not completely static since they recompute the Jacobian after each outer step of Gauss-Seidel. In addition, it theoretically allows for the choice of more than n projections.

In settings more similar to ours, Granvilliers and Hains [9] (gh) try to optimize bc3 by applying bc3revise on all projections at the beginning, and by selecting only the ones that tightened variables' domains the most. The choice of the projections retained is reconsidered whenever some splitting occurs in Algorithm BaP. For problems that lead to a lot of splitting, the method becomes computationally expensive.

Table 4 compares these three methods with bcrl-j. Note that these results should be taken with great care since we used our own implementation of these methods in the same environment as for bcrl-j, and that our only sources of information are the papers in which they were originally presented. With that *proviso* in mind, we see that bcrl-j outperforms hr and gh most of the times. With the exception of problem

Table 4 Comparing related work with solving by learning

Problem	hr	gh	sot	bcrl-j ($\alpha = 0.9$)
bb10000	1760	TO	TO	47
bt24	228	TO	198	121
cap	TO	237	112	115
dbvf100	41	TO	35	44
ecl12	2266	677	NA	268
ef200	TO	329	TO	9
mc200	3	860	TO	39
te1.12	TO	155	NA	26
te3.8000	1	71	NA	5
trig1.20	116	303	NA	86
tro200	48	TO	NA	45
yam10	48	3133	38	47

Times in seconds on an Intel Pentium IV at 3.8 GHz (rounded to the nearest sec.).
TO: Time out (7200s) reached.
NA: Not applicable (problem with non-polynomial expressions).

bb10000 where it takes more than 150 times longer than bcrl-j, the method sot is quite good on the benchmarks it can handle, that is those with strictly polynomial expressions, which should lead us to investigate how to harness the power of its heuristics in our settings. Performance of hr is stunning on *mc200*. We suspect that it is because this problem has only one solution that can be obtained without any splitting.

Lastly, in a somewhat orthogonal approach, Lebbah and Lhomme [17] try to identify cycles in the propagation inside bc3 to avoid slow convergence phenomena. One direction for future research might indeed be to try taking advantage of their methods in bcrl.

6 Conclusion

Reinforcement learning shows all its potential for difficult problems where no static transversal exists since it realizes a good tradeoff between considering all projections equally (bc3) and gambling on n projections only (standard first-order methods). The additional cost incurred by the weights update appears negligible, though the same might not hold for their smart initialization. Experimental evidences tend to show that the approach taken with bcrl-j, sophisticated as it is, still incurs a very reasonable overhead in view of its benefits.

Despite its simplicity, the rwa method as we embedded it in bc3 leads to a robust solving algorithm with reliable performances for all kinds of test problems.

There is still room for improvements nonetheless: as is shown on Fig. 7, despite the amelioration obtained with bcrl-j w.r.t. bc3, there still are problems for which many calls to bc3revise do not lead to any domain reduction (most notably, *bt24* and *ef200*). These problems also require a lot of splitting; this may be because bc3revise is itself not a powerful enough algorithm to tighten the domains for these instances (it may be the case when the numerical expressions involved are ill-conditioned).

We have used reinforcement learning to dynamically select the best projections to consider with bc3revise. There is, however, more potential to our method. In particular, we could reuse the principle at the root of bcrl to select both the

projection and the pruning method to use with. We would no longer assign weights to projections only, but to pairs of *narrowing operators* (projection/pruning method). This would lead to consider more than n^2 operators by creating pairs between the at most n^2 projections and several pruning methods, such as bc3revise, a *unary Newton-Raphson contractor*, or other methods, letting the learning procedure dynamically sift the best narrowing operators.

References

1. Auer, P., Cesa-Bianchi, N., Freund, Y., & Schapire, R. E. (2002). The non-stochastic multi-armed bandit problem. *SIAM Journal on Computing, 32*(1), 48–77.
2. Auer, P., Cesa-Bianchi, N., Freund, Y., & Shapire, R. E. (1995). Gambling in a rigged casino: The adversarial multi-armed bandit problem. In *Proceedings of the 36th annual symposium on foundations of computer science* (*FOCS '95*) (pp. 322–331). IEEE Computer Society Press.
3. Benhamou, F. (2001). Interval constraints, interval propagation. In P. M. Pardalos & C. A. Floudas (Eds.), *Encyclopedia of optimization* (vol. 3, pp. 45–48). Kluwer Academic Publishers.
4. Benhamou, F., McAllester, D., & Van Hentenryck, P. (1994). CLP(Intervals) revisited. In *Proceedings international symposium on logic program* (pp. 124–138). The MIT Press.
5. Dixon, L. C. W., & Szegö, G. P. (1978). The global optimization problem: An introduction. In L. C. W. Dixon & G. P. Szegö (Eds.), *Towards global optimization 2* (pp. 1–15). North-Holland.
6. Duff, I. S. (1981). On algorithms for obtaining a maximum transversal. *ACM Transactions on Mathematical Software, 7*(3), 315–330, (September).
7. Goualard, F. (2005). On considering an interval constraint solving algorithm as a free-steering nonlinear gauss-seidel procedure. In *Proceedings of the 20th annual ACM symposium on applied computing* (*reliable computation and applications track*) (vol. 2, pp. 1434–1438). The Association for Computing Machinery, Inc, (March).
8. Goualard, F., & Jermann, C. (2006). On the selection of a transversal to solve nonlinear systems with interval arithmetic. In V. N. Alexandrov et al. (Eds.), *Proceedings international conference on computational science 2006, Lecture Notes in Computer Science* (vol. 3991, pp. 332–339). Springer-Verlag.
9. Granvilliers, L., & Hains, G. (2000). A conservative scheme for parallel interval narrowing. *Information Processing Letters, 74*, 141–146.
10. Hansen. E. R., & Sengupta, S. (1981). Bounding solutions of systems of equations using interval analysis. *Bibliothek Information Technologie, 21*, 203–211.
11. Herbort, S., & Ratz, D. (1997). *Improving the efficiency of a nonlinear-system-solver using a componentwise newton method*. Research Report 2/1997, Institut für Angewandte Mathematik, Universität Karslruhe (TH).
12. Hickey, T. J., Ju, Q., & Van Emden, M. H. (2001). Interval arithmetic: From principles to implementation. *J. ACM, 48*(5), 1038–1068, (September).
13. IEEE (1990). *IEEE standard for binary floating-point arithmetic*. Technical Report IEEE Std 754-1985, Institute of Electrical and Electronics Engineers, 1985. Reaffirmed 1990.
14. INRIA Project COPRIN: Contraintes, OPtimisation, Résolution par INtervalles. *The COPRIN examples page*. Web page at http://www-sop.inria.fr/coprin/logiciels/ALIAS/Benches/benches.html.
15. Kearfott, R. B., Hu, C., & Novoa, M. III (1991). A review of preconditioners for the interval Gauss-Seidel method. *Interval Computations, 1*, 59–85.
16. Kearfott, R. B., & Shi, X. (1996). Optimal preconditioners for interval gauss-seidel methods. In G. Alefeld & A. Frommer (Eds.), *Scientific computing and validated numerics* (pp. 173–178). Akademie Verlag.
17. Lebbah, Y., & Lhomme, O. (2002). Accelerating filtering techniques for numeric csps. *Artificial Intelligence, 139*(1), 109–132.
18. Luksan, L., & Vlcek, J. (1998). *Sparse and partially separable test problems for unconstrained and equality constrained optimization*. Research Report V767-98, Institute of Computer Science, Academy of Science of the Czech Republic, (December).
19. Mackworth, A. K. (1977). Consistency in networks of relations. *Artificial Intelligence, 1*(8), 99–118.

20. Moore, R. E. (1966). *Interval Analysis*. Prentice-Hall, Englewood Cliffs, N. J.
21. Moré, J. J., & Cosnard, M. Y. (1979). Numerical solutions of nonlinear equations. *ACM Transactions on Mathematical Software, 5*, 64–85.
22. Neumaier, A. (1990). *Interval methods for systems of equations, Encyclopedia of Mathematics and its Applications*, vol. 37. Cambridge University Press.
23. Ortega, J. M., & Rheinboldt, W. C. (1970). *Iterative solutions of nonlinear equations in several variables*. Academic Press Inc.
24. Ratschek, H., & Rokne, J. (1995). Interval methods. In *Handbook of global optimization* (pp. 751–828). Kluwer Academic.
25. Sotiropoulos, D. G., Nikas, J. A., & Grapsa, T. N. (2002). Improving the efficiency of a polynomial system solver via a reordering technique. In *Proceedings 4th GRACM congress on computational mechanics* (vol. III, pp. 970–976).
26. Sutton, R., & Barto, A. (1998). *Reinforcement learning: An introduction*. MIT Press.