# Experimental comparison of BTD and intelligent backtracking: Towards an automatic per-instance algorithm selector

L. Blet[1,3], S. N. Ndiaye[1,2], and C. Solnon[1,3]

[1] Université de Lyon - LIRIS
[2] Université Lyon 1, LIRIS, UMR5205, F-69622 France
[3] INSA-Lyon, LIRIS, UMR5205, F-69621, France
{loic.blet,samba-ndojh.ndiaye,christine.solnon}@liris.cnrs.fr

**Abstract.** We consider a generic binary CSP solver parameterized by high-level design choices, i.e., backtracking mechanisms, constraint propagation levels, and variable ordering heuristics. We experimentally compare 24 different configurations of this generic solver on a benchmark of around a thousand instances. This allows us to understand the complementarity of the different search mechanisms, with an emphasis on Backtracking with Tree Decomposition (BTD). Then, we use a per-instance algorithm selector to automatically select a good solver for each new instance to be solved. We introduce a new strategy for selecting the solvers of the portfolio, which aims at maximizing the number of instances for which the portfolio contains a good solver, independently from a time limit.

## 1 Introduction

Backtracking approaches solve constraint satisfaction problems by building a search tree (or graph). In Chronological BackTracking (CBT) [1], this tree is explored in a depth-first way: When a failure occurs, the search backtracks to the last choice point. CBT is known to explore redundant subtrees when a failure is not due to the last decision (trashing). To overcome trashing, intelligent backtrackings have been proposed, such as Conflict-directed BackJumping (CBJ) [2], Dynamic BackTracking (DBT) [3] and Decision Repair (DR) [4]: They dynamically exploit the structure of the problem to directly backjump to failure causes thus avoiding trashing. Backtracking with Tree Decomposition (BTD) [5] uses a different idea to avoid trashing: It captures the static problem structure by identifying independent subproblems which are solved separately.

CBJ, DBT and CBT have already been experimentally compared (e.g., [6]). BTD has also been experimentally compared to CBT and CBJ (e.g., [5]). However, BTD has never been compared to CBJ and DBT on a wide benchmark, and it has never been compared to DR. It is interesting to compare them as they all exploit structure to guide the search: CBJ, DBT and DR exploit a dynamic structure thanks to explanations, whereas BTD exploits a static structure thanks to decompositions. Furthermore, these backtracking mechanisms may be combined with different constraint propagation mechanisms, such as Forward-Checking (FC) and Maintaining Arc Consistency (MAC), and with different variable ordering heuristics. In particular, [7] proposes to exploit information

about previous states of the search when selecting the next variable to be assigned. In some sense, this heuristic also exploits the structure of the instance to guide the search.

In this paper, we describe a generic CSP solver which has three parameters: (i) the search strategy, which may be instantiated to CBT, CBJ (with or without variable reordering), DBT, DR, or BTD; (ii) the constraint propagation mechanism, which may be instantiated to FC or MAC; and (iii) the variable ordering heuristic, which may be instantiated to minDomain over dynamic degree or over weighted degree.

A first contribution of the paper is to experimentally compare the 24 configurations of this generic solver on a wide benchmark of around a thousand instances. In particular, we compare BTD-based variants with other variants based on intelligent backtracking frameworks. This extensive experimental study shows us that, even though one configuration has better global success rates than all others, some configurations (such as BTD-based ones) which have low global success rates are very good on a large number of instances. In particular, we identify a minimal subset of 13 complementary configurations such that, for every instance of our benchmark, there is always at least one of these 13 configurations which is good for it, i.e., which is not significantly outperformed by any other configuration on this instance.

The next step is to exploit the complementarity of these configurations to improve success rates. This may be done by hybridizing mechanisms. In particular, we have proposed to combine BTD with approaches which dynamically exploit the structure (CBJ and DR) in [8]. Such hybrid approaches are able to solve more efficiently some instances but they are outperformed by some other configurations on other instances. Recent works on portfolios and per-instance algorithm selectors (e.g., [9,10,11,12,13]) have shown us that we may much more significantly improve success rates by learning selection models, which are able to choose a good solver for each new instance to be solved. Therefore, we combine our generic solver with a per-instance algorithm selector. Like other recent approaches, we extract features from instances, and we use machine learning techniques to learn a selection model. A key point is to choose a subset of solvers that may be selected by the selector: The goal is to keep a subset $S$ of solvers with complementary performances so that $S$ contains a solver which performs well on every instance of the training set. We compare two different strategies for achieving this task, called *Solved* and *Good*. The *Solved* strategy maximizes the number of instances solved at a given CPU time limit (ties are broken by minimizing CPU time), as proposed in [12]. The *Good* strategy maximizes the number of instances for which $S$ contains a good solver, and uses statistical tests to decide whether a solver is good for an instance. We experimentally show that this new strategy outperforms *Solved*.

The paper is organized as follows. In Section 2 we describe our generic framework for solving CSPs. In Section 3, we experimentally compare different configurations of this framework. In Section 4, we describe the per-instance algorithm selector and the two selection strategies. In Section 5, we experimentally compare the two selection strategies. We conclude in Section 6 with ideas for some further works.

## 2 Generic framework for binary CSPs

*Background.* A CSP instance is defined by a triplet $(X, D, C)$. $X$ is a finite set of variables. $D$ associates a finite set of values $D(x_i)$ with every variable $x_i \in X$. $C$ is a set of constraints. Each constraint is defined over a subset of variables and defines tuples of values that can be assigned simultaneously to its variables. In this paper, we consider binary CSPs, which only contain binary constraints defined over 2 variables. A solution is an assignment of all variables satisfying all constraints.

We focus on backtracking approaches which structure the assignment space in a search tree (or graph for DBT and DR) whose nodes correspond to variable/value assignments. We introduce a generic algorithm which is parameterized by the backtracking mechanism, the constraint propagation mechanism and the variable ordering heuristic. This generic algorithm allows us to compare in a unified framework state-of-the-art backtracking approaches for binary CSPs. It basically extends the generic algorithm of [6] by adding 3 new backtracking mechanisms (CBJR, DR and BTD).

*Backtracking.* In *Chronological Backtracking* (CBT) [1], the tree is explored with a depth-first search. When a failure occurs, the search backtracks to the last choice point. When the cause of the failure is not due to the last decision, but to an earlier one, CBT explores redundant subtrees. To overcome this issue, *Conflict directed BackJumping* (CBJ) [2] backtracks immediately to the last assigned variable involved in the failure, and unassigns all variables assigned after it. In this study, we use improvements proposed in [14,15] to get a version of CBJ similar to the one in [6]. It maintains for each value unsuccessfully tried the set of assigned variables involved in this failure. Moreover, if this set is empty, the value is permanently removed from the problem.

*Dynamic BackTracking* (DBT) [3] does not unassign variables between the current node and the cause of the failure, but simply backjumps over them since they are not involved in the current failure. Due to the poor performance of DBT combined with FC and a good variable ordering [16], [17] proposes a new version of CBJ combined with a retroactive ordering of already assigned variable (CBJR). After each new assignment, the variable ordering heuristic is used to try to replace the assigned variable higher in the search tree in light of the current state of the problem (for example if many values are removed from the domain of a variable, the ordering heuristic may move up this variable in the tree). Yet, to ensure completeness, the assigned variable cannot be moved before a variable involved in the filtering or failure of a value in its domain.

*Decision Repair* (DR) [18] is a generic framework that generalizes [4] with several parameters. Each instantiation of this framework corresponds to a different hybridization between tree search, local search and constraint propagation. We consider the *DR(mindestroy, uvar)* instantiation of this framework, as proposed in [18]. It performs a depth first search and performs Forward-Checking (FC) at each node of the search. When a failure occurs, the current assignment is repaired by unassigning one variable among those involved in the failure (not necessarily the last) and removing all explanations involving this variable. To ease inconsistency proofs, a variable minimizing the number of removed explanations is chosen randomly. We have extended DR to allow its combination with arc consistency (MAC) instead of FC. Note that DR does not guarantee a complete exploration of the search space.

Finally, *Backtracking with Tree Decomposition* (BTD) [5] uses a tree-decomposition of the constraint graph which captures the problem structure by identifying independent subproblems. BTD computes the order in which the subproblems must be solved, resulting in a partial order on the variables. Moreover, it records *goods* associated with subproblem solutions, and *nogoods* associated with subproblem failures. This information is exploited to avoid solving the same subproblem more than once. In this study, the tree decomposition is computed using the minimum-fill heuristic [19] to triangulate the constraint graph.

*Constraint propagation.* At each node of the search tree, constraints are propagated in order to filter domains and detect local inconsistencies. In this study, we consider two well-known filtering mechanisms [1]: *Forward Checking* (FC), which removes values which are not arc-consistent with the last variable/value assignment; and *Maintaining Arc Consistency* (MAC), which ensures arc consistency of all constraints. For CBJ, DBT and DR, we maintain arc consistency with AC3 [20], whereas for CBT and BTD we use AC2001 [21]. AC2001 considers an ordering of the values in the domains and records for each value its first compatible value in the other domains. If this compatible value is removed, AC2001 searches for a new compatible value starting from the position of the removed one.

*Variable ordering heuristics.* At each node of the tree, the search chooses the next variable to be assigned among the set of all non assigned variables. It uses a variable ordering heuristic to guide this choice. A classical variable ordering heuristic is minDomain, which chooses a variable which has the smallest domain. In this study, we consider two well-known improvements of this heuristic [22,7]: *minDomain over dynamic degree* (d), which chooses a variable $x$ which minimizes the ratio between the size of $D(x)$ and the number of unassigned variables sharing a constraint with $x$; and *minDomain over weighted degree* (w), which chooses a variable $x$ which minimizes the ratio between the size of $D(x)$ and the sum of weights of constraints which involve $x$ with another unassigned variable (where the weight of a constraint is the number of failures it has generated since the beginning of the search).

Note that when the backtracking mechanism is BTD, the variable ordering heuristic is used to choose the next variable within the current cluster of the tree decomposition, and not within the set of all unassigned variables (see [23]).

*Generic framework.* [6] defines a first generic framework that encompasses several state-of-the-art backtracking algorithms. In this study, we have extended this framework with three new backtracking mechanisms, namely CBJR, DR and BTD. From this generic framework, we can obtain configurations denoted by triplets $(b, c, o)$ where $b \in$\{CBT, CBJ, DBT, CBJR, DR, BTD\} defines the backtracking mechanism, $c \in$\{FC, MAC\} the constraint propagation mechanism, and $o \in$\{d,w\} the variable ordering heuristic.

For all configurations, we first decompose the constraint graph into its set of connected components to obtain independent subproblems which are solved independently and consecutively. Also, each subproblem is made arc consistent before starting the solving process.

| Class | #Instances | #Variables | | | #Values | | | #Constraints | | | Constraint tightness | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | min | avg | max | min | avg | max | min | avg | max | min | avg | max |
| ACAD | 75 | 10 | 116 | 500 | 2 | 146 | 2187 | 45 | 691 | 4950 | 0.001 | 0.692 | 0.998 |
| PATT | 238 | 16 | 263 | 1916 | 3 | 66 | 378 | 48 | 4492 | 65390 | 0.002 | 0.795 | 0.996 |
| QRND | 80 | 50 | 220 | 315 | 7 | 11 | 20 | 451 | 2968 | 4388 | 0.122 | 0.578 | 0.823 |
| RAND | 206 | 23 | 37 | 59 | 8 | 36 | 180 | 84 | 282 | 753 | 0.095 | 0.613 | 0.984 |
| REAL | 193 | 200 | 628 | 1000 | 2 | 152 | 802 | 1235 | 6394 | 17447 | 0.0 | 0.519 | 1.0 |
| STRUCT | 300 | 150 | 257 | 500 | 20 | 23 | 25 | 617 | 1641 | 3592 | 0.544 | 0.647 | 0.753 |

**Table 1.** Classes of the benchmark. For each class, the table displays its name, number of instances, number of variables, domain sizes, number of constraints and constraint tightness (ratio of forbidden tuples over number of possible tuples): minimum, average and maximum values.

All configurations are non deterministic: When choosing variables, ties are randomly broken; furthermore, we do not consider any value ordering heuristic and values are randomly chosen.

## 3 Experimental comparison

*Benchmark.* Our benchmark is composed of 1092 instances grouped into 6 classes described in Table 1. The first 5 classes come from the CSP'08 competition. We have only considered the binary instances. If classes contained too many similar instances we only took the first 10 instances. We have removed from the benchmark every instance which has not been solved by any of our 24 configurations within a time limit of 30 minutes among 15 runs for each instance. The last class (STRUCT) contains structured instances which are randomly generated as described in [24]. These instances have a structure similar to RLFAP instances which are real-world instances. This structure is defined by a tree of variable clusters, and the level of structure depends on the density of constraints in clusters and the sizes of the clusters. The class contains subclasses of instances with different levels of structure, sizes and constraint tightness.

*Experimental results.* Table 2 compares the success rates of the 24 configurations at different CPU-time limits on an Intel(R) Xeon(R) CPU E5-2670 0 @ 2.60GHz, 20480 KB cache size, 3GB RAM. As all configurations are non deterministic, we have performed 15 runs for each instance and each configuration. Table 2 also gives success rates of Gecode (with the model proposed in [25]) with 3 different propagation levels: ICL_VAL, ICL_DOM and ICL_DEF. It shows us that our implementation is competitive with Gecode. Of course, our implementation is dedicated to binary CSPs, whereas Gecode is a generic solver which has not be tailored for solving binary CSPs.

(CBT,MAC,w) is the best configuration when considering global success rates. This result is not surprising and has already been observed, for example, in [6]. Without surprise, we also note that configurations which use weighted degrees for ordering variables outperform configurations which consider dynamic degrees (as already observed in [7]). However the gain depends on the considered backtracking mechanism as pointed out in [26]. In particular, using weighted degrees greatly improves the solution process for CBT, DBT and DR whereas the improvement is not so high for CBJ.

| | | | 1 | 5 | 10 | 50 | 100 | 500 | 1000 | 1800 |
|---|---|---|---|---|---|---|---|---|---|---|
| CBT | FC | d | 37.0 | 45.2 | 47.6 | 52.7 | 55.3 | 60.0 | 61.1 | 61.7 |
| | | w | 41.8 | 51.7 | 56.8 | 65.9 | 69.4 | 77.8 | 81.5 | 83.2 |
| | MAC | d | 43.0 | 51.7 | 56.7 | 65.5 | 69.1 | 75.3 | 76.5 | 77.7 |
| | | w | **47.1** | **61.5** | **68.3** | **80.5** | **85.2** | **92.3** | **94.3** | **95.4** |
| CBJ | FC | d | **41.3** | 50.4 | 55.2 | 66.9 | 70.5 | 81.9 | 85.6 | 88.0 |
| | | w | 39.6 | 51.0 | 55.0 | 67.8 | 72.6 | 84.0 | 88.1 | 91.0 |
| | MAC | d | 38.0 | 50.2 | 54.3 | 68.2 | 74.2 | 85.3 | 88.8 | 90.4 |
| | | w | 39.7 | **53.1** | **57.6** | **73.7** | **79.6** | **90.7** | **93.5** | **95.1** |
| CBJR | FC | d | **39.9** | 49.4 | 53.3 | 63.3 | 66.9 | 75.8 | 78.1 | 79.5 |
| | | w | 39.1 | **50.5** | **55.0** | **67.5** | **72.6** | **84.2** | **88.3** | **90.9** |
| | MAC | d | 29.2 | 37.3 | 41.1 | 46.4 | 48.5 | 53.9 | 55.4 | 56.5 |
| | | w | 31.6 | 40.1 | 44.9 | 55.0 | 58.9 | 67.7 | 69.4 | 70.7 |
| DBT | FC | d | 33.8 | 38.0 | 38.8 | 40.8 | 41.5 | 43.9 | 45.8 | 46.7 |
| | | w | 37.7 | 47.4 | 50.5 | 61.9 | 66.5 | 77.0 | 80.3 | 83.7 |
| | MAC | d | 35.8 | 46.2 | 49.4 | 56.6 | 60.0 | 66.6 | 68.0 | 69.3 |
| | | w | **37.9** | **49.5** | **54.1** | **68.6** | **74.5** | **85.7** | **89.5** | **91.8** |
| DR | FC | d | 32.7 | 37.5 | 39.2 | 41.9 | 42.6 | 44.0 | 44.6 | 45.0 |
| | | w | **35.1** | **44.4** | 48.1 | 55.4 | 59.8 | 71.9 | 76.3 | 79.4 |
| | MAC | d | 32.5 | 41.6 | 45.1 | 51.9 | 53.8 | 59.0 | 60.8 | 62.3 |
| | | w | 34.4 | 44.3 | **48.7** | **57.8** | **62.5** | **75.2** | **80.3** | **84.5** |
| BTD | FC | d | 31.4 | 45.1 | 52.2 | 65.1 | 69.2 | 76.1 | 77.3 | 78.0 |
| | | w | 33.5 | 48.0 | 55.5 | 70.2 | 74.9 | 82.1 | 83.9 | 84.5 |
| | MAC | d | 32.8 | 45.1 | 53.9 | 70.1 | 75.8 | 84.6 | 86.0 | 87.1 |
| | | w | **37.4** | **51.3** | **61.9** | **77.6** | **83.3** | **91.9** | **93.6** | **94.2** |
| Gecode ICL_DEF | | | 29.7 | 34.9 | 38.1 | 48.9 | 55.4 | **66.7** | 69.3 | 71.9 |
| Gecode ICL_VAL | | | 27.7 | 32.9 | 35.2 | 45.3 | 51.3 | 63.8 | 67.2 | 70.4 |
| Gecode ICL_DOM | | | **29.9** | **35.8** | **38.9** | **50.9** | **56.6** | **66.7** | **70.5** | **73.4** |

**Table 2.** Comparison by means of global success rates. Each line successively displays the parameters of the solver and the percentage of successful runs at different CPU time limits (in seconds, over 15 runs on 1092 instances). For each backtracking mechanism we highlight in bold the best configuration. We highlight in blue the best over all configurations.

Also, configurations using DBT or DR as backtracking mechanism and FC for propagating constraints perform poorly. It was already hinted that DBT could lead to poor performances in [16].

These global success rates on the 1092 instances of our benchmark hide very different results when we look at each instance separately. In particular, some configurations which have rather low success rates on the whole benchmark are the best performing ones on some instances. We apply a simple rule to decide if a configuration is the *best* for an instance $i$: we first compare the number of successful runs within a 30 minute CPU time limit, and we break ties by comparing the CPU time of the successful runs.

Line (b) of Table 3 displays the percentage of instances for which a configuration is the best among the whole set of configurations. It shows us that, even though (CBT,FC,d) only solves $61.7\%$ instances of the whole benchmark after 30 minutes of CPU time, it

| | CBT | | | | CBJ | | | | CBJR | | | | DBT | | | | DR | | | | BTD | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | FC | | MAC | | FC | | MAC | | FC | | MAC | | FC | | MAC | | FC | | MAC | | FC | | MAC | |
| | d | w | d | w | d | w | d | w | d | w | d | w | d | w | d | w | d | w | d | w | d | w | d | w |
| (b) | **27.7** | 4.5 | 11.3 | 9.2 | 2.0 | 1.2 | 1.0 | 0.7 | 0.7 | 1.1 | 1.6 | 0.9 | 1.4 | 0.4 | 0.7 | 0.1 | 1.0 | 3.4 | 0.5 | 0.2 | 14.2 | 12.3 | 0.7 | 3.1 |
| (b/h) | 17.8 | 2.3 | 8.8 | 11.1 | 1.1 | 0.3 | 0.3 | 1.1 | 0.8 | 1.6 | 0.0 | 0.8 | 0.8 | 0.2 | 0.5 | 0.0 | 0.5 | 2.7 | 0.5 | 0.2 | **23.3** | 18.8 | 1.3 | 5.1 |
| (g/h) | 27.7 | 9.8 | 12.9 | 19.9 | 4.0 | 3.4 | 2.7 | 7.1 | 2.3 | 4.8 | 2.3 | 2.3 | 1.3 | 5.5 | 2.7 | 2.7 | 2.7 | 7.1 | 2.9 | 2.9 | **42.1** | 26.5 | 5.3 | 10.5 |
| (sb/h) | 6.4 | 0.3 | 5.5 | 5.8 | 0 | 0 | 0.2 | 0.5 | 0 | 1.1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1.9 | 0 | 0 | **14.0** | 7.2 | 0.2 | 1.3 |
| (sb2/h) | 6.4 | 0.6 | 6.1 | 7.1 | 1.6 | - | 0.2 | 0.8 | - | 1.4 | - | - | - | - | - | - | - | 2.3 | - | - | **14.6** | 7.9 | 0.3 | 1.3 |

**Table 3.** For each configuration $c$, line (b) gives the percentage of instances for which $c$ is the best configuration; line (b/h) (resp. (g/h)) gives the percentage of hard instances for which $c$ is the best configuration (resp. $c$ is a good configuration); line (sb/h) (resp. (sb2/h)) gives the percentage of hard instances for which $c$ is the best configuration and all other configurations are significantly worse than $c$ (resp. all other configurations except (CBJ,FC,w), (CBJR,FC,d), (CBJR,MAC,*), (DBT,*,*), (DR,FC,d) and (DR,MAC,*) are significantly worse than $c$).

is the best configuration for $27.7\%$ of the 1092 instances. Of course, it is well known that simple configurations like (CBT,FC,d) outperform more complicated configurations on very simple instances, for which there is no need for intelligent but expensive mechanisms, whereas they usually have very poor performance on harder instances. In line (b/h) of Table 3, we have removed easy instances from the benchmark: we consider that an instance is *easy* if it has been solved in less than one second by (CBT,MAC,w), for each of the 15 runs. With this definition, 470 instances of our benchmark are easy, and 622 are more difficult. When focusing on these harder instances, line (b/h) of Table 3 shows us that some configurations (such as those using DBT) only have very few instances for which they are the best.

As several configurations may have close results for a given instance, we also study the number of instances for which a configuration performs well: We consider that a configuration is *good* for an instance $i$ either if it is the best one, or if there is no statistical difference between its 15 runs and the 15 runs of the best configuration for $i$. We used the Student's t-test with $p = 0.01$ to decide whether a configuration is not significantly different from another one on a given instance. Line (g/h) of Table 3 displays the percentage of hard instances for which a configuration is good. Again, we note that configurations which are good for many instances do not always have high global success rates on the whole benchmark. In particular, the two configurations which are good for the largest numbers of instances (i.e., (BTD,FC,d) and (CBT,FC,d)) are far from having the highest success rates in Table 2.

All configurations are good for at least one instance of the benchmark. However, it may happen that some configurations are good only for instances for which other configurations are also good, i.e., some configurations are dominated by other ones. To study this, line (sb/h) of Table 3 displays the percentage of hard instances for which a configuration is the best and all other configurations are significantly worse than it. It shows us that 12 configurations are dominated by the other configurations: (CBJ,FC,*), (CBJR,FC,d), (CBJR,MAC,*), (DBT,*,*), (DR,FC,d) and (DR,MAC,*). Hence, we have removed these configurations from our study, except (CBJ,FC,d): there are 10 instances for which all good configurations belong to the set of 12 dominated configurations; as for these 10 instances (CBJ,FC,d) is good (and the only configuration to be good on those 10

| | Number of hard instances | | | | | | | Sep size | Tree width |
|---|---|---|---|---|---|---|---|---|---|
| | ACAD | PATT | QRND | RAND | REAL | STRUCT | Total | (avg) | (avg) |
| Decompose | 7 | 8 | 1 | 14 | 5 | 177 | 212 | 4,7% | 17,1% |
| Don't decompose | 5 | 50 | 12 | 23 | 77 | 63 | 230 | 25,3% | 31,8% |
| Don't know | 9 | 35 | 5 | 99 | 3 | 29 | 180 | 32,9% | 54,5% |

**Table 4.** Description of the 3 sets of instances. For each set, the table displays the number of hard instances in each benchmark class, and the average tree width and separator size (in percentage of the number of variables) of the tree decomposition.

instances), we keep (CBJ,FC,d). Finally, line (sb2/h) of Table 3 gives the percentage of hard instances for which a configuration is the best and all other configurations except (CBJ,FC,*), (CBJR,FC,d), (CBJR,MAC,*), (DBT,*,*), (DR,FC,d) and (DR,MAC,*) are significantly worse than it. The set composed of the 13 remaining configurations contains a good configuration for every instance. This set is minimal as each of these 13 solvers is the only one to be good for at least one instance.

BTD is very effective on many instances. In particular, (BTD,FC,d) is good on more than 42% of the hard instances, and it is significantly better than all other configurations on more than 14% of the hard instances. However, on some other instances it also performs poorly so that its global success rate is rather low compared to other approaches. In order to give an insight into which instances are better solved by BTD, we partitioned the 622 hard instances in 3 sets:

- The *Decompose* set contains all hard instances which are best solved by one of the 4 BTD-based configurations (i.e., (BTD,*,*)), and for which none of the 9 non BTD-based configuration (i.e., (CBT,*,*), (CBJ,FC,d), (CBJ,MAC,*), (CBJR,FC,w)), and (DR,FC,w)) is good;
- The *Don't decompose* set contains all hard instances which are best solved by one of the 9 non BTD-based configurations and for which none of the 4 BTD-based configuration is good;
- The *Don't know* set contains all other instances.

Table 4 shows us how the instances of the benchmark are distributed into these sets. Many instances of the *Decompose* set come from the STRUCT class, which contains structured instances. This is not a surprise that BTD-based approaches perform better than other approaches on these instances (see, e.g., [5]). As BTD has never been compared with intelligent backtracking approaches, it is interesting to note that BTD outperforms them on many of these structured instances. Only 35 instances of the 2008 competition belong to the *Decompose* set: Many instances of this benchmark do not exhibit static structures that can be exploited by BTD. When looking at parameters of the tree decomposition, we note that instances of the *Decompose* set have a smaller tree width (half the size of the *Don't decompose* set) and a smaller separator size (one fifth the size of the *Don't decompose* set). Instances of the *Don't know* set have large tree width. Actually, when the tree width is close to the number of variables, BTD behaves like CBT as nearly all the variables belong to the same cluster.

On some instances, most notably some *rlfap* instances, the decomposition is good but the instance is in *Don't decompose*. These instances are easy (the solution is found

very quickly by (CBT,FC,d)) but huge (up to 900 variables). Restricting the search to clusters can be detrimental as we forbid the search from going directly to the solution.

## 4 Per-instance algorithm selector

Experimental results reported in the previous section have shown us that the best performing configurations on some instances may have very bad performance on other instances so that they are far from having the best average success rates on the whole benchmark. This illustration of the well-known no-free-lunch theorem motivates our study on a per-instance algorithm selector which aims at selecting a good configuration for each new instance to be solved.

In this study, we do not aim at improving the state-of-the-art of per-instance algorithm selectors such as, e.g., CPHydra [9], ISAC [11] or EISAC [27], but we focus on a key point of these approaches, *i.e.*, the selection of the solvers to be included in the portfolio. Indeed, [12] shows us that better performance may be obtained with smaller portfolios. However, it is also important that the portfolio contains a large enough number of solvers so that there is a good solver for every instance. Experimental results reported in the previous section may be used to definitely remove some solvers from the portfolio: The 11 configurations which are dominated by the 13 other configurations can be removed from the portfolio without significantly changing the performance of a *Virtual Best Selector* (VBS), which always selects the best solver in the portfolio. In this section, we describe and compare two different strategies for selecting a subset of these 13 solvers in the portfolio: the strategy used in [12], and a new strategy. Before describing these two strategies, we describe the basic framework of our per-instance algorithm selector.

### 4.1 Basic framework of the selector

We consider a classical framework similar to the "off-the-shelf" framework of [12]. The idea is to train a supervised classifier by giving a training set of labeled CSP instances to it: Each instance of the training set is described by an input vector of features and is associated with an output label, corresponding to the best solver for this instance. This training phase allows the classifier to learn a selection model. Then, this model is used to select solvers for new instances to be solved, given their input feature vectors.

*Features.* Each CSP instance is described by a vector of features. We consider classical features, similar to those used in [9,12], for example. The main difference is that we also extract features from the tree decomposition, as Table 4 has shown us that the performance of BTD depends on tree widths and separator sizes.

More precisely, we extract the following static features from each instance: Number of variables, number of constraints, size of domains (average and standard deviation), constraint tightness, i.e., ratio of forbidden tuples with respect to all possible tuples in the relation (average and standard deviation), and variable degree in the constraint graph (average and standard deviation). As the constraint graphs of some instances are not connected, we also extract the following features: Number of connected components

in the constraint graph, number of variables in a connected component (average and standard deviation), and number of constraints in a connected component (average and standard deviation). Finally, we also extract features from a tree decomposition which is computed using the greedy algorithm *minFill* of [19] to triangulate the constraint graph: Number of clusters, maximum separator and cluster size, and density of constraints in a cluster (average and standard deviation).

In order to gather more information on the instance to be solved, we also perform a short run on it and extract dynamic features from this run. We have limited the time of this run to 1 second. As (CBT,MAC,w) is the best configuration within this time limit, we have chosen to run (CBT,MAC,w). Furthermore, this configuration allows us to gather information on variable weights (used by the variable ordering heuristic) and the number of values filtered by MAC. We collect the following dynamic features: Number of nodes in the search tree, maximum depth of a node in the search tree, number of failed nodes, number of values removed by MAC (average and standard deviation), and weight of a variable (average and standard deviation). In order to gather insights into the dynamics of the run, we collect these features for 3 time limits, i.e., 0.25, 0.5 and 1 second.

*Training.* Given a portfolio of solvers and a training set $I$ of instances such that each instance $i \in I$ is described by a vector of features and is associated with the solver of the portfolio which is the best for $i$, the goal is to train a classifier to associate instances with solvers. This is a classical supervised classification problem and there exist different well-known approaches to solve this problem [28]. In this study, we have used the Weka library [29,30] to perform this task. We have compared the different supervised classifiers which are implemented in Weka. The best classification results are obtained with *ClassificationViaRegression* with default parameters [31] so that we have used this classifier in our experiments.

Once the classifier has been trained on the training set of instances, we can use it to dynamically choose the best configuration of our generic solver for each new instance to be solved. More precisely, to solve a new instance $i$ we proceed as follows: We first run (CBT,MAC,w) on $i$ with a CPU-time limit of 1 second; if $i$ is not solved within this time limit, we extract static features from $i$, and dynamic features from the run of (CBT,MAC,w); we give these features to the classifier which returns a configuration and we run this configuration on $i$. Note that the time spent to extract the features and classify $i$ is very short (less than 0.1 seconds on average).

### 4.2 Selection of a subset of solvers

A key point of per-instance algorithm selection is to select the solvers to be included in the portfolio. The goal is to select solvers with complementary behaviors so that the portfolio contains a good solver for every instance. We may include in our portfolio the 13 non dominated solvers identified in Section 3. However, the larger the portfolio, the harder the learning task. Therefore, better results may be obtained with fewer configurations, as observed in [12].

We compare two strategies for selecting a subset $S_k$ of $k$ solvers (where $k \in [2; 13]$ is a parameter to be fixed). The first strategy, called *Solved*, is the one used in [12]. It selects in $S_k$ the $k$ solvers which maximize the number of instances solved by a VBS

at the CPU time limit. Further ties are broken by minimizing the solving time of the VBS. The second strategy, called *Good*, selects in $S_k$ the $k$ solvers which maximize the number of instances for which $S_k$ contains a good solver (i.e., a solver which is not statistically different from the best solver for this instance). Further ties are broken by maximizing the number of instances solved by a VBS at the CPU time limit.

For both strategies, finding the optimal subset $S_k$ is NP-hard: It is a set covering problem between solvers and instances, where a solver $s$ covers an instance $i$ if $s$ is able to solve $i$ (for *Solved*) or if $s$ is good for $i$ (for *Good*). In this study, we approximately solve it in a greedy way: Starting from the subset $S_1$, which contains the solver which covers the largest number of instances, we define $S_i$ from $S_{i-1}$ by adding to $S_{i-1}$ the solver which most increases the number of covered instances.

When considering the 15 runs of our 13 solvers on the 622 hard instances, we obtain the following orders:

- Order of selection of solvers with the *Solved* strategy:
  1-(CBT,MAC,w), 2-(BTD,FC,w), 3-(CBJR,FC,w), 4-(DR,FC,w), 5-(CBJ,MAC,w), 6-(CBT,MAC,d), 7-(BTD,FC,d), 8-(BTD,MAC,w), 9-(CBT,FC,d), 10-(CBJ,FC,d), 11-(CBJ,MAC,d), 12-(BTD,MAC,d), 13-(CBT,FC,w)
- Order of selection of solvers with the *Good* strategy:
  1-(BTD,FC,d), 2-(CBT,MAC,w), 3-(BTD,FC,w), 4-(CBT,FC,d), 5-(CBT,MAC,d), 6-(DR,FC,w), 7-(CBJ,MAC,w), 8-(BTD,MAC,w), 9-(CBJ,FC,d), 10-(CBJR,FC,w), 11-(CBT,FC,w), 12-(BTD,MAC,d), 13-(CBJ,MAC,d)

Of course, this order strongly depends on the composition of the benchmark. For example, if we remove half of the STRUCT instances, the *Good* strategy selects (CBT,FC,d) in third position and (BTD,FC,w) in fourth position, while all other positions are unchanged. However, if we remove all STRUCT instances, the order becomes very different and the best BTD-based approach is (BTD,FC,w) and it is selected in fourth position.

Let us note $S_k^s$ the subset which contains every solver whose rank is lower than or equal to $k$ for the strategy $s \in \{Solved, Good\}$, and VBS($S_k^s$) the VBS associated with $S_k^s$. This VBS selects the best solver of $S_k^s$ for each instance to be solved so that any selector built upon $S_k^s$ cannot outperform VBS($S_k^s$).

Table 5 compares the two strategies by means of VBS success rates. Let us first note that VBS($S_k^{Solved}$)=VBS($S_k^{Good}$) when $k = 10$ or $k = 13$ as $S_k^{Solved} = S_k^{Good}$ in these two cases. Also, VBS($S_k^{Solved}$) outperforms VBS($S_k^{Good}$) at the time limit of 1800s, when $k \leq 9$, and both approaches are equivalent when $k \geq 10$. This comes from the fact that the *Solved* strategy maximizes the number of solved instances at the time limit. As a counterpart, VBS($S_k^{Good}$) outperforms VBS($S_k^{Solved}$) for lower time limits or smaller values of $k$. This comes from the fact that the *Good* strategy maximizes the number of instances for which the portfolio contains a good solver, independently from the time limit.

For example, when $k = 4$, the difference between the success rates of VBS($S_4^{Good}$) and VBS($S_4^{Solved}$) is equal to 2.2, 3.5, 2.4, 1.3, and 0.2 when the time limit is equal to 1, 5, 10, 50, and 100s, respectively, whereas it becomes negative after 100s. However, the difference is less important ($-0.4$, $-0.7$, and $-0.7$ at 500, 1000 and 1800s, respectively). Actually, with $S_3^{Solved}$={(CBT,MAC,w), (BTD,FC,w), (CBJR,FC,w)}, a VBS is able to solve $99.4\%$ of the runs, but $S_3^{Solved}$ contains a good solver for only 294 of the

| | 1 | | 5 | | 10 | | 50 | | 100 | | 500 | | 1000 | | 1800 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | *Solved* | *Good* | *Solved* | *Good* | *Solved* | *Good* | *Solved* | *Good* | *Solved* | *Good* | *Solved* | *Good* | *Solved* | *Good* | *Solved* | *Good* |
| 2 | 52.0 | 52.7 | 68.9 | 70.6 | 75.4 | 77.2 | 87.5 | 88.7 | 91.8 | 92.3 | 96.7 | 96.9 | 98.2 | 98.2 | 98.7 | 98.6 |
| 3 | 52.4 | 53.2 | 69.2 | 71.6 | 75.9 | 77.8 | 87.8 | 89.0 | 92.5 | 92.8 | 97.6 | 97.3 | 99.0 | 98.4 | 99.4 | 98.8 |
| 4 | 52.5 | 54.7 | 69.4 | 72.9 | 76.1 | 78.5 | 88.2 | 89.5 | 92.8 | 93.0 | 97.8 | 97.4 | 99.1 | 98.4 | 99.5 | 98.8 |
| 5 | 52.7 | 55.8 | 69.4 | 73.1 | 76.1 | 78.7 | 88.3 | 89.7 | 93.0 | 93.0 | 97.8 | 97.4 | 99.2 | 98.4 | 99.6 | 98.8 |
| 6 | 54.5 | 55.9 | 69.7 | 73.3 | 76.4 | 79.0 | 88.8 | 90.3 | 93.1 | 93.6 | 97.8 | 97.8 | 99.2 | 98.8 | 99.7 | 99.3 |
| 7 | 55.6 | 56.0 | 72.3 | 73.3 | 78.7 | 79.0 | 90.2 | 90.3 | 94.0 | 93.9 | 98.3 | 98.1 | 99.3 | 99.2 | 99.7 | 99.6 |
| 8 | 55.7 | 56.1 | 72.3 | 73.3 | 78.8 | 79.2 | 90.4 | 90.5 | 94.0 | 93.9 | 98.3 | 98.1 | 99.4 | 99.2 | 99.7 | 99.6 |
| 9 | 56.2 | 56.4 | 73.4 | 73.5 | 79.3 | 79.3 | 90.5 | 90.6 | 94.1 | 94.1 | 98.4 | 98.2 | 99.4 | 99.2 | 99.7 | 99.6 |
| 10 | 56.5 | 56.5 | 73.5 | 73.5 | 79.5 | 79.5 | 90.7 | 90.7 | 94.2 | 94.2 | 98.4 | 98.4 | 99.4 | 99.4 | 99.7 | 99.7 |
| 11 | 56.5 | 56.5 | 73.5 | 73.7 | 79.5 | 79.6 | 90.8 | 90.7 | 94.3 | 94.2 | 98.4 | 98.4 | 99.4 | 99.4 | 99.7 | 99.7 |
| 12 | 56.5 | 56.5 | 73.5 | 73.7 | 79.5 | 79.6 | 90.8 | 90.7 | 94.3 | 94.2 | 98.4 | 98.4 | 99.4 | 99.4 | 99.7 | 99.7 |
| 13 | 56.6 | 56.6 | 73.7 | 73.7 | 79.6 | 79.6 | 90.8 | 90.8 | 94.3 | 94.3 | 98.4 | 98.4 | 99.4 | 99.4 | 99.7 | 99.7 |

**Table 5.** Comparison of *Solved* and *Good*. Each line successively displays: the number $k$ of solvers selected in $S_k$ and, for different time limits in seconds, percentages of successful runs of virtual best selectors built upon the sets defined with *Solved* and *Good* (over 15 runs on the 1092 instances). For each ($S_k$,time) couple, we highlight the strategy with the highest success rate.

622 hard instances. When adding new solvers to $S_3^{Solved}$, we only very slightly increase the success rate of the VBS. The solver which most increases the number of solved instances is (DR,FC,w) and it allows us to solve 26 more runs (among 622*15 runs). However, (DR,FC,w) is a good solver for a rather small number of instances and adding it to $S_3^{Solved}$ increases the number of hard instances for which we have a good solver by 24. As a comparison, adding (BTD,FC,d) to $S_3^{Solved}$ would allow us to solve 12 more runs (instead of 26, among 622*15) but it would increase the number of hard instances for which we have a good solver by 168 (instead of 24, among 622 instances).

## 5 Experimental evaluation

*Experimental setting.* We consider the 1092 instances of the benchmark described in Section 3, and the training set is composed of the 622 hard instances of this benchmark. We use a leave-one-out scheme: for each instance $i$ of the benchmark, if $i$ is a hard instance which belongs to the training set, then we remove $i$ from it and we train the classifier on all hard instances but $i$; finally we ask the classifier to select a solver for $i$.

*Comparison of classification rates obtained with different sets $S_k$.* The learnt solver of an instance $i$ is the solver returned by the classifier, and we say that $i$ is *well-classified* if its learnt solver is the best solver for $i$ among the set $S_k^s$ of candidate solvers (or if it is not statistically different from the best solver for $i$ in $S_k^s$). The second and third columns of the left part of Table 6 gives the percentage of well-classified hard instances for *Solved* and *Good*, respectively. It shows us that this percentage decreases when the number of configurations in $S_k^s$ increases, both for *Solved* and *Good*: It decreases from $81.7\%$ and $84.6\%$ with 2 configurations to $65.4\%$ with 13 configurations. However, the left part of Table 6 also shows us that the learnt solvers of instances which are not well

Ranking of the learnt solvers

| | 1 | | 2 | | 3 | | 4 | | 5 | | 6 | | ≥7 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | *Solved* | *Good* | *Solved* | *Good* | *Solved* | *Good* | *Solved* | *Good* | *Solved* | *Good* | *Solved* | *Good* | *Solved* | *Good* |
| 2 | 81.7 | 84.6 | 18.3 | 15.4 | | | | | | | | | | |
| 3 | 78.6 | 77.3 | 12.4 | 16.2 | 9.0 | 6.4 | | | | | | | | |
| 4 | 78.6 | 75.7 | 10.6 | 15.8 | 6.3 | 6.8 | 4.5 | 1.8 | | | | | | |
| 5 | 77.0 | 75.1 | 7.2 | 14.6 | 6.6 | 5.9 | 4.7 | 3.2 | 4.5 | 1.1 | | | | |
| 6 | 73.2 | 71.5 | 6.6 | 13.8 | 6.6 | 7.1 | 3.9 | 5.0 | 5.8 | 2.3 | 4.0 | 0.3 | | |
| 7 | 66.1 | 71.2 | 13.8 | 11.9 | 5.5 | 5.5 | 4.3 | 3.9 | 5.5 | 4.7 | 2.4 | 2.6 | 2.4 | 0.3 |
| 8 | 63.8 | 70.7 | 11.6 | 10.5 | 5.9 | 5.8 | 5.6 | 5.8 | 5.9 | 3.1 | 3.1 | 1.9 | 4.0 | 2.3 |
| 9 | 65.6 | 67.7 | 11.4 | 9.8 | 6.1 | 5.6 | 4.3 | 7.1 | 4.0 | 3.9 | 3.4 | 2.4 | 5.1 | 3.5 |
| 10 | 66.1 | 66.1 | 10.9 | 10.9 | 5.9 | 5.9 | 4.0 | 4.0 | 3.2 | 3.2 | 3.5 | 3.5 | 6.3 | 6.3 |
| 11 | 66.2 | 66.7 | 10.0 | 10.5 | 6.3 | 6.1 | 3.5 | 3.9 | 3.4 | 2.7 | 1.9 | 2.9 | 8.6 | 7.2 |
| 12 | 64.8 | 65.8 | 9.8 | 10.5 | 6.1 | 6.8 | 3.9 | 3.7 | 4.3 | 3.2 | 1.9 | 2.1 | 9.1 | 8.1 |
| 13 | 65.4 | 65.4 | 9.8 | 9.8 | 6.8 | 6.8 | 3.4 | 3.4 | 3.9 | 3.9 | 1.8 | 1.8 | 8.9 | 8.9 |

# good solvers

| | *Solved* | *Good* |
|---|---|---|
| 2 | 36.7 | 54.5 |
| 3 | 36.5 | 56.4 |
| 4 | 37.3 | 61.4 |
| 5 | 38.4 | 67.2 |
| 6 | 41.3 | 66.4 |
| 7 | 59.8 | 67.8 |
| 8 | 58.7 | 69.0 |
| 9 | 64.8 | 67.2 |
| 10 | 65.4 | 65.4 |
| 11 | 65.8 | 66.6 |
| 12 | 64.1 | 65.6 |
| 13 | 65.4 | 65.4 |

**Table 6.** Ranking and goodness of learnt solvers. For each set $S_k$ and each rank $j \in \{1, \ldots, k\}$, the left table displays the percentage of hard instances whose learnt configuration is the j$^{th}$ best among the $k$ configurations in $S_k$. For each set $S_k$, the right table gives the percentage of hard instances for which the learnt solver is a good solver.

classified often correspond to solvers which perform well: Given a set $S_k$ of solvers, and given an instance $i$, we rank each solver of $S_k$ from 1 to $k$ according to its performance on $i$ (the solver ranked 1 being the best one for $i$, and the solver ranked $k$ being the worst one). For example, let us look at the results for $S_{13}$: For $65.4\%$ of the instances, the learnt solver is the best one; for $9.8\%$ of the instances, it is the second best one; for $6.8\%$ it is the third best one; ...; and finally, for $8.9\%$ of the instances it is the seventh best one, or it is worse than the seventh best one.

The fact that the learnt solver is well-classified for an instance $i$ does not necessarily imply that it is good for $i$ (except when $k = 13$): This depends on whether $S_k$ contains a good solver for $i$ or not. The right part of Table 6 displays the percentage of hard instances for which the learnt solver is good (i.e., it is the best among the 13 solvers, or it is not statistically different from the best on this instance). For the *Solved* strategy, this percentage increases from $36.7\%$ with $S_2^{Solved}$ to $65.8\%$ with $S_{11}^{Solved}$, whereas for the *Good* strategy it increases from $54.5\%$ with $S_2^{Good}$ to $69\%$ with $S_8^{Good}$.

*Comparison of success rates.* Table 7 displays the percentage of instances solved at different time limits for the best solver, (CBT,MAC,w), and for the per-instance algorithm selector with different portfolios $S_k^s$ with $k \in [2; 13]$ and $s \in \{Solved, Good\}$, on average over 15 runs. We have used the Student's t-test with $p = 0.01$ to decide whether the 15 success rates at a given time $t$ and a given size $k$ are significantly different for the two strategies and we highlight in blue the best strategy when the test is positive. At one second, all variants of the selector have the same success rate as (CBT,MAC,w) because the selector runs (CBT,MAC,w) during one second before starting the selection process. However, after 5 seconds, all variants of the selector have better success rates than (CBT,MAC,w). The *Good* strategy is significantly better than the *Solved* one when

Success rates of per instance solver selectors (average on 15 runs):

| | 1 | | 5 | | 10 | | 50 | | 100 | | 500 | | 1000 | | 1800 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| k | *Solved* | *Good* | *Solved* | *Good* | *Solved* | *Good* | *Solved* | *Good* | *Solved* | *Good* | *Solved* | *Good* | *Solved* | *Good* | *Solved* | *Good* |
| 2 | 47.1 | 47.1 | 64.2 | 66.7 | 72.1 | 73.9 | 84.7 | 86.5 | 88.8 | **90.2** | 94.4 | 95.1 | **96.0** | 96.3 | **96.6** | 96.8 |
| 3 | 47.1 | 47.1 | 64.5 | 66.5 | 71.9 | 73.8 | 84.4 | 86.0 | 88.6 | 90.1 | 94.3 | 95.3 | 95.6 | 96.2 | 96.1 | 96.6 |
| 4 | 47.1 | 47.1 | 64.8 | 67.8 | 72.1 | 74.6 | 84.7 | 86.3 | 88.7 | 89.9 | 94.4 | 95.4 | 95.7 | 96.3 | 96.2 | 96.8 |
| 5 | 47.1 | 47.1 | 64.9 | 68.4 | 72.0 | **75.1** | 84.5 | 86.3 | 88.6 | 89.9 | 94.4 | **95.6** | 95.8 | **96.5** | 96.3 | **96.9** |
| 6 | 47.1 | 47.1 | 64.3 | **68.7** | 71.5 | 75.0 | 83.5 | **86.6** | 87.7 | 89.7 | 93.2 | 95.1 | 94.6 | 95.9 | 95.2 | 96.4 |
| 7 | 47.1 | 47.1 | 66.7 | 68.2 | 73.1 | 74.5 | 85.0 | 86.1 | 88.4 | 89.4 | 94.0 | 94.8 | 95.0 | 95.9 | 95.7 | 96.5 |
| 8 | 47.1 | 47.1 | 66.0 | 68.2 | 72.7 | 74.7 | 84.8 | 86.2 | 88.0 | 89.7 | 93.7 | 95.0 | 94.8 | 95.7 | 95.6 | 96.3 |
| 9 | 47.1 | 47.1 | 67.8 | 68.2 | 74.0 | 74.6 | 85.1 | 86.0 | 88.4 | 89.4 | 94.0 | 94.7 | 94.9 | 95.5 | 95.5 | 96.1 |
| 10 | 47.1 | 47.1 | 67.9 | 67.9 | 73.9 | 73.9 | 85.3 | 85.3 | 88.9 | 88.9 | 94.3 | 94.3 | 95.2 | 95.2 | 95.7 | 95.7 |
| 11 | 47.1 | 47.1 | 67.9 | 68.2 | 73.9 | 74.3 | 85.3 | 85.5 | 89.1 | 88.8 | 94.4 | 94.5 | 95.3 | 95.3 | 95.7 | 95.7 |
| 12 | 47.1 | 47.1 | 67.8 | 68.1 | 73.7 | 74.4 | 85.3 | 85.8 | 88.6 | 89.2 | 94.5 | 94.7 | 95.4 | 95.7 | 95.8 | 96.2 |
| 13 | 47.1 | 47.1 | **68.5** | 68.5 | **74.5** | 74.5 | **85.8** | 85.8 | **89.2** | 89.2 | **94.6** | 94.6 | 95.7 | 95.7 | 96.3 | 96.3 |

Success rates of (CBT,MAC,w) (average on 15 runs):

| 47.1 | 61.5 | 68.3 | 80.5 | 85.2 | 92.3 | 94.3 | 95.4 |
|---|---|---|---|---|---|---|---|

**Table 7.** Each line displays the size $k$ of the portfolio, followed by success rates of per-instance solver selectors built upon $S_k^{Solved}$ and $S_k^{Good}$ at different time limits (for 15 runs on the 1092 instances). For each time limit and each size $k$, we highlight in blue the cell with the best result if it is significantly better. For each time limit and each strategy $s \in \{Solved, Good\}$, we highlight in bold the highest success rate whatever the size $k$. The last line of the table recalls the success rates of (CBT,MAC,w).

$k \leq 9$, at all time limits. When $k \geq 10$, the two strategies often have results which are not significantly different.

For the *Solved* strategy, the best results are obtained with the largest portfolio, $S_{13}$, up to 500 seconds. After that, the best results are obtained with $S_2$. For the *Good* strategy, the best results are often obtained with a portfolio of 5 or 6 solvers. Figure 1 plots the evolution of the percentage of solved instances with respect to CPU time for the best solver (CBT,MAC,w) and for the selector with $S_5^{Good}$ and $S_{13}^{Solved}$. It also plots results of VBS($S_5^{Good}$) and VBS($S_{13}^{Solved}$).

## 6 Conclusion

We have extended the generic framework of [6] by adding three new backtracking mechanisms (CBJR, DR and BTD), thus defining a unified framework for comparing 24 different configurations corresponding to state-of-the-art approaches. As far as we know, this is the first time that approaches based on tree decomposition (BTD) are extensively compared with other search mechanisms such as CBJ, DBT, and DR, when combined with two different constraint propagation techniques (MAC and FC) and with two different variable ordering heuristics (d and w). Experiments have shown us that although BTD has lower global success rates than the best approaches, it also performs significantly better than them on many instances.

We have used a per-instance algorithm selector to choose a good configuration for each new instance to be solved. This selector is parametrized by the size $k$ of the port-

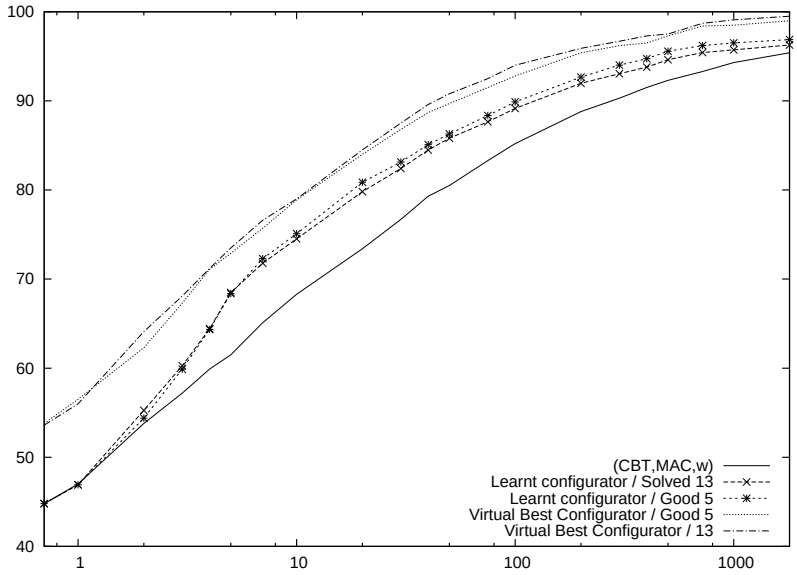


**Fig. 1.** Evolution of the percentage of solved instances with respect to CPU time (in seconds)

folio and we have introduced a new strategy for selecting the $k$ solvers. This strategy is independent from the CPU time limit and aims at maximizing the number of instances for which the portfolio contains a good solver. We compare this strategy with the one used in [12], which aims at maximizing the number of instances solved within a given CPU time limit. We experimentally show that our new strategy allows the selector to solve more instances.

In this first study, we have extracted rather simple features to characterize instances and we plan to study (i) the usefulness of these different features for the classification task and (ii) the possibility of adding new features such as other dynamic features gathered when running other algorithms (e.g., greedy search or local search). We also plan to extend this work to build runtime prediction models by using linear regression techniques, as done for example in SATzilla. This kind of prediction model could then be used to schedule configurations in a portfolio approach, as done for example in CPHydra. Further work will also concern the extension of our generic solver to n-ary constraints and to impact-based or activity-based variable ordering heuristics [32]. Finally, our generic framework allows us to change dynamically the configuration during the solving process. Therefore, we plan to extend our work to dynamic configuration as proposed, for example, in [33] or [34].

*Acknowledgements:* Many thanks to Pierre Flener and Justin Pearson for enriching discussions on this work.

## References


1. Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., New York, NY, USA, 2006.
2. Patrick Prosser. Hybrid algorithms for the constraint satisfaction problem. In *Computational Intelligence*, volume 9, pages 268–299, 1993.
3. Matthew Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, 1993.
4. Narendra Jussien and Olivier Lhomme. Local search with constraint propagation and conflict-based heuristics. *Artif. Intell.*, 139(1):21–45, 2002.
5. Philippe Jégou and Cyril Terrioux. Hybrid backtracking bounded by tree-decomposition of constraint networks. *Artif. Intell.*, 146:43–75, May 2003.
6. Christophe Lecoutre, Frederic Boussemart, and Fred Hemery. Backjump-based techniques versus conflict-directed heuristics. In *Tools with Artificial Intelligence, 2004. ICTAI 2004. 16th IEEE International Conference on*, pages 549–557. IEEE, 2004.
7. Frédéric Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais. Boosting systematic search by weighting constraints. In *ECAI*, volume 16, page 146, 2004.
8. Loïc Blet, Samba Ndojh Ndiaye, and Christine Solnon. A generic framework for solving csps integrating decomposition methods. In *CP doctoral program*, Quebec, Canada, 2012.
9. Eoin O'Mahony, Emmanuel Hebrard, Alan Holland, Conor Nugent, and Barry O'Sullivan. Using case-based reasoning in an algorithm portfolio for constraint solving. In *Irish Conference on Artificial Intelligence and Cognitive Science*, 2008.
10. Lin Xu, Holger Hoos, and Kevin Leyton-Brown. Hydra: Automatically configuring algorithms for portfolio-based selection. In *AAAI*, volume 10, pages 210–216, 2010.
11. Serdar Kadioglu, Yuri Malitsky, Meinolf Sellmann, and Kevin Tierney. Isac-instance-specific algorithm configuration. In *ECAI*, volume 215, pages 751–756, 2010.
12. Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. An empirical evaluation of portfolios approaches for solving csps. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 316–324. Springer, 2013.
13. Daniel J Geschwender, Shant Karakashian, Robert J Woodward, Berthe Y Choueiry, and Stephen D Scott. Selecting the appropriate consistency algorithm for csps using machine learning classifiers. In *Twenty-Seventh AAAI Conference on Artificial Intelligence*, 2013.
14. Fahiem Bacchus. Extending forward checking. In *Principles and Practice of Constraint Programming–CP 2000*, pages 35–51. Springer, 2000.
15. Narendra Jussien, Romuald Debruyne, and Patrice Boizumault. Maintaining arc-consistency within dynamic backtracking. In *Principles and Practice of Constraint Programming–CP 2000*, pages 249–261. Springer, 2000.
16. Andrew B. Baker. The hazards of fancy backtracking. In *AAAI*, pages 288–293, 1994.
17. Roie Zivan, Uri Shapen, Moshe Zazone, and Amnon Meisels. Retroactive ordering for dynamic backtracking. In *CP*, pages 766–771, 2006.
18. Cédric Pralet and Gérard Verfaillie. Travelling in the world of local searches in the space of partial assignments. In *CPAIOR*, pages 240–255, 2004.
19. Uffe Kjaerulff. Triangulation of graphs : Algorithms giving small total state space. Technical report, University of Aalborg, 1990.
20. Alan K Mackworth. Consistency in networks of relations. *Artificial intelligence*, 8(1):99–118, 1977.
21. Christian Bessière and Jean-Charles Régin. Refining the basic constraint propagation algorithm. In *IJCAI*, volume 1, pages 309–315, 2001.
22. Christian Bessiere and Jean-Charles Régin. Mac and combined heuristics: Two reasons to forsake fc (and cbj?) on hard problems. In *Principles and Practice of Constraint Programming–CP96*, pages 61–75. Springer, 1996.

23. Philippe Jégou, Samba Ndiaye, and Cyril Terrioux. Dynamic heuristics for backtrack search on tree-decomposition of csps. In *IJCAI*, pages 112–117, 2007.
24. P. Jégou, S. N. Ndiaye, and C. Terrioux. Strategies and Heuristics for Exploiting Tree-decompositions of Constraint Networks. In *Inference methods based on graphical structures of knowledge (WIGSK'06), ECAI workshop*, pages 13–18, 2006.
25. M. Morara, J. Mauro, and M. Gabbrielli. Solving xcsp problems by using gecode. In *26th Italian Conference on Computational Logic (CILC)*, volume 810 of *CEUR Workshop Proceedings*, pages 401–405. CEUR-WS.org, 2011.
26. Xinguang Chen and Peter van Beek. Conflict-directed backjumping revisited. *Journal of Artificial Intelligence Research*, 14:53–81, 2001.
27. Yuri Malitsky, Deepak Mehta, and Barry O'Sullivan. Evolving instance specific algorithm configuration. In *Symposium on Combinatorial Search (SOCS)*, 2013.
28. Roberto Battiti and Mauro Brunato. *The LION Way: Machine Learning plus Intelligent Optimization.* Lionsolver inc., 2013.
29. Geoffrey Holmes, Andrew Donkin, and Ian H Witten. Weka: A machine learning workbench. In *Intelligent Information Systems, 1994. Proceedings of the 1994 Second Australian and New Zealand Conference on*, pages 357–361. IEEE, 1994.
30. Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. The weka data mining software: an update. *ACM SIGKDD Explorations Newsletter*, 11(1):10–18, 2009.
31. Eibe Frank, Yong Wang, Stuart Inglis, Geoffrey Holmes, and Ian H Witten. Using model trees for classification. *Machine Learning*, 32(1):63–76, 1998.
32. Serdar Kadioglu, Eoin O'Mahony, Philippe Refalo, and Meinolf Sellmann. Incorporating variance in impact-based search. In *CP*, pages 470–477, 2011.
33. Hani El Sakkout, Mark G Wallace, and E Barry Richards. An instance of adaptive constraint propagation. In *Principles and Practice of Constraint Programming–CP96*, pages 164–178. Springer, 1996.
34. Giovanni Di Liberto, Serdar Kadioglu, Kevin Leo, and Yuri Malitsky. Dash: Dynamic approach for switching heuristics. *CoRR*, abs/1307.4689, 2013.