Constraints (2018) 23:87–122
https://doi.org/10.1007/s10601-017-9277-y

How efficient is a global constraint in practice?

A fair experimental framework

Sascha Van Cauwelaert[1] · **Michele Lombardi**[2] · **Pierre Schaus**[1]

Published online: 13 October 2017

Abstract Propagation is at the very core of t can provide signi: it can provide significant performance boosts as long as the search space reduction is not outweighed by the cost for running the propagators. A lot of research effort in the CP community is directed toward improving this trade-off. While experimental evaluation is here of the greatest importance, there exists no systematic and flexible methodology to measure the exact benefits provided by a given (new) filtering procedure. This work proposes such a framework by relying on *replaying* search trees to obtain more realistic assessments. Reducing propagation overhead is done chiefly by 1) devising more efficient algorithms or by 2) using on-line control policies to limit the propagator activations, i.e., mechanisms to reduce the number of propagator calls. In both cases, obtaining improvements is a long and demanding process with uncertain outcome. We propose a method to assess the potential gain of both approaches before actually starting the endeavor, providing the community with a tool to best direct the research efforts. In order to visualize benefits of actual global constraints and the potential of their improvement, we suggest the use of performance profiles. Our approach is showcased for well-known global constraints: ALLDIFFERENT, CUMULATIVE, BINPACKING and UNARY (with transition times).

Keywords Constraint programming · Propagator · Global constraint · Evaluation · Analysis · AllDifferent · Cumulative · BinPacking · Unary resource · Performance profiles

✉ Sascha Van Cauwelaert
sascha.vancauwelaert@uclouvain.be

Michele Lombardi
michele.lombardi2@unibo.it

Pierre Schaus
pierre.schaus@uclouvain.be

[1] Université Catholique de Louvain, Place Sainte Barbe, 2 bte L5.02.01, 1348 Louvain-la-Neuve, Belgium

[2] Università di Bologna, Viale del Risorgimento 2, Bologna, Italy

1 Introduction

Filtering (also known as Propagation) is a key ingredient of CP: it makes a constraint solver capable of pruning large portions of the search space, possibly saving significant exploration times. In practice, the strongest filtering algorithms are not always the winners on every problem. As explained in [52]: maintaining a higher level of consistency takes more time; on the other hand, if more values can be removed from the domains of the variables, the search effort will be reduced and this will save time. Whether or not the time saved outweighs the time spent depends on the problem, the algorithm, its implementation, the search heuristics, and the propagation queue strategy used in the solver.

The CP community has invested a lot of effort to improve this trade-off by researching the most efficient filtering algorithms. As an example, the SEQUENCE constraint was introduced in 1994 [5], but no poly-time Global Arc Consistency (GAC) algorithm was available until 2006 [28]. Then, the original GAC run time of $O(n^3)$ was not low enough to consistently beat weaker (but cheaper) propagators. This motivated improvement efforts that are still ongoing [7, 13, 14]. Other people proposed guarding techniques to reduce the number of times a heavy GAC algorithm is triggered. In the case of guarding propagation, [20] proposes a probabilistic model to estimate if ALLDIFFERENT (bound consistency) will be able to reduce a domain. In [45], the authors determine cases where domain propagators can simply be replaced by lighter bound propagators without increasing the search space.

The trade-off between computation time and pruning power is even more critical for NP-hard constraints. For example, Energetic Reasoning (ER) was proposed as a (powerful) filtering technique for CUMULATIVE in the nineties (see [4, 22]): however, the approach has never been widely employed due to its large run time. Improving the original $O(n^3)$ algorithm took in this case around 20 years [17], while an approach to reduce the overhead by guarding the ER activation with a necessary condition was presented only in 2011 [9].

Not surprisingly filtering is still an important research topic in the CP community. Unfortunately rigorous tools and methodologies to analyze the performance of filtering algorithms for global constraints are missing. This work introduces generic tools and a methodology to probe the potential of filtering techniques and to assess the likely impact of specific improvements (e.g., time complexity, better implementation). Such tools would allow the researchers to focus their efforts in the most promising directions. For example, a researcher may be interested in finding a more efficient way to enforce Generalized Arc Consistency (GAC) for a specific constraint: with the current methodologies, knowing if this line of research is worth investigating remains an open question until a new algorithm is actually devised and evaluated. With the approach we propose, instead, it becomes possible to estimate and bound a-priori the potential effectiveness of a propagator improvement.

Tools to analyze the solving process of CP are not new. Some interesting visual tools have already been introduced. For instance the Oz/Gecode explorer allows visualizing the search tree [43] and interacting with it through a GUI. CP-Viz is a generic visualization platform for CP [51] allowing an advanced post-mortem analysis of the solving process. In CP-Viz the user can visualize each constraint and its filtering, which is very useful for teaching CP or debugging models and constraints. The visual search tree profiling tool introduced in [48] allows comparing search trees visually with convenient navigation techniques, letting the user compare and understand the differences in terms of search space exploration between different configurations and models.

Unfortunately those visualization tools do not allow a fine grained analysis of the time benefits of adding a specific filtering to an existing model for a large set of instances. These

tools do also not allow evaluating what would be the benefit of reducing the computation time of a specific filtering procedure.

The notion of *global constraint* was formally defined a decade ago [12] through a comparison with its decomposition into simpler constraints. Several definitions are provided depending on the considered perspective. Intuitive descriptions for each perspective are:

- A constraint is global if no such decomposition exists (*expressiveness*).
- A constraint is global if its filtering algorithm is strictly stronger than the decomposition, at least in some cases (*quality of filtering*).
- A constraint is global if its filtering algorithm has a strictly better time and/or space complexity, compared to the filtering done by the decomposition (*computational efficiency*).

Those definitions provide an elegant framework to theoretically compare a global constraint with a decomposition, but do not explain how to assess the practical benefits provided by a global constraint on a set of representative instances. One of the contributions of this work is a practical and rigorous framework as an attempt to fill in this gap.

For preliminary analysis, standard profiling tools already allow discovering the fraction of time spent in each propagator, making it possible to estimate roughly potential speedups. This work (an extended version of [56]) goes one step further and tries to answer those questions by proposing a methodology and visual analysis tools inspired by performance profiles [19].

A typical approach to compare different filtering algorithms consists in measuring time and number of backtracks with respect to a baseline approach, on a set of benchmark instances that are solved to completeness. This allows assessing the propagator performance, but provides little or no information on the consequences of its speed-up. It is also common to use static search strategies (e.g., fixed variable heuristic, min value) to make the evaluation fair and rigorous since a stronger filtering has a guarantee to explore a reduced search tree. The rationale behind static strategies usage is that the search nodes order is known a priori and is not influenced by the current solver state, eg., by current domains filtered by the evaluated algorithms. A first drawback of this approach is the risk to bias the analysis, since dynamic strategies are often preferred in practice. Second, instances that can be solved to completeness (required to ensure that the same search space is visited) are generally small, which may not be the case for real applications. Third, differences in the complexity of filtering algorithms become more relevant as the instance size grows: therefore, being forced to focus on small instances may lead to misjudging the performance gap between different propagators.

Contributions We propose to extend the traditional evaluation approach with two main contributions:

1. A method to compare propagators in a principled fashion, by storing and *replaying* search trees (see [56]), in order to enable fair comparisons with arbitrary search strategies and instance sizes. Its main asset is that it enables to measure the exact impact of a propagator on the solving of a given problem. Shishmarev et al. already noticed the importance of replaying, in the context of search tree visualization [48, 50] and better understanding of learning solvers behavior [49].
2. A simple model to evaluate the potential for improvement that a propagator has. This is achieved by instrumenting the solver to collect information about the constraint whose potential is to be evaluated.

Paper outline This paper first motivates the need for our framework. It then introduces our replay technique used to make a fair evaluation of filtering procedures, and describes how to implement it. We then propose our simple approach to assess the impact of a propagator improvement. For ease of access, our method has been made an integral part of the OscaR solver [37], and Section 5 explains how the method can be used. We finally give a case study about different propagators for several constraints using our evaluation approach: ALLDIFFERENT, CUMULATIVE, BINPACKING and UNARY with transition times. As for the impact of propagator improvements, we focused on CUMULATIVE and the Revisited Cardinality Reasoning for BINPACKING. Applying our method allowed obtaining valuable insights: for example, we found that (somehow counter-intuitively) Energetic Reasoning cannot provide improvements on a number of typical scheduling instances, *even if the run-time of the propagator is reduced to zero*. Conversely, investigating different, complementary forms of filtering for CUMULATIVE has a much greater potential. We also observed that changing the search strategy may have a significant impact on the effectiveness of some propagators.

2 Motivation

This section provides the motivation that impelled us to propose our approach. We wish to design methods that allow a thorough analysis of the behavior of propagators in CP and to understand their potential. More precisely, we want to characterize exactly how much search space reduction and time gain is provided by the additional use of a given propagator. This is usually done by comparing the execution with and without the evaluated propagator. Currently, the comparison is made using dynamic and static search strategies. Both methods have their merits, but also substantial limitations:

- Dynamic strategies have a significant impact on which part of the search space is visited first, sometimes providing tremendous solving speed-up as compared to static strategies. They therefore intuitively make for more realistic evaluations since they are the ones programmers use in practice. Nevertheless, they allow poor control and limited insights in the behavior and potential of the propagator itself, since it is impossible to quantify how much additional filtering and search decisions were influenced by each other.
- On the other hand, static strategies allow measuring exactly the search space reduction provided by a propagator. However, they are rarely used in practice because they generally provide poor solving time performance. At the same time, dynamic and static search strategies perform differently, so an evaluation of a propagator with a static strategy can hardly be generalized to the usage with a dynamic strategy. This makes the comparison with static strategies somewhat artificial, since it is not representative of practical usage.

We argue that the designers of global constraints are currently missing an additional methodology that enables the same degree of control and ease of analysis of static strategies, in an experimental setup that is almost as realistic as that of dynamic strategies. In our opinion, as a key feature, such an approach should retain the ability of static strategies to distinguish clearly the benefits that are provided by inference and those that come from the search strategy. One of the main contributions of this work is a framework to perform global constraints evaluation that owns these important characteristics.

Section outline The motivation for our approach is presented as follows: we start by introducing a formalism to describe the search process in a CP solver, and we identify the conditions that enable one to measure precisely the impact of inference on the search performance. We use the formalism to discuss the current evaluation methods and to point out their limitations. Finally, we show how the approach we propose fills in the gap between the traditional evaluation methods, and allows the designer of a global constraint to obtain even more insights in the algorithm behavior and its potential.

2.1 Search formalization

We call a *model M* a set of filtering procedures[1] that allow one to solve a given problem p in a sound and complete manner using search. $M \cup \phi_1 \ldots \cup \phi_n$ denotes the model M where the set of filtering procedures $\{\phi_1, \ldots, \phi_n\}$ are used additionally in the fix point algorithm.

As described in [54], in a Depth-First Search backtracking algorithm, a node $p = \{b_1, \ldots, b_j\}$ in the search tree is identified by a set of branching constraints where b_i, $1 \leq i \leq j$ is the branching constraint posted at level i of the search tree. A node p is extended by adding the k branches $p \cup \{b^1_{j+1}\}, \ldots, p \cup \{b^k_{j+1}\}$ for some branching constraints b^i_{j+1}, $1 \leq i \leq k$.

The branches are often dynamically ordered using a heuristic, with the left-most branch being the most promising. To ensure completeness, the constraints posted on all the branches from a node must be exhaustive (for efficiency reasons, they are typically also mutually exclusive). Usually, branching strategies consist in posting unary constraints (e.g., $X \leq a$ and $X > a$) or binary constraints (e.g., $X \leq Y$ and $X > Y$). In this case, a variable ordering heuristic is used to select the next variable to branch on and the ordering of the branches is determined by a value ordering heuristic.

Definition 1 A **branching procedure** is a function that, given a search node $p = \{b_1, \ldots, b_j\}$ at level j of the search tree, computes the branching constraints at the next level: $\beta(p) = \langle b^1_{j+1}, \ldots, b^k_{j+1} \rangle$. The branching constraints are contracting [6], i.e., domains can only be reduced. Moreover, we assume that after imposing a constraint, at least one domain is reduced. Finally, the procedure has implicitly access to the current state of the constraint store (not written explicitly for notation brevity).

Example 1 β_{ff} is the first-fail binary branching procedure. On the left branch, it assigns the variable with the smallest domain cardinality to its smallest value, and it removes this value from the variable domain on the right branch. Formally, it returns two branching constraints c and $\neg c$, with $c \equiv \arg\min_{x \in X}(|D(x)|) = \min(D(x))$, where X is the set of current unbound decision variables and $|D(x)|$ is the cardinality of the domain of x.

Definition 2 A Constraint Branching Tree (CBT) t rooted at node p is a (possibly empty) ordered finite sequence of pairs $(\langle b^1, t^1 \rangle, \ldots, \langle b^i, t^i \rangle, \ldots, \langle b^k, t^k \rangle)$, where $\beta(p) = \{b^1, \ldots, b^k\}$ are the branching constraints (returned by a branching procedure) and t^i is a CBT for the node $p \cup b^i$. Intuitively, it can be thought of as a tree with branches labeled with

[1] The terms *constraint* and *propagator* are also used interchangeably in this paper.

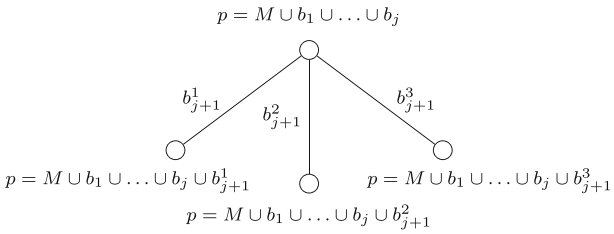

Fig. 1 Example of a CBT

branching constraints, to be traversed with a Depth-First Search. The definition of branching procedure ensures the structure is finite. An empty CBT is written (). $\mathbb{T}$ is the set of all CBTs. An example of a CBT is given in Fig. 1

2.2 Evaluation of global constraints

Formally, we consider the problem of evaluating a filtering function ϕ that maps a set of domains $D_1, \ldots D_n$ to a second set of domains $D'_1, \ldots D'_n$ such that $D'_i \subseteq D_i$.[2] In practice, ϕ may represent a propagator for enforcing GAC or a domain-specific consistency level (e.g., Energetic Reasoning), or it can be some kind of meta-propagation scheme such as Singleton Arc Consistency [10].

Like many other approaches, we measure the performance of the algorithm to compute ϕ by comparing the time needed to solve a target CSP using a *baseline* model $M \cup \phi_M$ and an extended model $M \cup \phi$. We call ϕ_M the *baseline filtering function* (e.g., a decomposition of a global constraint) that is to be replaced in the extended model by the algorithm ϕ that we want to evaluate. We also require the property that ϕ_M is subsumed by ϕ, i.e., ϕ performs *at least* the same deductions as ϕ_M (for instance, in the case of an AllDifferent constraint, ϕ_M can be a binary decomposition while ϕ would be the GAC filtering).

Notice that it often happens that we wish to evaluate the use of a stronger propagator while keeping weaker algorithms in the propagation queue with a higher priority, as stronger propagation can come at the cost of a higher time complexity. This also has to be done if we want to compare ϕ with a baseline filtering function ϕ_M that ϕ does not subsume. In this case, we compare the model $M \cup \phi_M$ with $M \cup \phi_M \cup \phi$, i.e., we construct the extended model by *adding* ϕ to the baseline model instead of replacing ϕ_M by ϕ. To reduce the notation, when ϕ is added to the baseline model instead of replacing ϕ_M, one can denote the baseline model by M and the extended model by $M \cup \phi$.

To ensure the measured difference between the two models is only due to propagation alone, two conditions must be respected:

C.1 The two runs must explore the same search space;
C.2 All search nodes (and therefore the solution nodes) that are visited by both runs are visited in the same order.

[2] Some particular CP approaches, such as the ones using Decision Diagrams [8], perform inference on internal data structures. But in the end, potential partial assignments are removed by propagation (e.g., by removing an arc of a Decision Diagram) and the inference made on Decision Diagrams can always be projected to variable domains.

The first requirement is always met as long as $M \cup \phi_M$ and $M \cup \phi$ are semantically equivalent (i.e., they have the same solutions) and the problem is solved to completeness (feasibility or optimality). Without the second requirement, $M \cup \phi_M$ or $M \cup \phi$ could get an unfair advantage if the search strategy quickly allows hitting a feasible solution (and stops, for feasibility problems), or a high-quality solution (and gets a good bound, for optimality problems). The next section discusses existing evaluation methods from the perspective of those two requirements.

2.3 Current evaluation methods and their limitations

There exist two families of search strategies in CP: *static* and *dynamic* strategies. For static strategies, also known as lexicographic-order search strategies, the order in which the branching constraints are posted is (implicitly) known prior to the search process, meaning that the pruning happening at a search node has no effect on this order. An example is a branching procedure that always returns two constraints C and $\neg C$, such that C assigns, according to a fixed variable order that is known a priori, the first unassigned variable to its smallest domain value. On the other hand, dynamic strategies do take the search state into account when creating the branching constraints. They are an essential asset of CP to get good performances. An example of dynamic strategy is the branching procedure described in Example 1.

While static and dynamic strategies are important, they both have strong limitations from an evaluation perspective, as we argue hereafter.

Dynamic strategies can be used for evaluation while respecting condition **C.1** under the condition that instances are solved to completeness. This is already a strong limitation, since it prevents evaluation on large-scale instances. Moreover, dynamic strategies cannot guarantee that condition **C.2** is satisfied in general. Let us illustrate with a first hypothetical example how this can lead to unfair conclusions (see Fig. 2). On the left tree, a model $M \cup \phi_M$ is used, and the branching procedure returns the branching constraints $\langle C, \neg C \rangle$ at the root node. This leads the search toward a region where the only solution of the problem (found in the green node) lies. Let us now consider the right tree: the same branching procedure is used but the model is extended to $M \cup \phi$, such that more domain pruning occurs at the root node. It is possible, due to the dynamic nature of the search strategy, that the branching constraints are swapped so that the constraint $\neg C$ is imposed on the left branch. The whole left subtree will have to be traversed before the correct search region can even be considered. It would be unfair to impute this bad performance to ϕ solely, since the propagator was in fact able to prune (perhaps significantly) more, and could help in practice with a slightly different context (e.g., with a different search strategy). When using dynamic strategies, the unpredictable effects of combined search and propagation prevent measurement of benefits/harm induced by the additional pruning.[3] The effects of ϕ should be isolated to really quantify the amount of pruning that a new propagator can perform.

Let us insist on this point with a real example. Example 2 illustrates how using dynamic strategies to evaluate a global constraint can lead to unfair conclusions.

[3] Notice that a similar reasoning could be done for another example where the left and right tree are attributed to $M \cup \phi$ and $M \cup \phi_M$, respectively.

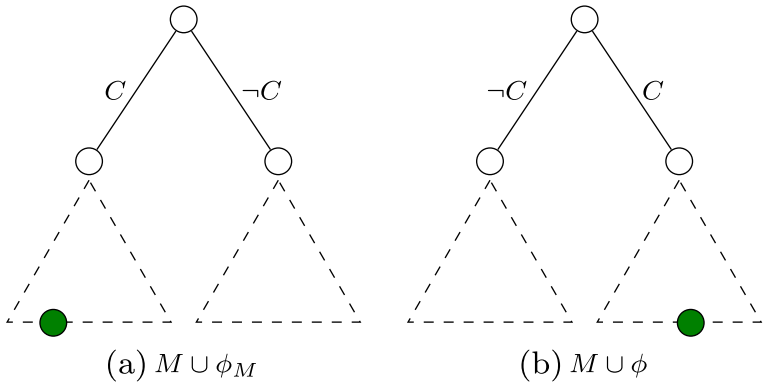

Fig. 2 Comparison of searches by two different models $M \cup \phi_M$ and $M \cup \phi$ with the same dynamic search strategy

Example 2 Consider the first *BL* instance [3] with 20 activities for the Resource Constrained Project Scheduling Problem (RCPSP). The Time-Tabling propagator and Energetic Reasoning Checker [4] are used for the CUMULATIVE constraint in the baseline model M. If the branching procedure β_{ff} of Example 1 is used, 100 nodes are required for finding the optimal solution. However, if the Energetic Reasoning Propagator ϕ is used **additionally** (i.e., together with the Energetic Reasoning Checker and Time-Tabling propagator, and hence additional pruning is possible in $M \cup \phi$), then 124 nodes are required. While the model $M \cup \phi$ has been able to prune more, more nodes were required to find the optimal solution. The blame for this counterintuitive behavior is on the interleaving of propagation and branching, rather than on the Energetic Reasoning propagator itself. Indeed, in this case, more propagation occured at a certain node, and the branching procedure generated a different sequence of branching constraints such that a behavior similar to the one illustrated in Fig. 2 happened. So, if the solution time is used as a metric for the evaluation, in such a case a clear non-beneficial bias[4] would apply against Energetic Reasoning. Notice an opposite bias is also possible.

Static strategies allow ensuring that condition **C.2** is fulfilled, by definition. Condition **C.1** is easily respected, by solving instances to completeness, possibly after having reduced the search space by adding constraints at the root node. This explains why researchers sometimes use those strategies to evaluate a global constraint. However, a major issue with that approach is that static search strategies are rarely used in practice, since they are often outperformed by dynamic ones. So the obtained conclusions may be biased, as the propagator will rarely be used in that context.

With Example 3, let us briefly showcase that static strategies do not line up with reality.

Example 3 We consider the same comparison made in Example 2. Let us use a lexicographic branching strategy on the starts of the activities: on the left branch, the activity start is assigned to its minimum value, on the right branch this value is removed from its domain. Without Energetic Reasoning, one obtains 20081 nodes to prove optimality, while with the

[4] In principle, one might remove most of the bias by using statistics, i.e., by running experiments with many instances and many search strategies. However, this is an expensive process and it is not always viable.

additional propagation one only requires 8789 nodes. Since the strategy is static, we need less nodes with more pruning, as expected. However, in Example 2, we made the opposite conclusion with a dynamic strategy. More importantly, the results reported here are quite different from those a user would get in practice: both approaches require much more nodes and the ratio of the number of nodes required by $M \cup \phi$ by the number of nodes required by M goes from 1.24 to ~ 0.44.

Finally, Example 4 should convince the reader that when evaluating a filtering algorithm, one may get opposite outcomes depending on wether a static or a dynamic search strategy is used.

Example 4 Let us consider the 15th *BL* instance [3] with 25 activities for the RCPSP. We wish to evaluate if adding Energetic Reasoning to solve the instance is beneficial. Let us first use the static strategy of Example 3. The required number of nodes to solve the instance without Energetic Reasoning is 193038 while only 24140 nodes are necessary if it is used additionally. Moreover, it takes ~ 16 and ~ 13 s[5] to solve the instance without and with the additional propagator, respectively. One could therefore conclude that Energetic Reasoning should be used. However, if we make the same comparison using the *SetTimes* dynamic strategy from [32], the results differ: only 51754 (respectively 18064) nodes are required without (respectively with) Energetic Reasoning. The ratio is therefore quite different ($24140/193038 \simeq 0.125$ as compared to $18064/51754 \simeq 0.349$), but more importantly, the solution time is ~ 1.5 s in the first case and ~ 4 s in the latter case. We would therefore consider in this case that Energetic should *not* be used to solve the instance. This illustrates that there is a need to line up with dynamic strategies that are actually used in practice when we perform the evaluation of a global constraint, since static branchings can lead to opposite conclusions. Although illustrated on a single instance, this phenomenon is not rare on a complete benchmark suite.

2.4 Aim of this work

In this paper, we suggest a framework to evaluate a global constraint ϕ while respecting conditions **C.1** and **C.2** so that any difference in execution can only be attributed to additional propagation provided by ϕ. This point was already raised in [56] and then in [48, 50]. At the same time, we wish for the ability to use search strategies that are as close as possible to those actually employed in practice, and we want to keep the possibility of using large instances, so we wish to avoid to force completeness to ensure condition **C.1**.

In addition, to better understand the potential of improving propagators from a computation time point of view, we suggest the simple yet very informative concept of *fictional propagator*. In brief, they allow estimating the impact on the solving time of a problem, assuming that a propagation time improvement has been found (e.g., reduction of the time complexity of a given propagator).

Notice that the approach we propose is not necessarily meant to replace existing ones. We simply think that our framework can strengthen the conclusions of an evaluation. But one could use our framework together with traditional evaluations: for instance, by comparing the results we obtain with those of a traditional evaluation based on dynamic strategies, it is possible to measure the effect of the interplay between the stronger inference and the search.

[5] With a 2.2 GHz Intel Core i7 processor.

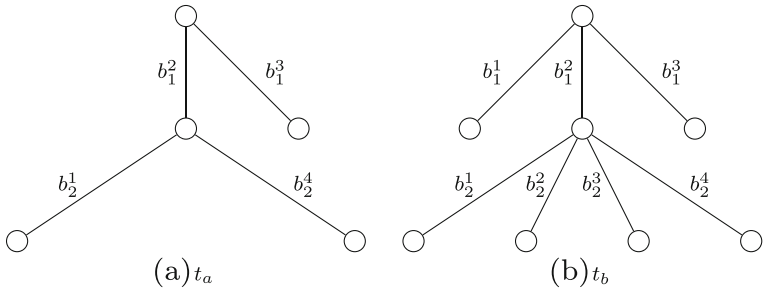

Fig. 3 CBT inclusion: $t_a \subseteq t_b$

In order to ensure conditions **C.1** and **C.2** are satisfied during our evaluation, the compared models must traverse CBTs linked by a well-defined relation, that we call *CBT inclusion* (Fig. 3 provides an illustration of CBT inclusion). It is formally defined with:

Definition 3 A CBT $t_1 = (\langle b_1^1, t_1^1\rangle, \ldots, \langle b_1^i, t_1^i\rangle, \ldots, \langle b_1^n, t_1^n\rangle)$ is *included* in a CBT $t_2 = (\langle b_2^1, t_2^1\rangle, \ldots, \langle b_2^j, t_2^j\rangle, \ldots, \langle b_2^m, t_2^m\rangle)$, denoted as $t_1 \subseteq t_2$ iff:

- $\exists (1 \leq l_1 < \ldots < l_n \leq m)$ s.t. $(b_1^i = b_2^{l_i} \wedge t_1^i \subseteq t_2^{l_i})\, \forall\, 1 \leq i \leq n$
- or $t_1 = ()$ (empty sequence)

In the next section, we propose an approach based on *replaying* CBTs that ensures the CBT traversed by the extended model $M \cup \phi$ (in a depth-first manner) is included in the CBT traversed by the baseline model $M \cup \phi_M$.

3 Fair evaluation through the *Replay* technique

This section first describes our *replay* technique (introduced in [56]) from a high-level perspective. It then discusses its limitations and provides details on how to implement the approach.

3.1 Replaying (high-level description)

The goal behind the replaying technique is to be able to evaluate only the effect of additional propagation, while using a search heuristic that is as similar as possible to the ones used in practice. To do so, we first generate a CBT, and replay it using the models to be evaluated. Replaying is therefore a two-step procedure:

1. Generation of a CBT to be replayed.
2. CBT replay with one (or several) models to be evaluated.

CBT generation is done using a function $generate(M, \beta) \rightarrow \mathbb{T}$ that returns a CBT from a model M and a branching procedure β. It relies on the functions $fixPoint(M)$ and

isSolved(M), defined hereafter. These definitions are provided for sake of characterizing the semantic of our approach, yet they do not necessarily correspond to the implementation.

Definition 4 If X is a set of variables, and M is a model that constrains elements of X, the function *fixPoint*$(M) \rightarrow \{\bot, ?\}$ reduces domains of the variables in X. It returns $\bot$ if a domain is wiped out (i.e., a constraint cannot be satisfied, hence no solution can be found), or ? if all domains are still non-empty (infeasibility cannot be proven).

Definition 5 The function *isSolved*$(M) \rightarrow \{True, False\}$ returns the status of the model M, solved or not. This corresponds to the fact that a solution to the modeled problem is found. That is, for the set of constraints C imposed on X, we have: $\bigwedge_i C_i \wedge \forall x \in X : |D(x)| = 1$, i.e., all variables of the model have a valid assignment (all the constraints are satisfied by the assignment).

The function *generate*(M, β) is then:

$$generate(M,\beta) \rightarrow \mathbb{T} = \begin{cases} (\langle \beta(M)^1, \\ generate(M \cup \beta(M)^1, \beta)\rangle, \ldots, \\ \langle \beta(M)^k, \\ generate(M \cup \beta(M)^k, \beta)\rangle, \ldots, \\ \langle \beta(M)^n, \\ generate(M \cup \beta(M)^n, \beta)\rangle) & \text{if } fixPoint(M) \neq \bot \\ & \quad \wedge \neg isSolved(M) \\ () & \text{otherwise} \end{cases}$$

It can be computed with a classic CP Depth-First Search. Example 5 illustrates a CBT generation, and Fig. 4a provides a visual example of a generated CBT that will be replayed further in this paper.

Example 5 Consider the branching procedure β_{ff} of Example 1 and the model $M = \{x > 0 \implies y > 2\}$, where $x \in \{0, 1, 2, 3\}$ and $y \in \{0, 1, 2, 3, 4\}$. *generate*$(M, \beta_{ff}) = (\langle x = 0, (\ldots)\rangle, \langle x \neq 0, (\langle y = 3, (\ldots)\rangle, \langle y \neq 3, (\ldots)\rangle)\rangle)$

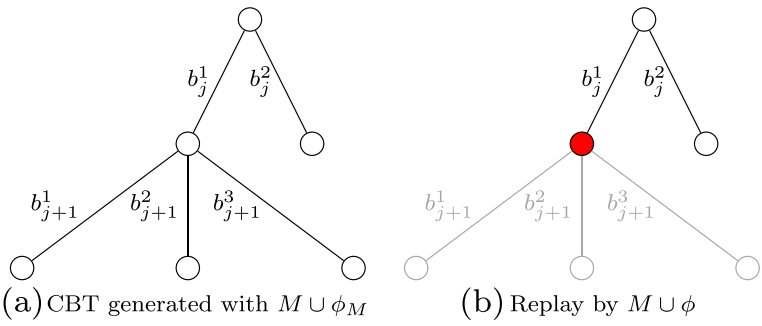

Fig. 4 A CBT generated by a model M and replayed by an extended model $M \cup \phi$

CBT replay Let $t = (\langle b^1, t^1\rangle, \ldots, \langle b^i, t^i\rangle, \ldots, \langle b^n, t^n\rangle)$. Let us define the function $replay(t, M)$ whose purpose is intuitively to re-traverse a CBT t using a model M and to return a new CBT t_{incl} such that $t_{incl} \subseteq t$:

$$replay(t, M) \rightarrow \mathbb{T} = \begin{cases} (\langle b^1, replay(t^1, M \cup b^1)\rangle, \ldots, \\ \langle b^k, replay(t^k, M \cup b^k)\rangle, \ldots, \\ \langle b^n, replay(t^n, M \cup b^n)\rangle) & \text{if } fixPoint(M) \neq \bot \\ & \quad \wedge \neg isSolved(M) \\ () & \text{otherwise} \end{cases}$$

The rationale behind this function is to make sure that exactly the same search space is visited. Moreover, we want to guarantee that the modifications made to the constraint store (i.e., adding or removing constraints to the store, and modifying the variable domains accordingly) are done in the exact same order. From the definitions, we can see that, as long as β is a deterministic function and ϕ_M is subsumed by ϕ, we have that (Property 1 and 2):

Property 1 $\forall \beta : replay(generate(M, \beta), M) = generate(M, \beta)$

Property 2 $replay(t, M \cup \phi) \subseteq replay(t, M \cup \phi_M)$

In other words, replaying with the original model leads to the original CBT, and extending the model leads to a CBT that is included in the original one.

An illustration of a replay of the CBT of Fig. 4a is given in Fig. 4b. In this figure, the extended model $M \cup \phi$ is able to prove infeasibility at the red node, i.e., before the baseline model $M \cup \phi_M$. The time required to visit the 3 gray nodes is therefore saved.

Propagator evaluation The evaluation of a propagator ϕ is simply done by computing in sequence:

$$t \leftarrow generate(M \cup \phi_M, \beta) \quad (1)$$

$$replay(t, M \cup \phi_M) \quad (2)$$

$$replay(t, M \cup \phi) \quad (3)$$

Then we compare the results of the two latter runs, which both use *replay* and hence incur the same search overhead. It is important that the generate run is done with the baseline problem $M \cup \phi_M$, because, thanks to the additional propagation performed by ϕ, the run with $M \cup \phi$ may skip some parts of t. However, all of the runs will always explore the same search space and visit the shared nodes in the same order.

This approach offers two significant advantages: 1) it allows tackling arbitrarily large instances, since a limit (e.g., time or number of nodes) can be enforced on the first run and the replays will still be guaranteed to explore the same search space. 2) It allows using any search strategy, including dynamic ones, making the evaluation more realistic.

Interestingly, if a limit to the generation is imposed, $M \cup \phi$ might actually find one additional solution. This occurs if the generation is stopped at an internal node with a partial assignment that is part of a solution, not found by $M \cup \phi_M$ since the generation was stopped due to the imposed limit. In this case, $M \cup \phi$ can remove more domain values inconsistent with the partial assignment, and by doing so, it might reduce the domains up the point where a total assignment is found, i.e., an additional solution is discovered.

Finally, let us designate as $metric(t, M)$ the metric quantity required to replay the CBT t with the model M. In particular, we respectively write $time(t, M)$ and $backtracks(t, M)$, the time and number of backtracks needed to traverse t.

3.2 Limitations

There are a few limitations to our approach that we must acknowledge. First, one can only use *monotonic* [46, 53] propagators in the baseline model M. A monotonic propagator is a propagator ϕ such that $D_i^1 \subseteq D_i^2 \implies \phi(D_i^1) \subseteq \phi(D_i^2)$, for any variable i considered by the propagator. Intuitively, this means that the more the domains are reduced, the more the propagator can infer inconsistent values. This property is required because once the model is extended with ϕ, more pruning might happen because of ϕ, implying reduced domains. If some propagators of M are not monotonic, they might prune less than when the CBT was generated. This has undesirable implications. For instance, one could reach a leaf solution node while still having unbound solution variables.

Another limitation is that we only allow the evaluation of a propagator ϕ by comparing a model $M \cup \phi_M$ with an *extended* model $M \cup \phi$. This requirement is due to ensure that $replay(t, M \cup \phi) \subseteq replay(t, M \cup \phi_M)$. This means that if we have two propagators ϕ_1 and ϕ_2 for a constraint that do not subsume each other, one cannot use the model $M \cup \phi_1$ as a baseline and replay with only $M \cup \phi_2$. One would have to replay with $M \cup \phi_1 \cup \phi_2$, or generate with M only (if it is sufficient to solve the problem in a sound manner) and replay with $M \cup \phi_1$ and with $M \cup \phi_2$.[6]

Finally, one could argue that when using the replay, we are comparing the stronger model $M \cup \phi$ in a search tree that would never be visited in practice, since it has been generated using the *baseline* model. We argue that in practice the replayed search tree is often similar to what it would be using the real dynamic search. We conducted a few small experiments to illustrate this. The results are given in Table 1. We first experimented with the Golomb ruler (length 11). The CBT was generated with the branching procedure of Example 1 using forward checking as the filtering technique for the ALLDIFFERENT constraints. The replay uses GAC ALLDIFFERENT constraints. The number of nodes was decreased from 7386480 to 2929035 using the stronger filtering. Even though the search tree is drastically reduced (more than divided by two), we measured that there are $\sim$99% of (local) *matching decisions*, i.e., decisions imposed during the replay that are exactly the same as the ones that would locally be returned by the real dynamic branching procedure if it was called at each search node of the replayed tree. This demonstrates that although being a static strategy, decisions of the replay are very similar to what would happen in practice with the dynamic strategy. We then considered an instance of the Job Shop Scheduling problem (36 activities) with a Conflict Ordering Search strategy [24], a state-of-the-art dynamic search strategy. The baseline model $M \cup \phi_M$ only uses binary constraints for the DISJUNCTIVE constraints, and the algorithms of [59] are used in $M \cup \phi$. In this case, the node ratio is $\sim$0.44 and the percentage of matching decisions is $\sim$84%, which is still very large. Finally, we experimented with the Traveling Salesman Problem with the branching procedure of Example 1. $M \cup \phi_M$

[6] A last solution is to use as a baseline the model that makes use of the *constructive disjunction* of ϕ_1 and ϕ_2 [29, 36, 61], that only prunes when both algorithm prune. However, in the general case, the time overhead for performing deductions only made by ϕ_1 or ϕ_2 cannot easily be deducted from the propagation time. This penalizes the baseline from a time perspective when it is replayed.

Table 1 Number of nodes of replays by M with and without ϕ, and percentages of locally matching decisions with the dynamic strategy

Problem	Golomb ruler	Job Shop	Traveling Salesman
# Nodes with $M \cup \phi_M$	7386480	898	82194
# Nodes with $M \cup \phi$	2929035	399	7177
% Same Decisions	99.86	84.42	21.77

uses a sum of ELEMENT constraints, while $M \cup \phi$ uses the MINIMUMASSIGNMENT constraint [23] with exact reduced costs, as proposed in [21]. We considered the instance *gr21* from the TSPLib [41]: the node ratio is ~ 0.09, and $\sim 22\%$ of the decisions are matching. This is less than for the other two problems, but the search space reduction is more drastic.

Clearly, the comparison between $M \cup \phi_M$ and $M \cup \phi$ remains artificial to some degree, because an actual dynamic strategy may behave differently for the run using $M \cup \phi$. Still, the ability to ensure conditions **C.1** and **C.2** and therefore isolate the contributions of ϕ, while using an arbitrary strategy, is a significant asset: one can exactly measure how fruitful/detrimental a filtering algorithm is in a realistic practical context.

3.3 Implementation of the replay technique

To implement the replay technique, we first generate a flat *linearized* version of the CBT by doing a *preorder taversal* using the baseline model $M \cup \phi_M$. This linearized CBT is simply a sequence of triples of the form $\langle b, c, d \rangle$ meant to represent a node of the original CBT. For a given triple $\langle b, c, d \rangle$:

- b is the branching constraint on the branch between the node it represents and its parent.
- c is the number of children of the node.
- d is the number of descendants of the node.

As we shall see, those triples are required to re-traverse the with a given extended with a given extended model $M \cup \phi$.

Linearizing the CBT must be done so that when the sequence is traversed, the behavior is the same as a CP Depth-First Search. The sequence must therefore represent the *preorder* traversal of the CBT. As an example, the sequence for the CBT given in Fig. 4a is:

$$\langle \top, 2, 5 \rangle, \langle b_j^1, 3, 3 \rangle, \langle b_{j+1}^1, 0, 0 \rangle, \langle b_{j+1}^1, 0, 0 \rangle, \langle b_{j+1}^3, 0, 0 \rangle, \langle b_j^2, 0, 0 \rangle$$

where $\top$ is the True clause.

Recording the sequence is done with Algorithm 1. This procedure defines in a recursive manner a classic CP Depth-First Search. More specifically, each time a branching constraint b is added to the model M, a triple $\langle b, c, d \rangle$ is added to a sequence S, where c is the number of branching constraints generated by the branching procedure β. However, unless the search proves infeasibility or finds a solution, the number of descendants d is only known after the recursive call. The triple is therefore updated at that moment. The calls to *RECORD_STATE*(M) and *RESTORE_STATE*(M) allow backtracking the state of the constraint store. We do not enter into details on how backtracking is performed, so that trail-based (inherited from [2, 60]) and copy-based [44] solvers fit into our proposed framework.

Algorithm 1 Linearized CBT generation algorithm

```
1  Algorithm generate(M, β)
       Input  : A model M and a branching procedure β
       Output: A sequence S that represents a generated CBT
2      S ⟵ () /* Empty Sequence                                   */
3      nVisitedNodes ⟵ 0
4      visit(M, β, S, ⊤) /* ⊤ is the True clause                  */
5      return S

6  Algorithm visit(M, β, S, b)
       Input  : A model M, a branching procedure β, a sequence S and a branching
                constraint b
       Output: A CBT t
7      nVisitedNodes ⟵ nVisitedNodes + 1
8      if fixPoint(M) = ⊥ then
9          S ⟵ S|⟨b, 0, 0⟩ /* Infeasibility                      */
10     else if isSolved(M) then
11         S ⟵ S|⟨b, 0, 0⟩ /* Solution                          */
12     else
13         (b^1, ..., b^i, ..., b^c) ⟵ β(M)
14         nodeIndex ⟵ nVisitedNodes /* Store the current node
               index.                                             */
15         S ⟵ S|⟨b, c, ?⟩ /* Number of descendants not known yet.
               */
16         for i ← 1 to c do
17             RECORD_STATE(M)
18             M ← M ∪ b^i /* Add b^i to the constraint store     */
19             visit(M, β, S, b^i)
20             RESTORE_STATE(M)
21         end
22         S_nodeIndex ⟵ ⟨b, c, nVisitedNodes − nodeIndex⟩ /* Store the number
               of descendants after the recursive call.           */
23     end
```

Replay algorithm A sequence generated with a baseline model $M \cup \phi_M$ can be replayed using any extended model $M \cup \phi$ with Algorithm 2. Notice that if the baseline model $M \cup \phi_M$ is replayed, this "linearized" process will behave exactly as the traditional search. Conversely, if the baseline model is extended with ϕ, some additional pruning might occur, implying that the store is either in a failed or in a solution state at an internal node n of the CBT (i.e., we may have $\mathit{fixPoint}(M) = \bot$ or $\mathit{isSolved}(M)$, see Definition 4). This event is illustrated in Fig. 4b, where the red node is failed due to additional pruning. When this happens, the replay process is able to directly skip all the descendants of the current node. This is illustrated by the light gray branching constraints in Fig. 4b. The replayed sequence by $M \cup \phi$ becomes:

$$\langle \top, 2, 5\rangle, \langle b^1_j, 3, 3\rangle, \underbrace{\langle b^1_{j+1}, 0, 0\rangle, \langle b^1_{j+1}, 0, 0\rangle, \langle b^3_{j+1}, 0, 0\rangle,}_{\text{Skip}} \langle b^2_j, 0, 0\rangle$$

Algorithm 2 Replay algorithm for a linearized CBT

```
1  Algorithm replay(M, S)
       Input  : A model M and a sequence of nodes obtained by linearizing the CBT t
       Output: time(t, M), backtracks(t, M)
2      nBacktracks ⟵ 0
3      replayNode(1)
4      return nBacktracks

5  Algorithm replayNode(index)
6      RECORD_STATE(M)
7      ⟨b, nChildren, nDescendants⟩ ⟵ S_index
8      newIndex ⟵ index + 1
9      M ← M ∪ b
10     if fixPoint(M) = ⊥ ∨ isSolved(M) then
11         nBacktracks ⟵ nBacktracks + 1
12         RESTORE_STATE(M)
13         return newIndex + nDescendants /* Skip the subtree          */
14     end
15     for i ← 1 to nChildren do
16         newIndex ⟵ replayNode(newIndex)
17     end
18     RESTORE_STATE(M)
19     return newIndex
```

Sharing sequences An interesting property of this approach is that the sequence can be serialized into files (as proposed in [16, 31]). We can therefore replay CBTs that do not fit in RAM (if a sequence is too large to fit in one file, it can simply be divided into chunks and put in several files). Additionally, those files can be shared among the community. Provided a well-defined format exists, all solvers implementing Algorithm 2 will be able to replay CBTs obtained using another solver. This opens the doors to common evaluation of propagators in the community.

4 Assessing the potential of a propagator

We assume we are interested in reducing the time for computing the output of a filtering procedure ϕ, without changing the function definition, i.e., without changing its input-output behavior. In particular, our goal is to assess the potential of two improvement directions: 1) increasing the efficiency of the current implementation/algorithm and 2) guarding the activation of ϕ with a necessary condition. Notice that the content of this section is actually orthogonal to the replay technique presented in Section 3. Yet, both can be combined, as exemplified in Section 6.

In order to assess the potential of improving the efficiency of ϕ or controlling its activation, we instrument the solver to collect detailed information about the propagator. Specifically, we store the total time for running ϕ, making a distinction between activations that actually lead to some pruning and fruitless activations. The two time statistics are respectively referred to as t_ϕ^+ and t_ϕ^-. Making those measurements only requires to write a procedure w_ϕ that wraps ϕ and is used instead. Every time w_ϕ is called during search,

it registers the current domain size of the decision variables and calls ϕ. If the size of any domain has changed, the CPU time required to execute ϕ is added to the t_ϕ^+ counter. If not, it is added to the other counter t_ϕ^-. The complexity of using w_ϕ is $\theta(n)$ where n is the number of decision variables. Once again, this approach is lightweight and easy to implement on most solvers.

It is now easy to get a rough, but valuable, estimate of the impact of specific measures on the solution time. First, we can estimate the impact of reducing the run time of ϕ by a factor $\mu \in [0, 1]$ by computing:

$$time(t, M \cup w_\phi) - \mu \cdot (t_\phi^+ + t_\phi^-) \tag{4}$$

i.e., by subtracting a fraction of the total computation time of ϕ. Similarly, we can assess the impact of guarding ϕ with a necessary condition that stops a fraction $\mu \in [0, 1]$ of the fruitless propagator activations. This is done by computing:

$$time(t, M \cup w_\phi) - \mu \cdot (t_\phi^-) \tag{5}$$

This simple, linear, approach allows us to compare *fictional* implementations of ϕ with real ones. By doing so, we get a chance to explore which values of μ would be necessary for beating the baseline, and we get a better understanding of the effort required to achieve such goal. In particular, we can approximately evaluate the impact of having an hypothetical time complexity for a fictional propagator. For instance, if the current implementation for ϕ is in $O(n^3)$ (where n is the number of variables), then we can estimate roughly what would be its cost for an $O(n^2)$ algorithm by choosing $\mu = (n-1)/n$ in (4).

4.1 Representative propagator evaluation with performance profiles

While it is interesting to make a quantified evaluation of filtering procedures for a given CBT, deeper and more general insights can be obtained by making use of benchmark suites. In order to aggregate the information and derive general conclusions, we rely on *performance profiles* [19]. A performance profile is a cumulative distribution function $F(\tau)$ of a given performance metric τ. In our case, the τ value is the ratio between the solving metric (typically, time or number of backtracks) of a target approach and that of the baseline $M \cup \phi_M$.

Formally, let $\phi_0, \phi_1, \ldots$ be the set of all considered implementations (possibly fictional, see Section 4.2) of ϕ, and let $\mathscr{T}$ be the set of all CBTs generated from the benchmark instances. Then the performance profile of ϕ_i is given by:

$$F_{M\cup\phi_i}(\tau) = \frac{1}{|\mathscr{T}|}\left|\left\{t \in \mathscr{T} : \frac{metric(t, M \cup \phi_i)}{metric(t, M \cup \phi_M)} \leq \tau\right\}\right| \tag{6}$$

For the sake of clarity, let us provide an introductory visual example in Fig. 5. In this plot,[7] one can see that the profile $F_{M\cup\phi_M}$ for the baseline model is a step function such that $F_{M\cup\phi_M}(\tau < 1) = 0$ and $F_{M\cup\phi_M}(\tau \geq 1) = 1$ (by definition, it will always be the case). Moreover, one can read that $F_{M\cup\phi}(2) = 0.75$. This means that the performance of $M \cup \phi$ is within a factor of 2 from the baseline in 75% of the benchmark problems. Assuming the benchmark is representative enough, the value of $F(\tau)$ can be interpreted as a probability.

An important value of a given performance profile $F_{M\cup\phi_i}(\tau)$ is in $\tau = 1$. For a given ϕ_i, $F_{M\cup\phi_i}(\tau = 1)$ gives the percentage of instances that can be solved using $M \cup \phi_i$ with

[7] The reader might be surprised by the x-axis of the plots, as there is a change of scale on the right-most side. This helps to have a long-term view of the profiles while keeping the focus on the τ region of interest.

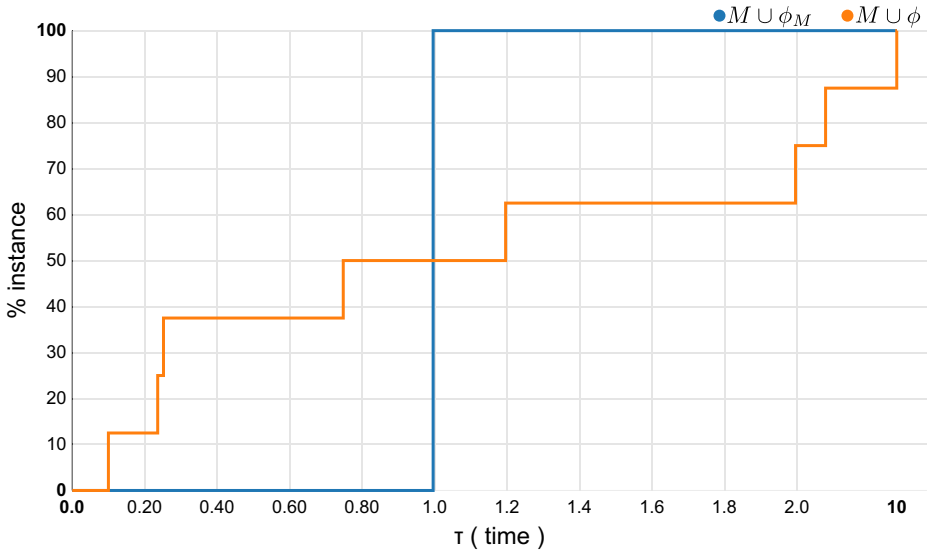

Fig. 5 Example of a Performance Profile to compare a baseline model $M \cup \phi_M$ with an extended model $M \cup \phi$

a value for the target metric that is less than (or equal to) the one of the baseline model $M \cup \phi_M$. For instance, in Fig. 5, 50% of the instances are solved by the extended model $M \cup \phi$ in a time smaller or equal to the one of the baseline. The space of τ is therefore divided in two important regions, $\tau < 1$ and $\tau \geq 1$. If $F_{M \cup \phi_i}(\tau) = 1$ for some $\tau < 1$, then using the model $M \cup \phi_i$ is always better than using the baseline, i.e., $M \cup \phi_i$ provides a speed-up for every instance. Unfortunately, this situation rarely happens in practice and it is thus interesting to read more carefully the performance profile. For a given pair ϕ_i, ϕ_j it is interesting to observe $F_{M \cup \phi_i}(\tau) - F_{M \cup \phi_j}(\tau)$, which indicates the *gain* of ϕ_i over ϕ_j. That is, $F_{M \cup \phi_i}(\tau) - F_{M \cup \phi_j}(\tau)$ reflects how many more (or less) instances can be solved by using $M \cup \phi_i$ instead of $M \cup \phi_j$ within a factor τ of the baseline metric value. Finally, the region above $F_{M \cup \phi}(\tau)$ for $\tau < 1$ is very informative, as it exhibits the gain of a given ϕ_i compared to the baseline $M \cup \phi_M$ **and** to $M \cup \phi$. Finally, instances with similar performance give rise to step-like changes in $F_{M \cup \phi}(\tau)$, while a linearly growing $F_{M \cup \phi}(\tau)$ is symptomatic of a diversified performance across the benchmark suite.

4.2 Fictional propagators

In order to assess the potential for improvements, we considered the following classes of fictional implementations:

- ϕ_{μ}^{cost}, i.e., an implementation for which the time is reduced by a factor μ.
- $\phi_{\mathcal{O}(f(n))}^{cost}$, i.e., an implementation for which the time complexity is $\mathcal{O}(f(n))$. It is a particular case of ϕ_{μ}^{cost} for which μ is well selected based on the actual complexity of ϕ and on the value of the parameter n.
- ϕ_{p}^{oracle}, i.e., an implementation that guards ϕ with a necessary condition causing useless activations with a probability p.

We then use performance profiles as described in Section 4.1 to derive general conclusions about the fictional propagators. For fictional implementations of ϕ, $time(t, M \cup \phi_i)$ is computed using (4) or (5).

Assuming the studied benchmark suite is representative enough, the joint use of performance profiles and fictional propagators allows us to provide quantitative and representative potential for improvements. The μ parameter in (4) or (5) plays an important role, as it allows quantifying how much reduction should be targeted to obtain the corresponding performance profile. In particular, the profile of ϕ_0^{oracle} (perfect necessary condition) bounds the gain that can be obtained by any necessary condition. The profile of $\phi_{\mathcal{O}(1)}^{cost}(\tau)$ (zero-cost implementation[8]) bounds the performance of any possible implementation. Against common intuition, $\phi_{\mathcal{O}(1)}^{cost}$ is not guaranteed to beat the baseline, since a weak filtering done by ϕ may trigger other (possibly expensive) propagators during fix point iteration.

4.3 Characterization of time efficiency and potential

The following definitions allow quantifying the gain obtained thanks to the extension of the baseline model $M \cup \phi_M$ with ϕ.

Definition 6 The *actual gain* $\mathcal{G}_{M\cup\phi_M}^{\phi}$ of a filtering procedure ϕ compared to a baseline model $M \cup \phi_M$ is the probability that $time(t, M \cup \phi) \leq time(t, M \cup \phi_M)$ for any CBT $t \in \mathbb{T}$. It can be estimated with $F_{M\cup\phi}(1)$.

The actual gain quantity represents the proportion over *all* existing CBTs for which $M \cup \phi$ is faster to traverse than $M \cup \phi_M$. While it is of course impossible to compute this value, we can estimate it with $F_{M\cup\phi}(1)$. The two next definitions provide the same quantity while considering the best fictional propagators that can be obtained out of ϕ.

Definition 7 The *upper bound gain* $\bar{\mathcal{G}}_{M\cup\phi_M}^{\phi}$ of a filtering procedure ϕ compared to a baseline model $M \cup \phi_M$ is the probability that $time(t, M \cup \phi_{\mathcal{O}(1)}^{cost}) \leq time(t, M \cup \phi_M)$ for any $t \in \mathbb{T}$. It can be estimated with $F_{M\cup\phi_{\mathcal{O}(1)}^{cost}}(1)$.

Definition 8 The *activation-control upper bound gain* $\hat{\mathcal{G}}_{M\cup\phi_M}^{\phi}$ of a filtering procedure ϕ compared to a baseline model $M \cup \phi_M$ is the probability that $time(t, M \cup \phi_0^{oracle}) \leq time(t, M \cup \phi_M)$ for any $t \in \mathbb{T}$. It can be estimated with $F_{M\cup\phi_0^{oracle}}(1)$.

The quantity $\mathcal{G}_{M\cup\phi_M}^{\phi}$ provides the probability that ϕ will actually be beneficial to solve an instance, if it is used to extend the model $M \cup \phi_M$. As long as it is non-zero, it means some gain could be obtained. Of course, the higher the value, the more ϕ is actually useful in practice in general. Quantities $\bar{\mathcal{G}}_{M\cup\phi_M}^{\phi}$ and $\hat{\mathcal{G}}_{M\cup\phi_M}^{\phi}$ are of great interest when compared to $\mathcal{G}_{M\cup\phi_M}^{\phi}$, as they allow quantifying the gap between the current gain, and the one that could be obtained by working on more efficient algorithms/implementations or finding necessary conditions for the algorithm to prune. Clearly, if $\bar{\mathcal{G}}_{M\cup\phi_M}^{\phi} - \mathcal{G}_{M\cup\phi_M}^{\phi} \approx 0^+$, devising a more efficient algorithm will not be very fruitful in terms of practical efficiency.

[8] Another way to think of $\phi_{\mathcal{O}(1)}^{cost}(\tau)$ is to consider additional inference made by ϕ to be integrated into the baseline model.

Potential of inference rules It is sometimes easier to find inference rules for a constraint than to directly propose an efficient algorithm to apply those rules. Instead of directly investing energy in order to find an efficient algorithm, one could postpone this work until the potential benefit of its discovery is known: an inefficient but easy algorithm to compute ϕ might be written in order to apply the inference rules. The value $\bar{\mathcal{G}}^{\phi}_{M\cup\phi_M}$ can then be used to quantify how fruitful in practice it would be to actually construct an efficient algorithm performing the inference rules. Again, if $\bar{\mathcal{G}}^{\phi}_{M\cup\phi_M}$ is very small, investing some time to find such an algorithm would not be beneficial in practice.

Global constraint maximal propagation In the same direction, another aspect that is useful and of great interest is the gain of the maximum propagation that can be performed by a global constraint with respect to a given consistency (e.g., GAC or GBC). In particular, studying constraints for which reaching a given consistency is NP-hard provides a lot of insight. An inefficient propagator to get the given consistency is straightforward: it is sufficient to embed a search process in the propagator. In the case of GAC, any instantiation of the *GAC-Schema* [11] can be used. From that, we can compute $\bar{\mathcal{G}}^{\phi^{max}}_{M\cup\phi_M}$, where ϕ^{max} is the inefficient procedure allowing to reach the level of consistency. $\bar{\mathcal{G}}^{\phi^{max}}_{M\cup\phi_M}$ gives the maximum gain that could ever be reached using the given level of consistency. Again, this can be compared with existing approaches in order to quantify how fruitful it would be to be able to prune more. The gain might again be negligible, meaning that research time should be better spent looking for new search strategies or models, rather than improving the consistency level.

5 Implementation in the OscaR solver

This section explains how the proposed framework is used in the OscaR solver [37]. It also gives some implementation details and design choices. The design of the replay framework was guided by the motivation of making it orthogonal to the existing OscaR search and without requiring any modification of existing default search heuristics (such as [24]). The existing search of OscaR was kept unmodified and agnostic to the replay framework. A search observer linearizes the search by capturing branching decisions into closures.

As an illustrative example, we use the well-known *n-Queens* problem. The OscaR model is provided in Fig. 6. It has been extended to integrate the replay technique. The additional instructions specific to the replay framework are highlighted in bold.

Initial model The model without replay is quite straightforward. The position variables of the queens are defined in lines 3–8. The constraints imposing that the queens cannot attack each other are declared in lines 10–12. They are however only added to the constraint store in the `startSubjectTo` bloc (line 20) where the search is effectively started (under some additional constraints that will eventually be removed from the model on search completion). In our case, this allows us to impose the allDifferent constraints before starting the search. We finally define the search heuristic in line 15 (the heuristic is here β_{ff}, from Example 1).

CBT generation The generation of the CBT is simply done by passing an additional search-listener parameter to the standard search. This listener stores the required sequence of triples used to replay (see Algorithm 1). The replaying searches will then re-use this exact

```
1  object Queens extends CPModel with App {
2
3    val nQueens = 10
4    // Number of queens
5    val Queens = 0 until nQueens
6
7    // Variables
8    val queens = Array.fill(nQueens)(CPIntVar.sparse(0, nQueens − 1))
9
10   val allDiffs = Seq(allDifferent(queens),
11     allDifferent(Queens.map(i => queens(i) + i)),
12     allDifferent(Queens.map(i => queens(i) − i)))
13
14   // Search heuristic
15   search(binaryFirstFail(queens))
16
17   val linearizer = new DFSLinearizer()
18
19   // Execution with FC allDifferent
20   val statsInit = startSubjectTo(){
21     add(allDiffs,Weak)
22   }(cp, linearizer)
23
24   // Replay with AC allDifferent
25   val statsReplayAC = cp.replaySubjectTo(linearizer, queens) {
26     add(allDiffs,Strong)
27   }
28 }
```

Fig. 6 Model for the n-Queens problem. The additional required instructions to replay and track a constraint are in bold. The rest of the model remains unchanged

same sequence. Notice that Depth First Limited Discrepancy Search [27] and Large Neighborhood Search [39] could also be used to perform the generation, as they are based on a regular OscaR search.

The listener interface defined in OscaR is given in Fig. 7. This interface allows defining the expected behavior when a node is expanded in Algorithm 1, i.e., after the branching procedure has been called. The only method (*onExpand*) takes an *Alternative* as an argument, which is basically a closure. This closure is to be applied when the branching is performed. The alternative therefore encapsulates any branching constraint addition to the model (i.e., $M \cup b_i$). The implementation of the linearizer (given in Fig. 8) is then direct: an internal buffer is filled with *preorder elements*, i.e., pairs with a branching constraint and a number of children, each time a node is expanded. The number of descendants of each node is then computed from this sequence after the generation is completed (without going into details, the sequence of triples is computed in two passes because the search in OscaR is sligthly different from Algorithm 1).

From a modeling point of view, to be able to generate the sequence during a search, the only requirement is to set up a *linearizer listener* (line 17) and pass it as a parameter, as in line 22. After this search is finished, the sequence is stored and we can replay it as many times as we want, potentially with constraints having a stronger pruning added to the model.

```
1 trait DFSearchListener {
2   /** Called on expand events */
3   def onExpand(currentBranchAlternative: Alternative, nChildren: Int): Unit
4 }
```

Fig. 7 Depth-First Search Listener interface

```
1 class DFSLinearizer extends DFSearchListener {
2 val nodes : ArrayBuffer[PreorderElement] = ArrayBuffer[PreorderElement]()
3 val nDescendantsOf : ArrayBuffer[Int] = ArrayBuffer[Int]()
4 def onExpand(currentBranchAlternative: Alternative, nChildren: Int) = nodes +=
      new PreorderElement(currentBranchAlternative, nChildren)
5 }
6 class PreorderElement(val alternative: Alternative, val nChildren: Int) extends
      Tuple2[Alternative, Int](alternative, nChildren){}
```

Fig. 8 Depth-First Search Linearizer. This class implements the interface of Fig. 7 to linearize the Depth-First Search of OscaR

CBT replay In order to replay the model, we call the *replaySubjectTo* procedure (line 25 in Fig. 6) that implements Algorithm 2. This procedure must know what variables must be assigned in order to detect solutions. OscaR indeed has no explicit store status when the problem is solved. During a replay, we consider that a solution is found once all constraints are satisfied (i.e., no failure during the fix point and no domain wipe-out) and all the variables passed to the replay primitive are assigned.

Once completed, the replay primitive returns the solution time, the number of found solutions, the number of backtracks and the number of nodes. Those statistics can be used to compare the performance of the baseline and that of the extended models.

Tracking a constraint It is optionally possible to track the activation of a propagator/constraint. This is useful to perform a study as described in Section 4. Basically, the modeler must just use the *track* function that returns the constraint passed as a parameter, augmented with code that implements the tracking behavior. This allows one to measure the time for each propagation call of the constraint. In order to separate pruning propagation time (t_ϕ^+) from non-pruning propagation time (t_ϕ^-), we must specify the variables for which we want to make this distinction. This is the second argument of the track function. The tracking behavior then verifies that some pruning happened for those variables by checking the size of their domains before and after propagation. The complexity for tracking is therefore $\mathcal{O}(n)$.

6 Experimentation

We applied our approach to several constraints and ran tests on AMD Opteron processors (2.7 GHz) using the Java Runtime Environment 8 and the constraint solver OscaR [37]. For each solved instance, we limited the run-time of $generate(M, \beta)$ to 600 s. Instances for which $generate(M, \beta)$ took less than 1 second were filtered out. The additional filtering put on top of the baseline model was executed with a lower priority by the constraint scheduler. The performance profiles were built with a public Web tool [57] made available[9] to the community.

6.1 AllDifferent

We analyzed the well-known ALLDIFFERENT constraint, since it is ubiquitous in Contraint Satisfaction Problems. The ALLDIFFERENT forward checking algorithm [18, 58] (written

[9] Accessible at http://performance-profile.info.ucl.ac.be/.

$allDiff_{FWC}$) is used in the baseline model, and we considered the following additional filtering methods:

- the bound consistent allDifferent, written $allDiff_{BC}$ [34].
- the counting-based allDifferent, written $allDiff_{CB}$, and described in [35].
- the arc consistent allDifferent [40], written $allDiff_{AC}$.

We used the 291 instances from the XCSP 2.1 benchmarks that contain allDifferent constraints, namely *bqwh-18-141_glb*, *medium*, *bqwh-15-106_glb*, *QG3*, *ortholatin*, *small*, *latinSquare*, *pigeons_glb*, *compet02* and *compet08*.

To assess the benefits of $allDiff_{BC}$, $allDiff_{CB}$ and $allDiff_{AC}$, we replayed with all the combinations of additional filtering procedures such that the replayed CBT is included into the generated one. We also considered models without $allDiff_{FWC}$ when possible ($allDiff_{CB}$ and $allDiff_{AC}$ subsume $allDiff_{FWC}$). Finally, the priority of $allDiff_{AC}$ in the propagation queue was the lowest, $allDiff_{BC}$ and $allDiff_{CB}$ had the same priority, and $allDiff_{FWC}$ had the highest priority. The branching procedure used to generate the CBTs is β_{ff}, as defined in Example 1.

Figure 9 provides the time performance profiles. Notice that we do not report the thirteen propagator combinations but only the profiles of the most different approaches in order to make the plots easier to read. From a time perspective, the approaches that are not shown have a profile with a shape that is generally in-between the curve of *FWCAndAC* and that of *CBAndBCAndAC*.

Our first observation is that even if *FWCAndBC* has an actual gain $\mathcal{G}^{\phi}_{M\cup\phi_M} \simeq 0.3$ (see Section 4.3) compared to the baseline (see the orange line in $\tau = 1$), it is clearly outperformed by the other approaches. *CBAndBC* (light green curve) and *FWCAndCBAndBC* (red curve) require a bit more time to approximately catch up with the other models. More importantly, we can see that while *FWCAndCB* has not the highest gain for τ values close to 0, it actually has the highest actual gain $\mathcal{G}^{\phi}_{M\cup\phi_M} \simeq 0.93$ (see the dark green line in $\tau = 1$) and

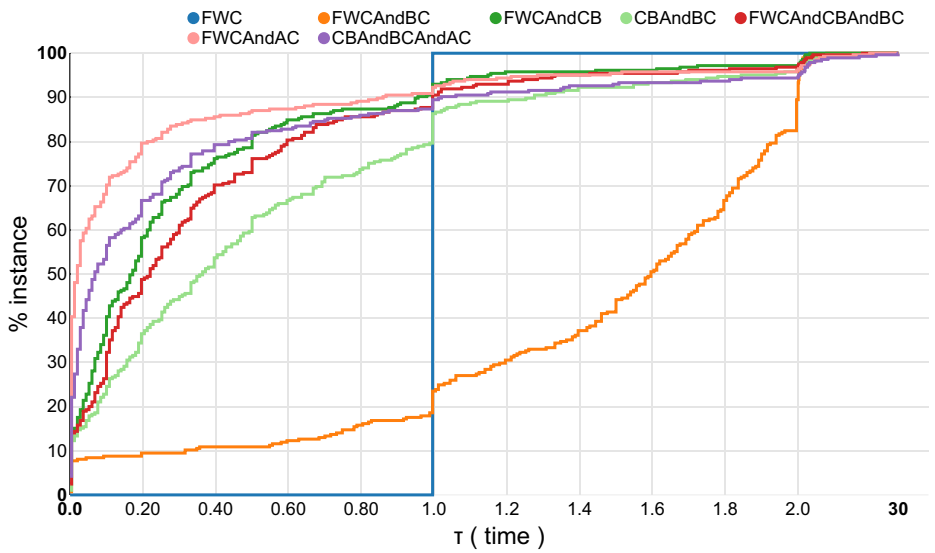

Fig. 9 Time performance profiles for combinations of AllDifferent propagators for XCSP instances

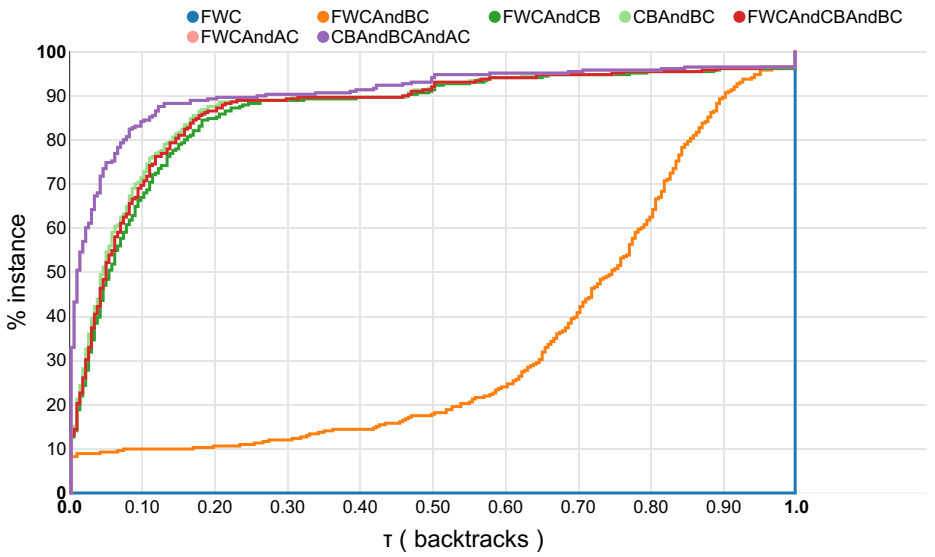

Fig. 10 Backtrack performance profiles for combinations of AllDifferent propagators for XCSP instances

stays better for larger τ values. This is of great interest as the counting-based allDifferent algorithm is actually very simple.

If we now look at the backtrack profiles of Fig. 10, we better understand why the *FWCAndBC* model is outperformed by the others, its gain in backtracks being way smaller than the other ones, especially for small τ values. As expected from the time profiles, the gains of the other models all have the same shape. Still, we can notice that *FWCAndCB* is one of the "worst" approaches in terms of backtrack gain, while it has the highest time gain, as just mentioned.

As a brief conclusion, we learned that Bound Consistency is not a sufficiently strong level of consistency for the ALLDIFFERENT constraint, from the point of view of both time and number of backtracks. On the contrary, the counting-based allDifferent infers almost as much as the Arc Consistency algorithm, allowing it to get similar time performances. Still, these conclusions must be taken with some care, as the problems we consider are quite structured.

6.2 Energetic Reasoning for the cumulative constraint

We analyzed the Energetic Reasoning propagator for the CUMULATIVE constraint [1, 3] on RCPSPs. The baseline model M employs the Time-Tabling algorithm from [33] and the ER Checker [4], which both run in $\mathcal{O}(n^2)$ [4, 17]. We did not use the improvements proposed in [17]. We use a dynamic search strategy, i.e., the classic *SetTimes* approach from [32]. We consider two benchmarks: the BL instances [3] (20–25 activities) and the PSPLIB (j30 and j90, with 30 and 90 activities) [30]. We focus on investigating, for the chosen benchmarks: 1) the potential benefit of having an ER algorithm running in $O(n^2)$ rather than in $O(n^3)$; 2) the potential benefit of a perfect necessary condition (see [55] and [9] for related works).

Figures 11/12 and 13/14 report profiles respectively for the BL and j90 instances. The real ER propagator has an actual gain $\mathcal{G}^{\phi}_{M\cup\phi_M} \simeq 0.5$ when BL instances are considered,

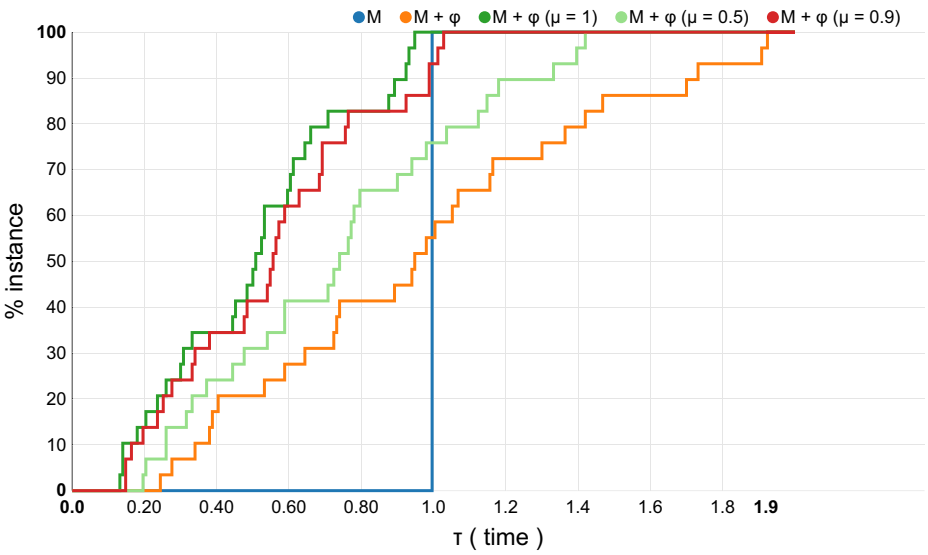

Fig. 11 Performance profiles for real and fictional (ϕ_{μ}^{cost}) ER propagators on the BL instances

but of only ~ 0.05 for the j90 instances (see the orange curves in $\tau = 1$ in Figs. 11/12 and 13/14). The larger problem size is a likely reason for the performance drop, so it is interesting to analyze the fictional, reduced-cost implementations (Figs. 11 and 13). In the BL benchmark a cost reduction translates to roughly proportional benefits. On j90, an $O(n^2)$ ER would lead to dramatic performance improvement, but it would beat the baseline in only 40% of the cases (see the purple curve in Fig. 13 in $\tau = 1$).

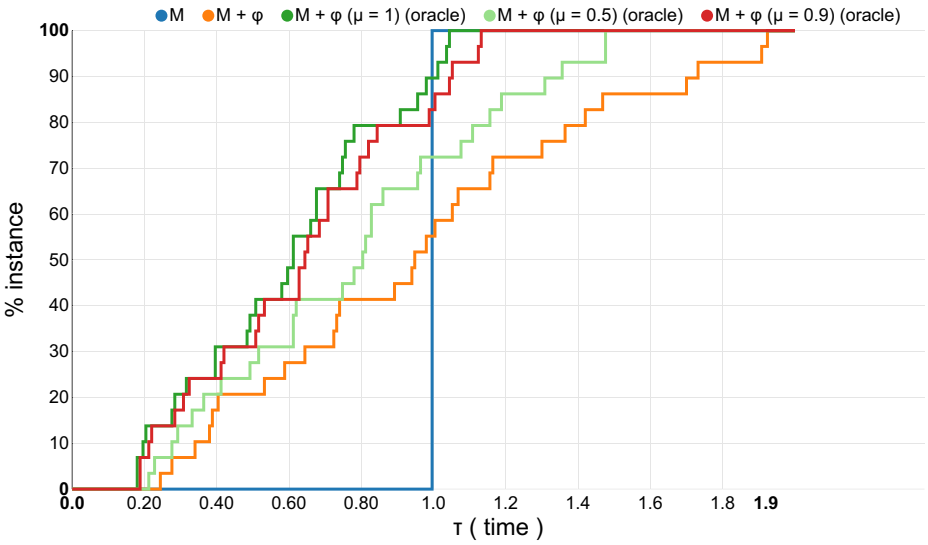

Fig. 12 Performance profiles for real and fictional (ϕ_{p}^{oracle}) ER propagators on the BL instances

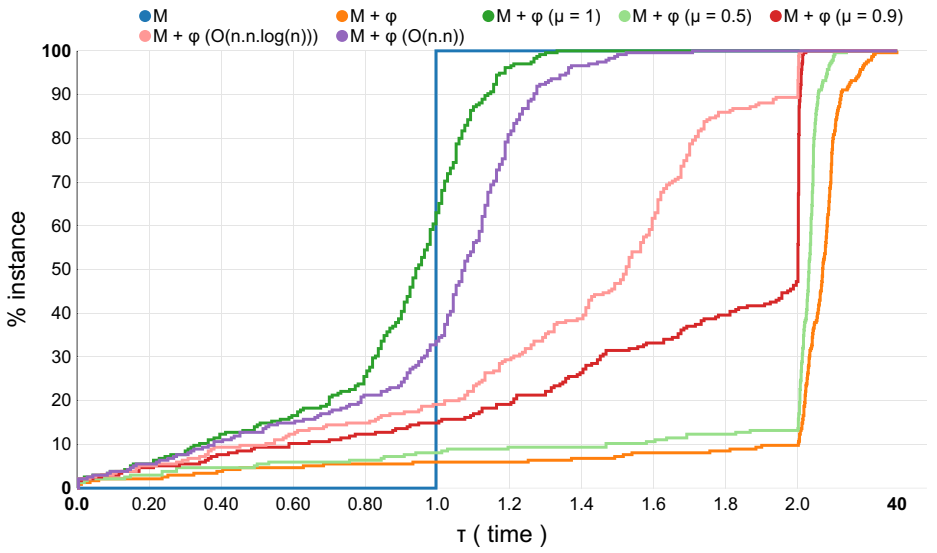

Fig. 13 Performance profiles for real and fictional (ϕ^{cost}_{μ} and $\phi^{cost}_{\mathcal{O}(f(n))}$) ER propagators on the j90 instances

More interestingly, for the upper bound gain we have $\bar{\mathcal{G}}^{\phi}_{M \cup \phi_M} \simeq 0.65$ (see the dark green curve Fig. 13, in $\tau = 1$), meaning there is about a 35% portion of instances where the baseline would win *no matter what the efficiency of ER is*, i.e., where the additional pruning of ER is sometimes detrimental rather than beneficial: despite an $\mathcal{O}(1)$ hypothetical ER, the additional ER filtering causes a larger number of iterations to reach the fix point. On such instances, ER cannot lead to benefits unless we find a way to activate it only when it

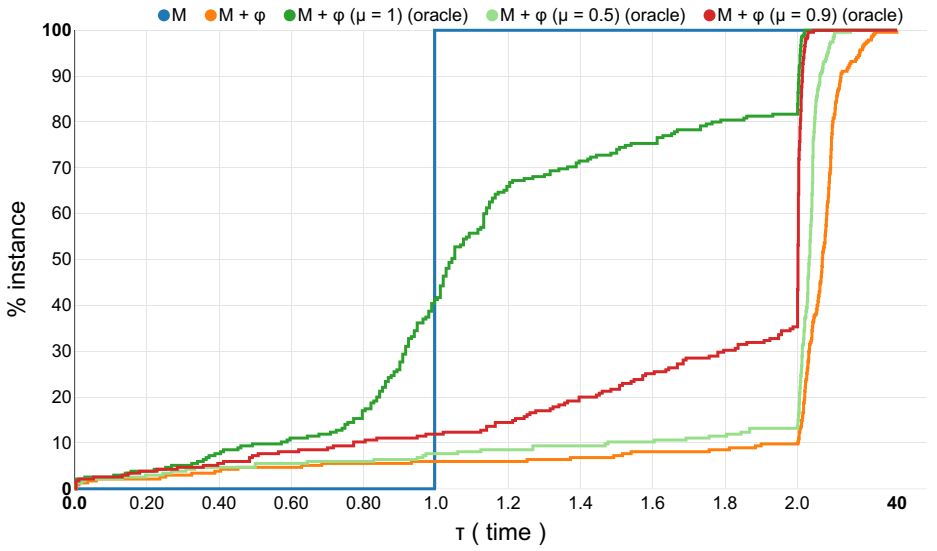

Fig. 14 Performance profiles for real and fictional (ϕ^{oracle}_{p}) ER propagators on the j90 instances

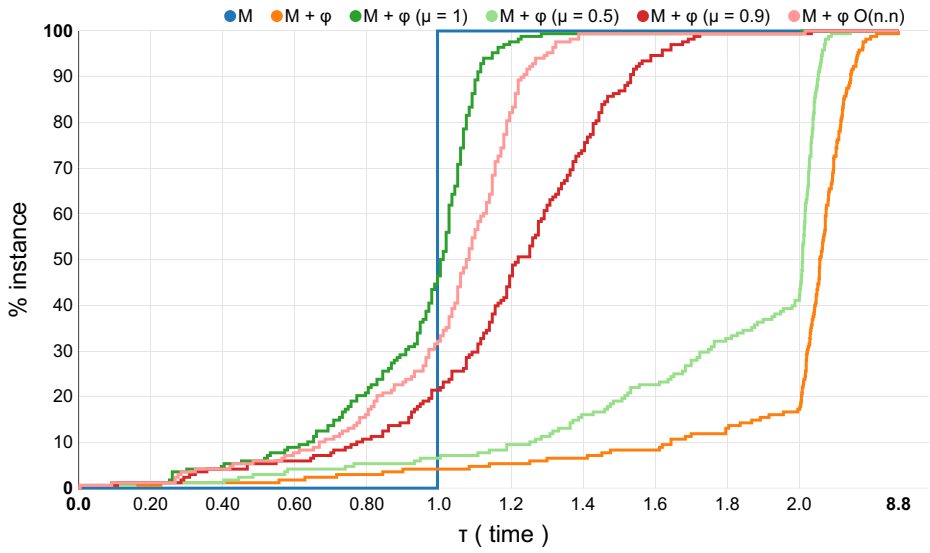

Fig. 15 Performance profiles for the SetTimes dynamic strategy

provides an actual advantage. As for using a necessary condition, a perfect approach would enable the same gain as that of a $O(n^2)$ ER (see the dark green in Fig. 14, in $\tau = 1$), but even a small mistake probability would cancel most of the benefits (in the same plot, compare the dark green curve with the red and light green curves).

Figures 15 and 16 compare profiles for different search strategies on the j30 instances (*SetTimes* and β_{ff}, as defined in Example 1): the potential gain of reducing the cost (e.g.,

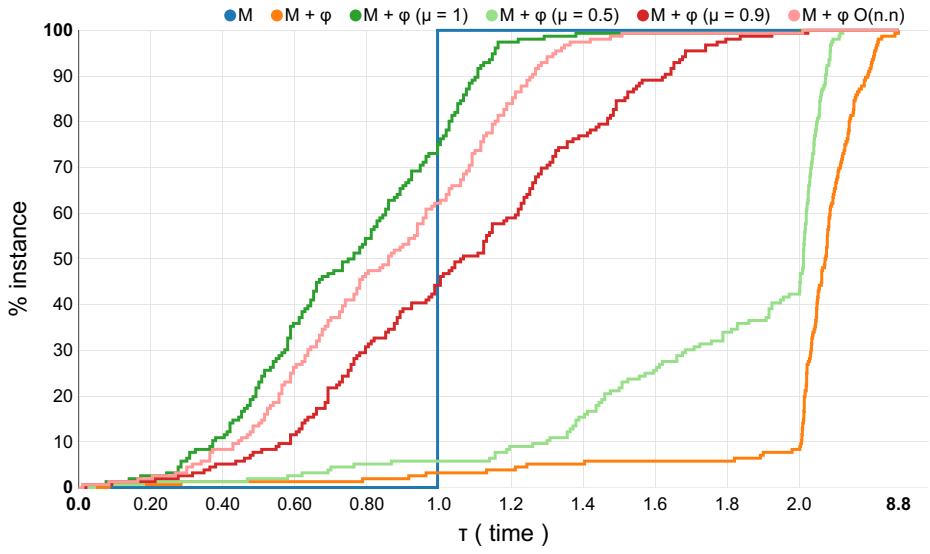

Fig. 16 Performance profiles for the binary static strategies

the dark green curves) is very different for the two strategies, even if the performance of the real propagator (orange curves) is roughly identical.

From this experiment, one can realize that although ER is one of the strongest filtering algorithms for the CUMULATIVE constraint, it does not provide much improvement for PSPLIB instances, even if we were able to perform its computation more efficiently. This illustrates that ER has two drawbacks when used in addition to Time-Tabling and the ER checker on those instances: 1) a heavier computation time, 2) a rather weak additional filtering in practice. Our simple method allows discovering that information before investing time in the research of a more efficient algorithm.

In addition, one can see that the possible performance improvements between the extended and the baseline models differ substantially depending on the kind of search strategies (static or dynamic) that we use. This points out the importance of having an approach for the rigorous comparison of propagators using practical search strategies.

6.3 Revisited cardinality reasoning for BinPacking

In our analysis of the RCRB propagator, we use as a benchmark the instances of the Balanced Academic Curriculum Problem (BACP) from [38, 42]. The baseline model M employs the BinPacking propagator from [47] and a GCC constraint (model **A** in [38]). The branching procedure is again β_{ff}, as defined in Example 1.

Figure 17 is very informative about the cost of RCRB. We can see for the actual gain that $\mathcal{G}^{\phi}_{M\cup\phi_M} \simeq 0.2$ (see the orange curve in $\tau = 1$ in Fig. 17), i.e., $\sim$20% of the instances are solved faster than the baseline model. However, reducing the propagator cost down to 0 provides only a small gain before $\tau = 1.1$ (see the dark green curve in Fig. 17): similarly to the ER case on j90, even a zero cost version of the propagator would not be able to beat the baseline in $\sim$55% of the instances (see the dark green curve in $\tau = 1$). For $\tau > 1.1$, reducing the propagator cost has a stronger effect, but a factor 0.9 reduction is required to

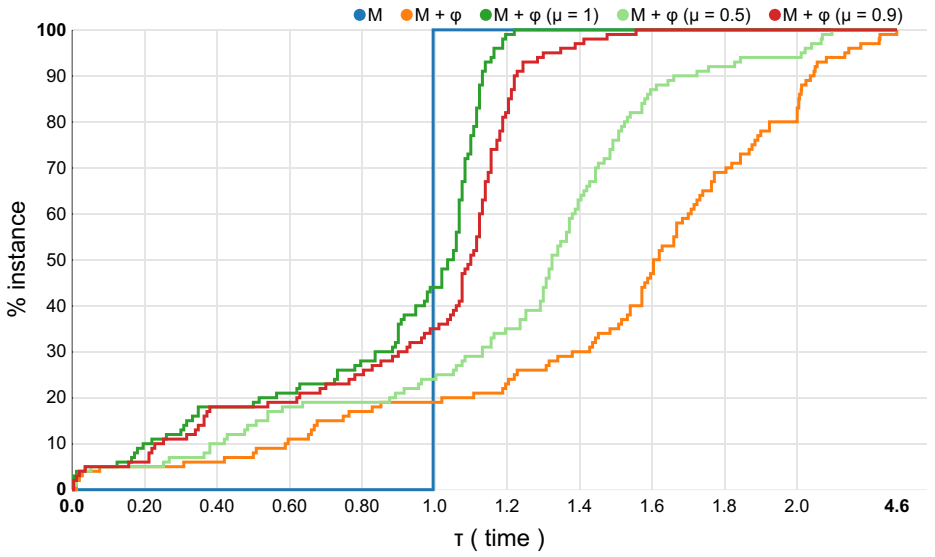

Fig. 17 Performance profiles with fictionally cost-reduced RCRB propagators

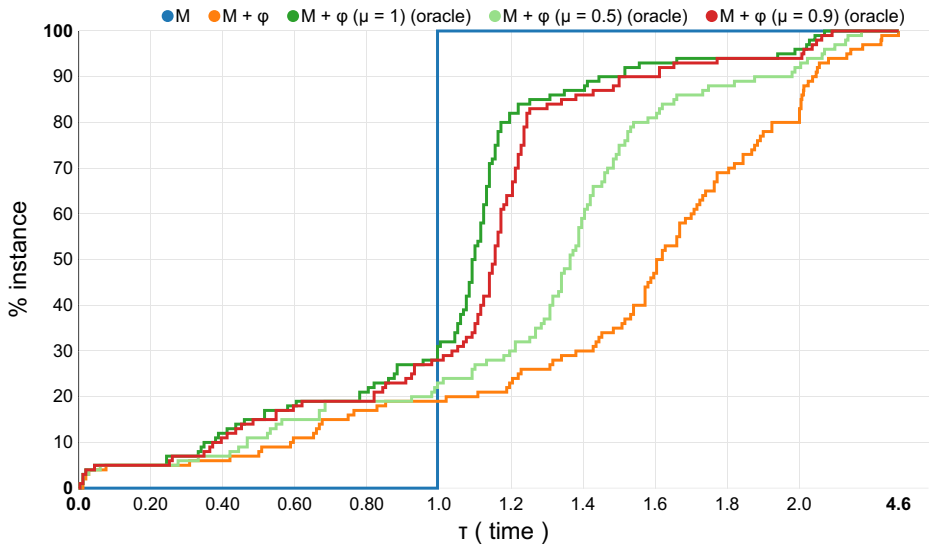

Fig. 18 Performance profiles with fictionally reduced RCRB propagators (necessary condition)

solve a lot more of the instances (see the red curve in Fig. 17). Hence, reducing the cost would improve the RCRB, but not that much compared to the baseline model as the benefits come "too late" in terms of τ. A similar analysis can be done for Fig. 18 for the potential gain of introducing a necessary condition.

6.4 Unary constraint with transition times

We here report results obtained with a recent approach of some of the authors [15]. This work extends the classic unary/disjunctive resource propagation algorithms to include propagation over *sequence-dependent* transition times between activities. In brief, the unary resource with transition times imposes the following relation:

$$\forall i, j : (\text{end}_i + tt_{i,j} \leq \text{start}_j) \vee (\text{end}_j + tt_{j,i} \leq \text{start}_i) \tag{7}$$

where start_j, end_i and $tt_{j,i}$ are the start and the end of an activity i, and the minimum transition time between activities i and j, respectively.

For each considered instance, the three following filterings for the unary constraint with transition times were used:

1. Binary constraints[10] (ϕ_b) given in (7). The baseline model M employs this constraint.
2. Binary constraints given in (7) with the Unary global constraint of [59] (ϕ_{b+u}).
3. The constraint of [15] (ϕ_{uTT}).

The search strategy used to generate the CBTs was *Conflict Ordering Search* [24]. Figures 19 and 20 respectively provide the profiles for time and number of backtracks for all the 960 instances. Figure 21 provides a "long-term" view of Fig. 19.

[10]For efficiency reasons, dedicated propagators have been implemented instead of posting reified constraints.

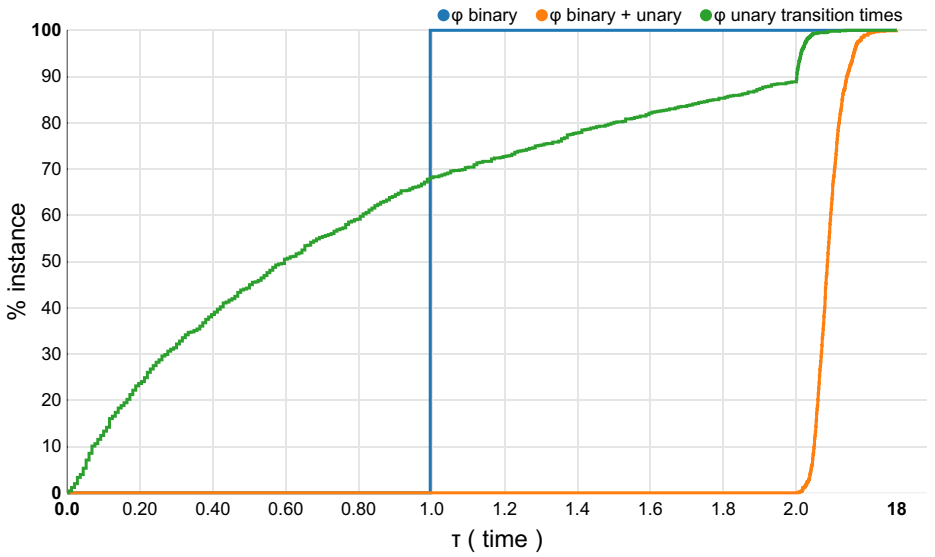

Fig. 19 Short-term time performance profiles for the Unary Resource with Transition Times

From Fig. 19, we can first conclude that ϕ_{b+u} (orange curve) is clearly worse than ϕ_{uTT} (green curve) and ϕ_b (blue curve) from a time perspective. Moreover, Fig. 20 shows that ϕ_{b+u} rarely offers more pruning than ϕ_b.

In comparison, we can see from Fig. 19 that for $\sim 25\%$ of the instances, ϕ_{uTT} is about 5 times faster than ϕ_b (see $F_{\phi_{uTT}}(0.2)$), and that $\sim 65\%$ of the instances are solved faster (see $F_{\phi_{uTT}}(1)$). Moreover, it offers more pruning for $\sim 100\%$ of the instances, meaning that the actual gain in terms of number of backtracks $\mathcal{G}_M^{\phi_{uTT}} \simeq 1$ (see $F_{\phi_{uTT}}(1)$, in Fig. 20).

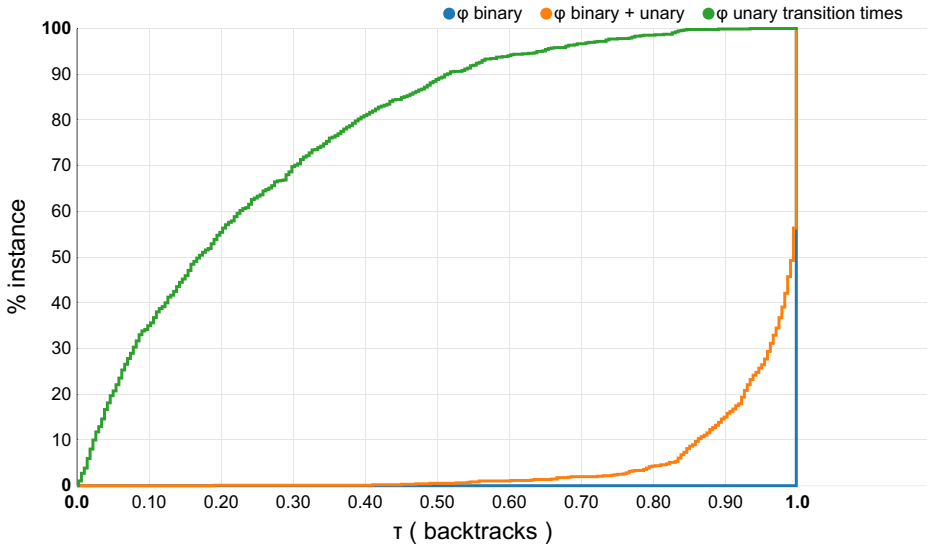

Fig. 20 Backtrack performance profiles for the Unary Resource with Transition Times

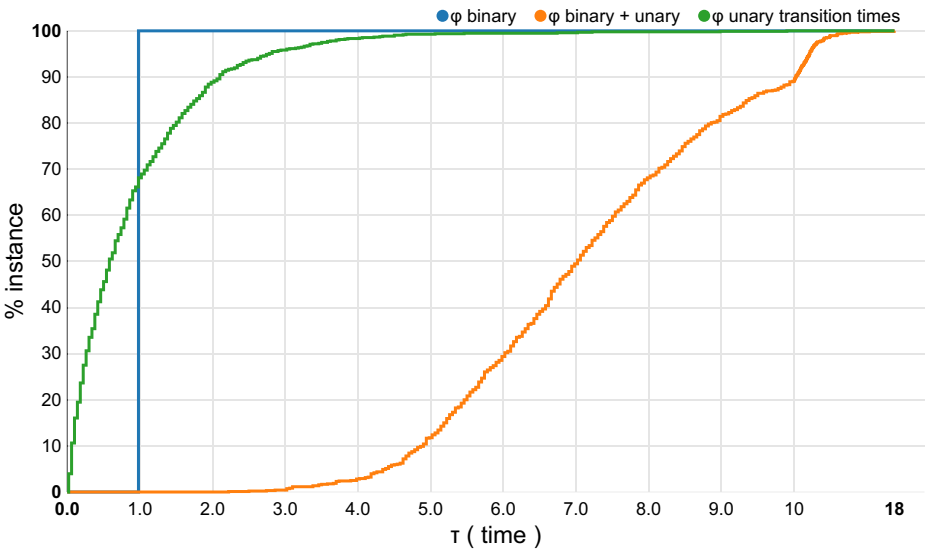

Fig. 21 Long-term time performance profiles for the Unary Resource with Transition Times

From Fig. 21, we can see that the constraint does not have too much overhead, as ϕ_{uTT} is at worst 2 times slower than ϕ_b for $\sim 20\%$ percent of the instances ($F_{\phi_{uTT}}(2) - F_{\phi_{uTT}}(1)$). It is a bit slower for the remaining $\sim 10\%$, but almost all instances are solved in a time at most 5 times slower than the baseline (since $F_{\phi_{uTT}}(5) = 1$).

The conclusion is clear: when transition times are involved, the unary resource algorithms that do not consider them provide almost no additional filtering and therefore only incur overhead. On the contrary, the unary resource with transition times [15] prunes much more and is therefore often beneficial.

6.5 Bound Consistent Cumulative

As proposed in Section 4.3, we studied the gain that would be provided by a Bound Consistent CUMULATIVE constraint, in order to estimate how far the current propagators are from the maximal pruning. The baseline model M uses poly-time propagators that suffice to achieve the strongest propagation we can get so far in the OscaR solver, namely *Energetic Reasoning* [4], *Not-First Not-Last* [4], and *Time-Table Disjunctive Reasoning* [26]. The Bound Consistent Cumulative Propagator was constructed as an exponential algorithm into which we basically embedded a search. A checker and a propagator were constructed and they were used to replay the generated CBTs. We only used the BL instances [3] as they remain quite small in terms of number of activities (20–25). We measured the backtracks and none of the instances were filtered out in this case. Figure 22 gives the performance profiles and Fig. 23 gives a zoomed version near $\tau = 0$. First, one can notice that there is still a lot to gain (it might not be possible as Bound Consistency for the Cumulative constraint is NP-hard), and this kind of measurements allows quantifying an upper bound on this gain. It is especially impressive to look near 0. For instance, for almost 30% of the instances, there is a potential gain factor of 100 (see the curves in $\tau = 0.01$ in Fig. 23) in terms of number of backtracks, even if we are already using the strongest poly-time pruning we know so far. This illustrates that, while working on efficient practical algorithms (e.g., [25]) is

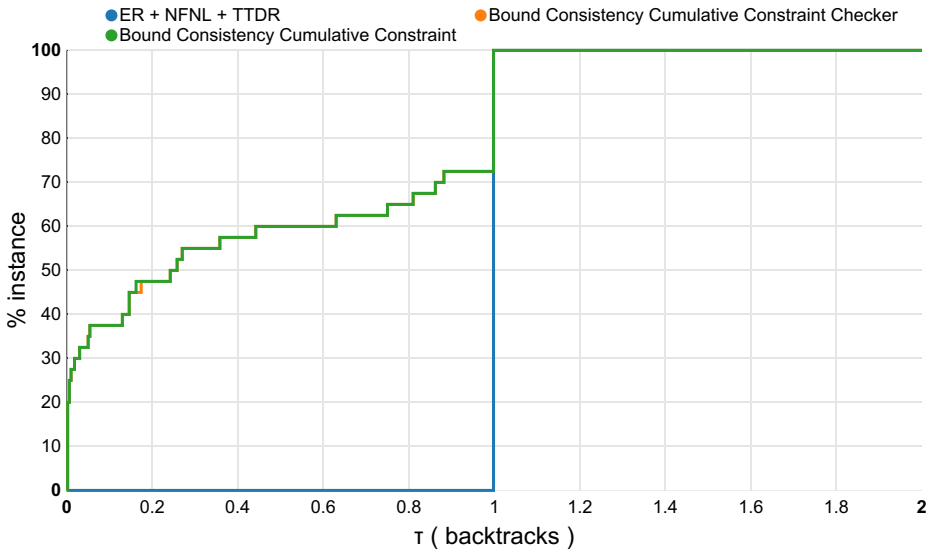

Fig. 22 Backtrack performance profiles for the Bound Consistent Cumulative Constraint

important, finding complementary poly-time and efficient algorithms to the ones used so far would clearly provide improvement. A last interesting point to notice is that the Bound-Consistent propagator almost provides no improvement compared to the checker. Devising new efficient checkers might actually suffice.

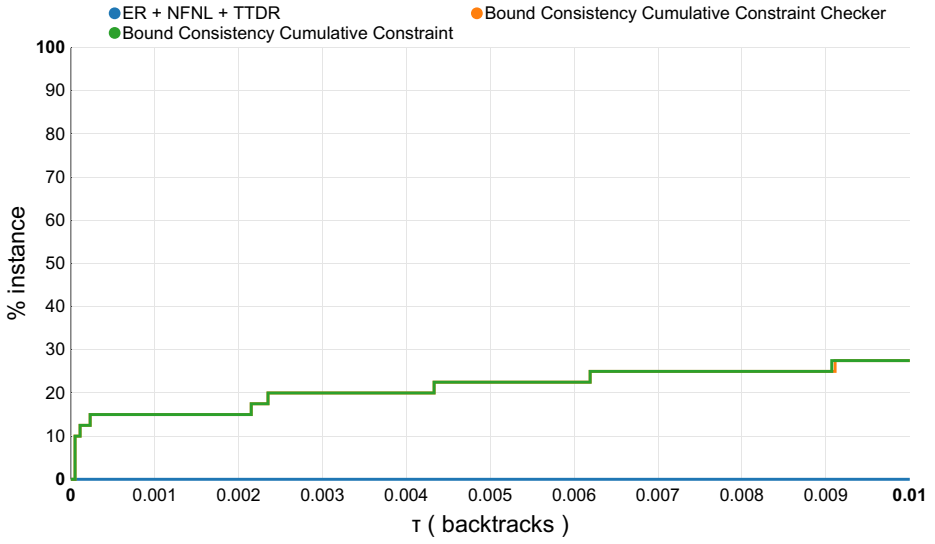

Fig. 23 Backtrack performance profiles for the Bound Consistent Cumulative Constraint (zoomed in)

7 Conclusion

Search heuristics can have a significant impact on the outcome of the evaluation of a global constraint (or more generally a filtering procedure). We therefore proposed a rigorous and yet simple framework that allows preventing any unfair advantages regarding the compared approaches, by *only measuring the effect of additional filtering*. Being able to measure exactly the time gain provided by a filtering algorithm permits to reduce the bias in empirical evaluations. We explained how to actually implement this framework.

Evaluating the potential advantages of reducing the cost of a given filtering procedure is of great importance to make our research efforts as fruitful as possible. As a first step in this direction, we proposed a systematic methodology to simulate the performance of *fictional implementations of a propagator having reduced activation cost*. This is done *before* starting time-consuming research activities to actually reduce the cost. A nice feature of deterministic replays is that measurements can be carried out in different replays, removing imprecisions due to measurement overhead.

We suggested in this work a broader usage of performance profiles in the CP community. We showcased in the results section that they allow deriving many informative conclusions. We evaluated several propagators for the following constraints on quite large benchmarks: ALLDIFFERENT, CUMULATIVE, BINPACKING and UNARY with transition times. In addition, a nice feature of our version of the performance profiles is that the whole community could continuously add new data and update them on a central repository, as a common effort to improve knowledge about the performance of propagators. This can be done as long as the baseline model remains the same.

As for the estimation of the impact of reducing the cost of a propagator, we illustrated the approach for Energetic Reasoning and Revisited Cardinality Reasoning for BinPacking over popular sets of instances. We found that reducing the propagator cost, *even to the point of making it negligible*, might actually be beneficial only on a small subset of a given instance set. Furthermore, this outcome can differ substantially depending on the considered benchmark and on the search strategy. We also briefly studied the shortfalls of not being able to achieve bound consistency for the cumulative constraint. Interestingly, from a pruning point a view, there is still a lot to gain.

Future work We might consider to generate more CBTs for a given instance, and gather the results. This would allow the evaluation approach to be even more robust. For instance, we could use several branching strategies or use Large Neighborhood Search in order to get more data for a given, large-scale, instance (and not only data from the beginning of the search tree). Another way to do so is to bound the search space to be replayed with a set of no-goods. Replaying a CBT with a model is not always possible, because the constraints used in the generator model must be subsumed by the one in the model, which is not always the case. For example, Time-Tabling and Edge-finding for the cumulative constraint do not subsume each other. Still we can generate a CBT into which all the replayed CBTs will be included in. To do so, when we generate the CBTs, we could use a model that prunes only when all the constraints used in replays are actually able to prune. Finally, regarding the potential of necessary conditions, we could study the gain of activating a propagator only when it prunes several variables or values.

References

1. Aggoun, A., & Beldiceanu, N. (1993). Extending CHIP in order to solve complex scheduling and placement problems. *Mathematical and Computer Modelling*, *17*(7), 57–73.
2. Ait-Kaci, H., & Des Flambertins, F. (1999). Warren's abstract machine—a tutorial reconstruction (pp. 114). Cambridge: MIT Press Cambridge.
3. Baptiste, P., & Le Pape, C. (2000). Constraint propagation and decomposition techniques for highly disjunctive and highly cumulative project scheduling problems. *Constraints*, *5*(1–2), 119–139.
4. Baptiste, P., Le Pape, C., & Nuijten, W. (2001). Constraint-based scheduling: applying constraint programming to scheduling problems. In *International Series in Operations Research & Management Science* (Vol. 39). Springer, Berlin.
5. Beldiceanu, N., & Contejean, E. (1994). Introducing global constraints in CHIP. *Mathematical and Computer Modelling*, *20*(12), 97–123.
6. Benhamou, F. (1996). Heterogeneous constraint solving. In *International conference on algebraic and logic programming* (pp. 62–76). Springer, Berlin.
7. Bergman, D., Ciré, A.A., & van Hoeve, W.J. (2014). MDD propagation for sequence constraints. *Journal of Artificial Intelligence Research (JAIR)*, *50*, 697–722.
8. Bergman, D., Cire, A.A., van Hoeve, W.J., & Hooker, J. (2016). Decision diagrams for optimization. Springer International Publishing.
9. Berthold, T., Heinz, S., & Schulz, J. (2011). An approximative criterion for the potential of energetic reasoning. In *Theory and practice of algorithms in (computer) systems* (pp. 229–239).
10. Bessière, C., & Debruyne, R. (2005). Optimal and suboptimal singleton arc consistency algorithms. In *International joint conference on artificial intelligence* (pp. 54–59).
11. Bessiere, C., & Régin, J.C. (1997). Arc consistency for general constraint networks: preliminary results. In *International joint conference on artificial intelligence* (pp. 398–404). Citeseer.
12. Bessiere, C., & Van Hentenryck, P. (2003). To be or not to be... a global constraint. In *International conference on principles and practice of constraint programming* (pp. 789–794). Springer, Berlin.
13. Brand, S., Narodytska, N., Quimper, C., Stuckey, P.J., & Walsh, T. (2007). Encodings of the sequence constraint. In *International conference on principles and practice of constraint programming* (pp. 210–224).
14. Cheng, K.C.K., & Yap, R.H.C. (2010). An MDD-based generalized arc consistency algorithm for positive and negative table constraints and some global constraints. *Constraints*, *15*(2), 265–304.
15. Dejemeppe, C., Van Cauwelaert, S., & Schaus, P. (2015). The unary resource with transition times. In *International conference on principles and practice of constraint programming* (pp. 89–104). Springer, Berlin.
16. Deransart, P., Hermenegildo, M.V., & Maluszynski, J. (2000). Analysis and visualization tools for constraint programming: constraint debugging. In *Lecture Notes in Computer Science* (Vol. 1870). Springer, Berlin.
17. Derrien, A., & Petit, T. (2014). A new characterization of relevant intervals for energetic reasoning. In *International conference on principles and practice of constraint programming* (pp. 289–297). Springer, Berlin.
18. Dincbas, M., Simonis, H., & Van Hentenryck, P. (1990). Solving large combinatorial problems in logic programming. *The Journal of Logic Programming*, *8*(1), 75–93.
19. Dolan, E.D., & Moré, J.J. (2002). Benchmarking optimization software with performance profiles. *Mathematical Programming*, *91*(2), 201–213.
20. Du Boisberranger, J., Gardy, D., Lorca, X., & Truchet, C. (2013). When is it worthwhile to propagate a constraint? A probabilistic analysis of alldifferent. In *Workshop on analytic algorithmics and combinatorics* (pp. 80–90). SIAM.
21. Ducomman, S., Cambazard, H., & Penz, B. (2016). Alternative filtering for the weighted circuit constraint: Comparing lower bounds for the tsp and solving tsptw. In *AAAI conference on artificial intelligence*.
22. Erschler, J., & Lopez, P. (1990). Energy-based approach for task scheduling under time and resources constraints. In *International workshop on project management and scheduling* (pp. 115–121).
23. Focacci, F., Lodi, A., Milano, M., & Vigo, D. (1999). Solving TSP through the integration of OR and CP techniques. *Electronic Notes in Discrete Mathematics*, *1*, 13–25.
24. Gay, S., Hartert, R., Lecoutre, C., & Schaus, P. (2015). Conflict ordering search for scheduling problems. In *International conference on principles and practice of constraint programming* (pp. 140–148). Springer.

25. Gay, S., Hartert, R., & Schaus, P. (2015). Simple and scalable time-table filtering for the cumulative constraint. In *International conference on principles and practice of constraint programming* (pp. 149–157). Springer, Berlin.
26. Gay, S., Hartert, R., & Schaus, P. (2015). Time-table disjunctive reasoning for the cumulative constraint. In *International conference on integration of artificial intelligence (AI) and operations research (OR) techniques in constraint programming* (pp. 157–172). Springer, Berlin.
27. Harvey, W.D., & Ginsberg, M.L. (1995). Limited discrepancy search. In *International joint conference on artificial intelligence* (pp. 607–615).
28. van Hoeve, W.J., Pesant, G., Rousseau, L., & Sabharwal, A. (2006). Revisiting the sequence constraint. In *International conference on principles and practice of constraint programming* (pp. 620–634).
29. Jefferson, C., Moore, N.C., Nightingale, P., & Petrie, K.E. (2010). Implementing logical connectives in constraint programming. *Artificial Intelligence*, *174*(16–17), 1407–1429.
30. Kolisch, R., Schwindt, C., & Sprecher, A. (1999). Benchmark instances for project scheduling problems. In *Project scheduling* (pp. 197–212). Springer, Berlin.
31. Langevine, L., Deransart, P., & Ducassé, M. (2003). A generic trace schema for the portability of CP(FD) debugging tools. In *International workshop on constraint solving and constraint logic programming* (pp. 171–195). Springer, Berlin.
32. Le Pape, C., Couronné, P., Vergamini, D., & Gosselin, V. (1994). Time-versus-capacity compromises in project scheduling. In *Workshop of the UK planning and scheduling*. Citeseer.
33. Letort, A., Beldiceanu, N., & Carlsson, M. (2012). A scalable sweep algorithm for the cumulative constraint. In *International conference on principles and practice of constraint programming* (pp. 439–454). Springer, Berlin.
34. López-Ortiz, A., Quimper, C.G., Tromp, J., & Van Beek, P. (2003). A fast and simple algorithm for bounds consistency of the alldifferent constraint. In *International joint conference on artificial intelligence* (Vol. 3, pp. 245–250).
35. McCreesh, C., & Prosser, P. (2015). A parallel, backjumping subgraph isomorphism algorithm using supplemental graphs. In *International conference on principles and practice of constraint programming* (pp. 295–312). Springer, Berlin.
36. Müller, T., & Würtz, J. (1995). Constructive disjunction in oz. In *Workshop Logische Programmierung*. Citeseer.
37. OscaR Team: OscaR: Scala in OR (2012). Available from https://bitbucket.org/oscarlib/oscar.
38. Pelsser, F., Schaus, P., & Régin, J.C. (2013). Revisiting the cardinality reasoning for binpacking constraint. In *International conference on principles and practice of constraint programming* (pp. 578–586). Springer, Berlin.
39. Pisinger, D., & Ropke, S. (2010). Large neighborhood search. In *Handbook of metaheuristics* (pp. 399–419). Springer, Berlin.
40. Régin, J.C. (1994). A filtering algorithm for constraints of difference in CSPs. In *AAAI conference on artificial intelligence* (Vol. 94, pp. 362–367).
41. Reinelt, G. (1991). Tsplib—a traveling salesman problem library. *ORSA Journal on Computing*, *3*(4), 376–384.
42. Schaus, P. (2009). Solving balancing and bin-packing problems with constraint programming. Ph.D. thesis, Université catholique de Louvain, Louvain-la-Neuve.
43. Schulte, C. (1997). Oz explorer: a visual constraint programming tool. In L. Naish (Ed.) *Proceedings of the fourteenth international conference on logic programming* (pp. 286–300). Leuven: The MIT Press.
44. Schulte, C. (1999). Comparing trailing and copying for constraint programming. In *International conference on logic programming* (Vol. 99, pp. 275–289).
45. Schulte, C., & Stuckey, P.J. (2005). When do bounds and domain propagation lead to the same search space? *ACM Transactions on Programming Languages and Systems*, *27*(3), 388–425.
46. Schulte, C., & Tack, G. (2009). Weakly monotonic propagators. In *International conference on principles and practice of constraint programming* (pp. 723–730). Springer, Berlin.
47. Shaw, P. (2004). A constraint for bin packing. In *International conference on principles and practice of constraint programming* (pp. 648–662). Springer, Berlin.
48. Shishmarev, M., Mears, C., Tack, G., & de la Banda, M.G. (2015). Visual search tree profiling. In *International conference on principles and practice of constraint programming*. Springer, Berlin.
49. Shishmarev, M., Mears, C., Tack, G., & de la Banda, M.G. (2016). Learning from learning solvers. In *International conference on principles and practice of constraint programming* (pp. 455–472). Springer, Berlin.
50. Shishmarev, M., Mears, C., Tack, G., & de la Banda, M.G. (2016). Visual search tree profiling. *Constraints*, *21*(1), 77–94.

51. Simonis, H., Davern, P., Feldman, J., Mehta, D., Quesada, L., & Carlsson, M. (2010). A generic visualization platform for CP. In *International conference on principles and practice of constraint programming* (pp. 460–474). Springer, Berlin.
52. Smith, B.M. (2005). Modelling for constraint programming. In *Lecture notes for the first international summer school on constraint programming*.
53. Tack, G. (2009). Constraint propagation—models, techniques, implementation. Ph.D. thesis, Saarland University, Germany.
54. Van Beek, P. (2006). Backtracking search algorithms. In *Handbook of constraint programming* (pp. 85–134).
55. Van Cauwelaert, S., Lombardi, M., & Pierre, S. (2014). Supervised learning to control energetic reasoning: feasibility study. In *Proceedings of the doctoral program of CP2014*.
56. Van Cauwelaert, S., Lombardi, M., & Schaus, P. (2015). Understanding the potential of propagators. In *International conference on integration of artificial intelligence (AI) and operations research (OR) techniques in constraint programming* (pp. 427–436). Springer, Berlin.
57. Van Cauwelaert, S., Lombardi, M., & Schaus, P. (2016). A visual web tool to perform what-if analysis of optimization approaches. Tech. rep., UCLouvain.
58. Van Hentenryck, P., & Dincbas, M. (1987). Forward checking in logic programming. In *International conference on logic programming* (pp. 229–256).
59. Vilım, P. (2007). Global constraints in scheduling. Ph.D. thesis, Charles University in Prague, Faculty of Mathematics and Physics, Department of Theoretical Computer Science and Mathematical Logic, KTIML MFF, Universita Karlova, Praha 1.
60. Warren, D. (1983). *An abstract Prolog instruction set* (Vol. 309). Menlo Park: SRI International.
61. Würtz, J., & Müller, T. (1996). Constructive disjunction revisited. In Görz, G., & Hölldobler, S. (Eds.) *KI-96: Advances in artificial intelligence. KI 1996. Lecture Notes in Computer Science (Lecture Notes in Artificial Intelligence),* (Vol. 1137, pp. 377386). Berlin: Springer.