Bandit-Based Search for Constraint Programming

Manuel Loth
MSR–INRIA joint centre
Palaiseau, France

Michèle Sebag
Marc Schoenauer
CNRS–INRIA team TAO
Univ. Paris-Sud, France

Youssef Hamadi
Microsoft Research
Cambridge, UK

Christian Schulte
School of ICT
KTH Royal Inst. of Tech., SW

Abstract

Constraint Programming (CP) solvers classically explore the solution space using tree-search based heuristics. Monte-Carlo Tree-Search (MCTS) is a tree-search method aimed at optimal sequential decision making under uncertainty. At the crossroads of CP and MCTS, this paper presents the Bandit Search for Constraint Programming (BASCOP) algorithm, adapting MCTS to the specifics of CP search trees. Formally, MCTS simultaneously estimates the average node reward, and uses it to bias the exploration towards the most promising regions of the tree, borrowing the multi-armed bandit (MAB) decision rule. The two contributions in BASCOP concern i) a specific reward function, estimating the relative failure depth conditionally to a (variable, value) assignment; ii) a new decision rule, hybridizing the MAB framework and the spirit of local neighborhood search. Specifically, BASCOP guides the CP search in the neighborhood of the previous best solution, by exploiting statistical estimates gathered across multiple restarts. BASCOP, using Gecode as the underlying constraint solver, shows significant improvements over the depth-first search baseline on some CP benchmark suites. For hard job-shop scheduling problems, BASCOP matches the results of state-of-the-art scheduling-specific CP approaches. These results demonstrate the potential of BASCOP as a generic yet robust search method for CP.

Introduction

A variety of algorithms and heuristics have been designed in CP, determining which (variable, value) assignment must be selected at each point, how to backtrack on failures, and how to restart the search (van Beek 2006).

The choice of the algorithm or heuristic most appropriate to a given problem instance has been intensively investigated since the late 70s (Rice 1976). In the particular domain of search, many of these approaches, e.g. (Xu et al. 2008; O'Mahony et al. 2008; Samulowitz and Memisevic 2007; Streeter, Golovin, and Smith 2007; Hutter, Hoos, and Stützle 2007), rely *de facto* on supervised machine learning (ML; more in Sect.). In this approach, a database describing the behavior of a portfolio of algorithms or heuristics on many representative problem instances (using static and possibly dynamic features (Xu et al. 2008)), is exploited to estimate the best option for the problem instance at hand, either in an off-line or online fashion.

In this paper, the use of another ML blend, specifically reinforcement learning (Sutton and Barto 1998), is investigated to support the CP search. Formally, it is shown how Monte-Carlo Tree Search (MCTS) can be used to guide the exploration of the CP search tree. The best-known version of MCTS, referred to as Upper Confidence Tree (Kocsis and Szepesvári 2006), extends the multi-armed bandit framework (Lai and Robbins 1985; Auer, Cesa-Bianchi, and Fischer 2002) to the field of sequential decision making. Since its inception, MCTS has demonstrated its merits and wide applicability in the domains of games (Ciancarini and Favini 2009) or automated planning (Nakhost and Müller 2009) among many others.

MCTS proceeds by incrementally developing a search tree through consecutive tree walks. In each tree-walk and in each node, the choice of the child node to be visited enforces a careful balance between the exploration of the search space, and the exploitation of the best results found so far. The choice among the child nodes relies on the statistical assessment of the reward expectation attached to each node, e.g. the frequency of winning the game after visiting this node.

The use of MCTS within the CP search raises two main difficulties. The first one concerns the definition of an appropriate reward attached to a tree node (a partial assignment of the branching variables). The second difficulty is that the CP search frequently involves multiple restarts (Luby, Sinclair, and Zuckerman 1993) to enforce an optimal exploration depth. Multiple restarts forbid the use of MCTS as is, since different nodes are considered in each restart (and memorizing them all is clearly infeasible).

The presented algorithm, called BASCOP (*Bandit-based Search for Constraint Programming*), proceeds by associating to each (variable, value) assignment its *relative failure depth*, averaged over all previous tree-walks. This estimate is maintained over the sequence of restarts. It is used to guide the search, possibly in conjunction with a left bias according to a value-ordering heuristic.

As proof of principle, BASCOP is implemented on the top of Gecode, and experimentally validated on the job-shop problem (JSP) (Beck 2007), balanced incomplete block design (BIBD) (Mathon and Rosa 1985), and car-sequencing

problem.

This paper is organized as follows. Section briefly reviews the state of the art in ML applied to CP, and describes Monte-Carlo Tree-Search for the sake of completeness (Sect.). Section gives an overview of the BASCOP algorithm, hybridizing MCTS and local neighborhood search with restarts. Section presents the experimental setting for the empirical validation of BASCOPand reports on the results. The paper concludes with some perspectives for further research.

Machine Learning for Constraint Programming

After briefly reviewing some ML-based approaches aimed at the control of search algorithms, this section discusses which one out of supervised ML, or reinforcement learning settings, is more appropriate to search control. For the sake of self-containedness, this section also describes the MCTS reinforcement learning algorithm.

Supervised Machine Learning

As mentioned, most ML approaches to the control of search algorithms exploit a set of scalar features, describing the problem instance using static (and possibly dynamic) information (Xu et al. 2008; Hutter et al. 2006). Specifically, datasets associating to the feature-based representation of each problem instance the target indicator (e.g. the runtime of a solver) are built and exploited by supervised machine learning to achieve portfolio algorithm selection, or hyper-parameter tuning. In SATzilla (Xu et al. 2008), a regression model predicting the runtime of each solver on a problem instance is built, and used to select the algorithm with minimal expected run-time. CPHydra (O'Mahony et al. 2008) uses a similarity-based approach and builds a switching policy based on the most efficient solvers for the problem instance at hand. (Samulowitz and Memisevic 2007) likewise applies ML to adjust the CP heuristics online. The Adaptive Constraint Engine (Epstein et al. 2002) can be viewed as an ensemble learning approach, where each heuristic votes for a possible variable/value decision to solve a CSP. Combining Multiple Heuristics Online (Streeter, Golovin, and Smith 2007) and Portfolios with deadlines (Wu and Van Beek 2008) are designed to build a scheduler policy in order to switch the execution of black-box solvers during the resolution process.

This paper more specifically focuses on online algorithm selection, where the ML component is in charge of selecting the appropriate heuristic/assignment at any time step depending on the current description of the instance. The proposed approach is based on Monte-Carlo Tree-Search which is a particular technique of the reinforcement learning framework.

Monte Carlo Tree Search

The best known MCTS algorithm, referred to as Upper Confidence Tree (UCT) (Kocsis and Szepesvári 2006), extends the Upper Confidence Bound algorithm (Auer, Cesa-Bianchi, and Fischer 2002) to tree-structured spaces. UCT simultaneously explores and builds a search tree, initially restricted to its root node, along N tree-walks. Each tree-walk involves three phases:

The **bandit phase** starts from the root node and iteratively selects an action/a child node until arriving in a leaf node. Action selection is handled as a multi-armed bandit problem. The set $\mathcal{A}_s$ of admissible actions a in node s defines the child nodes (s,a) of s; the selected action a^* maximizes the Upper Confidence Bound:

$$\bar{r}_{s,a} + C\sqrt{\log(n_s)/n_{s,a}} \tag{1}$$

over a ranging in $\mathcal{A}_s$, where n_s stands for the number of times node s has been visited, $n_{s,a}$ denotes the number of times a has been selected in node s, and $\bar{r}_{s,a}$ is the average cumulative reward collected when selecting action a from node s. The first (respectively the second) term in Eq. (1) corresponds to the exploitation (resp. exploration) term, and the exploration vs exploitation trade-off is controlled by parameter C. In a deterministic setting, the selection of the child node (s,a) yields a single next state $tr(s,a)$, which replaces s as current node. The bandit phase stops upon arriving at a leaf node of the tree.

The **tree building phase** takes place upon arriving in a leaf node s; some action a is (randomly or heuristically) selected and $tr(s,a)$ is added as child node of s. Accordingly, the number of nodes in the tree is the number of tree-walks.

The **roll-out phase** starts from the new leaf node $tr(s,a)$ and iteratively (randomly or heuristically) selects an action until arriving in a terminal state u; at this point the reward r_u of the whole tree-walk is computed and used to update the cumulative reward estimates in all nodes (s,a) visited during the tree-walk:

$$\begin{aligned} \bar{r}_{s,a} &\leftarrow \frac{1}{n_{s,a}+1}\left(n_{s,a} \times \bar{r}_{s,a} + r_u\right) \\ n_{s,a} &\leftarrow n_{s,a}+1; \qquad n_s \leftarrow n_s + 1 \end{aligned} \tag{2}$$

The Rapid Action Value Estimation (RAVE) heuristic is meant to guide the exploration of the search space and the tree-building phase (Gelly and Silver 2007). In its simplest version, $RAVE(a)$ is set to the average reward taken over all tree-walks involving action a.

Overview of BASCOP

This section presents the BASCOP algorithm, defining the proposed reward function and describing how the reward estimates are exploited to guide the search. Let us first describe the structure of the BASCOP search tree, restricting ourselves to binary variables for the sake of readability[1].

Tree structure

The complete CP search tree is structured as follows. Each node inherits a partial assignment s (including the constraint propagation achieved by the CP solver); it selects a variable X to be assigned, fetched from the variable-ordering heuristics; its child nodes are defined from the literals ℓ_X and $\ell_{\bar{X}}$.

[1]The extension to n-ary variables is straightforward, and will be considered in the experimental validation of BASCOP (section).

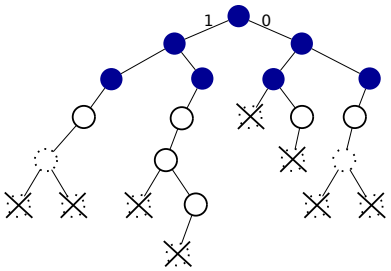

Figure 1: Structure of the BASCOP search tree: the top-tree (filled nodes), the DFS parts (unfilled nodes) including closed nodes (dotted nodes).

Each branch, associated to $s \wedge \ell$ (with $\ell = \ell_X$ or $\ell_{\bar{X}}$) is associated a status, ranging in: closed (the subtree associated to $s \wedge \ell$ has been fully explored); open (the subtree is being explored); or to-be-opened (not yet considered).

In the particular case of the depth-first-search (DFS) strategy, one systematically selects the leftmost branch among those that have not been closed yet. Only the tree-path from the root node to the currently visited node has to be maintained: nodes in the left part of the complete tree w.r.t. the current tree-path have been fully explored, nodes in the right part of the tree remain to be considered.

As mentioned (section), MCTS maintains a subset of the complete tree. This tree, initialized to the root node, gradually deepens along consecutive tree-walks, as a child node is added to every leaf node which is being visited, or at the k-th visit to this leaf node; parameter k is referred to as *expand rate*. Within the tree, the selection of the branch to be visited is achieved through the UCB criterion (Eq. 1). Below the leaf node, branches are iteratively selected using a default policy, referred to as roll-out policy and depending on the problem domain.

In BASCOP, the MCTS strategy is embedded within the CP exploration as follows. On the one hand, BASCOP maintains the upper-part of the tree being explored (how to accommodate the multiple restarts will be discussed in section). On the other hand, the roll-out policy is set to the depth-first-search strategy, thus enabling a systematic, and ultimately exhaustive, exploration of the subtree.

Fig. 1 depicts the general structure of the BASCOP search tree: the upper nodes (the filled nodes, referred to as *top-tree*) are explored using a UCB-like decision rule (see below). In the meanwhile, a depth-first-search tree-path is attached to each leaf s of the top-tree, enabling the exhaustive exploration of the sub-tree rooted in s. At each time step, a node in the BASCOP tree is labelled as top node (respectively bottom node) if it belongs to the top tree (resp. to a DFS path). The trade-off between the respective size of the top-tree and the DFS part is controlled from the expand rate k.

Relative Failure Depth Reward

As mentioned, the exploration of the BASCOP top tree is achieved using an UCB-like selection rule, based on a reward which remains to be defined. The most natural option in the MCTS spirit would be to reward each top node with some average success rate of the tree-walks visiting this node.

However, a heuristics commonly involved in the CP search is that of *multiple restarts*. Upon each restart, the current CP search tree is erased; the memory of the search is only reflected through some indicators (e.g. weighted degree, weighted dom-degree, impact, activity, or no-goods) maintained over the restarts. When rebuilding the CP search tree from scratch, a new variable ordering computed from the indicators is considered, hopefully resulting in more efficient and shorter tree-paths.

Naturally, BASCOP must accommodate the multiple restarts if it is to define a generic CP search strategy. For tractability reasons, BASCOP can hardly maintain all top nodes (partial assignments) ever considered along multiple restarts; the BASCOP search tree must thus also undergo the multiple restarts. It then becomes irrelevant to associate to each top node s an average reward, as this reward would be estimated from insufficiently many samples (tree-walks).

It thus comes to associate a reward to each literal ℓ, in the spirit of the RAVE heuristics (section). The statistics are thus managed orthogonal to the tree rather than in its nodes. On the positive side, such rewards can be maintained over multiple restarts. On the negative side, these rewards must measure the average impact of the literal ℓ on the search, regardless of the assignment s conditionally to which ℓ has been selected. Another motivation for this approach lies in the primary goal of RAVE: reward sharing boosts the search in its initial phase. Although this comes at the price of losing asymptotic convergence guarantees in the pure MCTS setting, this drawback is of no concern here: firstly, seeking convergence – that is asymptotically repeating optimal tree-walks – is not relevant since explored sub-trees are pruned; secondly, in the context of multiple-restarts, the search is usually restricted to such an initial phase, and a quick focus on promising regions is required.

The proposed reward is defined as follows. The quality of a tree-walk s is measured by its failure depth[2] d_s: intuitively, a shorter tree-walk contains more bad literals than a longer tree-walk, everything else being equal. The instant reward of a literal ℓ involved in tree-walk s cannot be directly set to d_s: the depth $d_{s,\ell}$ of the literal (the depth of the node where ℓ has been selected) in s is bound to fluctuate with the multiple restarts and the variable orderings, thus hindering the estimation of ℓ impact. Finally, the instant reward associated to literal ℓ over a tree-walk s is set to $d_s - d_{s,\ell}$.

$$RAVE(\ell) = \text{ Average } \{d_s - d_{s,\ell},\ s \text{ tree-walk involving } \ell\}$$

Overall, BASCOP thus maintains for each literal ℓ its average reward $RAVE(\ell)$ and the number n_ℓ of tree-walks visiting ℓ (either as a top node or as a DFS node).

Selection Rules

The exploration strategy in BASCOP involves two different selection rules, depending on whether the current node is part of the top-tree or of the bottom-tree.

[2]Other measures could have been considered, and are left for further study.

Depth-first-search is used for the bottom-tree. Note that the left-preference in DFS usually implements a suitable value ordering. In particular, a local neighborhood search (Beck 2007) can be implemented by exploring first the branch corresponding to the literal ℓ which is satisfied by the last found solution, as will be used for the job shop problem (section).

In the top-tree, selection rules combining the UCB rule and the left-preference are investigated. Letting X denote the current variable to be assigned, with ℓ_X (respectively $\ell_{\bar{X}}$) denoting the associated left (resp. right) branch:

- **Balanced SR** selects alternatively ℓ_X and $\ell_{\bar{X}}$;
- **ϵ-left SR** selects ℓ_X with probability $1-\epsilon$ (thus corresponding to a stochastic variant of the limited discrepancy search (Harvey and Ginsberg 1995)) and $\ell_{\bar{X}}$ otherwise;
- **UCB SR** selects the assignment maximizing the confidence bound of the RAVE estimate (Eq. 1)

$$\text{select } \arg\max_{\ell \in \{\ell_X, \ell_{\bar{X}}\}} RAVE(\ell) + C\sqrt{\frac{\log(n_{\ell_X} + n_{\ell_{\bar{X}}})}{n_\ell}}$$

- **UCB-Left SR**: same as UCB SR, with the difference that different exploration constants are attached to literals ℓ_X and $\ell_{\bar{X}}$ ($C_{\text{left}} = \rho C_{\text{right}}, \rho > 1$, in order to bias the exploration toward the left branch.

Note that balanced and ϵ-left selection rules are not adaptive; they are considered to comparatively assess the merits of the adaptive UCB and UCB-Left selection rules.

The pseudo-code of the BASCOP algorithm is shown in Figure 2.

Figure 2: BASCOP Algorithm

Input: number N of tree-walks, restart schedule, variable ordering heuristic VO, selection rule SR, expand rate k.
Data structure: a node stores
- a $state$: partial assignment,
- the $variable$ to be assigned next,
- children nodes corresponding to assigning different values to the $variable$,
- a top flag marking it as subject to SR or DFS.

Every time a new node must be created, its state is computed in the solver by adding the corresponding literal, and its variable is fetched from VO.

search tree $\mathcal{T} \leftarrow$ **new** `Node`(empty state)
$n_\ell \leftarrow 0$ for each (variable,value) couple ℓ
For $i = 1$ **to** N
  If restart **then** $\mathcal{T} \leftarrow$ **new** `Node`(empty state)
  If Tree-Walk($\mathcal{T}$) is successful
    process returned solution
EndFor
Tree-Walk($node$) **returns** $(depth, state)$ **:**
  $\langle$Check-node-state$\rangle$
  If $node$.top = `false`
    once every k, $node$.top $\leftarrow$ `true`
    otherwise, **Return** DFS($node$)
  Use SR to select $value$ among admissible ones
  $(d, s) =$ Tree-Walk($node$'s child associated to $value$)
  $\ell = (node.variable, value)$
  $n_\ell \leftarrow n_\ell + 1$
  $RAVE(\ell) \leftarrow RAVE(\ell) + (d - RAVE(\ell))/n_\ell$
  Return $(d+1, s)$
DFS($node$) **returns** $(depth, state)$ **:**
  $\langle$Check-node-state$\rangle$
  $(d, s) =$ **DFS**(leftmost admissible child)
  Return $(d+1, s)$
$\langle$**Check-node-state**$\rangle \equiv$
  If $node$'s state is terminal (failure,success)
    close the node, and its ancestors if necessary
    return $(0, node.state)$

Experimental Validation

This section reports on the empirical validation of BASCOP on three binary and n-ary CP problems: job shop scheduling (Taillard 1993), balance incomplete block design (BIBD) and car sequencing (the last two problems respectively correspond to pbs 28 and 1 in (Gent and Walsh 1999)). BASCOP is integrated within the state-of-the-art Gecode framework (Gecode Team 2012) and its results are comparatively assessed, using the depth-first-search, the balanced and ϵ-left (Section) strategies as baselines. In all experiments, the expand rate was set to 5.

Job Shop Scheduling

	Ta11	Ta12	Ta13	Ta14	Ta15	Ta16	Ta17	Ta18	Ta19	Ta20
DFS	1365	1367	1343	**1345**	1350	1360	1463	1397	1352	1350
UCB	**1357**	1370	**1342**	**1345**	**1339**	1365	**1462**	1407	**1332**	1356

Table 2: Best makespans obtained over 11 runs of 200 000 tree-walks on second set of Taillard instances, by DFS and UCB with parameters $C = 0.05, \rho = 2$. Bold numbers indicate best-known results.

Job shop scheduling, aimed at minimizing the schedule makespan, is modelled as a binary CP problem (Beck 2007). Upon its discovery, a new solution is used to i) update the model (requiring further solutions to improve on the current one); ii) bias the search toward the neighborhood of this solution along a local neighborhood search strategy. The search is initialized using the solutions of randomized Werner schedules, and the reported results are averaged over 11 independent runs. The variable ordering heuristics is based on wdeg-max (Boussemart et al. 2004). Multiple restarts are used, as the model involves a great many redundant variables, along a Luby sequence with factor 64.

The performance indicator is the relative distance to the best known makespan, monitored over 50 000 tree-walks for BASCOP using a UCB-left selection rule(Section) and DFS.

The results over the first four series of Taillard instances are reported in Table 1 (relative distance $\times$ 100), showing that BASCOP robustly outperforms DFS for a wide range

Instances	DFS	Bal.	ϵ-left					UCB																	
			ϵ	0.05	0.1	0.15	0.2	ρ	1				2				4				8			
								C	0.05	0.1	0.2	0.5	0.05	0.1	0.2	0.5	0.05	0.1	0.2	0.5	0.05	0.1	0.2	0.5
1–10	0.51	**0.39**		0.57	0.45	0.58	0.46		0.35	0.39	0.41	0.42	**0.32**	0.40	0.43	0.55	**0.34**	0.43	0.44	0.40	0.36	0.45	0.46	0.29
averages over parameters				0.51					**0.39**				0.43				0.40				**0.39**			
11–20	2.07	1.76		1.58	1.65	1.46	1.67		1.61	1.53	1.52	**1.39**	1.51	1.57	1.48	1.77	1.57	1.55	1.53	**1.40**	1.51	1.52	1.51	1.51
averages over parameters				1.59					**1.51**				1.58				**1.51**				**1.51**			
21–30	2.31	2.00		1.58	1.74	1.63	1.88		1.59	1.51	1.65	1.71	**1.47**	1.49	1.48	1.67	1.60	1.68	1.63	**1.42**	1.62	1.59	1.62	1.65
averages over parameters				1.71					1.62				**1.53**				1.58				1.62			
31–40	13.55	3.29		2.56	2.24	2.37	2.55		2.24	2.34	2.57	2.37	2.22	2.16	2.37	2.38	2.19	2.33	2.39	2.46	2.04	2.33	2.39	2.55
averages over parameters				2.43					2.38				**2.28**				2.34				2.33			

Table 1: BASCOP experimental validation on the Taillard job shop problems, 100 $\times$ relative distance to the best-known makespan, averaged over 11 runs, 50 000 tree walks.

of parameter values. Furthermore, the adaptive UCB-based search is shown to significantly improve on the non-adaptive balanced and ϵ-left strategies (except for the 1-10 series).

Further experiments, shown in Table 2, show that BASCOP discovers some of the current best-known makespans, previously established using dedicated CP and local search heuristics (Beck, Feng, and Watson 2011), at similar computational cost (200 000 tree-walks, circa one hour on Intel Xeon E5345, 2.33GHz).

Balance Incomplete Block Design (BIBD)

BIBD is a Boolean satisfaction problem. The goal of BASCOP is to find all solutions. Accordingly, no multiple restarts were considered; we did neither use dynamic variable ordering, nor local neighborhood search. Instances from (Mathon and Rosa 1985), characterized from their $v, k,$ and λ parameters, are considered; trivial instances and those for which no solution could be discovered by any method within 50 000 tree-walks are omitted. Table 3 reports the number of tree-walks required to find 50% of the solutions on the instances for which all solutions were found (top), and the number of solutions found after 50 000 tree-walks on other instances (bottom).

Overall, BASCOP consistently outperforms DFS (though to a lesser extent for large exploration constants, $C > .5$), which itself consistently outperforms the non-adaptive balanced strategy.

v	k	λ	DFS	bal.	UCB 0.05	UCB 0.1	UCB 0.2	UCB 0.5	UCB 1
number of iterations for half of solutions									
9	3	2	8654	8000	8862	8860	7473	7317	**7264**
9	4	3	13291	15144	12821	12824	**12794**	13524	13753
10	4	2	156	215	**153**	**153**	**153**	**153**	181
11	5	2	45	45	45	45	45	45	45
13	4	1	40	40	40	40	40	40	40
15	7	3	5007	5254	**1877**	1878	**1877**	1961	2773
16	4	1	**322**	394	377	379	378	392	340
16	6	2	1677	1947	**1130**	1131	1133	1139	1270
21	5	1	507	799	**484**	**484**	**484**	495	537
average			3300	3538	2865	2866	**2709**	2785	2911
number of solutions after 50K iterations									
10	3	2	19925	11136	17145	17172	17031	**18309**	**22672**
10	5	4	1454	1517	1552	1554	1550	1556	**1558**
13	4	2	824	1457	16597	**16654**	16596	2063	1898
15	3	1	21884	2443	22496	22505	22497	**23142**	15273
16	4	2	190	6	4726	**4727**	4725	247	392
16	6	3	180	-	416	416	**425**	306	64
19	3	1	18912	-	**19952**	**19952**	**19952**	15794	10190
19	9	4	-	-	18	18	18	**36**	-
21	3	1	-	-	16307	16289	**16329**	14764	9058
25	5	1	416	260	**460**	**460**	**460**	**460**	420
25	9	3	-	-	-	**12**	-	8	-
31	6	1	253	34	**347**	342	**347**	**347**	342
average			7388	3279	**9173**	8473	**9166**	6684	6516

Table 3: BASCOP experimental validation on BIBD: number of instances needed to find 50% of the solutions if all solutions are found in 50 000 tree-walks (top) or number of solutions found after 50 000 tree-walks (bottom).

Car Sequencing

Car sequencing is a CP problem involving circa 200 n-ary variables, with n ranging over $[20, 30]$. As mentioned, the UCB decision rule straightforwardly extends beyond the binary case; Multiple restarts were not considered eventually as they did not bring improvements; variable ordering based on *activity* (Michel and Van Hentenryck 2012) was used together with a static value ordering. 70 instances (ranging in 60-01 to 90-10 from (Gent and Walsh 1999)) are considered; the algorithm performance is the violation of the capacity constraint (number of extra stalls) averaged over the solutions found during 10 000 tree-walks.

The experimental results (Table 4) shows that CP solvers are still far from reaching state-of-the-art performance on these problems, especially when using the classical relaxation of the capacity constraint (Perron and Shaw 2004). Still, while DFS and balanced exploration yield the same results, BASCOP (with UCB selection rule) modestly but significantly (after a Wilcoxon signed-rank test) improves on DFS; the improvement is robust over a range of parameter settings, with C ranging in [.05, .5].

Discussion and Perspectives

The paper introduces BASCOP as a generic hybridization of MCTS and CP, and demonstrates its ability to provide good and robust results. BASCOP adapts MCTS to the specifics

	DFS	bal.	UCB 0.05	UCB 0.1	UCB 0.2	UCB 0.5
average gap	17.1	17.1	16.6	16.7	16.6	16.5
z-score vs DFS	-	0	3.21	2.59	3.44	3.20

Table 4: BASCOP experimental validation on car-sequencing: average violation after 10 000 tree-walks and significance of the improvement over DFS after Wilcoxon signed-rank test.

of CP tree search while preserving the generality of the underlying constraint engine and the applicability to any CP model. It is evaluated on three different domains, showing significant improvements over an efficient DFS baseline augmented with up-to-date dynamic variable ordering heuristics.

This work opens several perspectives for further research. A first perspective is to build and exploit node-based rewards in the no-restart context. Another potential source of improvements lies in the use of progressive-widening (Coulom 2006) to deal with many-valued variables.

Another perspective concerns the parallelization of BASCOP. Parallelization of MCTS has been studied in the context of games (Chaslot, Winands, and van den Herik 2008). Further work will consider how these approaches can be adapted within BASCOP, and assess their merits comparatively to parallel tree search based on work stealing (Chu, Schulte, and Stuckey 2009). In particular, parallel BASCOP might alleviate a current limitation of work stealing, that is, being blind to the most promising parts of the tree.

References

Auer, P.; Cesa-Bianchi, N.; and Fischer, P. 2002. Finite-time analysis of the multiarmed bandit problem. *Machine Learning* 47(2-3):235–256.

Beck, J.; Feng, T.; and Watson, J.-P. 2011. Combining constraint programming and local search for job-shop scheduling. *INFORMS Journal on Computing* 23(1):1–14.

Beck, J. C. 2007. Solution-guided multi-point constructive search for job shop scheduling. *JAIR* 29:49–77.

Boussemart, F.; Hemery, F.; Lecoutre, C.; and Sais, L. 2004. Boosting systematic search by weighting constraints. In *ECAI*, 146–150.

Chaslot, G.; Winands, M. H. M.; and van den Herik, H. J. 2008. Parallel monte-carlo tree search. In *Computers and Games*, 60–71.

Chu, G.; Schulte, C.; and Stuckey, P. 2009. Confidence-based work stealing in parallel constraint programming. In *CP*, 226–241.

Ciancarini, P., and Favini, G. 2009. Monte-Carlo Tree Search techniques in the game of Kriegspiel. In *IJCAI*, 474–479.

Coulom, R. 2006. Efficient selectivity and backup operators in Monte-Carlo Tree Search. In *Computers and Games*, 72–83.

Epstein, S.; Freuder, E.; Wallace, R.; Morozov, A.; and Samuels, B. 2002. The adaptive constraint engine. In *CP*, 525–542.

Gecode Team. 2012. Gecode: Generic constraint development environment. Available from `www.gecode.org`.

Gelly, S., and Silver, D. 2007. Combining online and offline knowledge in UCT. In *ICML*, 273–280. ACM.

Gent, I., and Walsh, T. 1999. CSP_{LIB}: A benchmark library for constraints. In *CP*, 480–481.

Harvey, W., and Ginsberg, M. 1995. Limited discrepancy search. In *IJCAI*, 607–615.

Hutter, F.; Hamadi, Y.; Hoos, H.; and Leyton-Brown, K. 2006. Performance prediction and automated tuning of randomized and parametric algorithms. In *CP*, 213–228.

Hutter, F.; Hoos, H.; and Stützle, T. 2007. Automatic algorithm configuration based on local search. In *AAAI*, 1152–1157.

Kocsis, L., and Szepesvári, C. 2006. Bandit based Monte-Carlo planning. In *ECML*, 282–293.

Lai, T., and Robbins, H. 1985. Asymptotically efficient adaptive allocation rules*1. *Advances in Applied Mathematics* 6:4–22.

Luby, M.; Sinclair, A.; and Zuckerman, D. 1993. Optimal speedup of las vegas algorithms. *IPL* 47:173–180.

Mathon, R., and Rosa, A. 1985. Tables of parameters for BIBD's with $r \leq 41$ including existence, enumeration, and resolvability results. *Ann. Discrete Math* 26:275–308.

Michel, L., and Van Hentenryck, P. 2012. Activity-based search for black-box constraint programming solvers. In *CPAIOR*, 228–243.

Nakhost, H., and Müller, M. 2009. Monte-Carlo exploration for deterministic planning. In Boutilier, C., ed., *IJCAI*, 1766–1771.

O'Mahony, E.; Hebrard, E.; Holland, A.; Nugent, C.; and O'Sullivan, B. 2008. Using case-based reasoning in an algorithm portfolio for constraint solving. In *AICS*.

Perron, L., and Shaw, P. 2004. Combining forces to solve the car sequencing problem. In *CPAIOR*, 225–239.

Rice, J. 1976. The algorithm selection problem. *Advances in Computers* 65–118.

Samulowitz, H., and Memisevic, R. 2007. Learning to solve QBF. In *AAAI*, 255–260.

Streeter, M.; Golovin, D.; and Smith, S. 2007. Combining multiple heuristics online. In *AAAI*, 1197–1203.

Sutton, R., and Barto, A. 1998. *Reinforcement Learning: An Introduction*. MIT Press.

Taillard, E. 1993. Benchmarks for basic scheduling problems. *European Journal of Operational Research* 64(2):278 – 285.

van Beek, P. 2006. Backtracking search algorithms. In *Handbook of Constraint Programming*. chapter 4, 85–134.

Wu, H., and Van Beek, P. 2008. Portfolios with deadlines for backtracking search. In *IJAIT*, volume 17, 835–856.

Xu, L.; Hutter, F.; Hoos, H.; and Leyton-Brown, K. 2008. Satzilla: Portfolio-based algorithm selection for SAT. *JAIR* 32:565–606.