Bandit-Based Search for Constraint Programming

Manuel Loth[1], Michèle Sebag[2], Youssef Hamadi[3], and Marc Schoenauer[2]

[1] Microsoft Research – INRIA joint centre, Palaiseau, France
[2] TAO, INRIA – CNRS – LRI, Université Paris-Sud, Orsay, France
[3] Microsoft Research, Cambridge, United Kingdom

Abstract. Constraint Programming (CP) solvers classically explore the solution space using tree-search based heuristics. Monte-Carlo Tree Search (MCTS), aimed at optimal sequential decision making under uncertainty, gradually grows a search tree to explore the most promising regions according to a specified reward function. At the crossroad of CP and MCTS, this paper presents the Bandit Search for Constraint Programming (BASCOP) algorithm, adapting MCTS to the specifics of the CP search. This contribution relies on i) a generic reward function suited to CP and compatible with a multiple restart strategy; ii) the use of depth-first search as roll-out procedure in MCTS. BASCOP, on the top of the Gecode constraint solver, is shown to significantly improve on depth-first search on some CP benchmark suites, demonstrating its relevance as a generic yet robust CP search method.

Keywords: adaptive search, value selection, bandit, UCB, MCTS.

1 Introduction

A variety of algorithms and heuristics have been designed in constraint programming (CP), determining which (variable, value) assignment must be selected at each step, how to backtrack on failures, and how to restart the search [1]. The selection of the algorithm or heuristics most appropriate to a given problem instance, intensively investigated since the late 70s [2], most often relies on supervised machine learning (ML) [3–7].

This paper advocates the use of another ML approach, namely reinforcement learning (RL) [8], to support the CP search. Taking inspiration from earlier work [9–12], the paper contribution is to extend the Monte-Carlo Tree Search (MCTS) algorithm to control the exploration of the CP search tree.

Formally, MCTS upgrades the multi-armed bandit framework [13, 14] to sequential decision making [15], leading to breakthroughs in the domains of e.g. games [16, 17] or automated planning [18]. MCTS proceeds by growing a search tree through consecutive tree walks, gradually biasing the search toward the most promising regions of the search space. Each tree walk, starting from the root, iteratively selects a child node depending on its empirical reward estimate and the confidence thereof, enforcing a trade-off between the exploitation of the

C. Schulte (Ed.): CP 2013, LNCS 8124, pp. 464–480, 2013.

best results found so far, and the exploration of the search space (more in section 2.3). The use of MCTS within the CP search faces two main difficulties. The first one is to define an appropriate reward attached to a tree node (that is, a partial assignment of the variables). The second difficulty is due to the fact that the CP search frequently involves multiple restarts [19]. In each restart, the current search tree is erased and a brand new search tree is built based on a new variable ordering (reflecting the variable criticality after e.g. their weighted degree, impact or activity). As the rewards attached to all nodes cannot be maintained over multiple restarts for tractability reasons, MCTS cannot be used as is.

A first contribution of the presented algorithm, named *Bandit-based Search for Constraint Programming* (BASCOP), is to associate to each (variable, value) assignment its average *relative failure depth*. This average can be maintained over the successive restarts, and used as a reward to guide the search. A second contribution is to combine BASCOP with a depth-first search, enforcing the search completeness in the no-restart case. A proof of principle of the approach is given by implementing BASCOP on the top of the Gecode constraint solver [20]. Its experimental validation on three benchmark suites, respectively concerned with the job-shop (JSP) [21], the balanced incomplete block design (BIBD) [22], and the car-sequencing problems, comparatively demonstrates the merits of the approach.

The paper is organized as follows. Section 2 discusses the respective relevance of supervised learning and reinforcement learning with regard to the CP search control, and describes the Monte Carlo Tree Search. Section 3 gives an overview of the BASCOP algorithm, hybridizing MCTS with CP search. Section 4 presents the experimental setting for the empirical validation of BASCOP and discusses the empirical results. The paper concludes with some perspectives for further research.

2 Machine Learning for Constraint Programming

This section briefly discusses the use of supervised machine learning and reinforcement learning for the control of CP search algorithms. For the sake of completeness, the Monte-Carlo Tree Search algorithm is last described.

2.1 Supervised Machine Learning

Most approaches to the control of search algorithms exploit a dataset that records, for a set of benchmark problem instances, i) the description of each problem instance after appropriate static and dynamic features [3, 23]; ii) the associated target result, e.g. the runtime of a solver. Supervised machine learning is applied on the dataset to extract a model of the target result based on the descriptive features of the problem instances. In SATzilla [3], a regression model predicting the runtime of each solver on a problem instance is built from the known instances, and used on unknown instances to select the solver with minimal expected run-time. Note that this approach can be extended to accommodate several restart strategies [24]. CPHydra [4] uses a similarity-based

approach (case-based reasoning) and builds a switching policy based on the most efficient solvers for the problem instance at hand. In [5], ML is likewise applied to adjust the CP heuristics online. The *Adaptive Constraint Engine* [25] can be viewed as an ensemble learning approach, where each heuristic votes for a possible (variable,value) assignment to solve a CSP. The methods *Combining Multiple Heuristics Online* [6] and *Portfolios with Deadlines* [26] are designed to build a scheduler policy in order to switch the execution of black-box solvers during the resolution process. Finally, optimal hyper-parameter tuning [7, 27] is tackled by optimizing the estimate of the runtime associated to parameter settings depending on the current problem instance.

2.2 Reinforcement Learning

A main difference between supervised learning and reinforcement learning is that the former focuses on taking a single decision, while the latter is interested in sequences of decisions. Reinforcement learning classically considers a Markov decision process framework $(\mathcal{S}, \mathcal{A}, p, r)$, where $\mathcal{S}$ and $\mathcal{A}$ respectively denote the state and the action spaces, p is the transition model ($p(s, a, s')$ being the probability of being in state s' after selecting action a in state s in a probabilistic setting; in a deterministic setting, $tr(s, a)$ is the node s' reached by selecting action a in state s) and $r : \mathcal{S} \mapsto \mathbb{R}$ is a bounded reward function. A policy $\pi : \mathcal{S} \mapsto \mathcal{A}$, starting in some initial state until arriving in a terminal state or reaching a time horizon, gathers a sum of rewards. The RL goal is to find an optimal policy, maximizing the expected cumulative reward.

RL is relevant to CP along two frameworks, referred to as offline and online frameworks. The offline framework aims at finding an optimal policy w.r.t. a family of problem instances. In this framework, the set of states describes the search status of any problem instance, described after static and dynamic feature values; the set of actions corresponds e.g. to the CP heuristics to be applied for a given lapse of time. An optimal policy associates an action to each state, in such a way that, over the family of problem instances (e.g., on average), the policy reaches optimal performances (finds a solution in the satisfiability setting, or reaches the optimal solution in an optimization setting) as fast as possible.

The online framework is interested in solving a single problem instance. In this framework, the set of states corresponds to a partial assignment of the variables and the set of admissible actions corresponds to the (variable, value) assignments consistent with the current state. An optimal policy is one which finds as fast as possible a solution (or, the optimal solution) for the problem instance at hand.

In the remainder of the paper, only the online framework will be considered; *states* and *nodes* will be used interchangeably. This online framework defines a specific RL landscape. Firstly, the transition model is known and deterministic; the next state $s' = tr(s, a)$ reached from a state s upon the (variable,value) assignment action a, is the conjunction of s and the (variable,value) assignment. Secondly, and most importantly, there is no clearly defined reward to be attached to intermediate states: e.g. in the satisfiability context, intrinsic rewards (satisfiability or unsatisfiability) can only be attached to terminal states. Furthermore,

such intrinsic rewards are hardly informative (e.g. all but a negligible fraction of the terminal states are unsatisfiable; and the problem is solved in general after a single satisfiable assignment is found).

The online framework thus makes it challenging for mainstream RL approaches to adjust the Exploration *vs* Exploitation trade-off at the core of RL. For this reason, the Monte-Carlo Tree Search approach is considered.

2.3 Monte Carlo Tree Search

The best known MCTS algorithm, referred to as Upper Confidence Tree (UCT) [15], extends the Upper Confidence Bound algorithm [14] 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 (initial state) and iteratively selects a child node (action) until arriving in a leaf node of the MCTS tree. 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 **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. The growth rate of the MCTS tree can be controlled through an *expand rate* parameter k, by adding a child node after the leaf node has been visited k times. Accordingly, the number of nodes in the tree is N/k, where N is the number of tree-walks.

The **roll-out phase** starts from the 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:

$$\begin{array}{l} n_{s,a} \leftarrow n_{s,a} + 1; \qquad n_s \leftarrow n_s + 1 \\ \bar{r}_{s,a} \leftarrow \bar{r}_{s,a} + (r_u - \bar{r}_{s,a})/n_{s,a} \end{array} \tag{2}$$

Additional heuristics have been considered, chiefly to prevent over-exploration when the number of admissible arms is large w.r.t the number of simulations (the so-called many-armed bandit issue [28]). Notably, the *Rapid Action Value*

Estimate (RAVE) heuristics is used to guide the exploration of the search space and the tree-building phase [16] when node rewards are based on few samples (tree-walks) and are thus subject to a high variance. In its simplest version, $RAVE(a)$ is set to the average reward taken over all tree-walks involving action a. The action selection is based on a weighted sum of the RAVE and the Upper Confidence Bound (Eq. (1)), where the RAVE weight decreases with the number n_s of visits to the current node [16].

A few work have pioneered the use of MCTS to explore a tree-structured assignment search space, in order to solve satisfiability or combinatorial optimization problem instances. In [9], MCTS is applied to boolean satisfiability; the node reward is set to the ratio of clauses satisfied by the current assignment, tentatively estimating how far this assignment goes toward finding a solution. In [11], MCTS is applied to Mixed Integer Programming, and used to control the selection of the top nodes in the CPLEX solver; the node reward is set to the maximal value of solutions built on this node. In [10], MCTS is applied to Job Shop Scheduling problems; it is viewed as an alternative to Pilot or roll-out methods, featuring an integrated and smart look-ahead strategy. Likewise, the node reward is set to the optimal makespan of the solutions built on this node.

3 The BaSCoP Algorithm

This section presents the BaSCoP algorithm (Algorithm 1), defining the proposed reward function and describing how the reward estimates are exploited to guide the search. Only binary variables will be considered in this section for the sake of simplicity; the extension to n-ary variables is straightforward, and will be considered in the experimental validation of BaSCoP (section 4). Before describing the structure of the BaSCoP search tree, let us first introduce the main two ideas behind the proposed hybridization of MCTS and the CP search.

Among the principles guiding the CP search [29], a first one is to select variables such that an eventual failure occurs as soon as possible (*First Fail* principle). A second principle is to select values that maximize the number of possible assignments. The *First Fail* principle is implemented by hybridizing MCTS with a mainstream variable-ordering heuristics (wdeg is used in the experiments). The latter principle will guide the definition of the proposed reward (section 3.2).

A second issue regards the search strategy used in the MCTS roll-out phase. The use of random search is not desirable, among other reasons as it does not enforce the search completeness in the no-restart context. Accordingly, the roll-out strategy used in BaSCoP implements a complete strategy, the depth first search.

3.1 Overview

The overall structure of the BaSCoP search space is displayed in Fig. 1. BaSCoP grows a search tree referred to as *top-tree* (the filled nodes in Fig. 1), which is a subtree of the full search tree. Each node is a partial assignment s (after

Algorithm 1. BASCOP

input : number N of tree-walks, restart schedule, selection rule SR, expand rate k.

data structure: a node stores
- a *state* : partial assignment as handled by the solver,
- the *variable* to be assigned next,
- children nodes corresponding to its admissible values,
- a *top* flag marking it as subject to SR or DFS,
- statistics: number n of visits, average failure depth avg.

Every time a new node must be created (first visit), its state is computed in the solver by adding the appropriate literal, and its variable is fetched from the solver.
All numeric variables are initialized to zero.

main loop :
- search tree $\mathcal{T} \leftarrow$ **new** `Node(`*empty state*`)`
- **for** N *tree-walks* **do**
  - **if** *restart* **then** $\mathcal{T} \leftarrow$ **new** `Node(`*empty state*`)`
  - **if** `Tree-walk(`$\mathcal{T}$`).`*state.success* **then**
    - process returned solution

function `Tree-walk(node)` **returns** (depth, state) **:**
- **if** node.state *is terminal (failure,success)* **then**
  - *close the node, and its ancestors if necessary*
  - **return** (0, node.state)
- **if** node.top = false **then**
  - **once every** k, node.top $\leftarrow$ true
  - **otherwise**, **return** `DFS(node)`
- node.n $\leftarrow$ node.n + 1
- *Use SR to select* value *among admissible ones*
- (d, s) = `Tree-walk(node`*'s child associated to* value`)`
- node.avg $\leftarrow$ node.avg + (d − node.avg)/node.n
- **if** d > node.avg **then** reward = 1
- **else** reward = 0
- **let** ℓ = (node.variable, value):
  - $n_\ell \leftarrow n_\ell + 1$
  - $RAVE_\ell \leftarrow RAVE_\ell + (\text{reward} - RAVE_\ell)/n_\ell$
- **return** (d + 1, s)

function `DFS(node)` **returns** (depth, state) **:**
- **if** node.state *is terminal (failure,success)* **then**
  - *close the node, and its ancestors if necessary*
  - **return** (0, node.state)
- (d, s) = `DFS(`*leftmost admissible child*`)`
- **return** (d + 1, s)

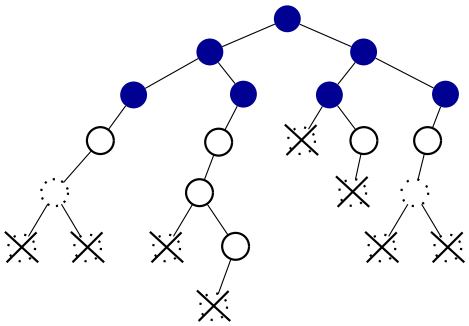

Fig. 1. Overview of the BASCOP search space. The top tree (filled nodes) is explored and extended along the MCTS framework. The bottom tree involves the tree paths under the top-tree leaves, iteratively updated by depth-first search. The status of a bottom-node is open (unfilled) or closed (dotted).

the constraint propagation achieved by the CP solver). The possible actions in s are to assign a fixed variable X (fetched from the variable-ordering heuristics) to value *true* or *false*, respectively represented as ℓ_X and $\ell_{\bar{X}}$ literals. Each child node of s (noted $s \wedge \ell$ with $\ell = \ell_X$ or $\ell_{\bar{X}}$) is associated a status: *closed* if the sub-tree associated to $s \wedge \ell$ has been fully explored; *open* if the sub-tree is being explored; *to-be-opened* if the node has not yet been visited. The value assigned to X is selected depending on the reward of the child nodes (section 3.2) and the selection rule (section 3.3).

BASCOP simultaneously explores and extends the top-tree along the MCTS framework, following successive tree-walks from the root until reaching a leaf node of the top-tree. The growth of the top-tree is controlled through the *expand rate* parameter k (section 2.3), where a child node is added below a leaf node s after s has been visited k times.

Upon reaching a leaf node of the top-tree, the BASCOP roll-out phase is launched until reaching a terminal state (failure or complete assignment). The roll-out phase uses the depth-first-search (DFS) strategy. DFS only requires to maintain a tree path below each leaf node; specifically, it requires to maintain the status of every node in these tree paths, referred to as bottom nodes (depicted as unfilled nodes in Fig. 1). By construction, DFS proceeds by selecting the left child node unless it is closed. Thereby, BASCOP enables a systematic exploration of the subtrees below its leaf nodes, thus enforcing a complete search in the no-restart setting.

3.2 Relative Failure Depth Reward

In the MCTS spirit, the choice among two child nodes must be guided by the average performance or reward attached to these child nodes, and the confidence thereof. Defining a reward attached to a partial assignment however raises several difficulties, as discussed in section 2. Firstly, the performance attached to

the terminal states below a node might be poorly informative, e.g. in the satisfiability context. Secondly and most importantly, 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 nogoods) maintained over the restarts. When rebuilding the CP search tree from scratch, a new variable ordering is computed from these indicators, expectedly resulting in more efficient and shorter tree-paths. Naturally, BASCOP must accommodate multiple restarts in order to define a generic CP search strategy. For tractability reasons however, BASCOP can hardly maintain all top-trees built along multiple restarts, or the rewards attached to all nodes in these top-trees. On the other hand, estimating node rewards from scratch after each restart is poorly informative too, as the rewards are estimated from insufficiently many tree-walks.

Taking inspiration from the RAVE heuristics (section 2.3), it thus comes to associate a reward to each ℓ_X and $\ell_{\bar{X}}$ literals, where X ranges over the variables of the problem. The proposed reward measures the impact of the literal on the depth of the failure, as follows. Formally, let s denote a current node with ℓ_X and $\ell_{\bar{X}}$ as possible actions. Let $\bar{d}_{s,f}$ denote the average depth of the failures occurring below s. Literal ℓ (with $\ell = \ell_X$ or $\ell_{\bar{X}}$) receives an instant reward 1 (respectively, 0) if the failure of the current tree-path occurs at depth $d > \bar{d}_{s,f}$ (resp., $d < \bar{d}_{s,f}$). The rationale for this reward definition is twofold. On the one hand, the values to be assigned to a variable only need to be assessed relatively to each other (recall that the variable ordering is fixed and external to BASCOP). On the other hand, everything else being equal, the failure due to a (variable, value) assignment should occur later than sooner: intuitively, a shorter tree-walk likely contains more bad literals than a longer tree-walk, everything else being equal.

Overall, the BASCOP reward associated to each literal ℓ, noted $r(\ell)$, averages the instant rewards gathered in all tree-paths where ℓ is selected in a top-tree node s. Indicator $n(\ell)$ counts the number of times literal ℓ is selected in a top tree node. As desired, reward $r(\ell)$ and indicator $n(\ell)$ can be maintained over multiple restarts, and thus based on sufficient evidence. Their main weakness is to aggregate the information from different contexts due to dynamic variable ordering (in particular the top-tree nodes s where literal ℓ is selected might be situated at different tree-depths) and due to multiple restarts. The aggregation might blur the estimate of the literal impact; however, the blurring effect is mitigated as the aggregation affects both ℓ_X and $\ell_{\bar{X}}$ literals in the same way.

3.3 Selection Rules

Let s and X respectively denote the current node and the variable to be assigned. BASCOP uses different rules in order to select among the possible assignments of X (literals ℓ_X and $\ell_{\bar{X}}$) depending on whether the current node s belongs to the top or the bottom tree.

In the bottom tree, the depth-first-search applies, always selecting the left child node unless its status is *closed*. Note that DFS easily accommodates value ordering: in particular, the local neighborhood search [21] biased toward the neighborhood of the last found solution (see section 4.2) can be enforced by setting the left literal to the one among ℓ_X and $\ell_{\bar{X}}$ which is satisfied by this last solution.

In the top-tree, several selection rules have been investigated:

- **Balanced SR** alternatively selects ℓ_X and $\ell_{\bar{X}}$;
- **ϵ-left SR** selects ℓ_X with probability $1 - \epsilon$ and $\ell_{\bar{X}}$ otherwise, thus implementing a stochastic variant of the limited discrepancy search [30];
- **UCB SR** selects the literal with maximal reward upper-bound (Eq. (1))

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

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

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.

3.4 Computational Complexity

BASCOP undergoes a time complexity overhead compared to DFS, due to the use of tree-walks instead of the optimized backtrack procedure, directly jumping to a parent or ancestor node. A tree-walk involves: i) the selection of a literal in each top-node; ii) the creation of a new node every k visits to a leaf node; iii) the update of the reward values for each literal. The tree-walk overhead thus amounts to h arithmetic computations, where h is the average height of the top-tree.

However, in most cases these computations are dominated by the cost of creating a new node, which involves constraint propagation upon the considered assignment.

With regard to its space complexity, BASCOP includes N/k top nodes after N tree-walks, where k is the expand rate; it also maintains the DFS tree-paths behind each top leaf node, with complexity $\mathcal{O}(Nh'/k)$, where h' is the average height of the full tree. The overall space complexity is thus increased by a multiplicative factor N/k; however no scalability issue was encountered in the experiments.

4 Experimental Validation

This section reports on the empirical validation of BASCOP on three binary and n-ary CP problems: job shop scheduling problems (JSP) [31], balance incomplete

block design (BIBD) and car sequencing (the last two problems respectively correspond to problems 28 and 1 in [32]).

4.1 Experimental Setting

BASCOP is implemented on the top of the state-of-the-art Gecode framework [20]. The goal of the experiments is twofold. On the one hand, the adaptive exploration *vs* exploitation MCTS scheme is assessed comparatively to the depth-first-search baseline. On the other hand, the relevance of the relative-depth-failure reward (section 3.2) is assessed by comparing the adaptive selection rules to the fixed balanced and ϵ-left selection rules (section 3.3).

The BASCOP expand rate parameter k is set to 5, after a few preliminary experiments showing good performances in a range of values around 5. The performances (depending on the problem family) are reported versus the number of tree-walks, averaged over 11 independent runs unless otherwise specified. The computational time is similar for all considered approaches, being granted that the DFS baseline uses the same tree-walk implementation as BASCOP[1]. The comparison of the runtimes is deemed to be fair as most of BASCOP computational effort is spent in the tree-walk part, and will thus take advantage of an optimized implementation in further work.

4.2 Job Shop Scheduling

Job shop scheduling, aimed at minimizing the schedule makespan, is modelled as a binary CP problem [21]. 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, that is, using the insertion algorithm of [33] with randomized flips in the duration-based ranking of operations. The variable ordering heuristics is based on wdeg-max [34]. Multiple restarts are scheduled along a Luby sequence with factor 64.

The performance indicator is the *mean relative error* (MRE), that is the relative distance to the best known makespan m^* ($(makespan - m^*)/m^*$), averaged over the runs and problem instances of a series. MRE is monitored over 50 000 BASCOP tree-walks, comparing the following selection rules: *none*, which corresponds to DFS standalone; *balanced*, which corresponds to a uniform exploration of the top nodes; *ϵ-left*, where the exploration is biased towards the left child nodes, and the strength of the bias is controlled from parameter ϵ; *UCB-left*, where the exploration-exploitation trade-off based on the relative-depth-failure reward is controlled from parameter C, and the bias toward the left is controlled from parameter ρ. The results on the first four series of Taillard instances are

[1] This implementation is circa twice longer than the optimized tree-walk Gecode implementation − which did not allow however the solution-guided search procedure used for the JSP and car sequencing problems at the time of the experiments.

Table 1. BASCOP experimental validation on the Taillard job shop problems: mean relative error w.r.t. the best known makespan, averaged on 11 runs (50 000 tree walks)

Selection rule			Results on instance sets 1-10	11-20	21-30	31-40
None (DFS)			0.51	2.07	2.31	13.55
Balanced			0.39	1.76	2.00	3.29
ϵ-left		ϵ				
		0.05	0.57	1.58	1.58	2.56
		0.1	0.45	1.65	1.74	2.24
		0.15	0.58	1.46	1.63	2.37
		0.2	0.46	1.67	1.88	2.55
	average		0.51	1.59	1.71	2.43
UCB	ρ	C				
	1	0.05	0.35	1.61	1.59	2.24
	1	0.1	0.39	1.53	1.51	2.34
	1	0.2	0.41	1.52	1.65	2.57
	1	0.5	0.42	**1.39**	1.71	2.37
	2	0.05	0.32	1.51	1.47	2.22
	2	0.1	0.40	1.57	1.49	2.16
	2	0.2	0.43	1.48	1.48	2.37
	2	0.5	0.55	1.77	1.67	2.38
	4	0.05	0.34	1.57	1.60	2.19
	4	0.1	0.43	1.55	1.68	2.33
	4	0.2	0.44	1.53	1.63	2.39
	4	0.5	0.40	1.40	**1.42**	2.46
	8	0.05	0.36	1.51	1.62	**2.04**
	8	0.1	0.45	1.52	1.59	2.33
	8	0.2	0.46	1.51	1.62	2.39
	8	0.5	**0.29**	1.51	1.65	2.55
	average		0.40	1.53	1.59	2.33

Table 2. Best makespans obtained out of 11 runs of 200 000 tree-walks on the 11-20 series of Taillard instances, comparing DFS and BASCOP with UCB-Left selection rule with parameters $C = 0.05, \rho = 2$. Bold numbers indicate best known results so far.

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

reported in Table 1, showing that BASCOP robustly outperforms DFS for a wide range of parameter values. Furthermore, the adaptive UCB-based search improves on average on all fixed strategies, except for the 1-10 series.

Complementary experiments displayed in Table 2,show that BASCOP discovers some of the current best-known makespans, previously established using dedicated CP and local search heuristics [35], at similar computational cost (circa one hour on Intel Xeon E5345, 2.33GHz for 200 000 tree-walks).

4.3 Balance Incomplete Block Design (BIBD)

BIBD is a family of challenging Boolean satisfaction problems, known for their many symmetries. We considered instances from [22], characterized from their $v, k,$ and λ parameters. A simple Gecode model with lexicographic order of the rows and columns is used. Instances for which no solution could be discovered by any method within 50 000 tree-walks are discarded. Two goals are tackled: finding a single solution; finding them all.

Table 3. BASCOP experimental validation on BIBD: number of tree-walks needed to find the first solution. Best results are indicated in bold; '-' indicates that no solution was found after 50 000 tree-walks.

					BASCOP				
v	k	λ	DFS	bal.	C 0.05	C 0.1	C 0.2	C 0.5	C 1
9	3	2	49	49	49	49	49	49	49
9	4	3	45	45	45	45	45	45	45
10	3	2	63	63	63	63	63	63	63
10	4	2	45	45	45	45	45	45	45
10	5	4	333	669	357	355	355	**256**	509
11	5	2	45	45	45	45	45	45	45
13	3	1	**161**	331	176	176	176	243	265
13	4	1	40	40	40	40	40	40	40
13	4	2	**202**	935	216	216	216	499	463
15	3	1	131	131	131	131	131	131	131
15	7	3	567	1579	**233**	**233**	**233**	451	370
16	4	1	164	166	164	164	164	164	164
16	4	2	**639**	12583	1297	1279	1282	1324	2492
16	6	2	503	821	315	315	315	**314**	407
16	6	3	7880	-	3200	3198	**2559**	2594	4394
19	3	1	671	-	**493**	**493**	**493**	709	3541
19	9	4	-	-	26251	25310	25383	**2004**	-
21	3	1	-	-	**779**	**779**	**779**	1183	6272
21	5	1	261	634	**217**	**217**	**217**	**217**	277
25	5	1	3425	11168	636	636	636	643	**541**
25	9	3	-	-	-	35940	-	**30131**	-
31	6	1	13889	36797	**882**	**882**	**882**	953	893

After preliminary experiments, neither variable ordering nor value ordering (e.g. based on the local neighborhood search) heuristics were found to be effective. Accordingly, BASCOP with UCB selection rule is assessed comparatively to the DFS standalone and BASCOP with balanced selection rule.

Table 3 reports the number of iterations needed to find the first solution; a single run is considered. Satisfactory results are obtained for low values of the trade-off parameter C. On-going experiments consider lower C values.

The All-solution setting is considered to investigate the search efficiency of BASCOP. On easy problems where all solutions can be found after 50 000 tree-walks, same number of tree-walks is needed to find all solutions. The search

Table 4. BASCOP experimental validation on BIBD: Number of tree-walks needed to find 50% of the solutions when all solutions are found in 50 000 tree-walks

					BASCOP				
v	k	λ	DFS	bal.	C 0.05	C 0.1	C 0.2	C 0.5	C 1
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

Table 5. BASCOP experimental validation on BIBD: Number of solutions found in 50 000 tree-walks

					BASCOP				
v	k	λ	DFS	bal.	C 0.05	C 0.1	C 0.2	C 0.5	C 1
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

efficiency is therefore assessed from the number of tree-walks needed to find 50% of the solutions, displayed in Table 4. Likewise, there exists a plateau of good results for low values of parameter C.

For more complex problems, the number of solutions found after 50 000 tree-walks is displayed in Table 5.

Overall, BASCOP consistently outperforms DFS, particularly so for low values of the exploration constant C, while DFS consistently outperforms the non-adaptive balanced strategy. For all methods, the computational cost is ca 2 minutes on Intel Xeon E5345, 2.33GHz for 50 000 tree-walks).

4.4 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

Table 6. BASCOP experimental validation on car-sequencing: top line: violation after 10 000 tree-walks, averaged over 70 problem instances. bottom line: significance of the improvement over DFS after Wilcoxon signed-rank test.

			BASCOP			
	DFS	bal.	C 0.05	C 0.1	C 0.2	C 0.5
average gap	17.1	17.1	16.6	16.7	16.6	16.5
p-value	-	0	10^{-3}	$5\,10^{-3}$	10^{-3}	10^{-3}

extends beyond the binary case. After preliminary experiments, multiple restart strategies were not considered as they did not bring any improvements. Variable ordering based on *activity* [36] was used together with a static value ordering. 70 instances (ranging in 60-01 to 90-10 from [32]) are considered; the algorithm performance is the violation of the capacity constraint (number of extra stalls) averaged over the solutions found after 10 000 tree-walks.

The experimental results (Table 6) show that CP solvers are far from reaching state-of-the-art performance on these problems, especially when using the classical relaxation of the capacity constraint [37]. Still, while DFS and balanced exploration yield same results, BASCOP with UCB selection rule significantly improves on DFS after a Wilcoxon signed-rank test; the improvement is robust over a range of parameter settings, with C ranging in [.05, .5].

5 Discussion and Perspectives

The generic BASCOP scheme presented in this paper achieves the adaptive control of the variable-value assignment in the CP search along the Monte-Carlo Tree Search ideas. The implementation of BASCOP on the top of the Gecode solver and its comparative validation on three families of CP problems establish, as a proof of principle that cues about the relevance of some (variable,value) assignments can be efficiently extracted and exploited online.

A main contribution of the proposed scheme is the proposed (variable,value) assignment reward, enforcing the BASCOP compatibility with multiple restart strategies. Importantly, BASCOP can (and should) be hybridized with CP heuristics, such as dynamic variable ordering or local neighborhood search; the use of the depth-first search strategy as roll-out policy is a key issue commanding the completeness of the BASCOP search, and its efficiency.

This work opens several perspectives for further research. Focussing on the no-restart CP context, a first perspective is to apply the proposed relative failure depth reward to partial assignments. Another extension concerns the use of progressive-widening [38] or X-armed bandits [39] to deal with respectively many-valued or continuous variables.

A mid-term perspective concerns the parallelization of BASCOP, e.g. through adapting the parallel MCTS approaches developed in the context of games [40]. In particular, parallel BASCOP could be hybridized with the parallel CP approaches based on work stealing [41], and contribute to the collective identification of the most promising parts of the search tree.

Acknowledgments. The authors warmly thank Christian Schulte for his help and many insightful suggestions about the integration of MCTS within the Gecode solver.

References

1. van Beek, P.: Backtracking Search Algorithms. In: Handbook of Constraint Programming (Foundations of Artificial Intelligence), pp. 85–134. Elsevier Science Inc., New York (2006)
2. Rice, J.: The algorithm selection problem. In: Advances in Computers, pp. 65–118 (1976)
3. Xu, L., Hutter, F., Hoos, H., Leyton-Brown, K.: Satzilla: Portfolio-based algorithm selection for SAT. JAIR 32, 565–606 (2008)
4. O'Mahony, E., Hebrard, E., Holland, A., Nugent, C., O'Sullivan, B.: Using case-based reasoning in an algorithm portfolio for constraint solving. In: AICS (2008)
5. Samulowitz, H., Memisevic, R.: Learning to solve QBF. In: AAAI, 255–260 (2007)
6. Streeter, M., Golovin, D., Smith, S.: Combining multiple heuristics online. In: AAAI, pp. 1197–1203 (2007)
7. Hutter, F., Hoos, H.H., Leyton-Brown, K., Stützle, T.: Paramils: An automatic algorithm configuration framework. J. Artif. Intell. Res (JAIR) 36, 267–306 (2009)
8. Sutton, R., Barto, A.: Reinforcement Learning: an introduction. MIT Press (1998)
9. Previti, A., Ramanujan, R., Schaerf, M., Selman, B.: Monte-carlo style UCT search for boolean satisfiability. In: Pirrone, R., Sorbello, F. (eds.) AI*IA 2011. LNCS, vol. 6934, pp. 177–188. Springer, Heidelberg (2011)
10. Runarsson, T.P., Schoenauer, M., Sebag, M.: Pilot, Rollout and Monte Carlo Tree Search Methods for Job Shop Scheduling. In: Hamadi, Y., Schoenauer, M. (eds.) LION 2012. LNCS, vol. 7219, pp. 160–174. Springer, Heidelberg (2012)
11. Sabharwal, A., Samulowitz, H., Reddy, C.: Guiding combinatorial optimization with UCT. In: Beldiceanu, N., Jussien, N., Pinson, É. (eds.) CPAIOR 2012. LNCS, vol. 7298, pp. 356–361. Springer, Heidelberg (2012)
12. Loth, M.: Hybridizing constraint programming and Monte-Carlo Tree Search: Application to the job shop problem. In: Nicosia, G., Pardalos, P. (eds.) Learning and Intelligent Optimization Conference (LION 7), Springer, Heidelberg (2013)
13. Lai, T., Robbins, H.: Asymptotically efficient adaptive allocation rules. Advances in Applied Mathematics 6, 4–22 (1985)
14. Auer, P., Cesa-Bianchi, N., Fischer, P.: Finite-time analysis of the multiarmed bandit problem. Machine Learning 47(2-3), 235–256 (2002)
15. Kocsis, L., Szepesvári, C.: Bandit based monte-carlo planning. In: Fürnkranz, J., Scheffer, T., Spiliopoulou, M. (eds.) ECML 2006. LNCS (LNAI), vol. 4212, pp. 282–293. Springer, Heidelberg (2006)
16. Gelly, S., Silver, D.: Combining online and offline knowledge in UCT. In: International Conference on Machine Learning, pp. 273–280. ACM (2007)
17. Ciancarini, P., Favini, G.: Monte-Carlo Tree Search techniques in the game of Kriegspiel. In: International Joint Conference on Artificial Intelligence, pp. 474–479 (2009)

18. Nakhost, H., Müller, M.: Monte-Carlo exploration for deterministic planning. In: Boutilier, C. (ed.) International Joint Conference on Artificial Intelligence, pp. 1766–1771 (2009)
19. Luby, M., Sinclair, A., Zuckerman, D.: Optimal speedup of las vegas algorithms. Information Processing Letters 47(4), 173–180 (1993)
20. Gecode Team: Gecode: Generic constraint development environment (2012), `www.gecode.org`
21. Beck, J.C.: Solution-guided multi-point constructive search for job shop scheduling. Journal of Artificial Intelligence Research 29, 49–77 (2007)
22. Mathon, R., Rosa, A.: Tables of parameters for BIBD's with $r \leq 41$ including existence, enumeration, and resolvability results. Ann. Discrete Math. 26, 275–308 (1985)
23. Hutter, F., Hamadi, Y., Hoos, H., Leyton-Brown, K.: Performance prediction and automated tuning of randomized and parametric algorithms. In: Benhamou, F. (ed.) CP 2006. LNCS, vol. 4204, pp. 213–228. Springer, Heidelberg (2006)
24. Haim, S., Walsh, T.: Restart strategy selection using machine learning techniques. In: Kullmann, O. (ed.) SAT 2009. LNCS, vol. 5584, pp. 312–325. Springer, Heidelberg (2009)
25. Epstein, S., Freuder, E., Wallace, R., Morozov, A., Samuels, B.: The adaptive constraint engine. In: Van Hentenryck, P. (ed.) CP 2002. LNCS, vol. 2470, pp. 525–540. Springer, Heidelberg (2002)
26. Wu, H., Van Beek, P.: Portfolios with deadlines for backtracking search. In: IJAIT, vol. 17, pp. 835–856 (2008)
27. Schneider, M., Hoos, H.H.: Quantifying homogeneity of instance sets for algorithm configuration. In: Hamadi, Y., Schoenauer, M. (eds.) LION 2012. LNCS, vol. 7219, pp. 190–204. Springer, Heidelberg (2012)
28. Wang, Y., Audibert, J., Munos, R.: Algorithms for infinitely many-armed bandits. In: Advances in Neural Information Processing Systems, pp. 1–8 (2008)
29. Refalo, P.: Impact-based search strategies for constraint programming. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 557–571. Springer, Heidelberg (2004)
30. Harvey, W., Ginsberg, M.: Limited discrepancy search. In: International Joint Conference on Artificial Intelligence, pp. 607–615 (1995)
31. Taillard, E.: Benchmarks for basic scheduling problems. European Journal of Operational Research 64(2), 278–285 (1993)
32. Gent, I., Walsh, T.: Csplib: A benchmark library for constraints. In: Jaffar, J. (ed.) CP 1999. LNCS, vol. 1713, pp. 480–481. Springer, Heidelberg (1999)
33. Werner, F., Winkler, A.: Insertion techniques for the heuristic solution of the job shop problem. Discrete Applied Mathematics 58(2), 191–211 (1995)
34. Boussemart, F., Hemery, F., Lecoutre, C., Sais, L.: Boosting systematic search by weighting constraints. In: ECAI, pp. 146–150 (2004)
35. Beck, J., Feng, T., Watson, J.P.: Combining constraint programming and local search for job-shop scheduling. INFORMS Journal on Computing 23(1), 1–14 (2011)
36. Michel, L., Van Hentenryck, P.: Activity-based search for black-box constraint programming solvers. In: Beldiceanu, N., Jussien, N., Pinson, É. (eds.) CPAIOR 2012. LNCS, vol. 7298, pp. 228–243. Springer, Heidelberg (2012)
37. Perron, L., Shaw, P.: Combining forces to solve the car sequencing problem. In: Régin, J.-C., Rueher, M. (eds.) CPAIOR 2004. LNCS, vol. 3011, pp. 225–239. Springer, Heidelberg (2004)

38. Coulom, R.: Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In: van den Herik, H.J., Ciancarini, P., Donkers, H.H.L.M(J.) (eds.) CG 2006. LNCS, vol. 4630, pp. 72–83. Springer, Heidelberg (2007)
39. Bubeck, S., Munos, R., Stoltz, G., Szepesvári, C.: X-armed bandits. Journal of Machine Learning Research 12, 1655–1695 (2011)
40. Chaslot, G.M.J.-B., Winands, M.H.M., van den Herik, H.J.: Parallel Monte-Carlo Tree Search. In: van den Herik, H.J., Xu, X., Ma, Z., Winands, M.H.M. (eds.) CG 2008. LNCS, vol. 5131, pp. 60–71. Springer, Heidelberg (2008)
41. Chu, G., Schulte, C., Stuckey, P.: Confidence-based work stealing in parallel constraint programming. In: Gent, I.P. (ed.) CP 2009. LNCS, vol. 5732, pp. 226–241. Springer, Heidelberg (2009)