Bandit-based Search for Constraint Programming

Manuel Loth, Michèle Sebag, Youssef Hamadi, Marc Schoenauer

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, CNRS − INRIA − 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 point, 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 [25, 24, 23, 9], 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 [11, 12] to sequential decision making [10], leading to breakthrough in the domains of e.g. games [28, 13] or automated planning [14]. 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

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 branching variables). The second difficulty is due to the fact that the CP search frequently involves multiple restarts [15]. 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 *relative failure depth*. This estimate can be maintained over the successive restarts, and used 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 [32], and presenting its comparative experimental validation on three benchmark suites, respectively concerned with the job-shop (JSP) [16], the balanced incomplete block design (BIBD) [17], and the car-sequencing problems.

The paper is organized as follows. Section 2 briefly discusses the respective merits of supervised learning and reinforcement learning with regard to the CP search control, and describes 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 BASCOPand 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 self-containedness, 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 recording for a set of benchmark problem instances i) the description of each problem instance after appropriate static and dynamic features [3, 18]; 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, and used to select the solver with minimal expected run-time. Note that this approach can be extended

to accommodate several restart strategies [19]. 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 [20] 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 [6] and Portfolios with deadlines [21] 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, 22] is tackled by optimizing the estimate of the runtime associated to parameter settings depending on the current problem instance, which can be viewed as a surrogate-optimization problem.

2.2 Reinforcement Learning

A main difference between supervised learning and reinforcement learning is that the former focuses on taking single decisions, 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, $tr(s, a)$ denotes 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 to each state an action, in such a way that on average − over the family of problem instances − 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 insatisfiability) 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 insatisfiable; and the problem is solved 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) [10], extends the Upper Confidence Bound algorithm [12] 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 performing this building phase only once every k tree-walk (stochastically, or based on the number of visits). Accordingly, the number of nodes in the tree is linear in the number of tree-walks, with a factor k.

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 [27]). Notably, the *Rapid Action Value Estimate* (RAVE) heuristics is meant to guide the exploration of the search space and the tree-building early phases [28]. In its simplest version, $RAVE(a)$ is set to the average reward taken over all tree-walks involving action a.

A few work have pioneered the use of MCTS to explore a tree-structured assignment search space, in order to solve combinatorial optimization or satisfiability problem instances. In [23], 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 [24], 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.
In [25], 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 the assignment goes toward finding a solution.

3 Overview of BASCOP

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 readability[4]. Before describing the structure of the BASCOP search tree, let us first discuss the principles guiding the hybridization of MCTS and the CP search.

Among the principles guiding the CP search [26], a first one is to select variables in order to fail as soon as possible; a second one is to select values that maximize the number of possible assignments. The former *First Fail* principle is implemented by hybridizing MCTS with a mainstream variable-ordering heuristics (wdeg is used in the experiments). The latter one, understood as *deep-fail* principle will guide the definition of the proposed reward (section 3.2). As mentioned in the introduction, a main difficulty is to be compatible with a multiple restart strategy, erasing and rebuilding the search tree along a pre-defined schedule.

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 (depth-first search is used in the experiments).

[4] The extension to n-ary variables is straightforward, and will be considered in the experimental validation of BASCOP (section 4).

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 *iterations* **do**
  - **if** *restart* **then** $\mathcal{T} \leftarrow$ **new** `Node(`*empty state*`)`
  - **if** `Tree-walk(`$\mathcal{T}$`)` *is successful* **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*
- $(\mathsf{d}, \mathsf{s}) =$ `Tree-walk(`node*'s child associated to* value`)`
- node.avg $\leftarrow (\mathsf{d} -$ node.avg$)/$node.n
- **if** d $>$ node.avg **then** reward $= 1$
- **else** reward $= 0$
- **let** $\ell =$ (node.variable, value) **in**
  - $n_\ell \leftarrow n_\ell + 1$
  - $RAVE_\ell \leftarrow RAVE_\ell + (\mathsf{reward} - RAVE_\ell)/n_\ell$
- **return** $(\mathsf{d}+1, \mathsf{s})$

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$)$
- $(\mathsf{d}, \mathsf{s}) =$ `DFS(`*leftmost admissible child*`)`
- **return** $(\mathsf{d}+1, \mathsf{s})$

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

This tree is to be explored in the MCTS fashion, that is following successive tree-walks from the root to a leaf node. In this mode, the classical depth-first-search (DFS) strategy can be defined as systematically selecting the leftmost branch, among those that have not been closed yet. Only the tree-path from the root node to the last visited leaf has to be maintained: nodes in the left part of the complete tree w.r.t. the current tree-path have been fully explored, and all nodes in the right part remain to be considered.

As mentioned (section 2.3), 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 the reached leaf node once every k tree-walk, k being referred to as the *expand rate*. Within this partial tree, the selection of the branch to be visited is achieved through a exploration/exploitation criterion (eg UCB, 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 3.2). 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 sub-tree. The consequence of following such a complete search is that, contrary to the usual MCTS derivations, the roll-out phase requires some storage of the visited nodes. Thus, the BASCOP nodes are divided in two parts: an upper part handled by the adaptive policy, and the nodes handled by the systematic policy, beneath this upper part.

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.

3.2 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

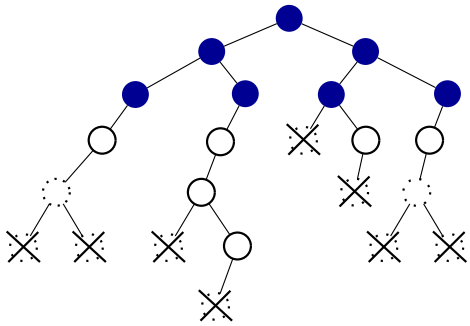

Fig. 1. Structure of the BASCOP search tree: the top-tree where the adaptive selection rule applies (filled nodes), the DFS-rollouts parts (unfilled nodes) including failed leaves (crossed) and closed nodes (dotted).

most natural option in the MCTS spirit would be gather statistics about the reward obtained from *each 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 2.3). 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 w is measured by its failure depth[5] d_w: intuitively, a shorter tree-walk contains more bad literals than a longer tree-walk, everything else being equal. More precisely, a reward of 1 is accumulated for each step taken down a tree-walk, each branch in a tree-walk w thus receiving a cumulative reward corresponding to the *relative* depth failure: $d_w - d_{w,\ell}$, where ℓ denotes the literal associated to the branch, $d_{w,\ell}$ the depth of the branch where ℓ was selected, and d_w the failure depth of w. Finally, this instant reward is aggregated in the RAVE statistics concerning all occurrences of ℓ in the top-tree:

$$RAVE(\ell) = \text{ Average } \{d_w - d_{w,\ell} | \text{ tree-walk } w \text{ involving } \ell\}$$

Overall, BASCOP thus maintains for each literal ℓ the number n_ℓ of tree-walks involving it in a top node, and its average reward $RAVE(\ell)$.

3.3 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.

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 neighbourhood search [16] 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 4.1).

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 [29]) 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.

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

3.4 Computational Complexity

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

However, in most cases such operations are dominated by the creation of the new node itself, which involves constraint propagation when updating the partial assignment.

As for the space-complexity, DFS stores a single tree-path of size h. With an expand rate k, BASCOP includes N/k top nodes after N tree-walks, and circa N/k DFS paths attached to each leaf node. The overall number of nodes 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) [30], balance incomplete block design (BIBD) and car sequencing (the last two problems respectively correspond to problems 28 and 1 in [31]). BASCOP is integrated within the state-of-the-art Gecode framework [32] and its results are comparatively assessed, using the depth-first-search, the balanced and ϵ-left (Section 3.3) strategies as baselines. In all experiments, the expand rate was set to 5, after a set of experiments consistently showing better results around this value. All results are reported in terms of number of iterations, the observed running times being equivalent for all methods. It should be noted, however, that the implementation of DFS was following BASCOP's tree-walk framework, with no specific optimization, which gives it about twice longer running times than Gecode's inner implementation. Since a large part of BASCOP's computation consists in DFS runs, it appeared that running the same implementation for the pure DFS allowed a fair comparison. Moreover, Gecode's implementation did not allow, at the time of our experiments, the solution-guided search procedure used for JSP and car sequencing.

4.1 Job Shop Scheduling

Job shop scheduling, aimed at minimizing the schedule makespan, is modelled as a binary CP problem [16]. 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 neighbourhood of this solution along a local neighbourhood 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 reported results are averaged over 11 independent runs. The variable ordering heuristics is based on wdeg-max [34]; accordingly, multiple restarts are used, 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 instances of a series. MRE is monitored over 50 000 tree-walks for BASCOP using DFS roll-outs and the following selection rules: *none*, which amounts to a single DFS run; *balanced*, for assessing the benefits of simply diversifying early decisions; *epsilon-left*, that stochastically biases this diversification towards the left branches; *UCB-left*, that combines left-biased diversification with the adaptive focus towards deep failure. Different levels of left bias are tested through the ϵ and ρ parameters for, respectively, ϵ-left and UCB-left rules.

The results over the first four series of Taillard instances are reported in Table 1, showing that BASCOP robustly outperforms DFS for a wide range 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 [35], at similar computational cost (200 000 tree-walks, circa one hour on Intel Xeon E5345, 2.33GHz).

4.2 Balance Incomplete Block Design (BIBD)

BIBD is a Boolean satisfaction problem, for which no variable-ordering and value-ordering heuristics appear to be useful. Accordingly, no multiple restart is needed, and a single search is performed in a static tree. We consider both problems of finding all solutions and finding a single one. Instances from [17], 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 iterations needed to find the first solution, while Table 4 concerns the search of all solutions.

The nature of the results in the latter differs according to the success of the search. When the 50 000 tree-walks were sufficient to exhaust the search space, the number of iterations for exhibiting all solutions was found to be equivalent for all methods, being almost equal to that of the exhaustive search. In this case, the information reported is the number of tree-walks after which 50% of the solutions were found, which illustrates the ability to focus the search on good regions in the first iterations. For the other instances, on which a complete search could not be performed within the 50 000 iterations, the number of solutions that were found is reported.

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.

Table 1. BASCOP experimental validation on the Taillard job shop problems, MRE in percentage, averaged over 11 runs of 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				
ϵ-left		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
	ρ	C				
UCB	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

4.3 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; no specific multi-armed bandit heuristics (e.g. [27]) was used. Multiple restarts were not considered eventually as they did not bring 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 [31]) 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 5) 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 [37]. 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;

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.

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

v	k	λ	DFS	bal.	UCB 0.05	UCB 0.1	UCB 0.2	UCB 0.5	UCB 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**

the improvement is robust over a range of parameter settings, with C ranging in [.05, .5].

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 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

Table 4. 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).

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

potential source of improvements lies in the use of progressive-widening [38] to deal with many-valued variables.

Another perspective concerns the parallelization of BASCOP. Parallelization of MCTS has been studied in the context of games [39]. 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 [40]. In particular, parallel BASCOP might alleviate a current limitation of work stealing, that is, being blind to the most promising parts of the tree.

Table 5. 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.

	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

References

1. van Beek, P.: Backtracking Search Algorithms. In: Handbook of Constraint Programming (Foundations of Artificial Intelligence). Elsevier Science Inc., New York, NY, USA (2006) 85–134
2. Rice, J.: The algorithm selection problem. Advances in Computers (1976) 65–118
3. Xu, L., Hutter, F., Hoos, H., Leyton-Brown, K.: Satzilla: Portfolio-based algorithm selection for SAT. JAIR **32** (2008) 565–606
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. (2007) 255–260
6. Streeter, M., Golovin, D., Smith, S.: Combining multiple heuristics online. In: AAAI. (2007) 1197–1203
7. Hutter, F., Hoos, H., Stützle, T.: Automatic algorithm configuration based on local search. In: AAAI. (2007) 1152–1157
8. Sutton, R., Barto, A.: Reinforcement Learning: An Introduction. MIT Press (1998)
9. Loth, M., Sebag, M., Hamadi, Y., Schoenauer, M., Schulte, C.: 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), Catania, Italy (January 2013)
10. Kocsis, L., Szepesvári, C.: Bandit based Monte-Carlo planning. In: ECML. (2006) 282–293
11. Lai, T., Robbins, H.: Asymptotically efficient adaptive allocation rules*1. Advances in Applied Mathematics **6** (1985) 4–22
12. Auer, P., Cesa-Bianchi, N., Fischer, P.: Finite-time analysis of the multiarmed bandit problem. Machine Learning **47**(2-3) (2002) 235–256
13. Ciancarini, P., Favini, G.: Monte-Carlo Tree Search techniques in the game of Kriegspiel. In: IJCAI. (2009) 474–479
14. Nakhost, H., Müller, M.: Monte-Carlo exploration for deterministic planning. In Boutilier, C., ed.: IJCAI. (2009) 1766–1771
15. Luby, M., Sinclair, A., Zuckerman, D.: Optimal speedup of las vegas algorithms. IPL **47** (1993) 173–180
16. Beck, J.C.: Solution-guided multi-point constructive search for job shop scheduling. JAIR **29** (2007) 49–77
17. Mathon, R., Rosa, A.: Tables of parameters for BIBD's with $r \leq 41$ including existence, enumeration, and resolvability results. Ann. Discrete Math **26** (1985) 275–308
18. Hutter, F., Hamadi, Y., Hoos, H., Leyton-Brown, K.: Performance prediction and automated tuning of randomized and parametric algorithms. In: CP. (2006) 213–228
19. Haim, S., Walsh, T.: Restart strategy selection using machine learning techniques. In: SAT. (2009) 312–325
20. Epstein, S., Freuder, E., Wallace, R., Morozov, A., Samuels, B.: The adaptive constraint engine. In: CP. (2002) 525–542
21. Wu, H., Van Beek, P.: Portfolios with deadlines for backtracking search. International Journal on Artificial Intelligence Tools **17**(05) (2008) 835–856
22. Schneider, M., Hoos, H.: Quantifying homogeneity of instance sets for algorithm configuration. In Hamadi, Y., Schoenauer, M., eds.: Learning and Intelligent Optimization. Lecture Notes in Computer Science, Springer Berlin Heidelberg (2012) 190–204

23. Sabharwal, A., Samulowitz, H., Reddy, C.: Guiding combinatorial optimization with uct. In: CPAIOR. (2012) 356–361
24. Runarsson, T.P., Schoenauer, M., Sebag, M.: Pilot, rollout and monte carlo tree search methods for job shop scheduling. In: LION. (2012) 160–174
25. 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: Artificial Intelligence Around Man and Beyond. Volume 6934 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2011) 177–188
26. Refalo, P.: Impact-based search strategies for constraint programming. In: CP. (2004) 557–571
27. Wang, Y., Audibert, J., Munos, R.: Algorithms for infinitely many-armed bandits. In: NIPS. (2008) 1–8
28. Gelly, S., Silver, D.: Combining online and offline knowledge in UCT. In: ICML, ACM (2007) 273–280
29. Harvey, W., Ginsberg, M.: Limited discrepancy search. In: IJCAI. (1995) 607–615
30. Taillard, E.: Benchmarks for basic scheduling problems. European Journal of Operational Research **64**(2) (1993) 278 – 285
31. Gent, I., Walsh, T.: $\mathrm{CSP}_{\mathrm{LIB}}$: A benchmark library for constraints. In: CP. (1999) 480–481
32. Gecode Team: Gecode: Generic constraint development environment (2012) Available from `www.gecode.org`.
33. Werner, F., Winkler, A.: Insertion techniques for the heuristic solution of the job shop problem. Discrete Applied Mathematics **58**(2) (1995) 191 – 211 ¡ce:title¿Workshop on Discrete Algoritms¡/ce:title¿.
34. Boussemart, F., Hemery, F., Lecoutre, C., Sais, L.: Boosting systematic search by weighting constraints. In: ECAI. (2004) 146–150
35. Beck, J., Feng, T., Watson, J.P.: Combining constraint programming and local search for job-shop scheduling. INFORMS Journal on Computing **23**(1) (2011) 1–14
36. Michel, L., Van Hentenryck, P.: Activity-based search for black-box constraint programming solvers. In: CPAIOR. (2012) 228–243
37. Perron, L., Shaw, P.: Combining forces to solve the car sequencing problem. In: CPAIOR. (2004) 225–239
38. Coulom, R.: Efficient selectivity and backup operators in Monte-Carlo Tree Search. In: Computers and Games. (2006) 72–83
39. Chaslot, G., Winands, M.H.M., van den Herik, H.J.: Parallel monte-carlo tree search. In: Computers and Games. (2008) 60–71
40. Chu, G., Schulte, C., Stuckey, P.: Confidence-based work stealing in parallel constraint programming. In: CP. (2009) 226–241