Hybridizing Constraint Programming and Monte-Carlo Tree Search: Application to the Job Shop problem

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

[1] Microsoft Research, Cambridge, UK
[2] TAO, CNRS–INRIA–LRI, Université Paris-Sud, France
[3] School of ICT, KTH Royal Institute of Technology, Sweden

Abstract. Constraint Programming (CP) solvers classically explore the solution space using tree search-based heuristics. Monte-Carlo Tree-Search (MCTS), a tree search-based method aimed at sequential decision making under uncertainty, simultaneously estimates the reward associated to the sub-trees, and gradually biases the exploration toward the most promising sub-trees. This paper examines the tight combination of MCTS and CP on the job shop problem (JSP). The contribution is twofold. Firstly, a reward function compliant with a multiple restart policy is proposed. Secondly, a biased MCTS node selection rule based on this reward is proposed. Its integration within the Gecode constraint solver is shown to compete with JSP-specific CP approaches on difficult JSP instances.

1 Goal

This paper focuses on hybridising Constraint Programming (CP) and Monte-Carlo Tree Search (MCTS) methods. The proof of concept of the approach is given on the job-shop problem (JSP), where JSPs are modelled as CP problem instances, and MCTS is hybridised with the Gecode constraint solver environment. Let us first briefly remind the reader of how to model JSPs in CP and of the background notations of MCTS, referring the interested reader to respectively [1] and [3] for a comprehensive presentation. The proposed hybrid approach, referred to as Fail-Depth Bandit-Search (*FDBS*), is thereafter described. The paper concludes with a discussion on the first experimental results on the difficult *Taillard 11-20* 20×15 problem instances.

CP-based resolution of JSP. The job-shop problem, one of the classical scheduling problems, is concerned with allocating jobs to machines while minimising the overall make-span. The CP modelling of JSP proceeds by considering a sequence of problems: after a given solution with make-span m has been found, the problem is modified by adding the constraint $makespan < m$.

Following [1], reportedly the best CP approach to JSP, we implemented JSP in Gecode as follows:

– the restart policy follows a Luby sequence: the search is restarted after the specified number of failures is reached or when a new solution is found;
– the variable ordering is based on the *weighted-degree* heuristics – found as Max Accumulated Failure Count (AfcMax) in Gecode[4];
– the search heuristics is a Depth-First-Search, starting from the last found solution, i.e. the left value associated to a variable is the value assigned to this variable in the previous best solution.

Monte-Carlo Tree Search. MCTS is concerned with optimal sequential decision under uncertainty, known in particular for its breakthrough in the game of Go [3]. MCTS proceeds by gradually and asymmetrically growing a search tree, carefully balancing the exploitation of the most promising sub-trees and the exploration of the rest of the tree. MCTS iterates N tree-walks, a.k.a simulations, where each tree-walk involves four phases:

– In the so-called bandit phase, the *selection rule* of MCTS selects the child node of the current node –starting from the root– depending on the empirical reward associated to each child node, and the number of times it has been visited. Denoting respectively $\hat{\mu}_i$ and n_i the empirical reward and the number of visits of the i-th child node, the most usual selection rule, inspired from the Multi-Armed Bandit setting, is:

$$\text{Select } i^* = \arg\max \left\{ \hat{\mu}_i + C\sqrt{\frac{\log \sum_i n_i}{n_i}} \right\} \tag{1}$$

– When MCTS reaches a leaf node, this node is attached a child node –the tree thus involves N nodes after N tree-walks– and MCTS enters the roll-out phase.
– In the roll-out phase, nodes (aka actions) are selected using a default (usually randomised) policy, until arriving at a terminal state.
– In this terminal state, the overall reward associated to this tree-walk is computed and used to update the reward $\hat{\mu}_i$ of every node in the tree-walk.

MCTS is frequently combined with the so-called RAVE (Rapid Action Value Estimate) heuristics, which stores the average reward associated to each action (averaging all rewards received along tree-walks involving this action). In particular, the RAVE information is used to select the new nodes added to the tree.

2 *FDBS*

The considered CP setting significantly differs from the MCTS one. Basically, CP relies on the multiple restart strategy, which implies that it deals with many,

[4] Indeed specific JSP-driven heuristics have been used in CP approaches, e.g. [2]. The paper scope however is to investigate the coupling of MCTS with generic CP search.

mostly narrow, trees. In contrast, MCTS proceeds by searching in a single, gradually growing and eventually very large tree. The CP and MCTS approaches were thus hybridised by attaching average rewards (section 2.1) to each value of a variable (section 2.2). Secondly, *FDBS* relies on redefining the selection rule in the bandit- and in the roll-out phases (sections 2.3,2.4).

2.1 Reward

Although an optimisation problem is addressed, the value to be optimised – the make-span– is of no direct use in the definition of the reward associated to each tree-walk. Indeed, all but a few of these tree-walks are terminated by a failure, that is an early detection of the infeasibility of an improvement over the last-found schedule. No significant information seems to be exploitable from the make-span's domain at a failure point. Hence, the depth of a failure is used as a base for the reward, as an indication of its closeness to success, with the following argument: the more variables are assigned the correct value, the deeper the failure.

Since the assignments of a given variable can occur at different depths within a tree and through the successive trees, it seemed reasonable, and was empirically validated, to consider the *relative failure depth* rather than the absolute one: after a tree-walk failing at depth d_f, for each variable v that was assigned, letting d_v be its assignment depth and x the assigned value, a reward $(d_f - d_v)$ is added to the statistics of (v, x).

2.2 RAVE

In the line of [3], the most straightforward option would have been to associate to each node in the tree (that is, a (variable, value) assignment conditioned by the former assignment nodes) the average objective associated to this partial assignment. This option is however irrelevant, because the frequent restarts make it ineffective to associate an average reward to an assignment *conditioned by other variable assignments*: there are not enough tree-walks to compute reliable statistics before they are discarded by the change of context (the new tree). Hence, a radical form of RAVE was used, where statistics are computed for each (variable,value) assignment, independently of the context (previous variables assignments).

2.3 Roll-Outs

The roll-out phase also presents a key difference from the usual MCTS settings, where these bottom parts of the tree-walks are not stored in memory, and are following a stochastic procedure as a form of Monte-Carlo sampling. In the CP setting addressed here, it is desirable to make the search complete, that is exhaustive when given enough time. This is one reason why the policy followed under each leaf of the MCTS tree, was chosen to be DFS. An other reason is

that left branches correspond to the last solution, around which it is reasonable to search for the next one. Contrary to random roll-outs, DFS requires node storage, yet only the last path of one DFS needs to be stored. Thus DFS provides a complete, simple, and low-storage roll-out policy, from which to gather statistics. This results in forms of *Interleaved Depth-First Search* ([4], with the key difference that the effort may be adaptively dispatched onto different regions of the search tree.

2.4 Selection rules

Let us remind that the left branch associated to each variable corresponds to the value assigned to this variable in the previous best solution. The selection rules thus determine the neighbourhood of the previous solution which is explored in the current tree.

Several rules have been considered:

- **Balanced**: selects alternatively the left and the right node;
- **ϵ-left**: selects the left node with probability $1 - \epsilon$, and can be seen as a stochastic emulation of Limited Discrepancy Search;
- **UCB**: selects a node according to eq. (1) with no bias towards the left branch;
- **UCB-Left**: same as UCB, where different constants C_{right} and C_{left} are used to enforce the bias toward the left branch.

Figure 1 illustrates the domains and shapes designed by the selection rule and roll-out policies, by an example using a *Balanced* selection rule and DFS roll-outs.

3 Results and discussion

Fig. 2 depicts the overall results in terms of mean relative error w.r.t. the best (non CP-based) solution found in the literature, on the *Taillard 11-20* problem suite (20×15), averaged on 11 independent runs, versus the number of tree-walks. The computational cost is ca 30mn on a PC with Intel dual-core CPU 2.66GHz. Compared to DFS, a simple diversification improves only on the early stages, while a left-biased one yields a significant improvement, of the same order as a failure-depth one, and improvements seem to add up when combining both biases.

An other lesson learnt is that *FDBS* matches the CP-based state of the art [1]: the use of MCTS was found to compensate for the lack of JSP-specific variable ordering.

Further work is concerned with reconsidering the variable ordering based on the MCTS rewards, combining per-node and per-variable statistics, as well as assessing the generality of the proposed approach on different problems.

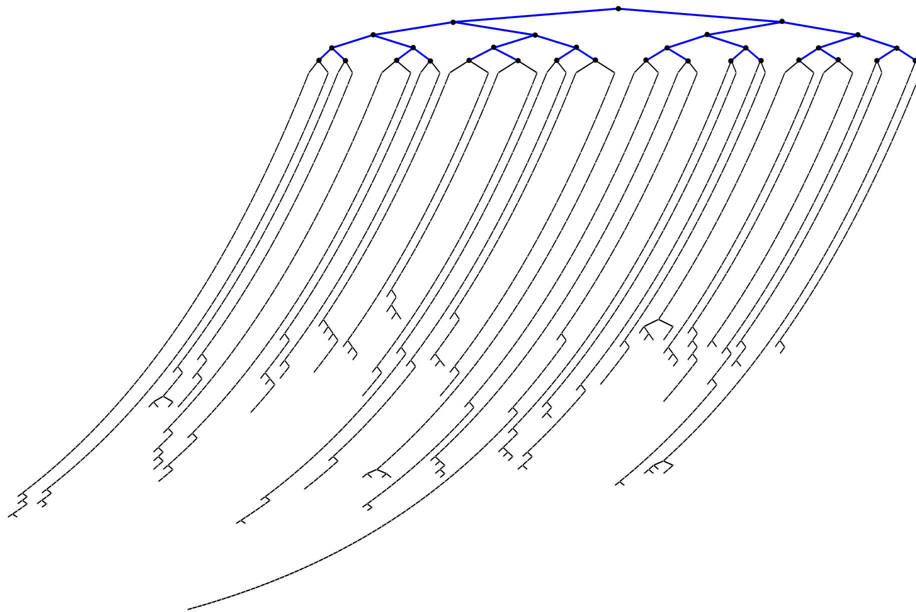

Fig. 1. Balanced + DFS search tree. Concurrent DFS are run under each leaf of the –growing– MCTS tree (dotted nodes).

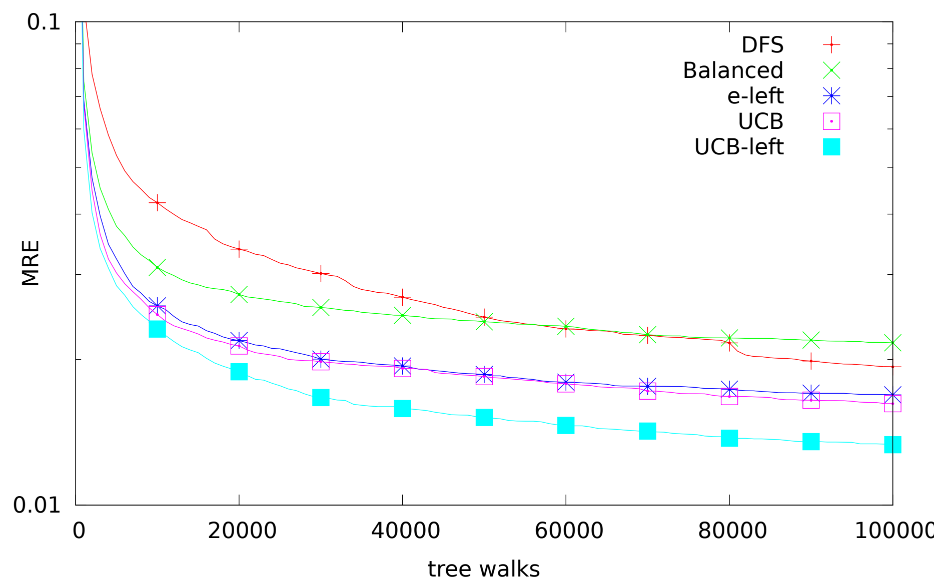

Fig. 2. Mean relative error, for 11 runs on the 10 20x20 Taillard instances

References

1. Beck J.C.: Solution-Guided Multi-Point Constructive Search for Job Shop Scheduling. Journal of Artificial Intelligence Research 29, pp 49–77 (2007)
2. Watson J.P., Beck J.C.: A Hybrid Constraint Programming / Local Search Approach to the Job-Shop Scheduling Problem. CPAIOR 2008: 263-277
3. Gelly S. et al: The grand challenge of computer Go: Monte Carlo tree search and extensions. Commun. ACM 55(3): 106-113 (2012)
4. Meseguer P.: Interleaved depth-first search. IJCAI 1997: vol. 2, 13821387