2015 IEEE 27th International Conference on Tools with Artificial Intelligence

RLBS: An Adaptive Backtracking Strategy Based on Reinforcement Learning for Combinatorial Optimization

Ilyess Bachiri, Jonathan Gaudreault, Claude-Guy Quimper
FORAC Research Consortium
Université Laval
Québec, Canada
ilyess.bachiri@cirrelt.ca
jonathan.gaudreault@forac.ulaval.ca
claude-guy.quimper@ift.ulaval.ca

Brahim Chaib-draa
Université Laval
Québec, Canada
brahim.chaib-draa@ift.ulaval.ca

***Abstract*—Combinatorial optimization problems are often very difficult to solve and the choice of a search strategy has a tremendous influence over the solver's performance. A search strategy is said to be *adaptive* when it dynamically adapts to the structure of the problem instance and identifies the areas of the search space that contain good solutions. We introduce an algorithm (RLBS) that learns to efficiently backtrack when searching non-binary trees. Branching can be carried on using any usual variable/value selection strategy. However, when backtracking is needed, the selection of the node to target involves reinforcement learning. As the trees are non-binary, we have the opportunity to backtrack many times to each node during the search, which allows learning which nodes generally lead to the best rewards (that is, to the most interesting leaves). RLBS is evaluated for a scheduling problem using real industrial data. It outperforms classic (non-adaptive) backtracking strategies (DFS, LDS) as well as an adaptive branching strategy (IBS).**

***Keywords*-Search; Backtracking; Learning; Optimization;**

I. Introduction

Combinatorial optimization problems are often very difficult to solve and the choice of a search strategy has a tremendous influence over the solver's performance. To solve a problem using search, one needs to choose a variable selection strategy (defining the order in which variables will be instantiated), a value selection strategy (defining the sequence in which we will try the variable possible values) and a backtracking strategy (that determines to which node we should backtrack/backjump, when a leaf is reached or a dead-end is encountered). Some backtracking policies are encoded into full deterministic algorithms (e.g. Depth-First Search, DFS) while others rely on more dynamic node evaluation mechanisms (e.g. Best-First Search). Others (e.g. Limited Discrepancy Search [9]) can be implemented as a deterministic iterative algorithm or as a node evaluator [3]. Whatever combination of variable selection, value selection, and backtracking strategy is used, the search remains complete. We only change visiting priorities in order to quickly find good solutions.

A strategy is said to be *adaptive* when it dynamically adapts to the structure of the problem and identifies the areas of the search space that contain good solutions in order to search those areas first. Some have proposed adaptive variable/value selection strategies (e.g. Impact-based Search (IBS) [17]) or adaptive backtracking strategies (e.g. Adaptive Discrepancy Search [7], proposed for distributed optimization problems).

In this paper, we consider a machine learning approach for backtracking/node evaluation, which improves the performance of the solver. More specifically, we use Reinforcement Learning (RL) to identify the areas of the search space that contain good solutions. The approach was developed for optimization problems for which the search space is encoded as a non-binary tree (decision variables are non-binary). As the trees are non-binary, we have the opportunity to backtrack multiple times to each node during the search. This allows learning which nodes generally lead to the best rewards (that is, to the most interesting leaves). Our backtracking strategy can be combined with any variable/value selection strategies.

Section II reviews some preliminary concepts regarding adaptive search and reinforcement learning. Section III explains how backtracking can be encoded as a reinforcement learning task and introduces the proposed algorithm (*Reinforcement Learning Backtracking Search*, or RLBS). Section IV presents results for a complex industrial problem that combines planning and scheduling. RLBS is compared to more classic (non-adaptive) search strategies (DFS, LDS) as well as an other adaptive branching strategy (IBS). Section V concludes the paper.

II. Background

Solving a combinatorial optimization problem using global search comes down to defining three key elements: a variable selection strategy, a value selection strategy, and a backtracking strategy [20]. The search space is structred as a tree based on the variable/value strategies. Regardless of the variable/value strategies applied, the search tree always covers the entire search space. The backtracking strategy defines in which order the nodes will be visited. Backtracking strategies can be implemented as iterative algorithms or node evaluation mechanisms [3].

A. Learning Variable/Value Selection Strategies

Some algorithms learn during the search which variables are the most difficult to instantiate, in order to dynamically change the order of the variables (e.g. YIELDS [10]). In [4] and [8], each time a constraint causes a failure, the priority of the variables involved in this constraint is increased.

In Impact Based Search (IBS) [17], the impact of the variables is measured by observing how their instantiation reduces the size of the search space. Since IBS picks the variable to assign and the value to try all at once, it can be considered learning a combination of a variable and value ordering strategies.

B. Learning to Backtrack

Approaches where the system learns to evaluate the quality of the nodes are of particular interest for backtracking strategies. Ruml [18] makes an interesting proposal regarding this. While a basic Limited Discrepancy Search (LDS) policy gives the same importance to any discrepancy, Best Leaf First Search (BLFS) dynamically attributes different weights to discrepancies according to their depth. BLFS uses a linear regression in order to establish the value of the weights. The model was not really used in order to define a backtracking strategy. Instead, the search algorithm proceeds by a series of successive descents in the tree. Ruml has achieved very good results with this algorithm (see [19]). It was the inspiration for the following algorithm.

Adaptive Discrepancy Search (ADS) [7] is an algorithm that was proposed for distributed optimization but it could be used in a classic COP context. During the search, it dynamically learns which nodes it pays the most to backtrack to (in order to concentrate on those areas of the tree first). For each node, it tries learning a function $Improvement(i)$ predicting how good would be the first leaf reached after backtracking to this node for the i-th time, in comparison to previous backtracks to the same node. The drawback of this method is that (1) whenever a new solution is found, a function needs to be learned for every ancestor node of the tree leaf corresponding to that solution, and (2) the learning process involves regression over a vector that grows over time [13]. Hence, the learning becomes more resource consuming as the search goes on.

C. Reinforcement Learning

The algorithm in Section III introduces a simplified learning mechanism based on a basic reinforcement learning technique.

The fundamental idea of Reinforcement Learning (RL) is to figure out a way to map actions to situations in order to maximize the total reward. The learner is not told which actions to take, it must discover by itself which actions lead to the highest reward (at long-term). Actions may affect not only the immediate reward but also the next situation and, through all, all subsequent rewards [2]. Moreover, the actions may not lead to the expected result due to the uncertainty of the environment.

RL uses a formal framework defining the interaction between the learner and the environment in terms of states, actions, and rewards. The environment that supports RL is typically formulated as a finite-state Markov Decision Process (MDP). In each state $s \in S$, a set of actions $a \in A$ are available to the learner, among which it has to pick the one that maximizes the cumulative reward. The evaluation of actions is entirely based on the learner's experience, built through its interactions with the environment. The goal of the learner is to find, through its interactions with the environment, an optimal policy $\pi : S \rightarrow A$ maximizing the cumulative reward. The cumulative reward is either expressed as a sum of all the rewards $R = r_0 + r_1 + \cdots + r_n$ or as a discounted sum of the rewards $R = \sum_t \gamma^t r_t$. The discount factor $0 \leq \gamma \leq 1$ is applied to promote the recent rewards. The discounted sum representation of the cumulative reward is mostly used for an MDP with no terminal state.

In a RL task, each action a, in each state s, is associated with a numeric value $Q(s, a)$ that represents the desirability to take the action a in the state s. These values are called Q-Values. The higher the Q-Value, the more likely the action is going to lead to a good solution, according to the learner's judgment. Every time a reward is returned to the learner, the learner must update the Q-Value of the action that has led to this reward. However, the older Q-Value should not be completely forgotten, otherwise the learner would be acting based on the very last experience every single time. To do so, we keep a part of the old Q-Value and we update it with a part of the new experience. Also, we assume that the learner is going to act optimally afterward. Moreover, the expected future rewards need to be discounted to express the idea of the sooner a reward is received, the better.

Let s be the current state, s' the next state, a an action, r the returned reward after having taken the action a, α the learning rate, and γ the discount factor. The update formula for the Q-Values is as follows:

$$Q_{t+1}(s_t, a_t) \leftarrow (1-\alpha) Q_t(s_t, a_t) + \alpha [r_{t+1} + \gamma \max_{a'} Q_t(s'_{t+1}, a')] \quad (1)$$

This update formula comes in handy when the learner has to learn an action-value representation, like in Q-Learning [21]. The update is almost instantaneous.

D. Reinforcement Learning and Search

The idea of using RL in solving combinatorial problems is supported by many publications [14], [16], [22]. Some researches tried to apply RL to solve optimization problems and some have considered solving Constraint Satisfaction Problems (CSP) using RL techniques.

For instance, Xu et al. [22] proposed a formulation of a CSP as a RL task. A set of different variable ordering heuristics is provided to the algorithm that learns which one to use, and when to use it, in order to solve a CSP in a shorter amount of time. The learning process is accomplished in an offline manner and applied on different instances of the same CSP. The states are the instances or sub-instances of the CSP and the actions are defined as the variable ordering heuristics. A reward is assigned each time an instance is solved. This approach relies on Q-learning to learn the optimal variable ordering heuristic at each decision point of the search tree, for a given (sub)-instance of the CSP.

Moreover, Loth et al. [11] have proposed the Bandit Search for Constraint Programming (BASCOP) algorithm that guides the seach (branching decisions) during the search, based on statistical gathered estimates BASCOP has been applied on a job shop problem in [12].

Miagkikh et al. [14] also proposes a local search technique using RL. This approach solves COPs based on a population of RL agents. The pairs $\langle \text{variable}, \text{value} \rangle$ are considered as the RL task states, and the branching strategies as the actions. Each RL agent is assigned a specific area of the search space where it has to learn and find good local solutions. The expertise of the entire population of RL agents is used. A new solution is produced by taking a part of the locally-best solution found by one agent and by completing it with the expertise of another agent.

Moll et al. [16] see the local search as a policy of a Markov Decision Process (MDP) where states represent solutions and actions define neighboring solutions. Reinforcement learning techniques can be used to learn a cost function in order to improve local search. One way to do so is to learn a new cost function over multiple search trajectories of the same instance. Boyan and Moore's STAGE algorithm [5] follows this approach and alternates between using the learned and the original cost function. By enhancing the predictive accuracy of the learned cost function, the guidance of the heuristics improves as the search goes on.

Another approach that uses reinforcement learning to improve local search in the context of combinatorial optimization is to learn a cost function off-line, and then use it on new instances of the same problem. Zhang and Dietterich's work [23] falls into this category.

III. RLBS: Backtracking as a Reinforcement Learning Task

This section introduces *Reinforcement Learning Backtracking Search* (RLBS). Branching is performed according to any usual variable/value selection heuristic. Each time we reach a leaf/solution, we need to select the node to backtrack to. To each available candidate (node with at least one unvisited child) corresponds a possible *action* ("backtracking to this node"). Once we select a node, the search continues from that point until we reach a new leaf/solution. The difference between the quality of this new solution and the best solution so far is the *reward* we get for performing the previous action. As we are searching a non-binary tree, we backtrack multiple times to each node during the search. This is an opportunity to identify the actions that pay the most (that is, nodes that are more likely to lead to interesting leaves/solutions). Since the method is implemented as a node evaluation mechanism, the search remains complete.

This situation reminds the k-armed-bandit problem [1], a single-state reinforcement learning problem. Many actions are possible (pulling one of the arms/levels of the slot machine). Each action may lead to a stochastic reward and we need balancing between exploration and exploitation. In our specific backtracking situation, performing an action makes us discover new nodes/actions, in addition to giving us a reward (which is stochastic and non-stationary).

A. Learning

As in classic reinforcement learning, the valuation (Q-value) of an action a is updated each time we get a reward after performing a. As we are in a single-state environment, the discount factor γ is equal to 0 and (1) reduces to (2):

$$Q_{t+1}(a_t) \leftarrow Q_t(a_t) + \alpha[r_{t+1}(a_t) - Q_t(a_t)] \qquad (2)$$

where $r(a)$ is the reward and α is the learning rate.

The next action to perform is selected based on those valuations. A node that paid well at first but never got good solutions afterward will see its Q-value decrease over time, until it becomes less interesting than other nodes/actions.

B. Initialization of the Algorithm

At the beginning of the search, we descend to the first leaf of the tree using a DFS. We then backtrack once to each open node (this is similar to the first 2 iterations of LDS), which allows computing their Q-Values. Then, we start using the Q-Values in order to choose the next node to backtrack to. Each time a new node is visited for the first time, its Q-Value is initialized using its parent's value.

IV. Experimentation using Industrial Data

The main goal of this research was to come up with a more efficient way to select the node to backtrack to during search, for (1) combinatorial optimization problems (2) for which we already know good variable/value selection strategies. We carried out experiments for a combined planning and scheduling problem from the forest-products industry (lumber planning and scheduling problem) for which good variable/value selection strategies are already known.

The problem is difficult as it involves *divergent processes* with *coproduction*: a single process simultaneously produces multiple products from one type of raw material. Moreover,

Table I
COMPUTATION TIME REDUCTION OF RLBS VS. LDS WITH DIFFERENT LEARNING RATE VALUES

α	Case 1	Case 2	Case 3	Case 4	Case 5	Average
0.1	$\downarrow 97.32\%$	$\downarrow 96.56\%$	$\downarrow 92.97\%$	$\downarrow 52.96\%$	$\downarrow 99.63\%$	$\downarrow 87.89\%$
0.2	$\downarrow 97.36\%$	$\downarrow 96.48\%$	$\downarrow 93.06\%$	$\downarrow 53.18\%$	$\downarrow 99.63\%$	$\downarrow$ **87.94%**
0.3	$\downarrow 97.38\%$	$\downarrow 95.44\%$	$\downarrow 90.26\%$	$\downarrow 53.64\%$	$\downarrow 99.63\%$	$\downarrow 87.27\%$
0.4	$\downarrow 97.40\%$	$\downarrow 95.27\%$	$\downarrow 93.47\%$	$\downarrow 53.90\%$	$\downarrow 99.63\%$	$\downarrow 87.93\%$
0.5	$\downarrow 97.43\%$	$\downarrow 86.87\%$	$\downarrow 93.67\%$	$\downarrow 54.09\%$	$\downarrow 99.63\%$	$\downarrow 86.34\%$
0.6	$\downarrow 97.47\%$	$\downarrow 80.54\%$	$\downarrow 93.68\%$	$\downarrow 54.28\%$	$\downarrow 99.63\%$	$\downarrow 85.12\%$
0.7	$\downarrow 97.47\%$	$\downarrow 78.64\%$	$\downarrow 93.77\%$	$\downarrow 54.39\%$	$\downarrow 99.63\%$	$\downarrow 84.78\%$
0.8	$\downarrow 97.48\%$	$\downarrow 77.24\%$	$\downarrow 93.82\%$	$\downarrow 54.70\%$	$\downarrow 99.63\%$	$\downarrow 84.58\%$
0.9	$\downarrow 97.48\%$	$\downarrow 76.40\%$	$\downarrow 93.88\%$	$\downarrow 54.87\%$	$\downarrow 99.63\%$	$\downarrow 84.45\%$

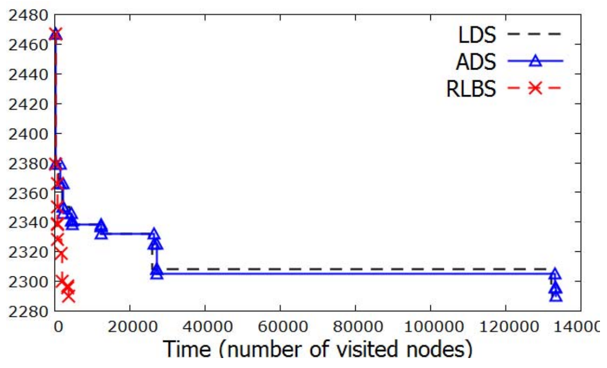

Figure 1. Objective function value according to computation time of LDS and RLBS for case #1 ($\alpha = 0.2$)

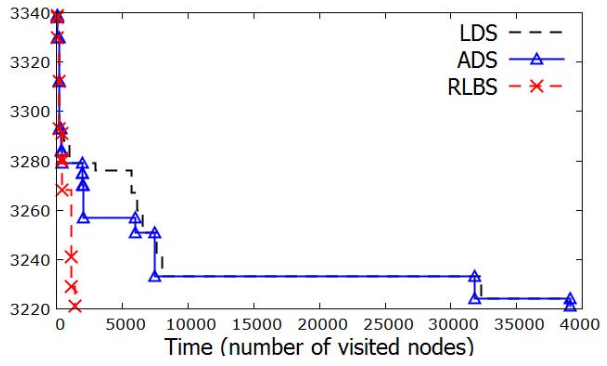

Figure 2. Objective function value according to computation time of LDS and RLBS for case #2 ($\alpha = 0.2$)

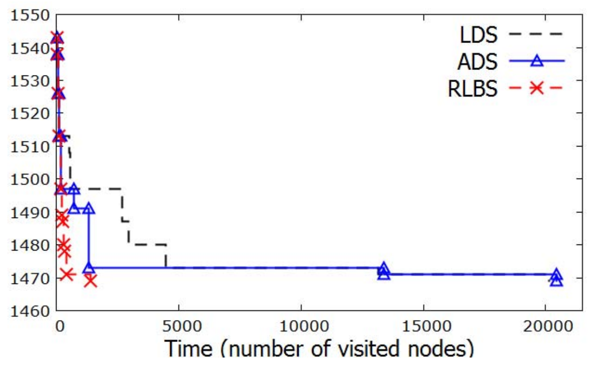

Figure 3. Objective function value according to computation time of LDS and RLBS for case #3 ($\alpha = 0.2$)

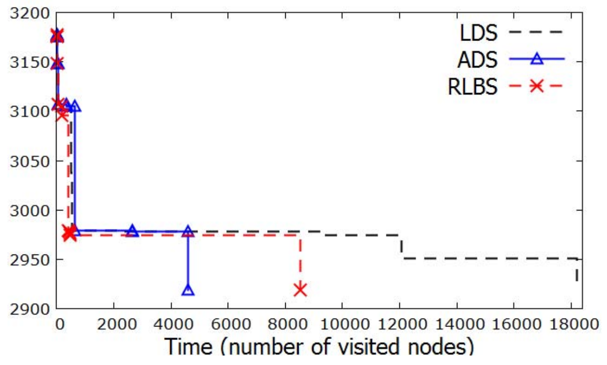

Figure 4. Objective function value according to computation time of LDS and RLBS for case #4 ($\alpha = 0.2$)

alternative processes can produce the same product. Finally, it involves complex setup rules. The objective is to minimize orders lateness.

The problem is fully described in [6] which provides a variable/value selection heuristic specific for it. In [7], this heuristic was used to guide the search in a constraint programming model. Provided with this branching strategy, LDS outperformed DFS as well as a mathematical programming approach. In [15], parallelization was used to improve performance (the visiting order of the nodes is the same as the centralized version, so it implements the same strategy).

We used the same variable/value selection heuristic as in previous work. We also used the same industrial data provided by a Canadian forest-products company. However, in order to be able to compare the algorithms according to the time needed to get optimal solutions, we reduced the size of the problems (5 periods instead of 44 periods).

We evaluated RLBS on 5 industrial instances of the lumber planing and scheduling problem using different learning rate values (see table I), and it turned out that, on average, RLBS performs best with a learning rate value of $\alpha = 0.2$ (with an average computation time reduction of 87.94%). Therefore, we use this learning rate value for all the experiments to compare RLBS to an LDS-based

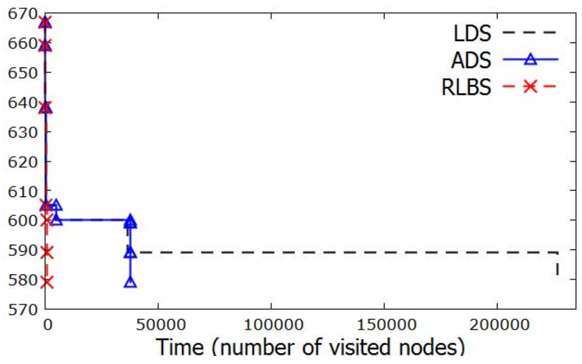

Figure 5. Objective function value according to computation time of LDS and RLBS for case #5 ($\alpha = 0.2$)

Table II
COMPUTATION TIME NEEDED TO GET THE BEST SOLUTION (RLBS AND ADS VS. LDS)

	Case 1	**Case 2**	**Case 3**	**Case 4**	**Case 5**	**Average**
LDS	132282	38861	20133	18197	226666	87227.8
ADS	133352	39079	20436	4612	38079	47111.6
RLBS	3494	1367	1398	8519	839	3123.4
Reduction ADS	$\uparrow 0.80\%$	$\uparrow 0.56\%$	$\uparrow 1.50\%$	$\downarrow 74.65\%$	$\downarrow 83.20\%$	$\downarrow 31.00\%$
Reduction RLBS	$\downarrow 97.36\%$	$\downarrow 96.48\%$	$\downarrow 93.06\%$	$\downarrow 53.18\%$	$\downarrow 99.63\%$	$\downarrow 87.94\%$

Table III
AVERAGE COMPUTATION TIME TO GET A SOLUTION OF A GIVEN QUALITY (RLBS AND ADS VS. LDS)

	Case 1	**Case 2**	**Case 3**	**Case 4**	**Case 5**	**Average**
LDS	3154147	724758	154336	964493	2716473	1542841.4
ADS	2897119	625379	98702	352457	848643	964460
RLBS	119949	68569	13776	518047	44561	152980.4
Reduction ADS	$\downarrow 8.15\%$	$\downarrow 13.71\%$	$\downarrow 36.05\%$	$\downarrow 63.46\%$	$\downarrow 68.76\%$	$\downarrow 38.03\%$
Reduction RLBS	$\downarrow 96.20\%$	$\downarrow 90.54\%$	$\downarrow 91.07\%$	$\downarrow 46.29\%$	$\downarrow 98.36\%$	$\downarrow 84.49\%$

policy (selecting the node showing the least discrepancies), and to ADS.

Figures 1 to 5 present the results for five different cases. Table II shows the reduction of computation time (measured as the number of visited nodes) needed to get an optimal solution. RLBS reduced computation time for each case (on average by 87.94%) while ADS reduced it for only 2 cases (with an average of 31.00%).

As in industrial context we usually do not have time to wait for the optimal solution, we also wanted to consider the time needed to get solutions of intermediate qualities. Table III shows, for each case, the average time needed to get a solution of any given quality. The last column shows that on average, for all problems and all needed solution qualities, the expected improvement of computation time provided by RLBS is 84.49%. ADS reduces the average computation time to get a solution of a given quality by 38.03% on average.

According to the tables II and III, we can conclude that RLBS is on average about 5 times faster than LDS, and about 3 times faster than ADS.

Finally, we generated small toy problems (Fig. 6) in order to compare RLBS with additional algorithms for which we were not able to solve the original problem in reasonable time (over 150 hours using Choco v2.1.5). We compared it to other approaches that do not change the branching heuristics (DFS, LDS). We also include results for Impact-Based Search (IBS) (which prevents us from using our specific variable/value selection heuristic) in order to illustrate that trying to learn how to branch instead of using good branching heuristics might not be the best option (at least not for the studied problem and IBS). IBS showed the worst result, presumably because it cannot make use of the specific branching strategy known to be efficient for this problem. DFS was also outperformed by LDS, as it is reported in the literature for this problem.

V. CONCLUSION

We proposed a simple learning mechanism based on reinforcement learning which allows a solver to dynamically learn how to backtrack without changing the branching strategy used. It was evaluated for a difficult industrial planning and scheduling problem which is only efficiently

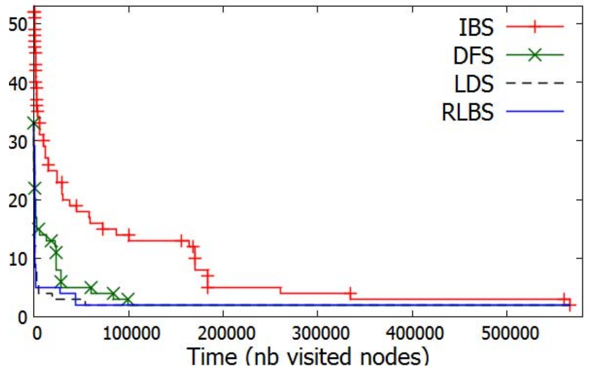

Figure 6. Objective function value according to computation time of RLBS, LDS, IBS and DFS for a toy problem ($\alpha = 0.2$)

solved when using specific branching heuristics. The proposed adaptive strategy greatly improved the performance in comparison with standard backtracking policies. This is made possible as the mechanism allows identifying which nodes are the most profitable to backtrack to and, thus, focusing on them first.

Using real industrial data showed the value of this approach. However, there are still open questions regarding how the algorithm should perform with problems for which we do not know good branching heuristics. In this situation, is it worth trying to identify which node we should backtrack to?

The combination of our adaptive *backtracking* strategy and adaptive *branching* strategies (like BASCOP [11] which uses reinforcement learning for branching) would be another interesting research opportunity.

References

[1] Ethem Alpaydin. *Introduction to machine learning*. MIT press, 2004.

[2] Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 1998.

[3] J. Christopher Beck and Laurent Perron. Discrepancy-bounded depth first search. In *Proceedings of the Second International Workshop on Integration of AI and OR Technologies for Combinatorial Optimization Problems (CPAIOR), Germany, Paderborn*, pages 7–17, 2000.

[4] Frédéric Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais. Boosting systematic search by weighting constraints. In *ECAI*, volume 16, page 146, 2004.

[5] Justin A Boyan and Andrew W Moore. Using prediction to improve combinatorial optimization search. In *Sixth International Workshop on Artificial Intelligence and Statistics*, 1997.

[6] Jonathan Gaudreault, Pascal Forget, Jean-Marc Frayret, Alain Rousseau, Sebastien Lemieux, and Sophie D'Amours. Distributed operations planning in the softwood lumber supply chain: models and coordination. *International Journal of Industrial Engineering: Theory Applications and Practice*, 17:168–189, 2010.

[7] Jonathan Gaudreault, Gilles Pesant, Jean-Marc Frayret, and Sophie D'Amours. Supply chain coordination using an adaptive distributed search strategy. *Journal of Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 42(6):1424–1438, 2012.

[8] Diarmuid Grimes and Richard J Wallace. Learning from failure in constraint satisfaction search. In *Learning for Search: Papers from the 2006 AAAI Workshop*, pages 24–31, 2006.

[9] William D Harvey and Matthew L Ginsberg. Limited discrepancy search. In *Proceedings of International Joint Conference on Artificial Intelligence (1)*, pages 607–615, 1995.

[10] Wafa Karoui, Marie-José Huguet, Pierre Lopez, and Wady Naanaa. YIELDS: A yet improved limited discrepancy search for CSPs. In *Proceedings of Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 99–111. Springer, 2007.

[11] Manuel Loth, Michele Sebag, Youssef Hamadi, and Marc Schoenauer. Bandit-based search for constraint programming. In *Principles and Practice of Constraint Programming*, pages 464–480. Springer, 2013.

[12] Manuel Loth, Michele Sebag, Youssef Hamadi, Marc Schoenauer, and Christian Schulte. Hybridizing constraint programming and monte-carlo tree search: Application to the job shop problem. In *Learning and Intelligent Optimization*, pages 315–320. Springer, 2013.

[13] Donald W Marquardt. An algorithm for least-squares estimation of nonlinear parameters. *Journal of the Society for Industrial & Applied Mathematics*, 11(2):431–441, 1963.

[14] Victor V Miagkikh and William F Punch III. Global search in combinatorial optimization using reinforcement learning algorithms. In *Proceedings of Evolutionary Computation, 1999.*

CEC 99. Proceedings of the 1999 Congress on, volume 1. IEEE, 1999.

[15] Thierry Moisan, Jonathan Gaudreault, and Claude-Guy Quimper. Parallel discrepancy-based search. In *Principles and Practice of Constraint Programming*, pages 30–46. Springer, 2013.

[16] Robert Moll, Andrew G Barto, Theodore J Perkins, and Richard S Sutton. Learning instance-independent value functions to enhance local search. In *Advances in Neural Information Processing Systems*. Citeseer, 1998.

[17] Philippe Refalo. Impact-based search strategies for constraint programming. In *Proceedings of Principles and Practice of Constraint Programming–CP 2004*, pages 557–571. Springer, 2004.

[18] Wheeler Ruml. *Adaptive tree search*. PhD thesis, Citeseer, 2002.

[19] Wheeler Ruml. Heuristic search in bounded-depth trees: Best-leaf-first search. In *Working Notes of the AAAI-02 Workshop on Probabilistic Approaches in Search*, 2002.

[20] Pascal Van Hentenryck. *The OPL optimization programming language*. Mit Press, 1999.

[21] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[22] Yuehua Xu, David Stern, and Horst Samulowitz. Learning adaptation to solve constraint satisfaction problems. In *Proceedings of Learning and Intelligent Optimization (LION)*, 2009.

[23] Wei Zhang and Thomas G Dietterich. A reinforcement learning approach to job-shop scheduling. In *Proceedings of International Joint Conferences on Artificial Intelligence*, volume 95, pages 1114–1120. Citeseer, 1995.