Learn, Compare, Search: One Sawmill's Search for the Best Cutting Patterns Across And/or Trees

Marc-André Ménard[1,2,3][0000−0002−8776−5166],
Michael Morin[1,2,4][0000−0002−1008−4303],
Mohammed Khachan[5][0000−0001−7402−3991],
Jonathan Gaudreault[1,2,3][0000−0001−5493−8836], and
Claude-Guy Quimper[1,2,3][0000−0002−5899−0217]

[1] FORAC Research Consortium
[2] CRISI Research Consortium for Industry 4.0 Systems Engineering
[3] Department of Computer Science and Software Engineering
[4] Department of Operations and Decision Systems
Université Laval, Québec, QC, Canada
[5] FPInnovations, Québec, QC, Canada
marc-andre.menard.2@ulaval.ca

Abstract. A sawmilling process scans a wood log and must establish a series of cutting and rotating operations to perform in order to obtain the set of lumbers having the most value. The search space can be expressed as an and/or tree. Providing an optimal solution, however, may take too much time. The complete search for all possibilities can take several minutes per log and there is no guarantee that a high-value cut for a log will be encountered early in the process. Furthermore, sawmills usually have several hundred logs to process and the available computing time is limited. We propose to learn the best branching decisions from previous wood logs and define a metric to compare two wood logs in order to branch first on the options that worked well for similar logs. This approach (Learn, Compare, Search, or LCS) can be injected into the search process, whether we use a basic Depth-First Search (DFS) or the state-of-the-art Monte Carlo Tree Search (MCTS). Experiments were carried on by modifying an industrial wood cutting simulator. When computation time is limited to five seconds, LCS reduced the lost value by 47.42% when using DFS and by 17.86% when using MCTS.

Keywords: Monte Carlo search · And/or trees · Tree search algorithms · Sawmilling · Learning.

1 Introduction

In North America, softwood lumber is a standardized commodity. Different dimensions and grades are possible and the hardware in the mill, thanks to embed-

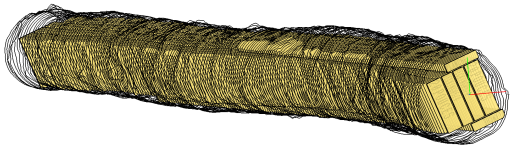


Fig. 1: A basket of products obtained from the log at the exit of the sawmill represented in a virtual log.

ded software, optimize cutting decisions for each log in order to maximize profit. Each lumber type has a specific value on the market, and the equipment aims to maximize the total basket value given a log. Processing a log leads to a basket of these standardized lumber products (and byproducts such as sawdust). Figure 1 shows an example of cut for a given log leading to a specific basket of lumber products. Two different combinations of cutting decisions (e.g., differences in trimming, edging and/or sawing) can lead to two baskets of different values.

The optimizer of sawmills' equipment can be used offline to measure the impacts related to changes in the configuration. This can also be done using sawing simulators such as Optitek [11], RAYSAW [30], and SAWSIM [14]. Providing an optimal cutting solution may take too much time, whether it is for a real-time cutting decision purpose, or to get a suitable forecast for decision-making.

The cutting decision optimization problem has been addressed in the literature in several ways (optimization, simulation, prediction by neural networks or other ML approaches) [16]. In this paper, we are concerned with a specific case where the problem is solved by an algorithm enumerating all possible cuts.

Obtaining the best cuts for a given log implies that all log cutting possibilities must be tested according to the possible cutting choices of each machine. This complete search for all possibilities can take several minutes per log. In practice we need to use a time limit for the search and therefore the sawmill lose value as there is no guarantee that a good cut for a log will be encountered early in the process. In this paper, we address this challenge.

To get the best possible solution according to the computation time limit, we need to test first the most promising cut choices. Although the past is no guarantee of the future in many cases, when sawing similar logs at the same sawmill, it might as well be. Based on this observation, we suggest an informed search algorithm which learns from the previous similar logs to guide the search process for the actual log. The assumption being that for two similar logs the cutting decisions leading to the best value will be the same or at least similar enough, it makes sense to guide the search with our knowledge of the best decisions for these logs.

The rest of the paper is divided as follows. First, we describe the problem more thoroughly. Second, we present the preliminary concepts. Third, we present our method for learning from the optimal cutting decision of already cut logs. Fourth, we present our results and finally we conclude.

2 Problem Description

In many sawing optimization systems, such as Optitek, logs are represented by a surface scan: a point cloud of the log's surface structured as a sequence of circular sections (see Figure 1 where the points of each section have been interpolated). A sawmill is defined as a set of machines. Each machine can make different transformations on the log. A machine that cuts the log into several parts creates different cutting sub-problems. Each part of the resulting log can go through the same or a different sequence of machines. At the end of the machine sequence, the sum of the values of each part of the initial log gives the value of the log. The value of a product is assessed through a value matrix according to the size and grade of the product. It is therefore not possible to know the exact value of the log before different cutting alternatives are explored.

The type of decisions varies from one machine to another. First, there are different sets of cutting patterns that can be applied by some machines. Second, for some machines, it is possible to rotate or to do a translation of the log before it passes through the machine. Each rotation or translation can greatly affect the basket of products obtained from the log.

The set of all decision sequence for a given log can be represented as a search tree. The root of the tree corresponds to the log at the entrance of the sawmill where no transformation has been done yet. Each level of the tree corresponds to a cutting decision taken on a part of the log by a machine. The decision can be a rotation, a translation or the choice of a cutting pattern. Each node of a level corresponds to a value for the decision at this level. A leaf corresponds to a solution, i.e., a basket of products for which a complete sequence of decisions has been made. To make sure we find the optimal solution, we have to go through the whole search tree (an exhaustive search is needed to prove the optimality).

The search tree can be represented by an and/or tree. When a decision cutting a log in more than one piece is made, we end up with several sub-problems. Each sub-problem (log part) then goes through different machines and cutting decisions. The sub-problems correspond to the "and" of an and/or tree. The subtree corresponding to each sub-problem must be searched to obtain a solution. Cutting decisions correspond to the "or" of an and/or tree. For an "or" node, we have a decision to make, whereas for an "and" node we have no choice but to solve the sub-problem, i.e., explore the subtree.

The width of the tree depends on the values to be tested for each cutting decision. Its maximum depth is bounded by the machine sequence and the number of unique types of cutting decisions to be tested. In our experiments and in practice, there are generally more cutting decisions to test than there are machines. The search tree is therefore wide and shallow.

Figure 2 shows a small search tree example. The log at the top of the tree enters the sawmill. Each circular node represents a decision that is possible to choose. For example, from the incoming log (root), we have the choice between two decisions. Each decision can then create sub-problems (rectangular nodes) by cutting the log until it reaches a leaf of the search tree. The sum of the leaves

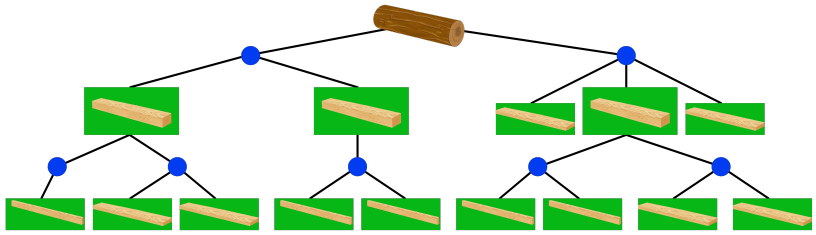


Fig. 2: Example of a search tree. The log at the top of the tree enters the sawmill. Decisions are made (circular nodes, "or") that can create new sub-problems (rectangular nodes, "and") by cutting the log.

corresponding to the best decisions for each sub-problem forms a product basket and the best product basket is returned.

Figure 3 shows an example of a (virtual) sawmill in the Optitek software. The rectangles are machines where cutting decisions can be optimized. The flow is generally from left to right (from the *Feed* to the *Sort* and *Chip* nodes), although there might be loops such as it is the case for the *Slasher* in our example. Each machine—except *Feed*, *Sort* and *Chip*—has inputs (left-hand side) and outputs (right-hand side). Products are routed from one machine to the other depending on their characteristics.

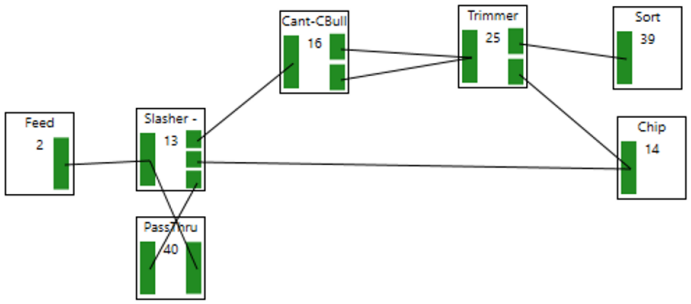


Fig. 3: A sawmill configuration. The rectangles are machines where cutting decisions can be optimized. The branches are the possible routes between a machine's output and another machine's input.

3 Preliminary Concepts

In this section, we present tree search algorithms from the literature. We will use them as a baseline of comparison for our approach we call Learn, Compare,

Search, or LCS. Each algorithm without LCS will be compared to a version where we injected LCS into the search process. We also review and/or trees and related works on learning to guide a search tree traversal and adaptive search.

3.1 Depth-First Search (DFS)

DFS starts at the root node and always branches on the leftmost node of the possible choices until it reaches a leaf in the tree. Then it backtracks until there is an unexplored child node and it resumes its exploration on the leftmost unexplored child. DFS visits the leaves of the tree from left to right.

DFS is the fastest way to test all decisions, but it has the disadvantage of not finding good solutions quickly if they are on the right-hand side of the search tree. For example, if we have a rotational optimization on the first machine and we want to test all degrees between 0 and 180 degrees, DFS will test the degrees in ascending order $(0, 1, 2, 3, \ldots)$. If the quality of a solution increases with rotation, DFS will find the optimal solution only at the end of the search and most of the time will be spent finding bad solutions.

3.2 Limited Discrepancy Search (LDS)

Harvey and Ginsberg [15] introduced the concept of discrepancy. A discrepancy is when the search heuristic is not followed and another node is visited instead. For example, in the DFS search algorithm, the heuristic is to branch left whenever possible. By branching right when we should have branched left in a binary tree, we deviate from the left-first heuristic of DFS, this is counted as one *deviation* (or discrepancy). If a node contains several children, there are two ways to count the deviations. First, the number of deviations can be the number of nodes skipped from left to right. For example, taking the first node counts as 0 deviation, taking the second counts as 1 deviation, taking the third counts as 2, and so on. The other method, and the one we use in this paper, is to count 1 deviation only whenever the first node is not taken.

The LDS [15] search algorithm explores the search tree iteratively. Each iteration visits the nodes that have less than k deviations. Starting from $k = 0$, the value of k is incremented at each iteration. In this article, we take the improved version of LDS (ILDS) presented by Korf [21] which avoids visiting a leaf of the search tree more than once.

3.3 Depth-Bounded Discrepancy Search (DDS)

DDS [31] is a search algorithm also based on deviations. At each iteration, DDS follows the DFS search algorithm until it reaches a node on level k where k is the iteration number. It then visits all the nodes on the next level, except the first node on the left, and continues the exploration by visiting only the nodes on the left for the remaining levels of the search tree. The idea of DDS is that the search heuristic is more likely to make a wrong choice at the top of the search tree than at the bottom.

3.4 Monte Carlo Tree Search (MCTS)

MCTS is a tree search algorithm using a compromise between exploration and exploitation [6]. MCTS works by iteration. Each iteration contains four phases: selection, expansion, simulation and backpropagation. There are several ways to implement MCTS. Our implementation is inspired by Antuori et al. [2]

Selection The selection phase starts at the root of the tree and ends when we reach a node that has not yet been visited in a previous iteration. When we are at a node, the next node to visit is chosen according to the formula (1) where $A(\sigma)$ represents a set of actions that can be done from the current node σ. Each action is represented by a branch in the search tree. Given node σ, $\sigma|a$ represents the child node reached by taking the action a. $\tilde{V}(\sigma|a)$ is the expected value of the node if the action a is chosen and corresponds to the exploitation term. $U(\sigma|a)$ corresponds to the exploration term. Finally, c is a parameter to balance the exploitation and the exploration (i.e., $\tilde{V}(\sigma|a)$ and $U(\sigma|a)$).

$$\arg\max_{a \in A(\sigma)} \tilde{V}(\sigma|a) + c \cdot U(\sigma|a) \tag{1}$$

There are different ways to compute an expected value $\tilde{V}(\sigma|a)$ for a node $\sigma|a$. For example, it is possible to take the average of the solutions found so far for the node. Instead, we use the best value found so far for this node as expected value. Keeping the best value is better in our case, because many of the solutions have a null value, i.e., the cutting decisions lead to a null value for one of the output products. The values of the different $\tilde{V}(\sigma|a)$ must be normalized to be compared with the exploration term $U(\sigma|a)$. We normalize this value between $[-1, 1]$ using the formula (2) where $N(\sigma)$ is the number of visits to the node σ, $V^+ = \max\{V(\sigma|a)|a \in A(\sigma), N(\sigma|a) > 0\}$ and $V^- = \min\{V(\sigma|a)|a \in A(\sigma), N(\sigma|a) > 0\}$ [2] which corresponds to the maximum and minimum value for the children nodes of the parent node.

$$\tilde{V}(\sigma|a) = \begin{cases} 2\frac{V^+ - V(\sigma|a)}{V^+ - V^-} - 1 & \text{if } N(\sigma|a) > 0 \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

The exploration term ($U(\sigma)$) is computed with the formula (3) where $Pr(\sigma)$ corresponds to the priority probability biases, $p(\sigma)$ corresponds to the parent node of the current node σ, and $N(\sigma)$ corresponds to the number of visits to the current node.

$$U(\sigma) = Pr(\sigma)\frac{\sqrt{N(p(\sigma))}}{N(\sigma) + 1} \tag{3}$$

Expansion The expansion phase creates a child node $\sigma|a$ for each action $a \in A(\sigma)$. For each child node $\sigma|a$, we initialize the number of visits $N(\sigma|a)$ to 0 and the expected objective value $V(\sigma|a)$ to 0. We also initialize the prior probability biases $Pr(\sigma|a)$ to a value if available and if not, we initialize $Pr(\sigma|a)$ according to a uniform distribution $\frac{1}{|A(\sigma)|}$.

Simulation The simulation phase, also called rollout, aims at finding a possible expected value for the node. The simulation phase must then visit at least one leaf of the search tree from the current node. The simulation can be done by making a random choice of nodes to visit at each level of the tree. It is also possible to make a weighted random choice with the different probabilities of each node $Pr(\sigma)$ if these probabilities are available.

Backpropagation The backpropagation phase will update the nodes visited during the selection phase. The number of visits $N(\sigma)$ is incremented by 1 and the expected value $V(\sigma)$ is updated if a better solution is found during the simulation phase.

3.5 Searching AND/OR Trees

The representation of the problem as an and/or tree allows having a search tree with less depth. The search algorithms can be adapted to the and/or tree.

For DFS, there is no change. For the LDS search algorithm, the algorithm must be modified. Larrosa et al. [22] presents the limited discrepancy and/or search (LDSAO) algorithm. This algorithm uses the LDS search algorithm, but for and/or trees. The difference between LDS and LDSAO is in the handling of the "and" nodes. For the "and" nodes, we have to find and solve these subproblems to get a solution. These nodes do not cause any discrepancy. For the "or" nodes, we do not cause any discrepancy if we follow the search heuristic. For the same number of discrepancies, Larrosa et al. [22] have shown that each iteration of LDSAO includes the search space of LDS on the original tree and more.

The same logic for going from LDS to LDSAO can be applied for DDS where we do not cause discrepancy when going through an "and" node, but only when we do not follow the search heuristic to go through an "or" node.

For the MCTS algorithm, in selection mode, we visit all the "and" nodes (subproblem) for a given level, but we visit only one "or" node (decisions). It is also the same principle in simulation mode, we visit all the "and" nodes (subproblem) for a given level, but we select only one of the decisions ("or" node).

3.6 Learning for Search Tree Traversal and Adaptative Search

Learning to guide a search tree traversal for search efficiency purposes is not a new concept. There are several adaptive search algorithms for variable choice heuristics and value choice heuristics such as *dom/wdeg* [5], solution counting based search [33], and activity-based search [25]. In this family of algorithms, we also find Impact-Based Search [27], Adaptive Discrepancy Search (ADS) [12], and the RBLS algorithm [3]. There is also Solution-guided multipoint constructive search (SGMPCS) [4] that guides the search from multiple solutions found during the search of the current instance. Our approach has some similarities to SGMPCS, but it starts with solutions found offline. MCTS [6], we described in Section 3.4, is also an adaptive algorithm. At the difference of the LCS approach,

these algorithms, in their original form, tend to adapt at runtime. They do not use prior knowledge of the problem.

The alternative is to guide the search with known solutions. Loth et al. [24] presented Bandit-Based Search for CP (BASCOP), an adaptation of MCTS using Reinforcement Learning (RL) to know where to branch next. They use a reward function based on the failure depth. This approach is interesting to keep the learning from one problem instance to another. In our problem, however, the search tree is determined by the order of the machines and the cutting decisions. There are no failures.

Other approaches are specific to Constraint Programming (CP). Chu and Stuckey [8], for instance, have presented a value ordering approach to tree search. When branching, a score is assigned to each value in the domain of the variable using domain-based features and machine learning.

There is also a whole literature on learning in the context of a branch and bound for Mixed-Integer Linear Programming which leads to search trees where branching splits the problem into subproblems instead of assigning a value to a variable (as it is usually the case in CP branch and bounds). In their literature review on learning to improve variable and value selection, Lodi and Zarpellon [23] define two categories of approaches: (1) incorporating learning in more traditional heuristics [13,18,10], and (2) using machine learning [1,19,20].

Finally, LCS is closer to informed search methods with a priori information than to adaptive search methods that learn at runtime. The idea of informed search is to add domain-specific knowledge to help the search algorithm [9,29]. Recent examples of informed search includes the work of Silver et al. [28], Ontanón [26], and Yang et al. [32]. Silver et al. [28] introduced AlphaGo, a system using MCTS along with a neural network trained by supervised learning on human expert movements, and improved through self-play RL. Ontanón [26] presents Informed MCTS for real-time strategy (RTS) games. The approach consists of using several Bayesian models to estimate the probability distribution of each action played by an expert or search heuristics. It then initializes the prior probabilities of the possible actions in the MCTS search. Yang et al. [32] present the Guiding MCTS which uses one or more deterministic scripts based on the human knowledge of the game to know which node to visit first for a RTS game. Our approach is similar to these approaches since we rely on past decisions to initialize the prior probabilities of actions.

4 Learning from Past Decisions in LCS

To find the best solution faster, we propose to first learn from past instances/logs. This allows us to know which decisions are more promising and should be visited first. In the context of sawing, a decision is considered more promising than another if it is more often found in the optimal cutting decisions of previously cut logs. The assumption is that for similar logs, the same cutting decisions will be made to find the best log value.

It is possible to get information on the decisions made for previous logs, because the structure of the search tree is similar from one log to another. Indeed, the order of the machines and the cutting decisions remain the same. However, the best choices may differ according to the characteristics of the logs. Even in the case where two logs (therefore their search trees) differ substantially, a knowledge transfer might still be possible between the two.

4.1 Adaptation of Search Algorithms to Learning

Each search algorithm presented in Section 3 can make use of past optimal cutting decisions of already cut logs. For DFS, LDS and DDS, we ordered the "or" child nodes representing a decision of an "and" node in descending order of the number of times that decision is made to obtain a solution of a previously cut log. We name this value for a node $Q(\sigma)$ where σ represent the node. The first node is therefore the decision that is most often in a solution of the previously cut logs compared to the other possible decisions, i.e., $\max_{a \in A}(Q(\sigma|a))$. For the algorithms using discrepancy, no discrepancy is caused when we take the node that is most often in a solution of the previously cut logs.

For the MCTS search algorithm, the prior probability biases $Pr(\sigma|a)$ are initialized according to Equation (4) representing the number of times a decision is chosen to obtain a solution from a previously cut log versus the number of times the parent decision is.

$$Pr(\sigma|a) = \frac{Q(\sigma|a)}{Q(\sigma)} \tag{4}$$

It may not be ideal to rely on the decisions made for all log sawed. Two logs that are very different may have different optimal cutting decisions. Therefore, it is best to rely only on the X decisions that led to the best cutting decisions for the logs most similar to the simulated log where X is a hyper-parameter. The following section presents the method for finding similar logs.

4.2 Finding Similar Logs

We assume each log is represented by a 3D point cloud. The point cloud is separated into sections and each section represents a part of the log. This is a standard representation for the logs in the industry.

Finding logs similar to the current log could be done by comparing the point clouds. For instance, by first using the *iterative closest point* (ICP) algorithm to align two point clouds, we could extract the alignment error and use it as a dissimilarity measure [7]. This, however, would be computationally intensive. The ICP algorithm is polynomial in the number of points [17], but each new log would need to be compared to every known log and log point clouds can have more than 30 thousand points. Our logs, for instance, have an average of 19326 points and the maximum number of points is 32893. To keep runtimes low, we compare the logs using a set of features that are easily computed. In fact, the

Table 1: Features of a log, calculated from a surface scan

Features	Description
Length	Length of the log
Small-end diameter	Diameter of the log at the top end
Large-end diameter	Diameter of the log at the stump end
Diameter at 25%	Log diameter at 25% of length from the stump end
Diameter at 50%	Log diameter at 50% of length from the stump end
Diameter at 75%	Log diameter at 75% of length from the stump end
Max sweep	Maximum value of the curvature of the log
Location max sweep	Percentage of length at which curvature is greatest
Thickness	Thickness of the log calculated from the point cloud
Width	Log width calculated from the point cloud
Volume	Volume of the log
Volume homogeneity	Average volume difference from section to section
Taper	Difference between the diameters of the two ends

features we use can be computed offline before the search. Table 1 shows the features used to represent a log.

To calculate the similarity between two logs, we first apply a MinMax normalization (Equation (5)) before comparing features.

$$b^c = \frac{b^c - \min(b^c)}{\max(b^c) - \min(b^c)} \quad \forall c \in C \tag{5}$$

This ensures two features that do not have the same order of magnitude have the same weight when computing the similarity, for example the length and diameter of a log. Then, we calculate the distance of the logs by taking the sum of their squared differences using Equation (6) where C is the set of features, b_1^c is the value of the feature c of the first log and b_2^c is the value of the feature c of the second log.

$$S(b_1, b_2) = \sum_{c \in C} (b_1^c - b_2^c)^2 \tag{6}$$

5 Experiments

To demonstrate the potential of LCS on our log sawing problem, we implemented it—along with the DFS, LDS, DDS, and MCTS tree search algorithms—in a state-of-the-art commercial sawing simulator (Optitek). We have eight different sawmill configurations which allows testing the approach on eight different industrial contexts. For each sawmill we proceed as follows. We virtually cut one hundred logs, each time exploring the search tree completely thus finding the max/optimal value.

Then, we randomly separate the hundred logs into two datasets. Fifty logs will be used as a training dataset to find the best hyper-parameter values for

each algorithm. The other fifty logs constitute the dataset used to compare the algorithms.

For these two datasets, another separation must be made. Thirty logs among the fifty are used to know the paths that most often led to the best solution for the methods that learn on the already simulated logs. The other twenty logs among the fifty will be used to compare the results obtained by each of the search algorithms. As the random separation impacts the method to learn on the already simulated logs, five replications of this separation are made to get an average of the results.

The hyper-parameters for MCTS are: the $c \in \{1, 2\}$ coefficient, used to balance exploitation and exploration; the number of simulations performed during the simulation phase, selected in $\{1, 2, 3, 4\}$; and the number of nodes visited for each level during the simulation, selected in $\{1, 2, 3\}$. If at a level, the number of child nodes is less than the number of child nodes to visit, we visit all of them.

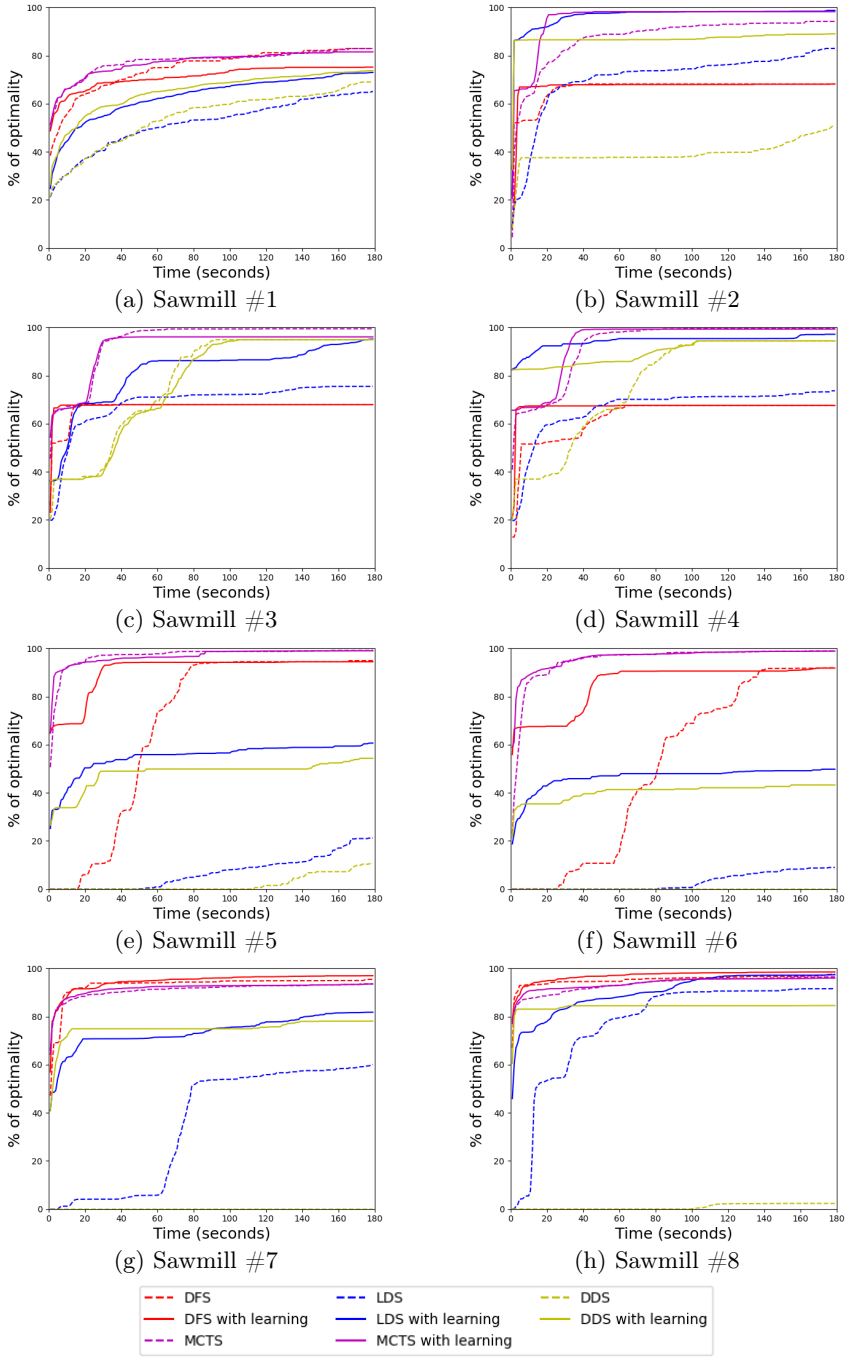
When processing a new log, we tested with 10, 20, and 30 as the number of similar logs we will use to guide the search.

6 Results

Figure 4 presents the results obtained on the 8 sawmill configurations according to the search algorithm. Figure 5 presents the averaged for the 8 sawmill configurations. For each sawmill configuration and search algorithm, we plotted the percentage of optimality attained on average on the 20 test logs (y-axis) against the solving time in seconds (x-axis). Each curve represents an average over five replications. At a given point in time, a search algorithm at 100% optimality has found all the optimal solutions for the 20 logs.

MCTS is better than DFS, LDS and DDS except for the sawmill configurations #7 and #8 where DFS is slightly better. By comparing each search algorithm with or without learning (with or without LCS), we see that the results are always better except for DFS for the sawmill configuration #1 and LDS for the sawmill configuration #3. However, there is little difference between MCTS and MCTS with learning. The fact that MCTS learns and adapt during the search, balancing exploitation and exploration, should reduce the gains obtained with learning on decisions made on previous logs.

For each search algorithm, Table 2 shows the percentage of improvement (reduction of the lost value, in percentage) provided by LCS after n seconds (with $n \in \{1, 2, \ldots, 5\}$) according to the search algorithm. After 1 second, the improvement is 23.58% for MCTS, the best search algorithm, whereas it is 23.2% for DFS, the simplest search algorithm. After 5 seconds, the improvement is 17.86% for MCTS and 47.42% for DFS. In our experiments, completing the search took 1 hour and 42 minutes per log on average. For industrial decisions, the "ideal" time limit depends on the hardware configuration of the sawmill and the available time to wait for a solution. With a time limit of one second, a thousand logs will take about 16.67 minutes whereas a time limit of 5 seconds would lead to a waiting time of 83.3 minutes.



(a) Sawmill #1 (b) Sawmill #2

(c) Sawmill #3 (d) Sawmill #4

(e) Sawmill #5 (f) Sawmill #6

(g) Sawmill #7 (h) Sawmill #8

Fig. 4: Percentage of optimality against solving time (in seconds) for each algorithm and sawmill configurations; average of 5 replications on 20 logs.

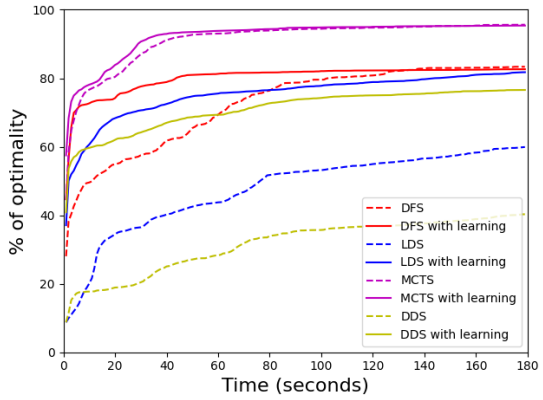


Fig. 5: Percentage of optimality against solving time (in seconds) for each tree search algorithm; averaged on all sawmill configurations.

Table 2: Reduction of the lost value when using LCS depending on the search algorithm according to the time in seconds.

	Elapsed time				
Search algorithms	1 sec.	2 sec.	3 sec.	4 sec.	5 sec.
DFS vs DFS with learning	23.2%	29.47%	42.45%	47.58%	47.42%
LDS vs LDS with learning	31.08%	44.55%	46.05%	46.62%	47.94%
DDS vs DDS with learning	35.22%	47.62%	47.79%	48.56%	48.63%
MCTS vs MCTS with learning	23.58%	23.73%	24.03%	22.44%	17.86%

7 Conclusion

We compared four search algorithms, namely DFS, LDS, DDS, and MCTS with and without learning (a total of eight variants). We showed how to improve each of them by learning from the best cutting decisions on previous logs, introducing a framework called LCS. In the context of our application, LCS, when injected in the search process, improved the cut quality obtained within the first 5 seconds of the search—a requirement for enabling better industrial decisions. Considering the thousands logs to process in sawmills and the need to forecast the impact of a decision, our approach has the potential to translate to tangible gains for the forest-product industry.

Although we applied LCS in the particular context of wood log cutting decision optimization, the framework is general and could be applied to other problems where the search space is represented as a tree. Learning from previous runs using a framework such as LCS, as long as the instances share sufficient similarities, appear to be a promising avenue for further research on combining learning and optimization.

References

1. Alvarez, A.M., Louveaux, Q., Wehenkel, L.: A machine learning-based approximation of strong branching. INFORMS Journal on Computing **29**(1), 185–195 (2017)
2. Antuori, V., Hébrard, E., Huguet, M.J., Essodaigui, S., Nguyen, A.: Combining monte carlo tree search and depth first search methods for a car manufacturing workshop scheduling problem. In: International Conference on Principles and Practice of Constraint Programming (CP) (2021)
3. Bachiri, I., Gaudreault, J., Quimper, C.G., Chaib-draa, B.: Rlbs: An adaptive backtracking strategy based on reinforcement learning for combinatorial optimization. In: 2015 IEEE 27th International Conference on Tools with Artificial Intelligence (ICTAI). pp. 936–942. IEEE (2015)
4. Beck, J.C.: Solution-guided multi-point constructive search for job shop scheduling. Journal of Artificial Intelligence Research **29**, 49–77 (2007)
5. Boussemart, F., Hemery, F., Lecoutre, C., Sais, L.: Boosting systematic search by weighting constraints. In: Proceedings of the 16th European Conference on Artificial Intelligence (ECAI). vol. 16 (2004)
6. Browne, C.B., Powley, E., Whitehouse, D., Lucas, S.M., Cowling, P.I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., Colton, S.: A survey of monte carlo tree search methods. IEEE Transactions on Computational Intelligence and AI in games **4**(1), 1–43 (2012)
7. Chabanet, S., Thomas, P., El Haouzi, H.B., Morin, M., Gaudreault, J.: A kNN approach based on ICP metrics for 3D scans matching: an application to the sawing process. In: 17th IFAC Symposium on Information Control Problems in Manufacturing (INCOM) (2021)
8. Chu, G., Stuckey, P.J.: Learning value heuristics for constraint programming. In: International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR). pp. 108–123. Springer (2015)
9. Drake, P., Uurtamo, S.: Move ordering vs heavy playouts: Where should heuristics be applied in monte carlo go,". In: Proceedings of the 3rd North American Game-On Conference. pp. 171–175. Citeseer (2007)
10. Fischetti, M., Monaci, M.: Backdoor branching. In: International Conference on Integer Programming and Combinatorial Optimization (IPCO). pp. 183–191. Springer (2011)
11. FPInnovations: Optitek 10. In: User's Manual (2014)
12. Gaudreault, J., Pesant, G., Frayret, J.M., D'Amours, S.: Supply chain coordination using an adaptive distributed search strategy. IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews) **42**(6), 1424–1438 (2012)
13. Glankwamdee, W., Linderoth, J.: Lookahead branching for mixed integer programming. In: Twelfth INFORMS Computing Society Meeting. pp. 130–150 (2006)
14. HALCO: Halco software systems ltd (2016), `http://www.halcosoftware.com`
15. Harvey, W.D., Ginsberg, M.L.: Limited discrepancy search. In: Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI). vol. 1, pp. 607–615 (1995)
16. Hosseini, S.M., Peer, A.: Wood products manufacturing optimization: A survey. IEEE Access **10**, 121653–121683 (2022)
17. Jost, T., Hügli, H.: Fast icp algorithms for shape registration. In: Pattern Recognition. pp. 91–99. Springer Berlin Heidelberg, Berlin, Heidelberg (2002)
18. Karzan, F.K., Nemhauser, G.L., Savelsbergh, M.W.: Information-based branching schemes for binary linear mixed integer problems. Mathematical Programming Computation **1**(4), 249–293 (2009)

19. Khalil, E., Le Bodic, P., Song, L., Nemhauser, G., Dilkina, B.: Learning to branch in mixed integer programming. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 30 (2016)
20. Khalil, E.B.: Machine learning for integer programming. In: Proceedings of the Twenty-fifth International Joint Conference on Artificial Intelligence (IJCAI). pp. 4004–4005 (2016)
21. Korf, R.E.: Improved limited discrepancy search. In: Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI). vol. 1, pp. 286–291 (1996)
22. Larrosa Bondia, F.J., Rollón Rico, E., Dechter, R.: Limited discrepancy and/or search and its application to optimization tasks in graphical models. In: Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence (IJCAI). pp. 617–623. AAAI Press (Association for the Advancement of Artificial Intelligence) (2016)
23. Lodi, A., Zarpellon, G.: On learning and branching: a survey. Top **25**(2), 207–236 (2017)
24. Loth, M., Sebag, M., Hamadi, Y., Schoenauer, M.: Bandit-based search for constraint programming. In: International Conference on Principles and Practice of Constraint Programming (CP). pp. 464–480. Springer (2013)
25. Michel, L., Van Hentenryck, P.: Activity-based search for black-box constraint programming solvers. In: Integration of AI and OR Techniques in Contraint Programming for Combinatorial Optimzation Problems (CPAIOR). pp. 228–243. Springer (2012)
26. Ontanón, S.: Informed monte carlo tree search for real-time strategy games. In: 2016 IEEE Conference on Computational Intelligence and Games (CIG). pp. 1–8. IEEE (2016)
27. Refalo, P.: Impact-based search strategies for constraint programming. In: International Conference on Principles and Practice of Constraint Programming (CP). pp. 557–571. Springer (2004)
28. Silver, D., Huang, A., Maddison, C.J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al.: Mastering the game of go with deep neural networks and tree search. nature **529**(7587), 484–489 (2016)
29. Świechowski, M., Godlewski, K., Sawicki, B., Mańdziuk, J.: Monte carlo tree search: A review of recent modifications and applications. Artificial Intelligence Review pp. 1–66 (2022)
30. Thomas, R.E.: Raysaw: A log sawing simulator for 3d laser-scanned hardwood logs. In: Proceedings, 18th Central Hardwood Forest Conference. vol. 117, pp. 325–334 (2012)
31. Walsh, T.: Depth-bounded discrepancy search. In: Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI). vol. 1, pp. 1388–1393 (1997)
32. Yang, Z., Ontanón, S.: Guiding monte carlo tree search by scripts in real-time strategy games. In: Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AAAI). vol. 15, pp. 100–106 (2019)
33. Zanarini, A., Pesant, G.: Solution counting algorithms for constraint-centered search heuristics. Constraints **14**(3), 392–413 (2009)