Exploratory Analysis of the Monte Carlo Tree Search For Solving The Linear Ordering Problem

Andoni I. Garmendia
University of the Basque Country
UPV-EHU
Donostia-San Sebastian, Spain
andoni.irazusta@ehu.eus

Josu Ceberio
University of the Basque Country
UPV-EHU
Donostia-San Sebastian, Spain
josu.ceberio@ehu.eus

Alexander Mendiburu
University of the Basque Country
UPV-EHU
Donostia-San Sebastian, Spain
alexander.mendiburu@ehu.eus

ABSTRACT

Monte-Carlo Tree Search has delivered great results in two-player game-playing and its current success has turned it into a popular choice of study in different use cases. Recently, many works have applied MCTS and, especially, its neural variant, as an end-to-end approach to solve Combinatorial Optimization Problems. However, its efficiency for solving regular Combinatorial Problems has still to be studied.

In this paper, we investigate the capability of the Monte-Carlo Tree Search algorithm to optimize permutation-based problems, making use of problem-specific knowledge. Particularly, we focus on the well-known Linear Ordering Problem (LOP), taking advantage of the easy computation of the expected and upper bound fitness of partial permutations. Moreover, we introduce a Multi-Objective Optimization approach to deal with the exploration-exploitation dilemma during the tree search. Conducted experiments show that MCTS obtains better results than classical constructive algorithms, though its performance is not obviously comparable to state-of-the-art results. Based on its ability for guiding structured searches, its scalability, convergence and search space coverage, MCTS could open new research trends in the optimization area.

CCS CONCEPTS

• **Mathematics of computing** → **Permutations and combinations**; • **Computing methodologies** → **Game tree search**; **Discrete space search**; **Artificial intelligence**.

KEYWORDS

Monte-Carlo Tree Search, Combinatorial Optimization, Linear Ordering Problem

ACM Reference Format:
Andoni I. Garmendia, Josu Ceberio, and Alexander Mendiburu. 2021. Exploratory Analysis of the Monte Carlo Tree Search For Solving The Linear Ordering Problem. In *2021 Genetic and Evolutionary Computation Conference Companion (GECCO '21 Companion), July 10–14, 2021, Lille, France.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3449726.3463163

GECCO '21 Companion, July 10–14, 2021, Lille, France

ACM ISBN 978-1-4503-8351-6/21/07…$15.00
https://doi.org/10.1145/3449726.3463163

1 INTRODUCTION

Combinatorial optimization (CO) is a branch of applied mathematics and computer science that studies how to find the optimal solution from a finite or countable infinite set of solutions. In order to solve Combinatorial Optimization Problems (COPs), scientists have developed a broad set of algorithms, which can be classified as exact and approximate methods. Exact methods perform a thorough search of the space and they are able to find the optimal solution to the problem, however, due to the exponential growth of the state space, the use of exact methods is computationally unaffordable and their usage is limited to small problem sizes. Among exact methods, tree search algorithms use the tree structure to traverse the state space in a organized manner. Especially, in those problems where obtaining upper and lower bounds of the solution for each node is easy, sub-trees can be efficiently discarded, and thus, the search is directed to a reduced space. In this context, Branch and Bound (BB) is an extension of the tree search algorithm that consists of visiting branches of the tree and checking if the bounds of the current branch can produce a better solution while discarding the rest.

Conversely, approximate methods, often called heuristic algorithms, try to find good (not necessarily optimal) solutions in a given computational budget. Among the approximation methods, metaheuristics are probably the most relevant [32]. Metaheuristics are defined as high level procedures that control lower level heuristics in an intelligent manner in order to efficiently explore and exploit the state space. They include, but are not limited to: local search [1], genetic algorithms [12], estimation of distribution algorithms [10], simulated annealing [2], tabu search [15], and their hybrids [8].

Besides the mentioned frameworks, recent works have tried to apply Machine Learning (ML) techniques to face COPs [7], using end-to-end schemes or combining ML with other optimization algorithms. Among the ML community, Reinforcement Learning (RL) has attracted the attention of researchers due to the fact that it is surprisingly capable of learning tasks or behaviours with neither previous domain knowledge nor labeled data. In a recent survey [23], the authors investigate the use of RL methods for COPs. RL algorithms have been applied to learn heuristics to solve combinatorial problems such as the Travelling Salesman [18], Vehicle Routing [24], Graph Coloring [16], Maximum Independent Set [3], Bin Packing [19] and the Knapsack problem [6] among others.

Monte-Carlo Tree Search (MCTS) is a tree search method combined with RL procedures that has shown a great performance in board games such as chess and GO [30]. MCTS records and estimates the statistics of problem states while exploring the most promising regions of the tree. MCTS can be divided in four stages:

Selection, Expansion, Evaluation and Backpropagation. A selection policy selects nodes until it reaches a leaf. Then, the leaf node is expanded, and a simulation policy performs a rollout to receive a reward. Finally, the reward is backpropagated to the visited nodes in such a way that the algorithm estimates an expected reward for each node.

Due to the recent success of MCTS, the machine learning community has applied it to solve COPs. The majority of works have used it as an end-to-end procedure, i.e., training the model to output solutions directly from the input instance [7]. However, the viability of using MCTS as an end-to-end procedure for large, real-world problems has not been demonstrated yet.

In order to investigate the feasibility of using MCTS as an end-to-end procedure, in this work, we analyze and explore the performance of the MCTS algorithm for permutation-based problems when problem-specific knowledge is available. Particularly, we focus on the Linear Ordering Problem (LOP), as expected rewards for a given node can be easily computed. Thus, this avoids the need to use neural networks to estimate the reward, making the algorithm simpler and more scalable. Furthermore, a property of the LOP allows sub-trees to be discarded if the partial permutation described by the root of the sub-tree cannot generate a local optima. The features above allow an efficient bounding procedure to be designed while navigating the tree. Finally, for the sake of maximizing the potential of MCTS on the LOP, and in order to better balance the exploration-exploitation trade-off during the search, we propose modifying the MCTS, multi-objectivizing the selection of the actions to take.

The remainder of the article is organized as follows. Section 2 gives a detailed description of the Linear Ordering Problem, and Section 3 provides some background about multi-objective optimization, which will be needed for later contributions. Afterwards, in Section 4, we describe the principles of the Monte-Carlo Tree Search algorithm. We present our approach in Section 5, putting together the knowledge we gained in previous sections. Next, in Section 6, the experimental setting and results are shown. A discussion on the potential of MCTS and future lines are given in Section 7, and the paper concludes in Section 8.

2 LINEAR ORDERING PROBLEM

The Linear Ordering Problem (LOP) [11, 21] is a classical Combinatorial Optimization Problem (COP). The LOP is a permutation problem that, in 1979, was proven to be NP-hard by Garey and Johnson [14]. Given a matrix $B = [b_{ij}]_{nxn}$, the goal is to find a simultaneous permutation of rows and columns such that the sum of the upper triangle entries is maximized. The objective function can be defined formally as in Eq. (1) where σ represents the permutation that simultaneously re-orders rows and columns of the original matrix and n is the problem size.

$$f(\sigma) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} b_{\sigma_i \sigma_j} \tag{1}$$

For illustration purposes, Fig. 1 shows an instance of size $n = 5$. If rows and columns were ordered as they are, that is, the identity permutation $\sigma = (1, 2, 3, 4, 5)$, summing up the entries in the upper triangle we get the objective value $f(\sigma) = 138$. The optimal solution

	1	2	3	4	5
1	0	16	11	15	7
2	21	0	14	15	9
3	26	23	0	26	12
4	22	22	11	0	13
5	30	28	25	24	0

(a)

	5	3	4	2	1
5	0	25	24	28	30
3	12	0	26	23	26
4	13	11	0	22	22
2	9	14	15	0	21
1	7	11	15	16	0

(b)

Figure 1: (a) Identity permutation and (b) optimum permutation of a $n = 5$ instance.

for this instance is obtained with $\sigma^* = (5, 3, 4, 2, 1)$ which has an optimal objective value of $f(\sigma^*) = 247$.

In what follows, we present a number of features of the LOP that will allow information of the problem to be incorporated into the design of the algorithm in latter sections. The first term of study is the contribution of an index to the objective value of the solution. Let item k be in the i^{th} position of the permutation σ, the contribution of item k to the objective value can be expressed as $c(\sigma, i) = \sum_{j=1}^{i-1} b_{\sigma_j \sigma_i} + \sum_{j=i+1}^{n} b_{\sigma_i \sigma_j}$ where b refers to the entry of the original matrix B. Note that the contribution of item k is independent of the ordering of the previous and posterior items.

The second term is the vector of differences, which is defined as the difference between the column-row pairs associated to each item of the permutation. Let d_i account for the vector of differences of item i as $d_i = (b_{\sigma_i 1} - b_{1\sigma_i}, ..., b_{\sigma_i n} - b_{n\sigma_i})$, that is to say, the element-wise difference between row i and column i.

The vector of differences can be used to determine whether, having a given item in certain position, that solution can be a local optimum or not [1]. Note that a solution σ^* is a local optimum if all neighbouring solutions have a lower objective value. As stated in [11], given a local optimum solution σ^*, for every item $\sigma_i^*, i = 1, ..., n$, all the partial sums of the differences between the associated entries located before i must be positive, while those located after i must be negative:

$$\sum_{j=i-1}^{z} (b_{\sigma_j^* \sigma_i^*} - b_{\sigma_i^* \sigma_j^*}) \geq 0, \qquad z = i-1, ..., 1 \tag{2}$$

$$\sum_{j=i-1}^{z} (b_{\sigma_j^* \sigma_i^*} - b_{\sigma_i^* \sigma_j^*}) \leq 0, \qquad z = i+1, ..., n \tag{3}$$

Indeed, with the help of the aforementioned property, discarding item-to-position assignments that can not be present in local optima solutions is straightforward. Specifically, item k does not generate a local optima at position i if Eq. (4) is true. Fig. 2 shows the calculation of the restricted positions for a particular item.

$$\sum_{z=i-1}^{1} (b_{\sigma_z k} - b_{k\sigma_z}) < 0 \quad or \quad \sum_{z=i+1}^{n} (b_{\sigma_z k} - b_{k\sigma_z}) > 0 \tag{4}$$

This condition can be extended to every item-position pair of values composing a binary *restrictions matrix* that can be computed with a computational complexity of $O(n^2 log(n))$. Fig. 3 shows the restrictions matrix corresponding to our example instance. A null

[1] By default, the neighborhood used to carry out studies and implement algorithms for the LOP is the "insert" neighborhood

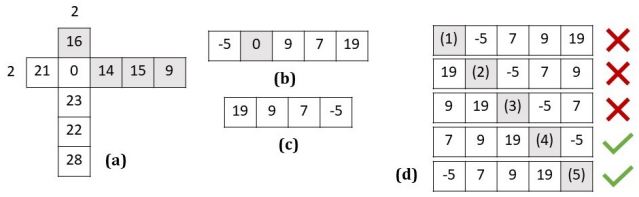

Figure 2: (a) Second row and column of the identity matrix. (b) The vector of differences associated to the second item. (c) Sorted differences in descending order. (d) The vector of differences in each position, the fourth and fifth positions are valid, the rest are restricted.

	1	2	3	4	5
1	0	0	0	0	1
2	0	0	0	1	1
3	1	1	1	0	0
4	0	1	1	1	1
5	1	0	0	0	0

Figure 3: restrictions matrix of our example instance of n=5.

entry, $r_{ij} = 0$, means that a local optimum will never have item i in position j.

The LOP, as a permutation problem, has $n!$ different solutions. The objective values of those solutions have a symmetrical distribution, i.e., each objective value is reflected around the average value of all the solutions. Let $\mathbb{E}[f]$ be the expected objective value of the problem, the objective value of half of the solutions in the search space is greater than $\mathbb{E}[f]$, while the other half has a lower objective value. In fact, each solution of the LOP can be obtained by reversing its opposite solution. This is an intuitive result since maximizing the upper triangle is the same as minimizing the lower triangle of the original matrix.

The computation of the expected value of f is shown in Eq. (5) where b_{ij} denotes the entry of the identity matrix of row i and column j and n is the problem size [2].

$$\mathbb{E}[f] = \frac{\sum_{i=1}^{n} \sum_{j=1}^{n} b_{ij}}{2} \tag{5}$$

A permutation whose positions have not been fully determined, are known as partial permutations. For instance, $(2, 3, -, -, -)$ is a partial permutation of our $n = 5$ sized problem, in which only the first two items have been placed.

Note that computing the expected objective value of a partial permutation (denoted by $\hat{\sigma}$) is straightforward: fixed items contribute with the corresponding entries on the upper triangle, while the sum of non-fixed entries is divided by two and added to the expected objective value. This is shown in Eq. (6) where m is the size of the partial permutation and $\hat{b}$ refers to the set of non-fixed entries.

$$\mathbb{E}[f(\hat{\sigma})] = \sum_{i=1}^{m-1} \sum_{j=i+1}^{m} b_{\sigma_i \sigma_j} + \frac{\sum \hat{b}}{2} \tag{6}$$

In a similar manner, an Upper Bound (UB) objective value for the problem (Eq. (7)) can be computed by taking the maximum values between pairs b_{ij} and b_{ji}.

$$UB[f] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \max\{b_{ij}, b_{ji}\} \tag{7}$$

Given a partial permutation $\hat{\sigma}$ with the first m positions fixed, the Upper Bound is computed by summing up the already placed entries in the upper triangle and the maximum values between non-fixed pairs $\hat{b_{kl}}$ and $\hat{b_{lk}}$ as shown in Eq. (8).

$$UB[f(\hat{\sigma})] = \sum_{i=1}^{m-1} \sum_{j=i+1}^{m} b_{\sigma_i \sigma_j} + \sum_{k=m+1}^{n-1} \sum_{l=k-1}^{n} \max\{\hat{b_{kl}}, \hat{b_{lk}}\} \tag{8}$$

3 MULTI-OBJECTIVE OPTIMIZATION

Multi-Objective Optimization (MOO) deals with problems in which the optimum is given by more than one objective-variable [13]. While single objective optimization tries to find the solution with the optimum objective value, the goal of MOO is to search for a set of optimal solutions called the Pareto optimal set. A solution being in the Pareto Optimal Set means that is not possible to find another solution superior to it with respect to all objectives.

Let R be a d-dimensional objective value vector corresponding to one of the solutions, where $R = (r_1, r_2, ..., r_d)$. For simplicity, it is assumed that each one of the d objectives is to be maximized. It is said that a solution x dominates or Pareto-dominates another solution y if and only if [31]:

$$r_i(x) \geq r_i(y) \quad \forall i = 1, 2, ..., d \tag{9}$$

and

$$r_i(x) > r_i(y) \quad \text{for at least one objective } d. \tag{10}$$

In the case when these two conditions do not apply, both solutions are said to be equally good. Thus, the Pareto Optimal Set is composed of solutions that are not dominated between themselves.

There are several approaches to tackle MOO problems. Although weighting the sum of the objective values is a classical approach to the problem that creates one single objective [20], this method implies properly adjusting the weights based on subject area knowledge or experimental trials and it only provides a linear approximation of the function. Instead, considering the whole Pareto Front as a set of values with equal probability of being chosen is a more conservative and secure idea, even though dominated solutions are never considered.

4 MONTE-CARLO TREE SEARCH

Monte-Carlo Tree Search is a heuristic algorithm used to solve problems with vast search spaces [9]. The algorithm is divided in four phases as shown in Fig. 4:

(1) *Selection.* The algorithm starts at the root node and descends the tree selecting a child based on a selection policy until it reaches a leaf node.

[2] We assume that the values in the diagonal are zero. If this is not the case, these values should be subtracted.

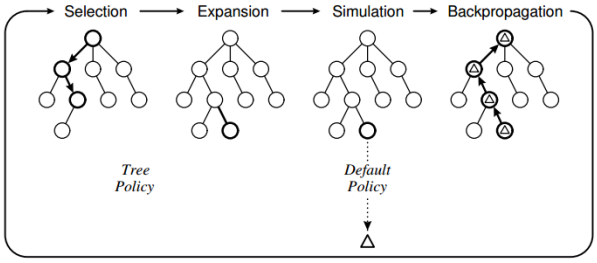

Figure 4: The four phases of the Monte-Carlo Tree Search Algorithm

(2) *Expansion.* Once a leaf node is found, new children nodes are expanded and added to the tree.
(3) *Simulation.* This phase starts from the expanded leaf node and travels down the tree until a final node is reached and a reward is obtained. Usually, action selection is performed by a random policy or heuristically.
(4) *Backpropagation.* The outcome is backpropagated through the previously selected nodes and node statistics are updated.

The algorithm starts from the root node and selects the most promising child recursively. This is done until a leaf node, or a non-visited node is found. Child selection is a multi-armed bandit problem. The policy selects an action (child) among the set of admissible actions based on maximizing the Upper Confidence Bound (UCB) vector defined by a sum of two elements, describing the exploitation and exploration factors [4]. Let S be the score value obtained by summing up the exploration and exploitation terms. Originally, the algorithm uses a constant c_e to balance both terms, $S = Q + c_e * U$ where Q and U are the exploitation and exploration terms respectively. The exploitation is usually given by the expected reward of a node, however, other values such as the upper-bound or a combination of both may be also used, while the exploration factor makes use of the visit count.

Eq. (11) depicts the original UCB score from [4]. The exploitation factor $Q = r_{sa}$ is the average reward when selecting child a in node s, while the exploration is given by a function of n_s and n_{sa}, referring to the total visit count of node s and the number of times child a has been selected in node s, respectively.

$$UCB = r_{sa} + c_e \sqrt{\frac{ln(n_s)}{n_{sa}}} \tag{11}$$

According to [17], and assuming the reward values range from 0 to 1, the exploration-exploitation ratio parameter value of $c_e = \sqrt{2}$ satisfies the Hoeffding inequality and is widely used in the literature.

Note that a child with no visits has an infinite UCB score, ensuring that all children must be visited at least once before exploring any other already visited child. As mentioned in [25], this behaviour results in a form of a local search. Conversely, in the recent work by DeepMind [29], called AlphaGO, they make use of an alternative score function replacing n_{sa} by $1 + n_{sa}$ where a non-visited child has a finite score value and, therefore, the local search component is reduced. Particularly, in [29] they introduce a new exploration factor, which is a variant of the original PUCT algorithm [26] and uses prior probabilities ($P_{s,a}$) to control the exploration term (see Eq. (12)).

$$U_{PUCT} = r_{sa} + c_e * P_{sa} * \sqrt{\frac{n_s}{1 + n_{sa}}} \tag{12}$$

Back in the MCTS procedure, once a leaf node is reached, children nodes are expanded and evaluated based on node statistics. Statistical information of nodes, i.e., either expected reward, visit counts or prior probabilities are gathered from a different phase called simulation-phase. In [29] two models were trained to infer the exploitation value and the prior probabilities. However, and based on the original MCTS algorithm, node statistics are obtained from random simulations (also known as rollouts).

Simulations (or rollouts) are performed from the expanded node until the terminal node. The policy to follow during the simulations can be stochastic, or it can follow some specific heuristic that helps incorporating problem knowledge to the decision-making. Simulations give an approximate view of the rest of the tree.

Once each rollout ends up reaching a terminal state, the final outcome is evaluated and propagated backwards through the visited nodes. The objective value is saved in the tree for future use. Often, the tree saves statistics about the total reward obtained from the current node and the number of times this node has been visited. That way, the expected reward of the node can be approximated dividing the total reward by the number of visits.

5 MO-MCTS FOR LOP

Once the LOP and MCTS have been introduced, in this section, we describe the design proposed for MCTS that takes advantage of incorporating the problem specific knowledge presented in Section 2. First, the general MCTS procedure is described, then, the Selection and Expansion phases are introduced and, finally, an unconventional procedure for the Simulation and Backpropagation phases is explained. This is an exploratory approach on the use of MCTS, thus, many other parameters, metrics and rules could be used. In order to illustrate the explanation, Fig. 5 is depicted.

The tree is defined as follows: The root node is equivalent to an empty solution and from there, n children are expanded, each one referring to put a particular item on the first position of the permutation. At the next level, any of the remaining $n-1$ items can be chosen, and so on (see the tree in Fig. 5). The tree is composed of n levels, corresponding to the number of positions. However, we may consider that the selection of the item in the last position is straightforward, so only the first $n-1$ levels really deserve our attention. The n^{th} level is composed of $n!$ nodes, one per permutation solution of the search space, while the total number of nodes in the whole tree is given by:

$$1 + \sum_{i=0}^{n-1} \prod_{j=0}^{i} (n - j) \tag{13}$$

Conventional MCTS chooses actions based on the score value S. Balancing the trade-off between exploitation, Q, and exploration, U is not straightforward, and it is a critical point for the MCTS. Thus, in this work, instead of searching for an appropriate C_e value, we redefine the problem as a bi-objective decision problem with two independent functions to be maximized, Q and U. Therefore, in

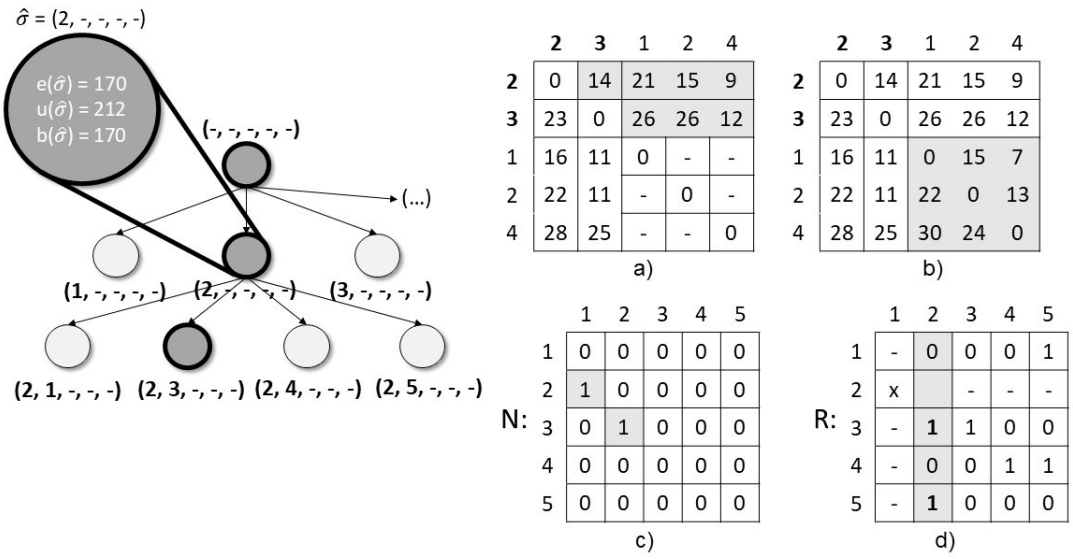

Figure 5: Illustration of the selection of an item in the second position after item 2 has been placed in the first position. Children nodes are expanded and expected and upper values are computed. a) If item 3 is selected in the second position, $\hat{\sigma} = (2, 3, -, -, -)$, the sum of fixed entries of the upper triangle, called S, is $S = 14 + 21 + 15 + 9 + 26 + 26 + 12 = 123$. b) The expected and the upper objective values are computed as follows: $\mathbb{E}[\hat{\sigma}] = S + (15 + 7 + 22 + 13 + 30 + 24)/2 = 178.5$ and $UB[\hat{\sigma}] = S + (22 + 30 + 24) = 199$. c) Matrix N represents the marginal visit count, where each entry $n_{i,j}$ denotes the number of times item i has been placed in the j^{th} position. As it is the first rollout, only two entries have a non-zero value. d) Finally, R is the restrictions matrix of the partial permutation $\hat{\sigma} = (2, -, -, -, -)$, where entry $r_{i,j}$ denotes that a local optimum for the insert neighborhood can have item i in position j only when $r_{i,j} = 1$. Therefore, it will be pruned whenever $r_{i,j} = 0$. The algorithm first calculates the restrictions matrix, and computes the expected and upper values only for those non-pruned children nodes. After selecting a child, the visit count matrix is updated.

each step all children are evaluated to form the pareto front, and an action among the set of non-dominated solutions (children nodes) is chosen. In this particular case, the probabilities of all the nodes in the Pareto Optimal Set are distributed uniformly.

The two objective functions make reference to the exploitation (Q) and exploration (U) factors, defined in Eq. (14) and 15 respectively.

In Eq. (14), $u(\hat{\sigma})$ refers to a function that, given a partial permutation $\hat{\sigma}$, yields its Upper Bound value. Similarly, $e(\hat{\sigma})$ and $b(\hat{\sigma})$ denote the functions that return the expected value and the best value found passing from the current node. The upper bound being the main guiding factor on the exploitation term, those children nodes with a higher value $b(\hat{\sigma})$ have an extra score. Thus, this gives advantage to already visited nodes where good solutions were found.

$$Q(\hat{\sigma}) = u(\hat{\sigma}) + (b(\hat{\sigma}) - e(\hat{\sigma})) \tag{14}$$

Regarding the exploration factor defined in Eq. (15), due to the large size of the tree and sparsity of the search paths, the visit count of a node rarely surpasses a single count when running short-time executions. Therefore, we will compute a marginal visit count matrix N in which a value n_{ij} represents the number of times item i has been placed in position j. Note that i is the last fixed item and j is its position in the partial permutation $\hat{\sigma}$.

$$U(\hat{\sigma}) = \frac{1}{1 + n_{ij}} \tag{15}$$

In the second step of the MCTS, the expansion phase, we introduce two tree-pruning schemes, which similarly to B&B, allow the number of nodes in the tree to be reduced by pruning sub-trees where the optimal solution cannot be found. As mentioned previously, the restrictions matrix of the LOP declares the positions at which items do not generate a local optimum solution for the insert neighborhood and, thus, neither for the global optimum [11]. Our algorithm performs a pruning of the tree whenever an expanded child is restricted. In the example of Fig. 5, only two items (3 and 5) can be placed in the second position, based on the restrictions matrix R (see Fig. 5d).

In addition, the second pruning strategy discards every child with an upper bound value lower than the value of the best solution found so far. This is an intuitive pruning strategy, since the maximum obtainable value is lower than the best found so far.

In the classical MCTS algorithm, once a child is expanded, a simulation phase starts and it collects statistics of the current sub-tree. However, we found that performing rollouts from the root node until the final node of the tree instead of using simulation phases obtained much better results. The reason for this may be the use of harsh restrictions in areas where the algorithm has been searching

for a while. Therefore, those efforts are wasted when reaching a node with restricted children. This may appear to be an unconventional MCTS algorithm, but the fact that rollouts repeatedly start from the root reduces the local search nature of the algorithm, and instead, focuses on a more global search.

The backpropagation, which is the final phase of an iteration of the MCTS, consists of gathering the final outcome and navigating back to the visited nodes to estimate how good a path is compared to others. We do not feel the need to save all the final outcomes since we already have the real expected value of each node, instead, the final objective value will be back-propagated only if it is greater than the best value found so far. In the example in Fig. 5, the best found value is initialized as the expected objective value ($b(\hat{\sigma}) = 170$), however, once the last level is reached and a better solution is found, $b(\hat{\sigma})$ is updated.

6 EXPERIMENTATION

In order to prove the validity of MCTS for solving the LOP, in what follows, a set of experiments is performed. First, a general performance analysis of the algorithm is presented together with a comparison to other methods. Then, the scalability of MCTS is studied, specifically focusing on its memory usage for different sized problems. Finally, an analysis of the search space coverage is conducted in order to give an intuition of the pruning efficiency.

6.1 Experimental setting

The experimental benchmark consists of a set of 30 instances from the LOLIB benchmark [22], 20 of them are from the RandomB type of size $n = 50$ and the other 10 instances are from the RandomAI sub-type, where the size of instances is $n = 100$. We also considered 39 instances of size $n = 150$ and 10 instances of size $n = 250$ from xLOLIB, a more challenging library generated by Schiavinotto et al. [28]. The best known values are taken from the supplementary material in [27] and correspond to the state-of-the-art values obtained by the CD-RVNS in that work.

We compare our algorithm with a classical constructive heuristic proposed by Becker et al. [5]. We also introduce a simple heuristic that tries to imitate our method in a naive manner. The so-called Upper Greedy algorithm is a deterministic constructive algorithm that selects the available item that would maximize the upper bound if selected and positioned in the partial permutation. Algorithm 1 shows the procedure of the Upper Greedy algorithm, where $UB(permutation, item)$ refers to the Upper Bound of the partial $permutation$ when appending $item$ to the first non-fixed position.

Finally, with the intention of putting into perspective where the obtained results are, i.e., what is the relative difficulty of an instance, a random search algorithm has been run. This method produces random permutations during the time limit while a record of the best solution found is saved.

We establish a computation time limit of 20 minutes as the stopping criterion for the algorithms, which are executed in a set of 20 repetitions, with the exception of Becker and Upper Greedy, due to their deterministic nature. Experiments are run on a cluster of 55 nodes, each one of which is equipped with two Intel Xeon X5650 CPUs and 64GB of memory, while the algorithms have been implemented in Python 3.6.

Algorithm 1: Upper Greedy constructive heuristic

```
permutation ← Empty List;
for position ← 1 to N do
    max ← 0;
    for item in Valid Items do
        u = UB(permutation, item);
        if u > max then
            max ← u;
            i ← item;
        end
    end
    permutation.append(i);
end
```

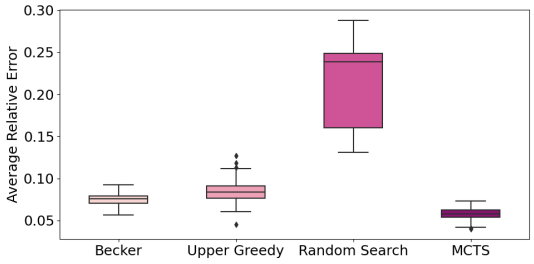

Figure 6: Overall performance of MCTS, Upper Greedy, Becker and Random Search algorithms, given by the Average Relative Error w.r.t. the best known objective value found by the CD-RVNS algorithm.

6.2 Experiment 1 - Performance Analysis

Fig. 6 presents the results obtained with MCTS, Becker and Upper Greedy for all 79 instances. The figure reveals that overall, MCTS performs better than Becker. In fact, MCTS obtained better results in 75 instances out of 79 (94.9%). However, MCTS is still far from state-of-the-art results obtained by more powerful and hybrid metaheuristics [27]. In particular, the overall results obtained by MCTS are 6% worse than those presented by CD-RVNS [27]. Regarding the Upper Greedy constructive method, it surpasses Becker only 20 of the times (25.3%). MCTS improves the solution given by Upper Greedy for all the instances, thus, the MCTS procedure contributes positively compared to the Upper Greedy algorithm.

6.3 Experiment 2 - Scalability

In order to analyze the capability of the algorithm to scale for larger sized problems, Fig. 7 divides the instances in 4 sub-groups based on instance size $n = \{50, 100, 150, 250\}$. In view of the results, we conclude that MCTS is the best performing algorithm for the four problem sizes. Upper Greedy and Random Search algorithms produce worse results as the problem size increases, while MCTS

Table 1: Memory consumption comparison for different instance sizes in a 20 minute execution.

Instance Size	Tree Size	Nodes/s	Memory	MB/s
50	291691	243.07	1209 MB	1.01
100	67131	55.94	1064 MB	0.89
150	27451	22.88	793 MB	0.66
250	8375	6.98	666 MB	0.56

and Becker show a better scalability maintaining the performance nearly uniform.

However, one of the drawbacks of MCTS is the amount of memory needed to run the algorithm. The main factor on the memory use is the growth of the tree and data stored in it. For each visited node, a set of statistics are saved (the expected objective value and the upper bound of the partial permutation described by the node) and they are calculated once, at the time of the expansion of the node. On the contrary, the visit count and the best value found of the current node are initialized to zero and to the expected value respectively at node expansion. Those values are updated during the execution. Table 1 shows a summary of the memory usage during the MCTS execution of 20 minutes. The node-discovery rate and the memory consumption rate are computed based on the values at the end of the execution, assuming they stay constant during execution (which is indeed the case). The algorithm is capable of visiting a larger set of nodes in smaller instances, since the node-processing time is reduced. Even though saved statistics in bigger sized instances have greater memory requirements, small instances present a higher memory consumption for the same amount of time, as they are able to visit significantly more branches of the tree.

6.4 Experiment 3 - Search Space Coverage

The percentage of visited and discarded nodes may give an idea of how capable MCTS is of finding the optimal solution. Fig. 8 shows the percentage of the tree that has been pruned during the 20 minute execution. Note that depending on the size of the problem, the shape of the curve is different.

At the beginning of the execution, when the root node is processed, a noticeable amount of sub-trees are discarded based on the restrictions matrix. Consider that discarding a child in the first level of the tree removes $\frac{1}{n}^{th}$ of the tree, and, thus, of the search space of solutions. Then, while the optimization goes on, fewer nodes are discarded since the restrictions come from lower levels of the tree. Once the majority of the nodes in the initial levels are removed using the restrictions matrix, the pruning rate decreases as it can be seen for the $n = 50$ instance in Fig. 8. Comparing both pruning strategies, on average, the upper bound strategy prunes smaller sub-trees, corresponding to lower levels, while the restrictions matrix strategy prunes the first levels. In fact, more than 99.99% of the pruned nodes have been discarded by the restrictions matrix.

7 DISCUSSION

In the performed experiments, MCTS has shown to be competitive compared to the best performing classical heuristic for the LOP. However, this is still far from being a state-of-the-art algorithm. MCTS underperforms when compared to a metaheuristic, such as CD-RVNS, proposed in [27]. This poses the question: is MCTS a valid approach to tackle the Linear Ordering Problem? In what follows we will try to answer to that question from different viewpoints.

7.1 Scalability

Even though MCTS performs competitively in terms of scalability, a key factor that limits its potential is the memory limit. Even though the 20 minute execution occupies 1GB of memory, larger duration executions would increase the memory consumption in an almost linear way. The use of Neural Networks to learn the tree-data may be a possible solution to reduce memory consumption, because many node statistics would be inferred, and there would be no need to save them. However, the goal of this paper is to study the intrinsic capacity of the algorithm to explore good solutions in vast search spaces, and adding an abstraction layer with neural networks would aggregate noise in the results and hinder the main objective.

7.2 Search Space Coverage

The rate in which sub-trees are pruned (pruning-rate) achieves its maximum in the initial step when the root node is being processed and goes decreasing in an asymptotic way. Conversely, the rate in which new leaves are extended (discovery-rate) is in orders of magnitude lower than the pruning rate at the beginning of the execution, and it decreases due to the increase of distance to new leaves. Taking as an example the instance of size $n = 100$, Fig. 8 shows that in the first step 50% of the tree is discarded, while at the end of the execution it increases to 80%. The tree has in total $n! = 100! = 9.33e + 157$ possible solutions, thereby, MCTS is capable of discarding $7.46e + 157$ solutions. This may appear to be a big percentage, however, the algorithm has still to explore among $1.87e + 157$ solutions. In fact, Table 1 indicates that MCTS visits 67131 different nodes during the 20 minute execution, so assuming a constant visit rate and that the discard rate is stopped, the search algorithm would need a countless number of years ($5.284562e+148$) to explore all the space.

7.3 Optimality

The exploration factor of the MOO approach in the node selection phase guarantees that every unrestricted node can be selected at least once. Some heuristics guarantee optimality if unlimited computation time is given. In our case, apart from the time limitation, we need to consider the growth of the tree and the memory needed to save it.

The MCTS design presented in this work is optimality-driven. By means of the restrictions matrix, those solutions that can not be (locally or even globally) optimum are discarded. However, we should note that solutions belonging to the neighborhood of the global optimum will never be considered, and thus solutions with (probably) good fitness values are ignored. On the other hand, thanks to the restrictions matrix, a wide area of the search space is discarded, focusing on the area in which the global optimum (optima) is. This is an interesting aspect to be studied as future work.

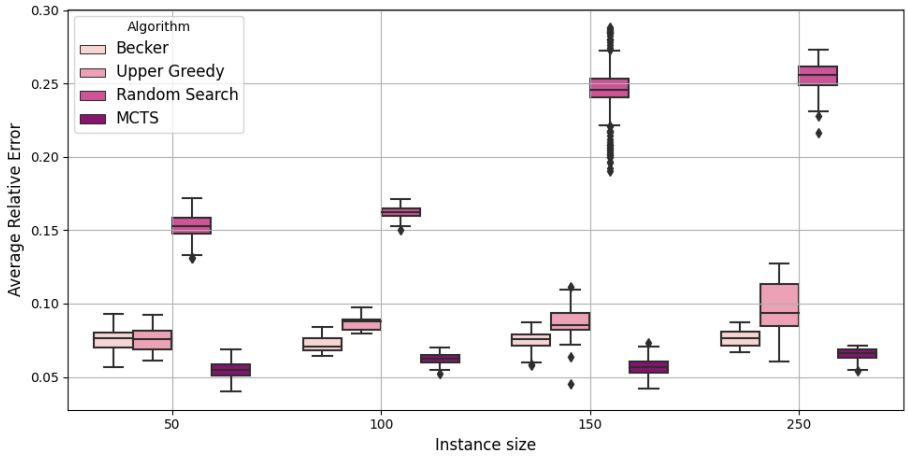

Figure 7: Performance comparison between MCTS, Upper Greedy, Becker and Random Search. Results are grouped in different instance sizes and shown as Average Relative Error w.r.t. the best known objective value found by CD-RVNS.

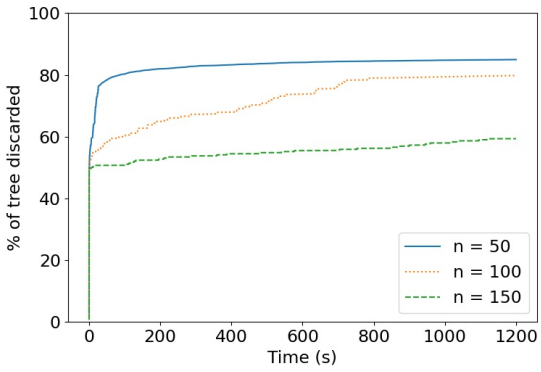

Figure 8: Discarded percentage space using restrictions for instances of size 50, 100 and 150.

7.4 Future Work

Instead of applying MCTS as an end-to-end approach, we feel that the use of collaborative schemes, i.e., MCTS alongside other optimization algorithms, may play a more promising role in future works [7].

Given that MCTS limits quite quickly the search space based on restrictions and Upper Bounds, it could be useful to launch another algorithm on the reduced space after this restriction step. It would be of interest to analyze whether making this first restriction step would have an impact on the performance of the other algorithm.

From the restrictions matrix, the most restricted areas are the beginning and the end of the permutation, making the selection of those positions more "certain", while middle positions are "uncertain". The fact that the LOP has this dual nature makes us think about the possibility of creating two opposed trees that construct a unique permutation. One begins from the first position and the other from the last position and iteratively both trees select left and right halves of the permutation.

8 CONCLUSION

In this work, we defined and explored the Monte Carlo Tree Search as a constructive algorithm that employs Multi-Objective Optimization to navigate the tree and balance the weight between exploration and exploitation. We selected the Linear Ordering Problem due to its properties which facilitate the computation of the expected objective value and the upper bound objective value of each node.

Conducted experiments showed that MCTS is competitive when compared to other constructive heuristics in terms of performance and scalability. However, time and memory play a key factor, limiting the performance of MCTS for large instances.

Moreover, our approach has shown a good capability pruning the tree for small and medium instances. Especially, the processing of the root node at the initial step discards around half of the solutions on average. There are still many aspects that deserve more study in order to gain more intuition about the possibilities of MCTS to solve this kind of problem.

9 ACKNOWLEDGEMENTS

Andoni I Garmendia acknowledges a predoctoral grant from the Basque Goverment (ref. PRE_2020_1_0023). This work has been partially supported by the Research Groups 2019-2020 (IT1244-19) and the Elkartek Program (Project Code KK-2020/00049) from the Basque Government, the PID2019-104933GB-10 and PID2019-106453GA-I00/AEI/10.13039/501100011033 research projects from the Spanish Ministry of Science, and the European Research Council H2020 (the EMPHATIC project).

REFERENCES

[1] Emile Aarts, Emile HL Aarts, and Jan Karel Lenstra. 2003. *Local search in combinatorial optimization*. Princeton University Press.

[2] Emile Aarts and Jan Korst. 1989. *Simulated annealing and Boltzmann machines: a stochastic approach to combinatorial optimization and neural computing*. John Wiley & Sons, Inc.

[3] Kenshin Abe, Issei Sato, and Masashi Sugiyama. 2019. Solving NP-Hard Problems on Graphs by Reinforcement Learning without Domain Knowledge. *Simulation* 1 (2019), 1–1.

[4] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. 2002. Finite-time analysis of the multiarmed bandit problem. *Machine learning* 47, 2 (2002), 235–256.

[5] O Becker. 1967. Das Helmstädtersche Reihenfolgeproblem—die Effizienz verschiedener Näherungsverfahren. *Computers uses in the social science* (1967).

[6] Irwan Bello, Hieu Pham, Quoc V Le, Mohammad Norouzi, and Samy Bengio. 2016. Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940* (2016).

[7] Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. 2020. Machine learning for combinatorial optimization: a methodological tour d'horizon. *European Journal of Operational Research* (2020).

[8] Christian Blum, Jakob Puchinger, Günther R Raidl, and Andrea Roli. 2011. Hybrid metaheuristics in combinatorial optimization: A survey. *Applied soft computing* 11, 6 (2011), 4135–4151.

[9] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. 2012. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games* 4, 1 (2012), 1–43.

[10] Josu Ceberio, Ekhine Irurozki, Alexander Mendiburu, and Jose A Lozano. 2012. A review on estimation of distribution algorithms in permutation-based combinatorial optimization problems. *Progress in Artificial Intelligence* 1, 1 (2012), 103–117.

[11] Josu Ceberio, Alexander Mendiburu, and Jose A Lozano. 2015. The linear ordering problem revisited. *European Journal of Operational Research* 241, 3 (2015), 686–696.

[12] Lawrence Davis. 1991. Handbook of genetic algorithms. (1991).

[13] Kalyanmoy Deb. 2014. Multi-objective optimization. In *Search methodologies*. Springer, 403–449.

[14] Michael R Garey and David S Johnson. 1979. Computers and Intractability: A Guide to the Theory of NP-Completeness. WH Freeman & Co. New York, NY, USA. (1979).

[15] Fred Glover. 1989. Tabu search—part I. *ORSA Journal on computing* 1, 3 (1989), 190–206.

[16] Jiayi Huang, Mostofa Patwary, and Gregory Diamos. 2019. Coloring big graphs with alphagozero. *arXiv preprint arXiv:1902.10162* (2019).

[17] Levente Kocsis and Csaba Szepesvári. 2006. Bandit based monte-carlo planning. In *European conference on machine learning*. Springer, 282–293.

[18] Wouter Kool, Herke Van Hoof, and Max Welling. 2018. Attention, learn to solve routing problems! *arXiv preprint arXiv:1803.08475* (2018).

[19] Alexandre Laterre, Yunguan Fu, Mohamed Khalil Jabri, Alain-Sam Cohen, David Kas, Karl Hajjar, Hui Chen, Torbjørn S Dahl, Amine Kerkeni, and Karim Beguir. 2019. Ranked Reward: Enabling Self-Play Reinforcement Learning for Bin packing. (2019).

[20] R Timothy Marler and Jasbir S Arora. 2010. The weighted sum method for multi-objective optimization: new insights. *Structural and multidisciplinary optimization* 41, 6 (2010), 853–862.

[21] Rafael Martí and Gerhard Reinelt. 2011. *The linear ordering problem: exact and heuristic methods in combinatorial optimization*. Vol. 175. Springer Science & Business Media.

[22] Rafael Martí, Gerhard Reinelt, and Abraham Duarte. 2012. A benchmark library and a comparison of heuristic methods for the linear ordering problem. *Computational optimization and applications* 51, 3 (2012), 1297–1317.

[23] Nina Mazyavkina, Sergey Sviridov, Sergei Ivanov, and Evgeny Burnaev. 2020. Reinforcement learning for combinatorial optimization: A survey. *arXiv preprint arXiv:2003.03600* (2020).

[24] Mohammadreza Nazari, Afshin Oroojlooy, Lawrence V Snyder, and Martin Takáč. 2018. Reinforcement learning for solving the vehicle routing problem. *arXiv preprint arXiv:1802.04240* (2018).

[25] Rui Jorge Rodrigues Rei. 2018. Monte Carlo Tree Search for Combinatorial Optimization. (2018).

[26] Christopher D Rosin. 2011. Multi-armed bandits with episode context. *Annals of Mathematics and Artificial Intelligence* 61, 3 (2011), 203–230.

[27] Valentino Santucci and Josu Ceberio. 2020. Using pairwise precedences for solving the linear ordering problem. *Applied Soft Computing* 87 (2020), 105998.

[28] Tommaso Schiavinotto and Thomas Stützle. 2004. The linear ordering problem: Instances, search space analysis and algorithms. *Journal of Mathematical Modelling and Algorithms* 3, 4 (2004), 367–402.

[29] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. 2016. Mastering the game of Go with deep neural networks and tree search. *nature* 529, 7587 (2016), 484–489.

[30] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. 2017. Mastering the game of go without human knowledge. *nature* 550, 7676 (2017), 354–359.

[31] David A Van Veldhuizen and Gary B Lamont. 1998. Evolutionary computation and convergence to a pareto front. In *Late breaking papers at the genetic programming 1998 conference*. Citeseer, 221–228.

[32] Mutsunori Yagiura and Toshihide Ibaraki. 2001. On metaheuristic algorithms for combinatorial optimization problems. *Systems and Computers in Japan* 32, 3 (2001), 33–55.