Local Optimization in Monte-Carlo Tree Search for the Traveling Salesman Problem

Clément DUMAS

Lycée Thiers, 13001 Marseille, France
butanium.contact@gmail.com

Abstract – In this study, we implement a Monte-Carlo Tree Search (MCTS) for the Traveling Salesman Problem (TSP). We begin with the implementation outlined by Shimomura and Takashima [2016], enhancing the approach with local optimization by implementing the 2-opt algorithm. The proposed method significantly outperforms that of Shimomura and Takashima [2016] and appears to improve upon iterated 2-opt.

1 Background

1.1 Traveling Salesman Problem

Consider a set of n cities $v_1, ..., v_n$ on a plane. Let G be the complete graph with cities as vertices and edges weighted by the integer part of the Euclidean distance between cities (1).

$$d(v_i, v_j) = \left\lfloor \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \right\rfloor \quad (1)$$

A tour $\pi = (a_1, ..., a_n)$ is defined as a circuit that passes through each vertex exactly once, with the weight w defined as:

$$w(\pi) = \sum_{i=1}^{n-1} d(a_i, a_{i+1}) + d(a_n, a_1)$$

The Traveling-Salesman Problem (TSP) for n cities aims to find the minimum weight tour π_m in G.

1.2 2-opt Algorithm

The 2-opt algorithm is a local optimization heuristic for TSP. Beginning with a tour π, it looks for edges $(a_i,\ a_{i+1})$ and $(a_j,\ a_{j+1})$ such that their removal and replacement with $(a_i,\ a_j)$ and $(a_{i+1},\ a_{j+1})$ result in a reduced weight of the tour. After an iteration of 2-opt:

$$\pi = a_1, \ldots, a_i, a_{i+1}, \ldots, a_j, a_{j+1}, \ldots, a_n$$

transforms into

$$\pi' = a_1, \ldots, a_i, a_j, a_{j-1}, \ldots, a_{i+2}, a_{i+1}, a_{j+1}, \ldots, a_n$$

if $w(\pi') < w(\pi)$. The process is shown in figure 1. Once no such (i, j) pairs can be found, the algorithm terminates, and the round is declared *2-optimized*. An example of applying the 2-opt algorithm is depicted in figure 2.

Empirical results in Table 3, provided by Johnson and McGeoch [1997], show that a 2-optimized tour has an average error of less than 5% compared to the exact solution. This leads to defining the *iterated 2-opt heuristic* as outlined in Algorithm 1.

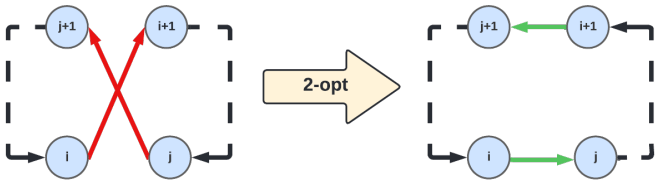
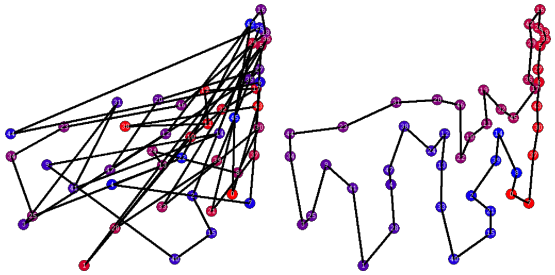


FIG. 1: A modification after one iteration of 2-opt. Dashed lines symbolize a path connecting 2 cities.



(a) Random tour (b) Random tour after 2-optimization

FIG. 2: 2-optimization of a random tour

1.3 Monte-Carlo Tree Search

Consider an agent playing a game, initially in state E_0. The agent must choose an action a_{0,i_0}, which leads to state E_1, followed by actions $a_{1,i_1}, \ldots, a_{n-1,i_{n-1}}$, culminating in a final state E_n. The goal is to determine the sequence maximizing the score w attributed to E_n. The Monte-Carlo Tree Search (MCTS) is a probabilistic approach to this challenge, iteratively constructing a tree of potential states, with the nodes of the tree representing different game states.

Monte Carlo Tree Search (MCTS) is an algorithm proposing a probabilistic approach to the problem at hand. Iteratively, a

Algorithm 1: Iterated 2-opt

$best\_tour \leftarrow [1, \ldots, n]$;
$w_{min} = \infty$;
while τ **do**
    $\pi \leftarrow$ random tour;
    `2-opt`(tour);
    if `w`*(tour)* < w_{min} **then**
        copy π in $best\_tour$;
        $w_{min} \leftarrow w(\pi)$
return $best\_tour$

search tree is constructed, where nodes represent different possible states. The children of a node E_i are the states reachable by taking an action from E_i. In each node, we store the number of visits and the average score $\overline{w}$ obtained by passing through that node.

Algorithm 2 starts with a tree composed of the initial state node R_0, without children. Then, a loop is initiated and continues for a duration of τ. At each iteration:

1. *Selection*: Identify the most promising part of the tree for expansion.
2. Create a new child node from the selected node using the `expand` function.
3. Complete a random play-out starting from the new node with the *simulation* policy until a final state.
4. Perform *backpropagation* by including the play-out score in the selected node and its ancestors.

Throughout the algorithm, we keep track of the sequence of actions yielding the best score.

Algorithm 2: MCTS Algorithm

$root \leftarrow R_0$;
$max\_w \leftarrow 0$;
$best\_actions \leftarrow$ empty list;
while *execution time is less than* τ **do**
    $node \leftarrow$ `select`$(root)$;
    $child \leftarrow$ `expand`$(node)$;
    $w, actions \leftarrow$ `simulate`$(child)$;
    if $w > max\_w$ **then**
        $max\_w \leftarrow w$;
        $best\_actions \leftarrow actions$
    `backpropagation`$(child, w)$
return $best\_actions$

1.3.1 Selection

Selection involves traversing the tree until finding a terminal node or a node with an untried action, indicating an undeveloped child. To achieve this, we recursively select the child E_{i+1} of E_i that maximizes (2) following Algorithm 3.

Algorithm 3: Selection Function

Argument: the root of the tree R
Returns : the node to expand
$node \leftarrow R$;
while *node is not terminal* ***and*** *node has all its children expanded* **do**
    $node \leftarrow$ child of $node$ maximizing UCT
return $node$

$$\text{UCT} = \boxed{\overline{w_{i+1}}} + \boxed{C_{exp} \cdot C_p \sqrt{\frac{\ln n_i}{n_{i+1}}}} \qquad (2)$$

Here, $\overline{w_{i+1}}$ is the average score of E_{i+1}, and the n_j are the number of times the j^{th} node has been visited during selection. C_{exp} is the exploration constant, and C_p is a constant proportional to the problem size. If the scores w obtained are in $[0, 1]$, then $C_p = 1$. C_{exp} defaults to $\sqrt{2}$ but can be empirically adjusted to favor exploration or exploitation.

Indeed, this equation is split into two terms:

1. The green term is the *exploitation* term, reflecting the quality of solutions that can be reached from this node.
2. The red term is the *exploration* term, favoring children that have been overlooked during previous selections, thereby encouraging the algorithm not to disregard paths that might seem unpromising at first glance.

If a terminal node is reached, backpropagation is performed with the score associated with the final state. Otherwise, the tree is expanded by creating one of the missing children of the final node and launching the *simulation* from there.

1.3.2 Simulation

Simulation from a node involves performing a sequence of random actions starting from the state of the node until reaching a final state. The score of this final state is then used in backpropagation. It should be noted that not all states traversed during the simulation are stored in the tree to prevent memory saturation.

1.3.3 Backpropagation

Backpropagation from a node E_i with a value w consists of incorporating w into the average score $\overline{w_i}$ of E_i and incrementing its n_i, then doing the same for all its ancestors. More specifically, for $node \in \{E_i \text{ and its ancestors}\}$, the following is performed:

$$\overline{w}_{node} \leftarrow \frac{n_{node} \times \overline{w}_{node} + w(\pi)}{n_{node} + 1}$$
$$n_{node} \leftarrow n_{node} + 1$$

2 Applying MCTS to the TSP

2.1 Initial MCTS Implementation for TSP

We base our initial MCTS implementation for TSP on the approach by Shimomura and Takashima [2016].

2.1.1 TSP-specific formulation for MCTS

For the TSP containing n cities, we define a game state as a partial path including $p \leqslant n$ distinct cities. With the agent starting with the path containing only the first city v_1, each successive action involves choosing an unvisited city. After $n-1$ actions, the agent completes a tour π by connecting the last visited city back to v_1, and the score is the weight $w(\pi)$.

2.1.2 Adapted Selection Policy

Since the problem is one of minimization, not maximization, we must reflect this in our selection policy. Hence, during selection, rather than choosing the child that maximizes (2), we will instead choose the one that minimizes (3).

$$\text{UCT2} = \overline{w_{i+1}} - C_{exp} \cdot C_p \sqrt{\frac{\ln n_i}{n_{i+1}}} \tag{3}$$

There is no general method to define C_p, so Shimomura and Takashima [2016] propose using either double the weight of the minimum spanning tree of G or double the standard deviation of the first n values found for w. The latter method is feasible because UCT2 is used only after all the children of the root have been expanded.

2.1.3 Adapted Simulation Policy

Simulation consists of randomly completing the partial path with the remaining cities. There are two ways to select cities at random. The first is to select completely at random, giving each city an equal chance to be chosen. The second is to select each subsequent city v' after having chosen v_d with the probability according to (4).

$$\mathbb{P} = \frac{1}{Z} \cdot \frac{1}{\text{dist}(v', v_d)} \quad \text{where} \quad Z = \sum_{v \text{ possible}} \frac{1}{\text{dist}(v, v_d)} \tag{4}$$

This method is referred to as *roulette* by Shimomura and Takashima [2016].

2.2 Related Work

The approach described in 2.1 represents the first implementation of MCTS applied to TSP. However, this method did not yield very good results. Several tests on the $att48$ configuration from TSPLIB (Reinelt [1991]) show that after 30 minutes, a tour π with 20% relative error as defined in (5) is obtained, which is not 2-optimized.

$$\text{Relative Error}(\pi) = \text{Err}(\pi) = \frac{w(\pi) - w_{\min}}{w_{\min}} \tag{5}$$

Since then, there have been two noteworthy publications combining MCTS and TSP (Fu et al. [2020], Xing and Tu [2020]). These methods differ significantly from the method by Shimomura and Takashima [2016], particularly because they involve neural networks. My work focuses on improving Shimomura and Takashima [2016]'s method, and the functioning of these two new methods is described in Appendix A.

3 Integrating 2-opt within MCTS

Our contention that non-2-optimized solutions from the Shimomura and Takashima [2016] method could benefit from 2-opt optimizations led to three distinct approaches for integration: hidden optimization, prepropagation, and optimized simulation.

3.1 Hidden Optimization

Hidden optimization involves applying the 2-opt algorithm to tours post-simulation and backpropagation. This ensures that the returned MCTS tour is 2-optimized. The modified algorithm is shown in Algorithm 4.

Algorithm 4: MCTS with hidden optimization

```
while temps d'exécution inférieur à τ do
    to_dev ← sélection(racine);
    dev ← développer(to_dev);
    π ← simulation(node);
    rétropropagation(dev, w(π));
    2 − opt(π);
    if w(π) < d_min then
        d_min ← w(π);
        π_min ← π
```

The inefficacy of the current method is illuminated by code profiling, with results showing that the algorithm dedicates 90% of its runtime to 2-optimization, despite its limited utility in Monte Carlo Tree Search (MCTS). Specifically, the 2-optimized tour is not propagated through the tree.

3.2 Prepropagation

It was these findings that led to the introduction of the concept of *prepropagation*. This approach involves injecting the 2-optimized tour into the tree. Starting from the root, the tree is descended by following the cities of the 2-optimized tour, updating the average weight $\overline{w_i}$ and visit count n_i of each encountered node, or creating them if they do not exist. This descent stops upon encountering the need to create a node whose

depth exceeds $prof$, where $prof$ is the depth of the node developed after selection. This process is encapsulated in Algorithm 5.

Algorithm 5: Prepropagation Algorithm

```
Function Prepropagate(R, π, prof):
    Function Auxiliary(node, i):
        w̄_node ← (n_node × w̄_node + w(π)) / (n_node + 1);
        n_node ← n_node + 1;
        if node is not terminal and the child F of node
          representing the choice of city π[i+1] is
          developed then
            Auxiliary(F, i + 1);
        else if i < prof then
            create F;
            Auxiliary(F, i + 1)
    Auxiliary(R, 0)
```

3.3 Simulation Optimization

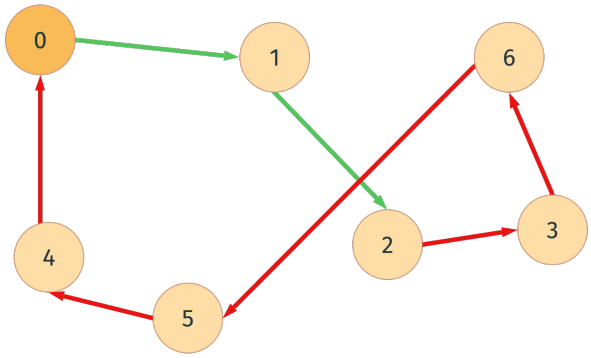
However, this method results in a significantly higher score being backpropagated through simulation as opposed to the one that is prepropagated after 2-optimization. To counterbalance this, the simulation phase is optimized with 2-opt. This is done by preserving the portion of the tour generated by selection, while performing 2-optimization on the remainder. More formally, if the developed node is the p^{th} city of the tour, then during the 2-optimization, only pairs (i, j) with $p < i$ and $i+1 < j$ are permitted. This optimization process is demonstrated in Figure 3.

3.4 Adapting the Selection

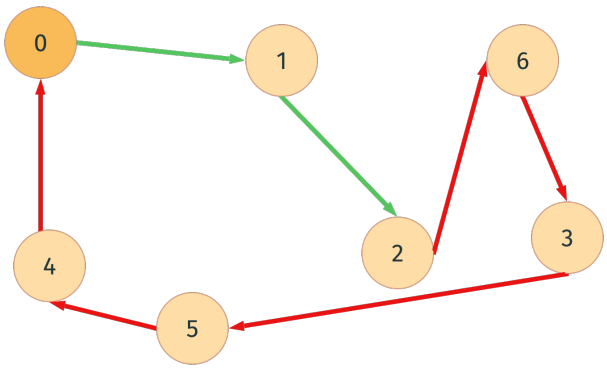
Shimomura and Takashima [2016] employ $\overline{w}$ in their UCT2 formula. However, the objective of the Traveling Salesman Problem (TSP) is not to find a combination of cities yielding low average scores but to identify the single best solution. Replacing $\overline{w}$ with w_{best}, the minimal weight found from a given node, therefore seems prudent. Empirical evidence supports this: tree nodes minimizing $\overline{w}$ are distinct from those minimizing w_{best}. Furthermore, to expedite the tree's deepening, we set the exploration constant C_{exp} to 0.01. It is important to note that altering the algorithm by Shimomura and Takashima [2016] in this way significantly reduces its efficiency.

3.5 Final Algorithm

The modified algorithm, referred to as MCTS 2-opt, is outlined in Algorithm 6. Changes and additions to the classic MCTS approach by Shimomura and Takashima [2016] are highlighted in orange.



(a) In green is the selection, in red is the simulation



(b) The tour after simulation optimization, with the selection portion left intact

FIG. 3: Simulation 2-Optimization

Algorithm 6: MCTS+2-opt Algorithm

```
root ← R_0;
w_min ← ∞;
π_min ← [1, ..., n];
while execution time is less than τ do
    to_develop ← selection(root);
    develop ← expand(to_develop);
    π ← simulation2-opt(develop);
    backpropagation(develop, w(π));
    2-opt(π);
    prepropagate(root, π);
    if w(π) < w_min then
        w_min ← w(π);
        π_min ← π
return π_min
```

4 Experimental Results

4.1 Experimental Conditions

The programming language selected for implementation is OCaml[1]. Tests are conducted on a Linux server powered by an AMD EPYC 7601 2.2 GHz processor with 8 GB of RAM. We gener-

[1]Source code available at: github.com/Butanium/monte-carlo-tree-search-TSP

ate 128 random instances of the TSP, with optimal tours computed using Gurobi Optimization, LLC [2022]'s solver.

4.2 Experimental Procedure

Each algorithm is tested **5** times on each of the **128** instances with $\tau \in \{10\,\text{s}, 30\,\text{s}\}$. The classic MCTS by Shimomura and Takashima [2016] fails to converge swiftly, resulting in subpar solutions. Nonetheless, it significantly outperforms `Greedy Random`, which generates random solutions sequentially and retains the best one. The `MCTS + 2-opt` corresponds to Algorithm 6 without the adjustments discussed in 3.4, while `MCTS select + 2-opt` includes them. These amendments prove crucial in leveraging the quality of 2-optimized tours, achieving a 50% reduction in mean relative error (5) compared to the already impressive `Iterative 2-opt`.

Algorithme	erreur relative moyenne	
	10 s	30 s
MCTS select + 2-opt	_	0.2 %
MCTS + 2-opt	0.67 %	0.41 %
2-opt itéré	0.69 %	0.48 %
2-opt une fois	10 %	_
MCTS classique	120 %	_
Greedy Random	_	430 %

TAB. 1: Résultats expérimentaux pour 100 villes

5 Future Work

Primarily, the C_{exp} value seems to greatly influence algorithmic performance. However, uncertainties in selecting C_p prevent precise determination of the optimal C_{exp}. For future research, I intend to eliminate the need for constant C_p by employing UCT3 (outlined in Appendix A.1) during selection.

I am also interested in exploring different expansion policies, such as developing all children instead of only one, as suggested by Xing and Tu [2020].

The 2-opt algorithm operates on average in $\mathcal{O}(n^3 \log(n))$ time complexity according to Johnson and McGeoch [1997]. Employing a faster heuristic—even at the cost of some efficacy—would allocate more time for MCTS to develop the tree. Thus, I plan to evaluate alternative heuristics in place of 2-opt to assess their outcomes.

Lastly, the sheer size of the search space for TSP ($n!$) hinders MCTS from achieving satisfactory exploration depth. A possible remedy involves reducing the search space size. It would be compelling to investigate algorithms that constrict this search space and to integrate them with MCTS.

References

Zhang-Hua Fu, Kai-Bin Qiu, and Hongyuan Zha. Generalize a small pre-trained model to arbitrarily large TSP instances. *CoRR*, abs/2012.10658, 2020. URL `https://arxiv.org/abs/2012.10658`.

Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2022. URL `https://www.gurobi.com`.

Keld Helsgaun. An extension of the lin-kernighan-helsgaun tsp solver for constrained traveling salesman and vehicle routing problems. *Roskilde: Roskilde University*, pages 24–50, 2017.

David S. Johnson and Lyle A. McGeoch. The traveling salesman problem: A case study in local optimization. In E. H. L. Aarts and J. K. Lenstra, editors, *Local Search in Combinatorial Optimization*, pages 215–310. John Wiley and Sons, Chichester, United Kingdom, 1997.

Gerhard Reinelt. TSPLIB—A Traveling Salesman Problem Library. *INFORMS Journal on Computing*, 3 (4):376–384, November 1991. doi: 10.1287/ijoc.3.4.376. URL `https://ideas.repec.org/a/inm/orijoc/v3y1991i4p376-384.html`.

Masato Shimomura and Yasuhiro Takashima. Application of monte-carlo tree search to traveling-salesman problem. In *The 20th Workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI)*, pages 352–356, 2016.

Zhihao Xing and Shikui Tu. A graph neural network assisted monte carlo tree search approach to traveling salesman problem. *IEEE Access*, 8:108418–108428, 2020. doi: 10.1109/ACCESS.2020.3000236.

Appendices

A Alternative MCTS Approaches for TSP

A.1 Xing and Tu [2020]

The technique proposed by Xing and Tu [2020] builds upon the work of Shimomura and Takashima [2016] through the integration of a neural network. During the selection phase, when considering the child node F of $node$, $\overline{w}$ is replaced with $\hat{Q}$. Let N denote the set of child nodes of $node$:

$$w_{\min} = \min_{j \in N} w_j$$

$$w_{\max} = \max_{j \in N} w_j$$

$$\hat{Q}_F = \frac{w_{\max} - w_F}{w_{\max} - w_{\min}}$$

Consequently, the child node of $node$ with the highest score attains $\hat{Q} = 1$, while the lowest score corresponds to $\hat{Q} = 0$. During selection, the child node F that maximizes (6) is chosen.

$$\text{UCT3} = \hat{Q}_F + C_{exp}\sqrt{\frac{\ln n_{node}}{n_F}} \tag{6}$$

This method eliminates the need for C_p, as $\hat{Q}$ is confined within the range [0,1].

Another distinction from Shimomura and Takashima [2016] is that simulations are performed by a neural network, which is tasked with estimating the length of the best tour that completes the partial path obtained during selection. The authors benchmark their results against other neural network-based methodologies on problem instances with fewer than 100 cities.

A.2 Fu et al. [2020]

Fu et al. [2020] adopt a distinct definition of the TSP game. A game state is characterized by a tour π. An action A is defined as a set of edges ranging from $2 \leqslant k \leqslant 10$, denoted by cities $(a_1, b_1, \ldots, a_k, b_k, a_{k+1})$, where b_i is the successor of a_i in π and $a_{k+1} = a_1$. The action involves substituting edges (a_i, b_i) with (b_i, a_{i+1}), thereby transforming π into π_A. MCTS is utilized to identify productive actions. This equates to a selection of the a_i since b_i is predetermined by a_i, and the action is defined if $a_i = a_1$. The authors make their choice of a_{i+1} solely based on b_i. As a result, they do not build a tree but use an $n \times n$ weight matrix instead. Here, $W_{i,j}$ indicates the likelihood of selecting v_j following v_i. Initialized by a neural network, this matrix is meant to favor the paired indices (i, j) most likely to be part of the optimal path. Any pairs initialized with an excessively low value are permanently discarded and never considered, significantly narrowing the search space. To select a_{i+1} during the selection process, Fu et al. [2020] aim to maximize UCT, replacing $\overline{w_j}$ with $Q_{b_i,j}$ and n_i with the number of actions performed during MCTS.

$$Q_{b_i,j} = \frac{W_{b_i,j}}{\Omega_{b_i}}$$

$$\Omega_{b_i} = \frac{\sum_{k \neq b_i} W_{b_i,k}}{\sum_{k \neq b_i} 1}$$

Once MCTS identifies an action A that yields $w(\pi_A) < w(\pi)$, the action is executed, and the MCTS process is resumed to seek a new action that could further optimize π_A. If no such action is found to be satisfactory, π is replaced by a randomly generated and 2-optimized tour π'. Subsequently, the MCTS process is restarted.

The authors compare their results on configurations ranging from 50 to 10,000 cities with those of other methods that leverage neural networks and the LKH heuristic by Helsgaun [2017]. Their method significantly outperforms other neural network-based methodologies, though it falls short of exceeding the performance of LKH.