Optimisation locale dans une recherche arborescente de Monte-Carlo pour le problème du voyageur de commerce

Clément DUMAS

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

Résumé – Dans ce TIPE, j'implémente une recherche arborescente de Monte-Carlo pour le problème du voyageur de commerce. Je pars de l'implémentation présentée par Shimomura et Takashima (2016) et j'y ajoute de l'optimisation locale avec l'algorithme de 2-opt. Ma méthode surpasse largement celle de Shimomura et Takashima (2016) et semble améliorer le 2-opt itéré.

1 Prérequis

1.1 Problème du voyageur de commerce

On considère n villes $v_1, ..., v_n$ dans le plan. Soit G le graphe complet dont les sommets sont les villes et dont les arêtes sont pondérées par la partie entière de la distance euclidienne entre deux villes (1).

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

On définit un tour $\pi = (a_1, ..., a_n)$ comme un circuit passant exactement une fois par chaque sommet. On définit son poids w par :

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

Le problème du voyageur de commerce (Traveling-Salesman Problem ou TSP) pour n villes consiste à trouver le tour π_m de G minimisant w.

1.2 Algorithme de 2-opt

Le 2-opt est un algorithme d'optimisation locale pour le TSP. On part d'un tour π, puis on cherche 2 arêtes $(a_i,\ a_{i+1})$ et $(a_j,\ a_{j+1})$ avec $i + 1 < j$ qui, une fois supprimées et remplacées par $(a_i,\ a_j)$ et $(a_{i+1},\ a_{j+1})$, font baisser le poids du tour. Plus précisément, après une itération de 2-opt :

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

est transformé en

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

si $w(\pi') < w(\pi)$.
Ce processus est schématisé dans la figure 1. Une fois qu'on ne peut plus trouver de tels (i, j), l'algorithme s'arrête et on dit que le tour est *2-optimisé*. Un exemple d'utilisation de l'algorithme est illustré dans la figure 2

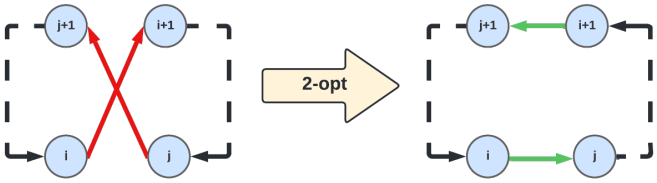
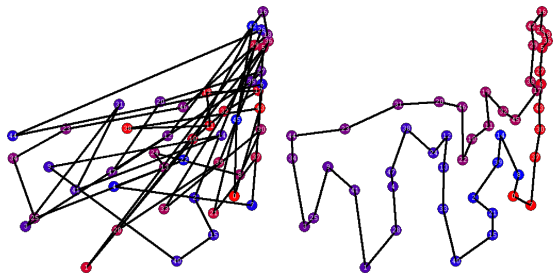


FIGURE 1 – Une modification après une itération de 2-opt. Les lignes en pointillé symbolisent un chemin reliant 2 villes



(a) Tour aléatoire (b) Tour après 2-optimisation

FIGURE 2 – 2-optimisation d'un tour aléatoire

Les résultats dans le Tableau 3, donné par Johnson et McGeoch (1997) (p. 21), montrent empiriquement qu'un tour 2-optimisé aura en moyenne moins de 5% d'erreur par rapport à la solution exacte. On définit alors l'heuristique *2-opt itéré* dans l'algorithme 1.

1.3 Recherche arborescente de Monte-Carlo

On considère un agent jouant à un jeu. Celui-ci est initialement dans un état E_0. Il doit choisir une action a_{0,i_0} pour $0 \leqslant i_0 \leqslant n_0$ qui le placera dans un état E_1, puis des actions $a_{1,i_1}, \ldots, a_{n-1,i_{n-1}}$ qui le placeront dans un état final E_n. On évalue alors l'agent en attribuant un score w à E_n.

Algorithme 1 : 2-opt itéré

```
meilleur_tour ← [1, ..., n];
w_min = ∞;
tant que temps d'exécution inférieur à τ faire
    π ← tour aléatoire;
    2-opt(tour);
    si w(tour) < w_min alors
        copier π dans meilleur_tour;
        w_min ← w(π)
retourner meilleur_tour
```

Le but de l'agent est de trouver la suite d'action maximisant w. Par la suite, on notera E_i l'état de jeu et le nœud qui lui est associé dans l'arbre.

La recherche arborescente de Monte-Carlo (Monte-Carlo Tree Search ou MCTS) est un algorithme proposant une approche probabiliste de ce problème. On construit itérativement un arbre de recherche dont les nœuds sont les différents états possibles. Les fils d'un nœud E_i sont donc les états atteignables en effectuant une action à partir de E_i. On stocke dans chaque nœud le nombre de visites ainsi que le score moyen $\overline{w}$ obtenu en passant par ce nœud.

L'algorithme 2 commence avec l'arbre composé du nœud de l'état initial R_0, sans fils. Ensuite, on lance une boucle que l'on termine au bout d'un temps τ. À chaque itération :

1. On détermine la partie de l'arbre la plus prometteuse à développer lors de la *sélection*. Celle-ci renvoie un nœud E_i à développer
2. On crée un nouveau fils E_{i+1} au nœud E_i avec la fonction `développer`
3. On termine la partie aléatoirement à partir de l'état E_{i+1} selon la politique de *simulation*, jusqu'à atteindre un état final E_f.
4. On inclut $w(E_f)$ dans le score moyen de E_{i+1} et tous ses ancêtres lors de la *rétropropagation*

Tout au long de l'algorithme, on garde en mémoire la suite d'actions donnant le meilleur score.

Algorithme 2 : Algorithme de MCTS

```
racine ← R_0;
w_max ← 0;
meilleures_actions ← liste vide;
tant que temps d'exécution inférieur à τ faire
    node ← sélection(racine);
    dev ← développer(node);
    w, actions ← simulation(dev);
    si w > w_max alors
        w_max ← w;
        meilleures_actions ← actions
    rétropropagation(dev, w)
retourner meilleures_actions
```

1.3.1 Sélection

La sélection consiste à parcourir l'arbre jusqu'à trouver un nœud terminal ou un nœud avec une action non essayée donc un fils non créé. Pour cela, on sélectionne récursivement le fils E_{i+1} de E_i qui maximise (2) suivant l'algorithme 3.

Algorithme 3 : Fonction de sélection

```
Argument : la racine de l'arbre R
Renvoie  : le noeud à développer
node ← R;
tant que node n'est pas final et node a tous ses fils développés faire
    node ← fils de node maximisant UCT
retourner node
```

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

Où $\overline{w_{i+1}}$ est le score moyen de E_{i+1} et les n_j sont le nombre de fois que le $j^{ème}$ nœud a été visité lors de la sélection. C_{exp} est la constante d'exploration et C_p une constante proportionnelle à la taille du problème. Si les scores w obtenus sont dans $[0, 1]$, alors $C_p = 1$. C_{exp} vaut $\sqrt{2}$ par défaut, mais peut être ajustée empiriquement pour privilégier l'exploration ou l'exploitation.

En effet, cette formule est divisée en 2 termes :

1. Le terme vert est le terme d'*exploitation*. Il témoigne de la qualité des solutions atteignables à partir de ce nœud
2. Le rouge est le terme d'*exploration*. Il privilégie les fils qui ont été délaissés lors des sélections précédentes, incitant ainsi l'algorithme à ne pas abandonner des pistes qui semblent aux premiers abords peu prometteuses.

Si on atteint un nœud terminal, on effectue la rétropropagation avec le score associé à l'état final. Sinon, on agrandit l'arbre en créant l'un des fils manquant du nœud final et on lance le *simulation* à partir de celui-ci.

1.3.2 Simulation

La simulation à partir d'un nœud consiste à effectuer, en partant de l'état du nœud, une suite d'actions aléatoires jusqu'à arriver à un état final. Le score de cet état final est ensuite utilisé dans la rétropropagation. À noter que tous les états parcourus lors de la simulation ne sont pas stockés dans l'arbre afin de ne pas saturer la mémoire.

1.3.3 Rétropropagation

La rétropropagation d'une valeur w à partir d'un nœud E_i consiste à inclure w dans le score moyen $\overline{w_i}$ de E_i et à incrémenter son n_i, puis à faire de même pour tous ses ancêtres. Plus

précisément, pour $node \in \{E_i \text{ et ses ancêtres}\}$ on effectue :

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

2 MCTS appliquée au TSP

2.1 Première approche

Comme première implémentation d'une MCTS pour le TSP je me suis basé sur le travail de Shimomura et Takashima (2016).

2.1.1 Formulation du TSP adapté à une MCTS

On considère une instance du TSP contenant n villes. On définit un état de jeu comme étant un chemin partiel contenant $p \leqslant n$ villes distinctes. Notre agent commence avec le chemin ne contenant que la première ville v_1. À chaque action, l'agent choisit une ville qu'il n'a pas encore visitée. Au bout de $n-1$ actions, l'agent a visité toutes les villes et on complète son chemin en un tour π en reliant la dernière ville visitée à v_1. Le score de l'agent est alors la $w(\pi)$.

2.1.2 Politique de sélection adaptée

Ce problème est donc un problème de minimisation plutôt que de maximisation, il faut donc le prendre en compte dans notre politique de sélection. Ainsi, lors de la sélection, plutôt que de choisir le fils qui maximise (2), on va plutôt choisir celui qui minimise (3).

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

Il n'y a pas de méthode générale pour définir C_p, ainsi Shimomura et Takashima (2016) proposent d'utiliser le double du poids de l'arbre couvrant minimal de G ou le double de l'écart type des n premières valeurs trouvées pour w. Cette deuxième méthode est possible, car UCT2 n'est utilisée qu'une fois tous les fils de la racine développés.

2.1.3 Politique de simulation adaptée

La simulation consiste à compléter aléatoirement le chemin partiel avec les villes restantes. On distingue 2 manières de sélectionner aléatoirement les villes. La première est de le faire de manière complètement aléatoire, chaque ville ayant la même probabilité d'être choisie. La deuxième est de choisir successivement v' après avoir choisi v_d avec la probabilité (4).

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

Cette méthode est appelée *roulette* par Shimomura et Takashima (2016).

2.2 Travaux connexes

L'approche décrite dans 2.1 est la première implémentation d'une MCTS pour le TSP. Cependant, cette méthode n'a pas de très bons résultats. Plusieurs tests sur la configuration $att48$ de TSPLIB (Reinelt (1991)) montrent qu'après 30 minutes, on obtient un tour π avec 20% d'erreur relative (5) qui n'est pas 2-optimisé.

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

Depuis, il y a eu deux publications alliant MCTS et TSP (Fu *et al.* (2020); Xing et Tu (2020)). Les méthodes utilisées diffèrent beaucoup de la méthode de Shimomura et Takashima (2016), notamment car elles utilisent des réseaux de neurones. Mes travaux visant à améliorer la méthode de Shimomura et Takashima (2016), le fonctionnement de ces deux méthodes est décrit dans l'appendice A.

3 Utilisation du 2-opt lors de la MCTS

Le fait que les solutions renvoyées par la méthode de Shimomura et Takashima (2016) ne soient pas 2-optimisées me laissa penser que le 2-opt pourrait grandement améliorer la MCTS. J'ai donc développé 3 moyens d'inclure du 2-opt dans la MCTS.

3.1 Optimisation cachée

L'optimisation cachée consiste à 2-optimiser le tour obtenu lors de la simulation après la rétropropagation. Cela permet d'être sûr que le chemin renvoyé par la MCTS sera 2-optimisé. La boucle dans l'algorithme de MCTS devient donc :

Algorithme 4 : MCTS avec optimisation cachée : la 2-optimisation est insérée à la ligne 6

```
1 tant que temps d'exécution inférieur à τ faire
2     to_dev ← sélection(racine);
3     dev ← développer(to_dev);
4     π ← simulation(node);
5     rétropropagation(dev, w(π));
6     2 − opt(π);
7     si w(π) < d_min alors
8         d_min ← w(π);
9         π_min ← π
```

Le problème de cette méthode est révélé par un profilage du code : 90% du temps de l'algorithme est consacré à la 2-optimisation alors que celle-ci ne sert pas vraiment dans la MCTS. En effet, le tour 2-optimisé n'est pas propagé dans l'arbre.

3.2 Prépropagation

C'est ainsi que j'ai introduit le concept de *prépropagation*. Le principe est d'injecter le tour 2-optimisé dans l'arbre. Pour

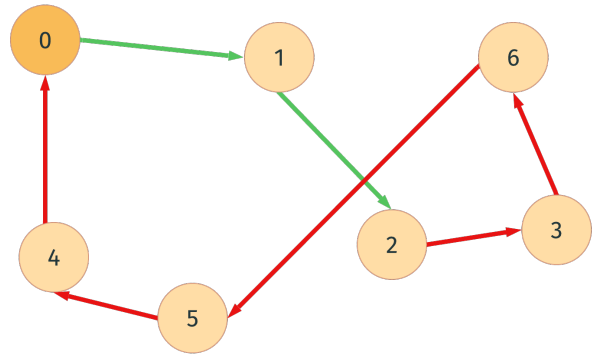
cela on part de la racine puis on descend l'arbre en suivant les villes du tour 2-optimisé. On actualise le $\overline{w_i}$ et le n_i de chaque nœud rencontré, ou on les crée s'ils n'existent pas. On s'arrête lorsqu'on tombe sur un nœud à créer dont la profondeur est supérieure à $prof$, $prof$ étant la profondeur du nœud développé après la sélection. Ce processus est décrit dans l'algorithme 5.

Algorithme 5 : Algorithme de prépropagation

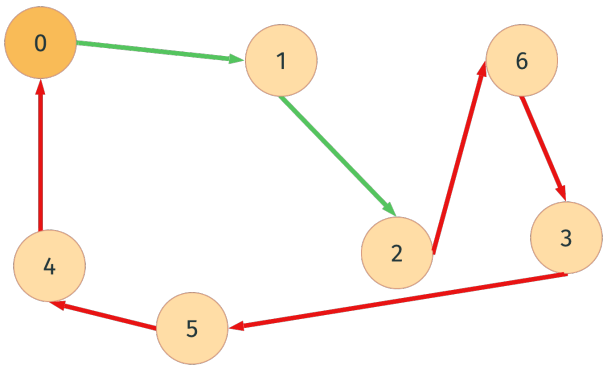
```
1  Fonction Prépropagation(R, π, prof) :
2      Fonction Aux(node, i) :
3          w̄_node ← (n_node × w̄_node + w(π)) / (n_node + 1);
4          n_node ← n_node + 1;
5          si node n'est pas final et le fils F de node
             représentant le choix de la ville π[i+1] est
             développé alors
6              Aux(F, i + 1);
7          sinon si i < prof alors
8              créer F;
9              Aux(F, i + 1)
10     Aux(R, 0)
```

3.3 Optimisation de la simulation

Cependant, cette méthode rend le score rétropropagé par la simulation bien plus grand que celui prépropagé après 2-optimisation. Pour compenser cela, on optimise la simulation avec du 2-opt. Pour cela, on garde intacte la partie du tour générée par la sélection, mais on 2 optimise le reste. Plus formellement, si le nœud développé est la $p_{\text{ème}}$ ville du tour, alors, lors de la 2-optimisation, on n'autorise que des couples i, j avec $p < i$ et $i + 1 < j$. Ce processus est illustré dans la figure 3.



(a) En vert la sélection, en rouge la simulation



(b) Tour après optimisation de la simulation, la partie sélection est laissée intacte

FIGURE 3 – 2-optimisation de la simulation

3.4 Sélection adaptée

Shimomura et Takashima (2016) utilisent $\overline{w}$ dans UCT2. Cependant, le problème du voyageur de commerce n'a pas pour but de trouver une combinaison de villes menant à de faibles scores en moyenne, mais bien à trouver **la** meilleure solution. Il paraît donc judicieux de remplacer $\overline{w}$ par w_{best}, le poids minimal trouvé à partir d'un nœud donné. Cela est confirmé empiriquement : les nœuds de l'arbre minimisant $\overline{w}$ et ceux minimisant w_{best} ne sont pas les mêmes. De plus, pour accélérer le développement en profondeur de l'arbre, on choisit $C_{exp} = 0.01$. À noter que modifier l'algorithme de Shimomura et Takashima (2016) de la sorte le rend beaucoup moins efficace.

3.5 Algorithme final

Finalement, on obtient l'algorithme 6, appelé MCTS 2-opt. Les modifications et ajouts apportés à la `MCTS classique` de Shimomura et Takashima (2016) sont reportés en orange.

Algorithme 6 : Algorithme de MCTS+2-opt

```
1  racine ← R_0;
2  w_min ← ∞;
3  π_min ← [1, ..., n];
4  tant que temps d'exécution inférieur à τ faire
5      to_dev ← sélection(racine);
6      dev ← développer(to_dev);
7      π ← simulation2-opt(dev);
8      rétropropagation(dev, w(π));
9      2-opt(π);
10     prépropagation(racine, π);
11     si w(π) < w_min alors
12         w_min ← w(π);
13         π_min ← π
14 retourner π_min
```

4 Résultats expérimentaux

4.1 Condition de l'expérience

Le langage de programmation choisi est OCaml [1]. Les expériences sont réalisées sur un serveur Linux avec un processeur AMD EPYC 7601 2.2 GHz et 8 Go de RAM. On génère

1. Le code source est disponible à l'adresse : github.com/Butanium/monte-carlo-tree-search-TSP

128 instances aléatoires du TSP. Les tours optimaux de ces instances sont calculés avec le solver de Gurobi Optimization, LLC (2022).

4.2 Déroulement de l'expérience

Chaque algorithme est testé **5** fois sur chacune des **128** instances avec $\tau \in \{10\text{ s}, 30\text{ s}\}$. La `MCTS classique` de Shimomura et Takashima (2016) n'a pas le temps de converger et renvoie donc des solutions très peu satisfaisantes. Néanmoins, elle surpasse largement `Greedy Random` qui consiste à générer des solutions aléatoires à la suite et à renvoyer la meilleure. La méthode `MCTS + 2 opt` correspond à l'Algorithme 6 sans les modifications de 3.4 tandis que `MCTS select + 2-opt` les inclut. Ces dernières modifications se révèlent essentielles pour tirer parti de la qualité des tours 2-optimisés. Cela permet de surpasser le `2-opt itéré` en divisant par 2 son erreur relative (5) moyenne déjà très basse

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 %

TABLE 1 – Résultats expérimentaux pour 100 villes

5 Travaux futurs

Tout d'abord, C_{exp} semble beaucoup influencer les performances de l'algorithme. Cependant, l'incertitude introduite dans le choix de C_p ne permet de déterminer avec exactitude le C_{exp} optimal. Pour de futurs travaux, j'aimerais me passer de la constante C_p en utilisant UCT3 (définie dans l'Appendice A.1) lors la sélection.

J'aimerais aussi tester d'autres politiques d'expansion, en développant tous les fils au lien d'un seul, comme le suggèrent Xing et Tu (2020).

Le 2-opt est en moyenne en $\mathcal{O}(n^3 \log(n))$ d'après Johnson et McGeoch (1997). L'utilisation d'une heuristique plus rapide (même si elle était moins efficace) laisserait plus de temps à la MCTS pour développer l'arbre. Je compte donc essayer de remplacer le 2-opt par d'autres heuristiques afin de comparer les résultats.

Enfin, le principal obstacle pour développer une MCTS pour le TSP est la taille de l'espace de recherche ($n!$). Cela empêche la MCTS d'atteindre une profondeur d'arbre satisfaisante. Pour pallier cela, on peut réduire la taille de cet espace. Il pourrait donc être intéressant d'étudier des algorithmes permettant de restreindre cet espace de recherche et de les combiner avec la MCTS.

Références

Zhang-Hua FU, Kai-Bin QIU et 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 et Lyle A. MCGEOCH : The traveling salesman problem : A case study in local optimization. *In* E. H. L. AARTS et J. K. LENSTRA, éditeurs : *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. URL `https://ideas.repec.org/a/inm/orijoc/v3y1991i4p376-384.html`.

Masato SHIMOMURA et 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 et Shikui TU : A graph neural network assisted monte carlo tree search approach to traveling salesman problem. *IEEE Access*, 8:108418–108428, 2020.

Appendices

A Autres MCTS pour le TSP

A.1 Xing et Tu (2020)

L'approche de Xing et Tu (2020) s'inspire de Shimomura et Takashima (2016) mais utilise un réseau de neurones. Lorsqu'on considère le fils F de $node$ lors de la sélection, $\overline{w}$ est remplacé par $\hat{Q}$. En notant N l'ensemble des fils de $node$:

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

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

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

Ainsi, le fils de $node$ avec le meilleur score a $\hat{Q} = 1$ et celui avec le pire score $\hat{Q} = 0$. Lors de la sélection, on choisit donc le fils F de $node$ maximisant (6).

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

Cette méthode permet de supprimer C_p car $\hat{Q} \in [0, 1]$.

Une autre différence avec Shimomura et Takashima (2016) est que la simulation est effectuée par un réseau de neurones. Celui-ci estime la longueur du meilleur tour complétant le chemin partiel obtenu lors de la sélection. Les auteurs comparent leur résultat avec d'autres méthodes utilisant des réseaux de neurones sur des instances comportant moins de 100 villes.

A.2 La méthode de Fu *et al.* (2020)

Fu *et al.* (2020), définissent leur jeu du TSP de manière différente. Un état de jeu est un tour π. Une action A est un ensemble de $2 \leqslant k \leqslant 10$ arêtes représentées par les villes $(a_1, b_1, ..., a_k, b_k, a_{k+1})$ avec b_i le successeur de a_i dans π et $a_{k+1} = a_1$. L'action consiste à remplacer les arêtes (a_i, b_i) par les (b_i, a_{i+1}) transformant π en π_A. La MCTS est utilisée pour chercher des actions. Cela revient à choisir les a_i car b_i est déterminé par a_i, de plus si on choisit $a_i = a_1$ l'action est définie. Les auteurs choisissent a_{i+1} uniquement en fonction de b_i. Ils ne construisent donc pas d'arbre, mais utilisent une matrice de poids de taille $n \times n$. $W_{i,j}$ correspond à la probabilité de choisir v_j après v_i. Cette matrice est initialisée par un réseau neuronal censé privilégier les couples (i, j) ayant le plus de chance de se trouver dans le chemin optimal. Les (i, j) initialisés avec une valeur trop petite sont définitivement exclus et ne seront jamais considérés, ce qui permet de restreindre grandement l'espace de recherche. Pour choisir a_{i+1} lors de la sélection, Fu *et al.* (2020) cherchent à maximiser UCT mais remplacent $\overline{w_j}$ par $Q_{b_i,j}$ et le n_i par le nombre d'actions effectuées lors de la MCTS.

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

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

Une fois que la MCTS a trouvé une action A telle que $w(\pi_A) < w(\pi)$, elle l'effectue et cherche une nouvelle action pour optimiser π_A. Si aucune action satisfaisante n'est trouvée, alors on remplace π par un tour généré aléatoirement puis 2-optimisé π'. Ensuite, on reprend la MCTS.

Les auteurs comparent leurs résultats sur des configurations allant de 50 à 10000 villes avec ceux d'autres méthodes utilisant des réseaux de neurones et à l'heuristique LKH de Helsgaun (2017). Leur méthode surpasse largement les autres méthodes utilisant des réseaux de neurones, mais pas LKH.