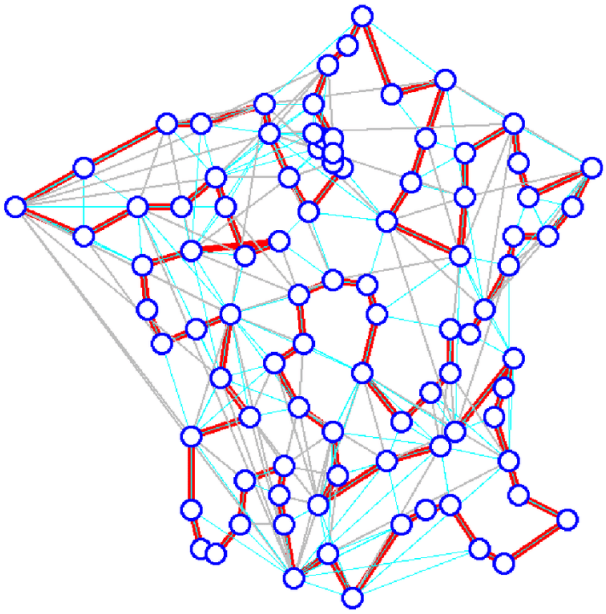
Projet semestriel
Monte Carlo Tree Search pour le problème du voyageur de commerce probabiliste

Yuchen MO, Hengshuo LI, Peiyao LI
06 Juin 2023



Introduction

Le problème du voyageur de commerce(TSP) est un problème classique d'optimisation combinatoire qui consiste à trouver le chemin le plus court entre un ensemble donné de villes, de sorte que le voyageur puisse visiter chaque ville une fois et revenir à la ville de départ. Dans le cas probabiliste du voyageur de commerce, en revanche, chaque ville possède une probabilité d'annulation, simulant une situation où un client annule une commande et où le voyageur n'a pas besoin de passer par cette ville annulée. Cela ajoute à la complexité et à la difficulté du problème.

Pour résoudre le problème du voyageur de commerce probabiliste, nous utiliserons l'algorithme de recherche arborescente de Monte Carlo(MCTS). L'algorithme de recherche arborescente de Monte Carlo est une méthode de recherche heuristique qui guide le processus de recherche par le biais de la simulation et du hasard afin de trouver une solution de haute qualité. Nous essayons également de résoudre le problème en utilisant k-opt combiné à l'algorithme de recherche arborescente de Monte Carlo. Nous proposons également une idée pour résoudre le problème en combinant la méthode de Branch and bound.

Dans ce projet, nous commencerons par modéliser le problème du voyageur de commerce et le problème du voyageur de commerce probabiliste, en construisant différents modèles pour différentes méthodes. Nous détaillerons ensuite les trois méthodes que nous avons étudiées pour résoudre le problème du voyageur de commerce probabiliste à l'aide de l'algorithme de recherche arborescente de Monte Carlo. L'implémentation finale est réalisée en C++ et testée à l'aide des instances de TSPLib.

Table des matières

1 Modélisation mathématique du problème du voyageur de commerce

1.1 Introduction au problème du voyageur de commerce

Le problème du voyageur de commerce, est un problème d'optimisation qui consiste à déterminer, étant donné une liste de villes et les distances entre toutes les paires de villes, le plus court chemin qui passe par chaque ville une et une seule fois. On ne connaît pas d'algorithme permettant de trouver une solution exacte rapidement dans tous les cas car c'est un problème NP-difficile.

L'énoncé du problème du voyageur de commerce est le suivant : étant donné n points (des « villes ») et les distances séparant chaque point, trouver un chemin de longueur totale minimale qui passe exactement une fois par chaque point et revienne au point de départ. Formellement, une instance est un graphe complet $G(V, A, \omega)$, avec V un ensemble de sommets, A un ensemble d'arêtes et ω une fonction de coût sur les arcs. Le problème est de trouver le plus court cycle hamiltonien dans le graphe G.[1]

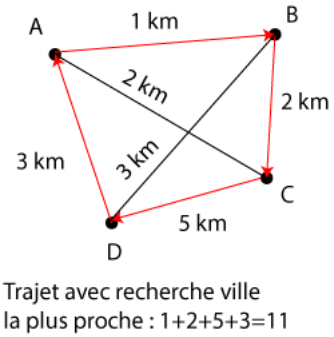


FIGURE 1 – Problème du voyageur de commerce

1.2 Modélisation mathématique du problème(Version 1)

On considère n villes $(\pi_1, ..., \pi_n)$ et on considère chaque solution du problème TSP comme une permutation de villes i.e. $\Pi = (\pi_2, \pi_1, ..., \pi_n)$, c'est-à-dire que on part de Π_1, suivons l'ordre des indices de Π, jusqu'à Π_n, et revenons enfin de Π_n à Π_1. On a aussi $l(\pi_i, \pi_j)$ est la distance entre la ville π_i et la ville π_j $i, j \in \{1, ..., n\}$. Donc, le problème est

$$min_{\Pi} \sum_{i=1}^{n-1} l(\Pi_i, \Pi_{i+1}) + l(\Pi_n, \Pi_1)$$

1.3 Modélisation mathématique du problème(Version 2)

Pour la forme de programmation en nombres entiers du problème du voyageur commerce probabiliste, nous voulons minimiser l'espérance du chemin en tenant compte des probabilités. Nous avons besoin de plusieurs contraintes. Tout d'abord, il ne peut y avoir qu'une seule arête par ville. Ensuite, il ne peut y avoir qu'une seule arête vers chaque ville. Ensuite, aucune sous-boucle ne peut être formée. Enfin, considérons le cas de l'annulation d'une ville. Nous pouvons obtenir la forme de planification en nombres entiers suivante.[2]

$$
\begin{cases}
\min \frac{1}{K} \sum_{1 \leq k \leq K} \sum_{ij \in A} l_{ij} x_{ij}^k \\
\text{Subject to} \\
\sum_{j \neq i} y_{ij} = 1 \qquad \forall i \in \{1, ..., n\} & (1) \\
\sum_{i \neq j} y_{ij} = 1 \qquad \forall j \in \{1, ..., n\} & (2) \\
\sum_{j \neq j} x_{ij}^k = 1 \qquad \forall i \in V_k \qquad \forall k \in \{1, ..., K\} & (3) \\
\sum_{i \neq j} x_{ij}^k = 1 \qquad \forall j \in V_k \qquad \forall k \in \{1, ..., K\} & (4) \\
\sum_{i,j \in S} y_{ij} \leq |S| - 1 \qquad \forall S \in \{1, ..., n\}, 2 \leq |S| \leq n - 2 & (5) \\
\sum_{i,j \in S} x_{ij}^k \leq |S| - 1 \qquad \forall S \in V_k, 2 \leq |S| \leq n - 2 \qquad \forall k \in \{1, ..., K\} & (6) \\
x_{ij}^k \geq y_{ij} \qquad \forall ij \in A_k \qquad \forall k \in \{1, ..., K\} & (7) \\
y_{ij} \in \{0, 1\} \qquad \forall ij \in A & (8) \\
x_{ij}^k \in \{0, 1\} \qquad \forall ij \in A_k \qquad \forall k \in \{1, ..., K\} & (9)
\end{cases}
$$

FIGURE 2 – Modélisation 2

1.4 Façon de résoudre le problème (Version 1)

On choisit d'abord un ordre de permutation arbitrairement Π. Chaque action A est une transformation qui convertit un état donné Π en un nouvel état Π*. Étant donné que chaque solution TSP consiste en un sous-ensemble de n arêtes, chaque action peut être considérée comme une transformation $k - opt$ $(2 \leq k \leq n)$, qui supprime d'abord k arêtes, puis ajoute k arêtes différentes pour former un nouveau tour. Nous laissons les k choix d'ajout dépendre des k choix de suppression, comme indiqué ci-dessous.

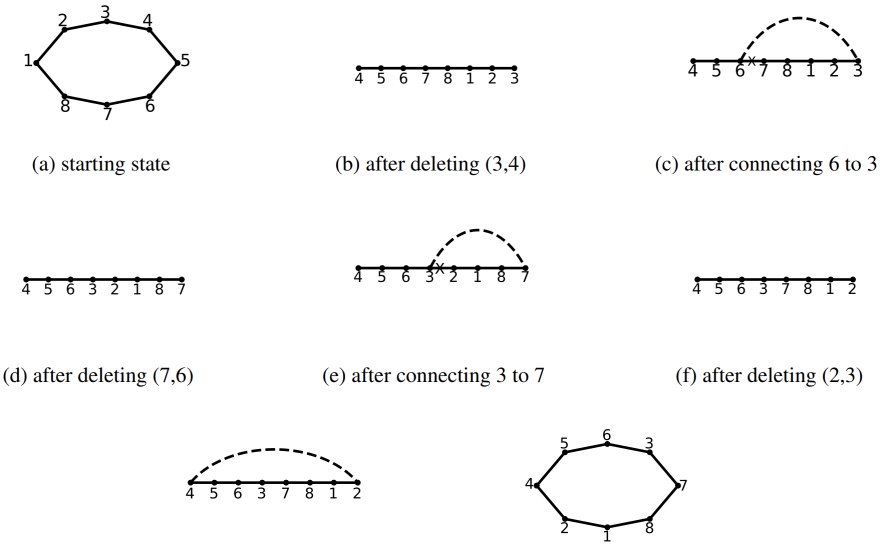


FIGURE 3 – Représentation de k-opt méthode

Sur la figure 2, la sous-figure (a) est l'état de départ $\Pi = (1, 2, 3, 4, 5, 6, 7, 8)$. Pour déterminer une action, nous décidons d'abord d'une ville a_1 et supprimons le bord entre a_1 et sa ville précédente b_1. Sans perte de généralité, supposons $a_1 = 4$, puis $b_1 = 3$ et l'arête $(3, 4)$ est supprimée, ce qui entraîne un statut temporaire illustré dans la sous-figure (b). De plus, nous décidons qu'une ville a_2 connectera la ville b_1, résultant généralement en une solution irréalisable contenant un cycle interne (sauf si $a_2 = a_1$). Par exemple, supposons $a_2 = 6$ et connectez-le à la ville 3, le statut temporaire résultant est illustré dans la sous-figure (c), où un cycle interne se produit et le degré de la ville a_2 augmente à 3. Pour rompre le cycle interne et réduire le degré de a_2 à 2, le bord entre la ville a_2 et la ville $b_2 = 7$ doit être supprimé, ce qui entraîne un statut temporaire illustré dans la sous-figure (d). Ce processus est répété, pour obtenir une série de villes a_k et b_k $(k \geqslant 2)$. Dans cet exemple, $a_3 = 3$ et $b_3 = 2$, correspondant respectivement aux sous-figures (e) et (f). Une fois $a_k = a_1$, la boucle se ferme et atteint un nouvel état (solution TSP réalisable). Par exemple, si $a_4 = a_1 = 4$ et connectez a_4 à b_3, le nouvel état résultant est montré dans la sous-figure (g), qui est redessinée comme un cycle dans la sous-figure (h).

Formellement, une action peut être représentée par $A = (a_1, b_1, a_2, b_2, ..., a_k, b_k, a_{k+1})$, où k est une variable et la ville de début doit coïncider avec la ville finale, i.e. $a_{k+1} = a_1$. Chaque action correspond à une transformation $k - opt$, qui supprime k arêtes, c'est-à-dire $(a_i, b_i), 1 \leqslant i \leqslant k$, et ajoute k arêtes, c'est-à-dire $(b_i, a_{i+1}), 1 \leqslant i \leqslant k$, pour atteindre un nouvel état. Notez que tous ces éléments ne sont pas facultatifs, puisqu'une fois a_i connu, b_i peut être déterminé de manière unique sans aucun choix facultatif. Par conséquent, pour déterminer une action, nous ne devons décider qu'une série de k sous-décisions, c'est-à-dire les k villes $a_i, 1 \leqslant i \leqslant k$.

Soit $L(\Pi)$ la longueur du tour correspondant à l'état Π, correspondant alors à chaque action $A = (a_1, b_1, a_2, b_2, ..., a_k, b_k, a_{k+1})$ qui transforme Π en un nouvel état Π^*, la différence $\triangle(\Pi, \Pi^*) = L(\Pi) - L(\Pi^*)$ pourrait être calculé comme suit :

$$\triangle(\Pi, \Pi^*) = \sum_{i=1}^{k} l(b_i, a_{i+1}) - \sum_{i=1}^{k} l(a_i, b_i)$$

Si $\triangle(\Pi, \Pi^*) < 0$, Π^* est meilleur (avec une longueur de tour plus courte) que Π.

Par conséquent, nous pouvons choisir une permutation de ville comme départ, utiliser la méthode ci-dessus pour obtenir en permanence de meilleures permutations et enfin obtenir une solution.[3]

1.5 Façon de résoudre le problème (Version 2)

Ici, nous considérons la deuxième approche de modélisation. Nous utilisons la méthode Branch and bound pour le résoudre. On choisit aléatoirment le root ville i, après on choisit un chemin départ de i aléatoirement. Si nous pouvons obtenir une solution entière, alors nous utiliserons la distance de chemin comme coût, et le minimum du coût obtenu sera utilisé comme limite, et sa solution sera utilisée comme solution temporaire. Le node qu'on a la solution entière n'a pas besoin de continuer à se développer. Pour le cas où la solution entière n'est pas obtenue, alors nous gardons le nœud et sélectionnons au hasard un chemin à développer. Si une solution entière peut être obtenue, comparez le chemin avec la limite. Si le chemin est inférieur à la limite, il sera être utilisée comme nouvelle limite, sa solution comme solution temporaire. Continuez à développer vers le bas jusqu'à ce que toutes les solutions réalisables soient trouvées. Enfin, une solution optimale locale est obtenue. En utilisant cette méthode, nous pouvons obtenir une situation optimale, puis nous sélectionnons à nouveau aléatoirement la ville de départ, et en utilisant la même méthode, nous pouvons obtenir une nouvelle solution optimale locale. Nous comparons les deux distances et choisissons la plus petite comme solution optimale temporaire. A ce moment, la probabilité de sélectionner au hasard le chemin est la même.[4]

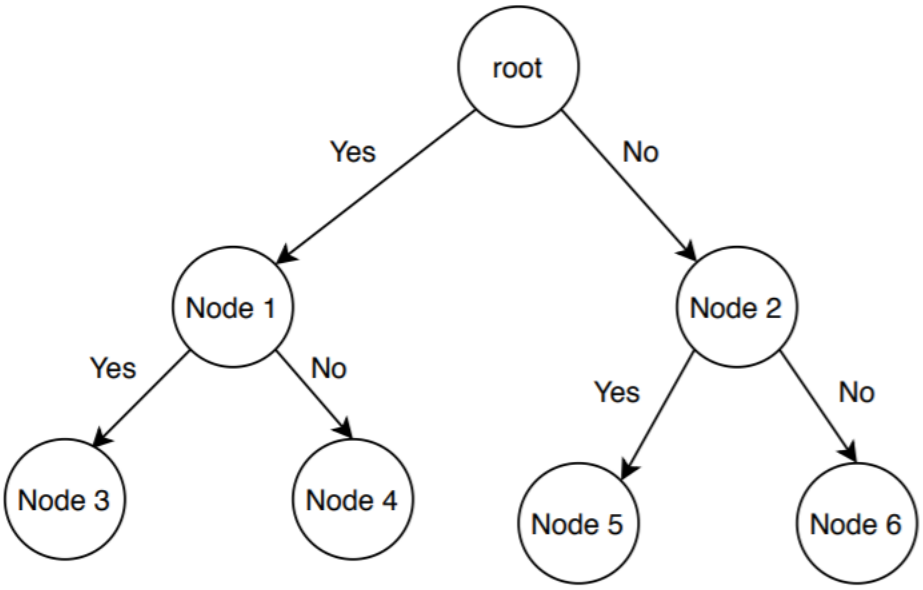


FIGURE 4 – Représentation de branch and bond méthode

2 Algorithme de recherche arborescente de Monte Carlo

2.1 Introduction de l'algorithme

La méthode de recherche arborescente de Monte Carlo peut véritablement simuler le processus physique réel, de sorte que la solution au problème est très cohérente avec la réalité et qu'un résultat très satisfaisant peut être obtenu. Par exemple, la simulation du processus de prise de décision dans les jeux stratégiques tels que le Go[5], la modélisation de la dynamique des fluides[6]. C'est aussi une méthode de calcul basée sur les méthodes théoriques des probabilités et des statistiques, et c'est une méthode utilisant des nombres aléatoires pour résoudre de nombreux problèmes de calcul. Reliez le problème à résoudre à un certain modèle de probabilité et utilisez un ordinateur pour réaliser une simulation statistique ou un échantillonnage afin d'obtenir une solution approximative au problème.

Lorsque le problème à résoudre est la probabilité d'un certain événement, ou la valeur attendue d'une variable aléatoire, ils peuvent obtenir la fréquence d'un tel événement ou la valeur moyenne de cette variable aléatoire par une certaine méthode "expérimentale", et utiliser eux comme solutions aux problèmes. C'est l'idée de base de la méthode de Monte Carlo. La méthode de Monte Carlo utilise des méthodes mathématiques pour simuler en appréhendant la quantité géométrique et les caractéristiques géométriques du mouvement des choses. Il est basé sur un modèle probabiliste, selon le processus décrit par ce modèle, à travers les résultats d'expériences de simulation, comme la solution approchée du problème.

L'algorithme de MCTS est principalement divisé en quatre étapes, qui sont la sélection, l'expansion, la simulation et le rétropropagation.

ÉTAPE 1 : Sélection

En partant du nœud racine, sélectionnez de manière récursive le nœud enfant optimal et atteignez enfin un nœud feuille. Juger les avantages et les inconvénients des nœuds en fonction des limites de confiance supérieures (UCB)(cette formule n'est pas fixe et varie en fonction des applications) :

$$UCB1(S_i) = V_i + c\sqrt{\frac{logN}{n_i}}$$

Parmi eux, V_i est la valeur moyenne du nœud ; c est une constante, généralement 2 ; N est le nombre total d'explorations ; n_i est le nombre d'explorations du nœud courant. Avec la formule UCB ci-dessus, vous pouvez calculer les valeurs UCB de tous les nœuds enfants et sélectionner le nœud enfant avec la plus grande valeur UCB pour l'itération.

ÉTAPE 2 : Expansion

Si le nœud feuille actuel n'est pas un nœud terminal, créez un ou plusieurs nœuds enfants et sélectionnez-en un pour l'expansion.

ÉTAPE 3 : Simulation

À partir du nœud d'expansion, exécutez une sortie simulée jusqu'à la fin du jeu. Par exemple, en partant du nœud d'expansion, après 10 simulations et en gagnant neuf fois à la fin, le score du nœud d'expansion sera relativement élevé, et vice versa.

ÉTAPE 4 : Rétropropagation

En utilisant les résultats de la troisième étape de simulation, les réverbérations sont propagées pour mettre à jour la séquence d'action actuelle.

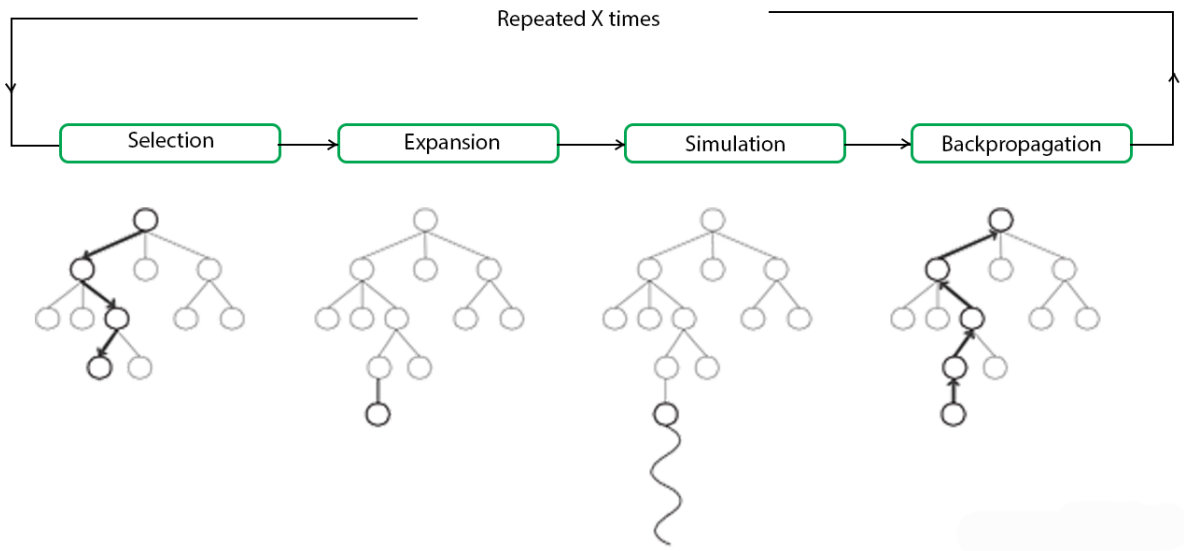


FIGURE 5 – Recherche arborescente de Monte Carlo

3 Le MCTS pour résoudre de problème TSP

3.1 Réalisation d'utiliser directement MCTS pour résoudre le problème

3.1.1 Représentation de la solution

La solution de voyage de commerce est représentée par la séquence des villes qui correspond à l'ordre de la villes. Donc pour notre arbre, chaque noeud est représenté une ville choisiat et le noeud suivant est représenté la ville prochaine choisit.

3.1.2 Sélection

La fonction d'évaluation pour la noeud N définie par:

$$h(N) = \frac{x_j - x_{min}}{x_{max} - x_{min}} - 2C_p \sqrt[2]{\frac{ln(n)}{n_j}}$$

Donc le noeud le plus interessant est le noeud minimisant le $h(N)$. Pour le sélection de C_p on fixe d'abord (après tester notre code, on choisie C_p entre $[0.2, 0.3]$ pour équilibrer les explorations et exploitations)

x_j : le minimum de la coût de simulation précédant.

n : le nombre totale de simulation pour le noeud parent.

x_{min} le cout minimum de le noeud parent.

x_{max} le cout maximum de le noeud parent.

n_j : le nombre de simulation pour cette noeud N.

Le premier terme pour uniformiser le x_j entre 0 et 1. On fait une référence de wikipédia[7] qui nous montrons que probabilité de gagner un jeux est entre 0 et 1.

3.1.3 Expansion

On crée les noeud enfants pour le noeud N (le union de solution des noeud enfants contient tous les solution de N), **on choisit aléatoirement un parmi les noeuds enfants C** .

Mais il y a une contrainte on ne peux pas choisir les villes qui est déjà choisie. Dans notre version de code, on crée un noeud enfant par une fois.

3.1.4 Simulation

On choisit une séquence des villes arbitrairement mais on fixe la ville qui est représenté par noeud avant C.

Mais quand on teste la méthode qui choisit une séquence des villes arbitrairement. C'est très moins performant. Donc on choisit une méthode de rollout, c'est à dire qu'on choisit le séquence des villes suivi la lois de probabilité lié à distance entre eux. Pour former les probabilités, on utilise la fonction softmax.

3.1.5 Mise à jour

Mise à jour n, n_j, x_j(si on trouve la coût de cette séquence est inférieur que ancien) et x_{max} pour les noeuds de cette branche.

Il y a un limite peut être mise à jour (le coût minimum on trouve).

3.2 Implémentation en C++ méthode MCTS directement

3.2.1 Classe Node

```
//parent node of the tree
Node* parent;

//the information of the node
Info content;

//the city number
int number;

// the location information of the node
int level;

// the extend status whether or not fully developped
bool extend;

//the path infomation of the node
int* tableux;
Node<T> ** childs;

// the number of childs
int nc;

//the maximum travel cost of the childs
double Xmax;
```

create _child(): pour créer une nouvelle fils qui représente ajoute une nouvelle ville dans le parcours sans répétition. Si on peut créer le nouvelle fils, tout d'abord on cherche les numéros des villes déjà passé, numéro de ville déjà passé par le fils. Parmi eux, on choisie aléatoirement et alloue l'espace pour nouvelle noeud.

rollout (): pour faire une simulation aléatoire à partir de ce noeud*. On va simuler trois fois pour récupérer le valeur minimum. Pour chaque fois, d'abord on va cumuler les numéros de citées qui n'ai pas arrivée. Ensuite nous allons tirer les cité par rapport la probabilité lié son distance. Finalement, on calcule la distance totale.

À la fin de cette fonction, content.X et Xmax sont mis à jour.

distribution(*permit,debut,fin): pour remplir le chemin hamiltonien suivi le probabilité par rapport de distance(calule par softmax).

insert(): pour que tous les fils forme un tas. Si tous les files sont développés, c'est à dire qu'on ne peut créer plus de fils, comme on choisit que le premier fils pour faire une simulation donc on applique downAjust(). Sinon les fils ne sont pas encore tous développés, on applique upAjust().

downAjust(int low ,int high) pour ajuster le tas de tête à pied. On choisit i=low et j=2i+1 son fils gauche(dans le tas) pour vérifier si les fils j,j+1 un est plus petite que racine i. On échange le fils plus petite et racine et commence à vérifier par la suite i=j,j=2i+1 jusqu'à les fils sont plus grande que racine ou touche le fonds c'est à dire il admet pas la fils.

upAjuste(int low , int high) pour ajuster le tas de pied à tête. Même principe que downAjuste() mais on commence à la fin de tas, et vérifie son parent est plus petite que lui ou pas.

back_up() pour mise à jour le content de parent et coût maximum de parent.

3.2.2 Classe Info

```
//the minimum value among the previous tirals
double X;

//the weight
static double Cp;

//the number of trials
int N;
```

UCT(int n, double Xmin , double Xmax)

$$\frac{X - Xmin}{Xmax - Xmin} - 2Cp\sqrt{\frac{log(n)}{N}};$$

3.2.3 Classe MCTS

selection(): pour déterminer quel Node faire développé. On commence à racine, si le noeud est développé et le noeud touch pas le fond. On refaire le vérifier de son fils childs[0] qui trier par tas par rapport de Info::UCT . Sinon on retourner lui même.

expansion(Node bestchild) pour créer une nouvelle fil de cet noeud. Si il touche le fond alors retourner lui même. Sinon créer une fils pour lui et le retourner.

simulation(Node child) pour estimer la parcours minimum de cet noeud. On exécute le rollout pour obtenir un parcours aléatoire et remplir le Info

update(Node child) commence à child mis à jour les information de son ancêtres.

Méthode1 Pour une racine fixé, on fait

```
while(n<iteration){
        t=selection();
        t=expansion(t);
        simulation(t);
        update(t);
        n++;
}
```

Méthode 2 Pareil que le méthode 1 sauf que on déplace le racine vers le plus meilleure fils, après certaine nombre de itération.

4 Utiliser algorithme k-opt et MCTS pour résoudre le problème

Comme nous l'avons dit dans la section 1.4, nous sélectionnons d'abord manuellement une ville comme point de départ, puis nous sommes prêts à démarrer la simulation. Au départ, la probabilité de choisir la prochaine ville est la même. Si nous obtenons $\triangle(\Pi, \Pi^*) < 0$, alors nous augmentons la probabilité d'être sélectionné pour chaque arête de poisson dans ce chemin. Revenez ensuite au point de départ et continuez d'essayer jusqu'à ce que vous ne puissiez pas trouver un meilleur chemin. Ensuite, nous choisissons au hasard un nouveau point de départ pour répéter la méthode. Nous devons également définir un temps d'arrêt de l'algorithme T.

4.1 Réalisation de l'algorithme k-opt et MCTS

4.1.1 Initialisation

Pour k-opt, on définit deux matrices symétriques $n \times n$, soit une matrice de poids W dont l'élément W_{ij} (tout initialisé à 1) contrôle la probabilité de choisir la ville j après la ville i, et une matrice d'accès Q dont l'élément Q_{ij} (tout initialisé à 0) enregistre les instants où le couple (i, j) est choisi lors des simulations. De plus, une variable M (initialisée à 0) est utilisée pour enregistrer le nombre total d'actions déjà simulées.

Pour Branch and bond, on garde le même initialisation.

4.1.2 Simulation

Pour k-opt, étant donné un état π, nous utilisons le processus de simulation pour générer de manière probabiliste un certain nombre d'actions. Comme expliqué dans la section 2.4, chaque action est représentée par $a = (a_1, b_1, a_2, b_2, ..., a_k, b_k, a_{k+1})$, contenant une série de sous-décisions $a_i, 1 \leqslant i \leqslant k$ (k est aussi une variable, et $a_{k+1} = a_1$), tandis que b_i pourrait être déterminé uniquement une fois que a_i est connu. Une fois b_i déterminé, pour chaque arête $(b_i, j), j \neq b_i$, nous utilisons la formule suivante pour estimer son potentiel $Z_{b_i j}$ (plus la valeur de $Z_{b_i j}$ est élevée, plus l'opportunité de l'arête (b_i, j) d'être choisi) :

$$Z_{b_i j} = \frac{W_{b_i j}}{\Omega_{b_i j}} + \alpha \sqrt{\frac{ln(M+1)}{Q_{b_i j} + 1}}$$

où $\Omega_{b_i j}$ désigne la valeur moyenne de $W_{b_i j}$ de toutes les arêtes par rapport à la ville b_i. Dans cette formule, la partie gauche $\frac{W_{b_i j}}{\Omega_{b_i j}}$ met l'accent sur l'importance des arêtes avec des valeurs élevées de $W_{b_i j}$ (pour améliorer la caractéristique d'intensification), tandis que la partie droite $\alpha \sqrt{\frac{ln(M+1)}{Q_{b_i j}+1}}$ préfère les arêtes rarement examinées (pour améliorer la diversification fonctionnalité). α est un paramètre utilisé pour atteindre un équilibre entre intensification et diversification, et le terme " $+1$" est utilisé pour éviter un numérateur négatif ou un dénominateur nul.

Pour prendre les sous-décisions séquentiellement, nous choisissons d'abord a_1 au hasard, et déterminons ensuite b_1. Récursivement, une fois a_i et b_i connus, a_{i+1} est décidé comme suit : (1) si boucler la boucle (connecter a_1 à b_i) conduirait à une action améliorante, ou$i \geqslant 10$, soit $a_{i+1} = a_1$. (2) sinon, considérons les 10 plus proches voisins de b_i avec $Z_{b_i l} \geqslant 1$ comme villes candidates, formant un ensemble X (excluant a_1 et la ville déjà connectée à b_i). Ensuite, parmi X chaque ville j est sélectionnée comme a_{i+1} avec probabilité p_j , qui est déterminée comme suit :

$$p_j = \frac{Z_{b_i j}}{\sum_{l \in X} Z_{b_i l}}$$

Une fois $a_{i+1} = a_1$, on ferme la boucle pour obtenir une action. De même, plus d'actions sont générées (formant un pool d'échantillonnage), jusqu'à rencontrer une action d'amélioration qui conduit à un meilleur état, ou le nombre d'actions atteint sa borne supérieure (contrôlée par un paramètre H).

Pour Branch and bond, Nous devons également choisir une ville de départ i, puis chaque fois que nous choisissons un petit chemin entre 2 villes(départ i) x_{ij}, nous utilisons la même formule

$$Z_{b_i j} = \frac{W_{ij}}{\Omega_{ij}} + \alpha\sqrt{\frac{ln(M+1)}{Q_{ij}+1}}$$

pour choisir.

4.1.3 Sélection

Pour k-opt, au cours du processus de simulation ci-dessus, si une action d'amélioration est rencontrée, elle est sélectionnée et appliquée à l'état actuel π, pour obtenir un nouvel état π^{new}. Sinon, si aucune action de ce type n'existe dans le pool d'échantillonnage, il semble difficile d'obtenir de nouvelles améliorations dans la zone de recherche actuelle. Ensuite, on choisit une nouvelle ville de départ (chaque ville de départ a la même probabilité d'être choisie).

Pour Branch and bond, une méthode similaire à k-opt, nous gardons la ville de départ, et chaque fois que la méthode Branch and bond est complétée, nous comparons la distance de sa solution optimale locale avec la distance de notre solution optimale temporaire($L(x)$). Si sa distance est plus petite, nous l'utilisons comme une nouvelle solution optimale temporaire et la distance est $L(x_{new})$. Si on ne peut pas obtenir de nouvelles améliorations, on change la ville de départ.

4.1.4 Rétropropagation

Pour k-opt, la valeur de M ainsi que les éléments des matrices W et Q sont mis à jour par rétropropagation comme suit. Au début, chaque fois qu'une action est examinée, M est augmenté de 1. Puis, pour chaque arête (b_i, a_{i+1}) qui apparaît dans une action examinée, soit $Q_{b_i a_{i+1}}$ augmenter de 1. Enfin, chaque fois qu'un état π est converti à un meilleur état π^{new} en appliquant l'action $a = (a_1, b_1, a_2, b_2, ..., a_k, b_k, a_{k+1})$, pour chaque arête $(b_i, a_{i+1}), 1 \leqslant i \leqslant k$, laisser :

$$W_{b_i a_{i+1}} \leftarrow W_{b_i a_{i+1}} + \beta(exp(\frac{L(\pi) - L(\pi^{new})}{L(\pi)}) - 1)$$

où β est un paramètre utilisé pour contrôler le taux d'augmentation de $W_{b_i a_{i+1}}$. Notez que nous ne mettons à jour $W_{b_i a_{i+1}}$ que lorsque nous rencontrons un meilleur état, car nous voulons éviter de fausses estimations (même dans une mauvaise action qui conduit à un état pire, il peut exister de bonnes arêtes (b_i, a_{i+1}). Avec ce processus de rétropropagation, le poids des bonnes arêtes serait augmenté pour augmenter leur chance d'être sélectionné, ainsi le processus d'échantillonnage serait de plus en plus ciblé. W et Q sont des matrices symétriques, donc soit $W_{b_i a_{i+1}} = W_{a_{i+1} b_i}$ et $Q_{a_{i+1} b} = Q_{b_i a_{i+1}}$ toujours.

Nous fixons un temps T et répétons l'algorithme dans le temps défini.

Pour Branche and bond, la valeur de M ainsi que les éléments des matrices W et Q sont mis à jour par rétropropagation comme suit. Au début, chaque fois qu'on obtient une nouvelle solution temporaire et la distance est $L(x_{new})$, M est augmenté de 1.Puis, pour chaque arête (i, j) qui apparaît dans solution temporaire, soit $Q_{i,j}$ augmenter de 1. Enfin, chaque fois on change la solution optimale temporaire, pour chaque arête $(i, j), 1 \leqslant i, j \leqslant n$, laisser :

$$W_{i,j} \leftarrow W_{i,j} + \beta(exp(\frac{L(x) - L(x_{new})}{L(x)}) - 1)$$

Nous fixons aussi un temps T et répétons l'algorithme dans le temps défini.

4.2 Implémentation en C++ avec méthode utilisant k-opt et MCTS

Nous présentons quelques-unes des fonctions les plus importantes dans le rapport, voir les commentaires dans le code pour les explications détaillées.

4.2.1 Structure de données

Comme décrit dans la première partie, nous convertissons le chemin en un permutation de villes. Par conséquent, nous avons créé une structure appelée Struct_Node, qui a deux variables, Pre_city, qui représente sa ville précédente, et Next_city, qui représente sa prochaine ville. Utilisez ensuite le tableau All_Node pour réaliser le permutation des villes et former un circuit hamiltonien. Nous avons également un tableau appelé Solution, qui est également utilisé pour stocker le permutation des villes, mais il est de type entier et ne stocke que la ville. De plus, il existe de nombreuses structures de données pour des opérations spécifiques, veuillez consulter les commentaires dans le code.

```
struct Struct_Node {
        int Pre_City;
        int Next_City;
};

Struct_Node *All_Node;        //Enregistrer un tour
int *Solution;
```

4.2.2 Utilisation des données

Pour les données, nous avons deux façons. L'une consiste à spécifier le nombre de villes, puis à générer aléatoirement la distance entre les deux villes et à les stocker dans le tableau Distance. L'autre consiste à utiliser l'instance de TSPLib. On lit le fichier .tsp, enregistrer les coordonnées horizontales et verticales de chaque ville, puis calculer la distance entre deux villes et enfin la stocker dans le tableau Distance. Nous convertissons uniformément les distances sous forme entière.

```
#define Inf_Cost           1000000000

typedef int    Distance_Type;
double *Coordinate_X;
double *Coordinate_Y;
Distance_Type **Distance;

int Calculate_Int_Distance(int First_City,int Second_City) {  //Calculer distance
↪  entre 2 ville a partir leur coordonnees
        return (int)(0.5 + sqrt(
        ↪  (Coordinate_X[First_City]-Coordinate_X[Second_City])
                *(Coordinate_X[First_City]-Coordinate_X[Second_City]) +
                ↪
                (Coordinate_Y[First_City]-Coordinate_Y[Second_City])*
                (Coordinate_Y[First_City]-Coordinate_Y[Second_City]) ) );
}
```

4.2.3 Processus de Markov

Nous savons que la méthode de k-opt ne modifie que partiellement le permutation des villes. Par conséquent, notre idée est la suivante : dans un temps donné, donnez d'abord au hasard un permutation des villes, et utilisez la combinaison de k-opt et MCTS pour opérer sur le permutation

des villes jusqu'à ce qu'aucune meilleure solution ne puisse être générée à l'aide de ce permutation. Ensuite, générez à nouveau aléatoirement un permutation des villes, puis utilisez la méthode de combinaison de k-opt et MCTS pour le résoudre. Répétez jusqu'à ce que le temps donné soit atteint. C'est un processus de Markov. Nous fixons le temps donné comme Param_T*nb_city. Plus il y a de villes, plus longtemps nous simulons.(Les permutations des villes décrits dans ce paragraphe sont stockés à l'aide du structure All_node et constituent donc une boucle.)

```
Distance_Type Markov_Decision_Process() //Répéter le MCTS jusqu'à temps limite
{
        MCTS_Init();         // Initialize MCTS parameters
        Generate_Initial_Solution();      // State initialization
        MCTS();                                    // MCTS
        while(((double)clock()-Current_Instance_Begin_Time)
        ↪  /CLOCKS_PER_SEC<Param_T*nb_city)
        {
                Jump_To_Random_State();
                MCTS();
        }

        Restore_Best_Solution(); //Copier le meilleur solution dans le Struct_Node
        ↪  *Best_All_Node ) à Struct_Node *All_Node
        if(Check_Solution_Feasible())
                return Get_Solution_Total_Distance();
        else
                return Inf_Cost;
}
```

4.2.4 MCTS

Comme décrit ci-dessus, nous utilisons le MCTS une fois pour chaque permutation des villes donnée au hasard. Nous calculons d'abord la solution pour la permutation des villes actuelle, puis nous changeons la permutation des villes à l'aide de la méthode k-opt. Best_delta est la différence entre la permutation des villes changée et la permutation originale. s'il est supérieur à 0, la solution changée augmente, et s'il est inférieur à 0, la solution changée diminue. Une rétro-propagation est ensuite effectuée. Si le nouveau chemin est plus court que le chemin original après l'utilisation de k-opt, nous conservons le nouvel permutations des villes et utilisons son chemin comme chemin le plus court. Tant qu'aucun chemin plus court ne peut être trouvé après la transformation k-opt basée sur le permutation des villes originale, nous mettons fin à ce MCTS. (Les permutations des villes décrits dans ce paragraphe sont stockés à l'aide du structure All_node et constituent donc une boucle.)

```
// Processus MCTS
void MCTS() {
        while(true)
        {
                Distance_Type Before_Simulation_Distance =
                ↪  Get_Solution_Total_Distance();

                //Simuler n fois(controlé par Param_H) actions
                Distance_Type Best_Delta=Simulation(Param_H*nb_city);
```

```
                // Utiliser l'information dans le meilleur action pour mise à jour
                ↪  les paramètres de MCTS par le rétropropagation
                ↪                   Back_Propagation(Before_Simulation_Distance,Best_Delta);
                ↪

                if(Best_Delta < 0)
                {
                        // Selectionner le meilleur action pour executer
                        ↪
                        Execute_Best_Action();

                        // Stocker le meilleur solution dans Struct_Node
                        ↪  *Best_All_Node
                        ↪
                        Distance_Type
                        ↪  Cur_Solution_Total_Distance=Get_Solution_Total_Distance();

                        if(Cur_Solution_Total_Distance <
                        ↪  Current_Instance_Best_Distance)
                        ↪
                        {
                                Current_Instance_Best_Distance =
                                ↪  Cur_Solution_Total_Distance;
                                ↪
                                cout<<"CHANGE "<<Current_Instance_Best_Distance;
                                ↪
                                cout<<"\n";
                                Store_Best_Solution();
                        }
                        ↪
                        Convert_All_Node_To_Solution();
                }
                else
                        break;          // Le MCTS terminer si il n'existe plus
                        ↪  d'action amélioré
        }
}
```

Simulation L'entrée de Simulation est une limite sur le nombre maximum de simulations. Elle est utilisée comme condition pour déterminer si une meilleure solution peut encore être trouvée pour ce cas. Action_delta est la différence entre le permutation des villes aprés une opération k-opt et le permutation des villes d'origine, on compare chaque fois, la plus petite étant la meilleure différence.

```
Distance_Type Simulation(int Max_Simulation_Times)
{
        Distance_Type Best_Action_Delta = 0;
        for(int i=0;i<Max_Simulation_Times;i++)
        {
                int Begin_City = Get_Random_Int(nb_city);
                Distance_Type Action_Delta =
                ↪  Get_Simulated_Action_Delta(Begin_City);
```

```
                Total_Simulation_Times++;
                if(Action_Delta<Best_Action_Delta){
                        Best_Action_Delta = Action_Delta;
                        break;
                }
        }
        return Best_Action_Delta;
}
```

Get_Simulated_Action_Delta Comme nous l'avons décrit dans la première partie, k-opt consiste à sélectionner deux ensembles de villes a_i, b_i, a_i et b_i étant déconnectés et b_i et a_{i+1} étant reconnectés. Ainsi, si on fait M-opt la différence entre le chemin de le permutation de villes après l'opération k-opt et le chemin de le permutation de villes original est calculée comme suit

$$\sum_{i=0}^{M} d(b_i, a_{i+1}) - \sum_{i=0}^{M} d(a_i, b_i)$$

avec $a_0 = a_{M+1}$. Nous plaçons a_i, b_i dans le tableau City_Sequence. Le code est si long que nous ne le refléterons pas dans le rapport.

Back_Propagation Après chaque simulation terminée, nous rétropropageons les arêtes correspondant au chemin le plus court. La formule de rétropropagation est présentée dans la 1ème partie. Les arêtes des chemins à ce stade sont équivalentes aux (b_i, a_{i+1}) dans le k-opt(dans le tableau City_Sequence).

```
//Si le delta pour un action est moins que zéro, utiliser l'information de cet
↪  action(Stocké dans le City_Sequence) pour renouveler les paramètres par
↪  rétropropagation
void Back_Propagation(Distance_Type Before_Simulation_Distance, Distance_Type
↪  Action_Delta) {
        for(int i=0;i<Pair_nb_city;i++)           {
                int First_City = City_Sequence[2*i];
                int Second_City = City_Sequence[2*i+1];
                int Third_City;
                if(i<Pair_nb_city-1)        Third_City = City_Sequence[2*i+2];
                ↪
                else        Third_City = City_Sequence[0];
                if(Action_Delta >0)                 {
                        double Increase_Rate = Beta*(pow(2.718, (double)
                        ↪  (-Action_Delta) / (double)(Before_Simulation_Distance)
                        ↪  )-1);
                        ↪                           Weight[Second_City][Third_City]
                        ↪  += Increase_Rate;
                        Weight[Third_City][Second_City] +=
                        ↪  Increase_Rate;
                        ↪
                }
        }
}
```

5 Monte Carlo Tree Search pour voyage de commerce probaliste

5.1 Introduction

Le problème probabiliste du voyageur de commerce (PTSP) a été introduit par Jaillet[8] comme une extension du problème classique du voyageur de commerce (TSP). Dans le PTSP, le vendeur ne doit pas nécessairement visiter chaque nœud, mais une probabilité d'exiger une visite est donnée. Probabilité d'exiger une visite est donnée pour chaque nœud. L'objectif est de trouver une tournée a-priori qui inclut chaque nœud et minimise la longueur attendue d'une tournée a-posteriori qui contient les noeuds avec les probabilités données et saute les autres. Une application réelle du PTSP consisterait, par exemple, à planifier une tournée quotidienne pour le facteur. tournée quotidienne pour le facteur. Après chaque livraison, il se rend toujours à l'adresse suivante de la tournée planifiée pour laquelle une livraison est requise.

Pour implémentation dans MCTS, on ne change que le méthode pour évaluer le x_j .

5.1.1 Méthode déterministe

Formellement, on nous donne un graphe complet $G =< V, E >$ avec V contenant n nœuds et E contenant m arêtes. Chaque arête$(i, j) \in E$ est affectée d'un coût $d_{ij} > 0$ et chaque noeud $v \in V$ a une probabilité $p(v)$ d'être visité. Une solution est une tournée a-priori $T =< v_1, ..., v_n >$ sur tous les nœuds avec v_n connecté à v_1 et ses coûts attendus sont définis comme suit:

$$c(T) = \sum_{i=1}^{2^n} p(R_i) \times L(R_i)$$

où R_i est une réalisation possible, c'est-à-dire un tour a posteriori possible, et $p(R_i)$ sa probabilité d'occurrence. Comme il existe un nombre exponentiel de réalisations différentes, il n'est pas pratique d'obtenir la valeur objective de cette manière. C'est pourquoi Jaillet [2] a montré que la longueur attendue peut être calculée en $O(n^3)$ en énumérant explicitement toutes les réalisations à l'aide du terme suivant :

$$c(T) = \sum_{i=1}^{n} \sum_{j=i+1}^{n} d_{v_i v_j} p_{v_i} p_{v_j} \prod_{k=i+1}^{j-1} (1 - p_{v_k}) + \sum_{j=1}^{n} \sum_{i=1}^{j-1} d_{v_i v_j} p_{v_i} p_{v_j} \prod_{k=j+1}^{n} (1 - p_{v_k}) \prod_{k=1}^{i-1} (1 - p_{v_k})$$

Le premier terme signifie le coût moyen de passe v_i vers v_j.

Le deuxième terme au contraire, signifie le coût moyen de passe v_j vers v_i.

Dans notre cas, par défaut, on supposons que le ville départ (0) est fixé avec son probabilité d'occurrence égale 1.

Par conséquence, il n'aura pas le cas de passe v_j vers v_i qui sont dans l'ordre $i < j$. donc on reformule terme:

$$c(T) = \sum_{i=1}^{n} \sum_{j=i+1}^{n+1} d_{v_i v_j} p_{v_i} p_{v_j} \prod_{k=i+1}^{j-1} (1 - p_{v_k})$$

où $v_1 = v_{n+1} = 0$.

5.1.2 Méthode approximation

Même si le terme $c(T)$ produit un temps d'évaluation polynomial pour le PTSP, le temps $O(n^3)$ résultant pour calculer coût de PTSP est encore très très long, en particulier pour les méthodes

métaheuristiques qui doivent évaluer à plusieurs reprises la valeur de la fonction objective $c(T)$. Dans cette étude, l'algorithme proposé doit comparer plusieurs fois deux solutions (c'est-à-dire la nouvelle solution avant et après la recherche locale) sur la base de la valeur de la fonction objective. Par conséquent, l'évaluation approximative a été utilisée pour augmenter l'efficacité de calcul de l'algorithme proposé.

$$E(T) = \sum_{i=1}^{n} \sum_{j=i+1}^{\min(n+1,i+\lambda)} d_{v_i v_j} p_{v_i} p_{v_j} \prod_{k=i+1}^{j-1} (1 - p_{v_k})$$

La seule différence entre $c(T)$ et $E(T)$ est le choix de la position de troncature λ dans $E(T)$. L'équation $E(T)$ aura une complexité de calcul de $O(n\lambda^2)$ au lieu de $O(n^3)$ dans $c(T)$. Il est facile de voir que $E(T)$ devient plus précis lorsque λ augmente. Cependant, une valeur plus élevée de λ nécessite plus d'efforts de calcul $E(T)$. L'équation E(T) peut donner une très bonne approximation de $c(T)$ avec une valeur petite de λ lorsque le probabilité $p(v_k)$ devient grande, parce que $\prod_{k=i+1}^{j-1}(1-p_{v_k})$produira un très faible valeur et peut être omise. Néanmoins, l'équation $E(T)$ aura besoin d'une plus grande valeur de λ pour effectuer une bonne approximation lorsque la valeur de $p(v_k)$ est petite.

Pour une tournée donnée, $E(T)$ est toujours inférieur à la valeur de la valeur de $c(T)$ en raison de la troncature dans le calcul de E. Laissons b et a désigner respectivement la tournée a priori avant et après l'application d'une méthode de recherche locale spécifique. Cela signifie qu'aucune amélioration n'a été trouvée après la recherche locale si $E(a) \geq E(b)$. $c(T)$ est utilisé pour évaluer exactement la solution après la recherche locale si $E(a) < E(b)$. Si la recherche locale donne une meilleure valeur $c(b)$ que celle de la solution originale (c'est-à-dire $c(a) < c(b)$), la nouvelle solution a remplacera la solution originale. Si aucune amélioration n'a été trouvée après la recherche locale, aucun remplacement n'est effectué.

5.1.3 Méthode estimation

Nous considérons le PTSP comme un problème d'optimisation combinatoire stochastique d'optimisation combinatoire stochastique qui peut être décrit comme suit[9] : Minimiser $F(x) = E(f(x, \Omega))$, sous réserve que $x \in S$, où x est une solution a priori, S est l'ensemble des solutions réalisables, l'opérateur E représente l'espérance mathématique, et $f(x, \Omega)$ est le coût de la solution a posteriori qui dépend d'une variable aléatoire Ω, qui est une distribution de Bernoulli à n variables, et d'une réalisation ω de la solution a posteriori. et une réalisation ω de Ω prescrit quels nœuds doivent être visités. Un estimateur sans biais de $F(x)$ d'une solution x peut être calculé sur la base d'un échantillon de coûts de solutions a posteriori obtenues à partir de M solutions indépendantes de la variable aléatoire Ω.

Dans les algorithmes d'amélioration itérative pour le PTSP, nous devons comparer deux solutions voisines x et x' afin de sélectionner celle dont le coût est le plus faible. Pour x , un estimateur sans biais de $F(x')$ peut être estimé analogiquement $F(x)$ en utilisant un ensemble différent de M' réalisations indépendantes de Ω. Cependant, afin d'augmenter la précision de cet estimateur, la technique bien connue de réduction de la variance appelée la méthode des nombres aléatoires communs peut être adoptée.

Méthode des nombres aléatoires communs, dans le contexte du PTSP, cette technique consiste à à utiliser le même ensemble de réalisations de Ω pour estimer les coûts $F(x')$ et $F(x)$. Par conséquent, nous avons M = M' et l'estimateur $\hat{F}_M(x') - \hat{F}_M(x)$ de la différence de coût est donné par :

$$\hat{F}_M(x') - \hat{F}_M(x) = \frac{1}{M} \sum_{r=1}^{M} (f(x', w_r) - f(x, w_r))$$

Nous avons mis en œuvre des algorithmes d'amélioration itérative qui utilisent cette façon d'estimer les différences de coût en exploitant une structure de voisinage qui utilise 2-opt. Pour rendre le calcul des différences de coût aussi efficient que possible, étant donné deux solutions a

priori voisines et une réalisation ω, l'algorithme doit identifier les arêtes qui ne sont pas communes aux deux solutions a posteriori. Ceci est réalisé comme suit :

pour chaque arête $< i, j >$ qui est supprimée de x, il faut finaliser l'arête correspondant à la solution a posteriori. pour chaque arête$< i, j >$ supprimée dans x, il faut trouver l'arête correspondante $< i', j' >$ qui est supprimée dans la solution a posteriori de x. Nous appelons cette arête l'arête a posteriori et nous l'obtenons comme suit : Si le nœud i nécessite une visite, alors $i' = i$, sinon, i' est le premier prédécesseur de i dans x tel que $\omega[i'] = 1$, c'est-à-dire le premier prédécesseur pour lequel la réalisation est un, ce qui indique qu'il nécessite une visite. Si le nœud j nécessite une visite, alors $j' = j$, sinon, j' est le premier successeur de j tel que $\omega[j'] = 1$. Rappelons que dans un mouvement 2-opt, les arêtes $< a, b >$ et $< c, d >$ sont supprimées de x et remplacées par $< a, c >$et$< b, d >$. Pour une réalisation ω donnée et les arêtes a-posteriori correspondantes, $< a', b' >$, $< c', d' >$, la différence de coût entre les deux solutions a posteriori est donnée par $d(a', c') + d(b', d') - d(a', b') - d(c', d')$, où $d(i, j)$ est le coût de l'arête $< i, j >$.

5.2 Implémentation en C++ avec méthode MCTS pour le cas probabiliste

```
double ETtsp(const int* sequence, const int M=MAX){
        double cout =0;
        for (int i=0;i<M;i++){
                cout +=d[sequence[i]][sequence[i+1]];
        }
        return cout;
}

double ETptsp1(const int* sequence ){
        double cout =0;
        for(int i=0; i<MAX;i++){
                for(int j=i+1;j<=MAX;j++) {
                        float
                        ↪  t=d[sequence[i]][sequence[j]]*P[sequence[i]]*P[sequence[j]];
                        float f=0;
                        for(int k = i+1;k<=j-1;k++){
                                f*=(1-P[sequence[k]]);
                        }
                        cout+=t*f;
                }
        }
        return cout;
}

double ETptsp2(const int* sequence ){
        double cout =0;
        int lambda=20;
        for(int i=0; i<MAX;i++){
                int l= min(MAX,i+lambda);
                for(int j=i+1;j<=l;j++) {
                        float
                        ↪  t=d[sequence[i]][sequence[j]]*P[sequence[i]]*P[sequence[j]];
                        float f=1;
                        for(int k = i+1;k<=j-1;k++){
```

```
                                f*=(1-P[sequence[k]]);
                        }
                        cout+=t*f;
                }
        }
        return cout;
}

double ETptsp3(const int* sequence){
        double cout=0;
        int it =100;
        int n=0;
        while(n++<it){
                int om[MAX];
                int c=0;
                for(int i=0;i< MAX;i++){
                        float r = random(0,1000)/999;
                        if(r<P[i]) {
                                om[i]=1;
                                c++;
                        }
                        else{
                                om[i]=0;
                        }
                }
                int* pc= new int[c+1];
                int l=0;
                for (int i=0;i<=MAX;i++){
                        if(om[sequence[i]]) pc[l++]=sequence[i];
                }
                cout +=ETtsp(pc,c);
                delete[] pc;
        }
        return cout/it;
}

double ETptsp4(const int* x1,int* CitySequence,int Pair_nb_city ){
        double diff=0;
        int it =100;
        int n=0;
        while(n++<it){
                int om[MAX];
                int c=0;
                for(int i=0;i< MAX;i++){
                        float r = random(0,1000)/999;
                        if(r<P[i]) {
                                om[i]=1;
                                c++;
                        }
                        else{om[i]=0;}
                }
```

```
                for(int i=0;i<Pair_nb_city;i++) {
                        int a=0;
                        int b=0;
                        int c=0;
                        int t=0;
                        if(om[CitySequence[2*i]]) a=CitySequence[2*i];
                        else {
                                int j=0;
                                for(;j<MAX;j++) {
                                        if(x1[j]==CitySequence[2*i])
                                        break;
                                }
                                while( om[x1[j]]==0 && j!=0){
                                        j--;
                                }
                                a = x1[j];
                        }
                        if(om[CitySequence[2*i+1]]) {
                                b=CitySequence[2*i+1];
                                c=CitySequence[2*i+1];
                        }
                        else{
                                int j=0;
                                for(;j<MAX;j++) {
                                        if(x1[j]==CitySequence[2*i+1])
                                        break;
                                }
                                int l=j;
                                while( om[x1[l]]==0 && j!=MAX){ l++; }
                                b = x1[l];
                                l= j;
                                while(om[x1[l]] == 0 && j!=0){ l--; }
                                c= x1[l];
                        }
                        if(om[CitySequence[2*i+2]])t=CitySequence[2*i+2];
                        else{
                                int j=0;
                                for(;j<MAX;j++) {
                                        if(x1[j]==CitySequence[2*i+2])
                                        break;
                                }
                                while( om[x1[j]]==0 && j!=MAX){ j++; }
                                t = x1[j];
                        }
                        diff+=(d[c][t]-d[a][b]);
                }
        }
        return diff/it;
}
```

6 MCTS + Réseaux profonds + Apprentissage Renforcement

Faire référence de [10]

6.1 MCTS

6.1.1 Selection

(Kocsis et Szepesvri 2006) ont proposé une stratégie de sélection appelée Upper Confidence bounds applied to Trees (UCT), qui a remporté un grand succès dans le jeu. Il existe certaines différences entre le jeu et les problèmes d'optimisation combinatoire.

Tout d'abord, une branche avec le taux moyen de gain le plus élevé est préférée dans le jeu, alors que l'optimisation combinatoire vise à trouver l'extrême, qui peut se situer dans la direction sans une bonne valeur moyenne. Ainsi, étant donné un nœud s, nous modifions la politique de l'UCT en sélectionnant l'enfant i de s qui maximise la la formulation suivante,

$$arg\max_{\hat{Q}_i}(\hat{Q}_i + 2 * C_p * \sqrt{\frac{\ln N_s}{N_i}})$$

où $\hat{Q}_i = -f(i)$, f est défini comme suit:

$$f(v) = g(v) + h(v)$$

où $g(v)$ est connu et représente la longueur réelle de la séquence ordonnée S du premier au dernier nœud, $h(v)$ est inconnu et supposé être la longueur optimale de v au début ville 0 (cycle) . Dans notre cadre, $h(v)$ est évalué par un réseau neuronal profond, qui sera décrit dans la section suivante. $\hat{Q}_i$ est la meilleure récompense trouvée sous la sous-arborescence du nœud i . N_s et N_i sont respectivement le nombre de visites du nœud $s$et du nœud i. $C_p > 0$ est un paramètre utilisé pour équilibrer l'exploitation et l'exploration

De plus, l'étendue de la valeur $\hat{Q}$ est différente entre les problèmes de jeu et les problèmes d'optimisation combinatoire. Dans les jeux, le résultat d'un jeu est composé d'une perte, d'un match nul et d'une victoire, c'est-à-dire $0, 0, 5, 1$. La récompense moyenne d'un nœud reste toujours comprise entre $[0, 1]$. Dans les problèmes d'optimisation combinatoire, une récompense arbitraire peut ne pas se situer dans l'intervalle prédéfini. dans l'intervalle prédéfini. Nous normalisons donc la meilleure récompense de chaque nœud c dont le parent est le nœud p à $[0, 1]$ avec la formulation suivante,

$$\hat{Q}_c = \frac{\hat{Q}_c - \hat{Q}_{min}}{\hat{Q}_{max} - \hat{Q}_{min}}$$

où $\hat{Q}_{max}$ et $\hat{Q}_{min}$ sont respectivement le maximum et le minimum parmi tous les nœuds enfants du nœud p, respectivement.

6.1.2 Expansion

Lorsqu'un nœud feuille I est atteint, nous développons le nœud jusqu'à ce que son nombre de visites atteigne un seuil prédéfini (nous avons fixé ce seuil à 40). Cette méthode permet d'éviter de générer un trop grand nombre de branches, ce qui distrait la recherche et économise les ressources de calcul. Comme l'algorithme A* (Hart, Nilsson et Raphael 1968), nous développons tous les nœuds enfants du nœud feuille l en même temps.

6.1.3 Simulation

Nous utilisons la fonction de valeur h pour évaluer tous les nœuds enfants qui sont développés dans l'étape d'expansion.

6.1.4 Rétro-propagation

Nous choisissons d'utiliser la meilleure récompense. de simulation parmi tous les nœuds enfants pour la rétro-propager vers la racine.

6.2 Graphes convolutions réseaux

Inspiré par le graph embedding network, nous proposons d'utiliser des convolutions de graphe pour extraire des caractéristiques du graphe. Chaque nœud du le graphe est représenté par un vecteur de caractéristiques et fusionne les informations de ses nœuds voisins de manière récursive en fonction de la topologie du graphe. Pour chaque nœud, la caractéristique est exprimée sous la forme d'un vecteur à 9 dimensions. Nous utilisons un élément 0 ou 1 pour indiquer si un nœud a été traversée ou non. En outre, les informations sur le nœud actuel (état de la traversée, coordonnées x, coordonnées y), nous prenons particulièrement en compte le premier et le dernier nœud du chemin parcouru. dernier nœud du chemin parcouru car le chemin de la solution est le chemin hamiltonien. De plus, nous utilisons le poids des arêtes(distance) comme caractéristique supplémentaire

Nous décrivons maintenant la paramétrisation des convolutions de graphes à l'aide de graphes embedding. Nous faisons correspondre les caractéristiques de chaque nœud v du graphe à l'espace dimension plus large en utilisant la formule suivante:

$$H_v^{t+1} = \sigma(\theta_1 x_v + \theta_2 \sum_{u \in \mathcal{N}(\sqsubseteq)} H_u^t + \theta_3 \sum_{u \in \mathcal{N}(\sqsubseteq)} \sigma(\theta_4 d_{v,u}))$$

où $\theta_1 \in \mathbb{R}^l, \theta_2, \theta_3 \in \mathbb{R}^{l \times l}$ et $\theta 4 \in \mathbb{R}^l$ sont les paramètres, et σ est l'unité linéaire rectifiée (relu).x_v et $d_{v,u}$ sont les caractéristiques du nœud et la distance entre les deux nœuds mentionnés ci-dessus, respectivement. Et $\mathcal{N}(\sqsubseteq)$ désigne les nœuds voisins du nœud v.

Après T itérations, chaque nœud est intégré dans le graphe, nous utiliserons ces informations d'intégration pour définir h(v), nous calculons h(v) comme suit,

$$h(v) = \theta_5 \sigma([\theta_6 \sum_{u \in V} H_u^T, \theta_7 H_v^T])$$

où $\theta_5 \in \mathbb{R}^{2l}, \theta_6, \theta_7 \in \mathbb{R}^{l \times l}$ et $[-,-]$désigne l'opérateur de concaténation.
On le suggère T=4.

6.3 Auto-apprentissage

6.3.1 Formulation de l'apprentissage par renforcement

Nous définissons les états, les actions et les récompenses dans le cadre de l'apprentissage par renforcement de la manière suivante :

- États : un état S est une séquence ordonnée de nœuds parcourus sur un graphe G. Nous utilisons graphe embedding pour coder chaque état sous la forme d'un vecteur dans l'espace à l dimension. L'état terminal $\hat{S}$ signifie que nous avons parcouru tous les nœuds.
- Transition : la transition est déterministe dans le problème du voyageur de commerce, et correspond à l'ajout du nœud sélectionné $v \in \bar{S}$ à S, où $\bar{S}$ et S sont respectivement la séquence parcourue et la séquence non parcourue, respectivement.
- Actions : une action v est un nœud de G dans la séquence $\bar{S}$ non parcourue.
- Récompenses : Lorsque tous les nœuds de G sont parcourus, la longueur d de la séquence ordonnée $\hat{S} = \{v_1, v_2, .., v_n\}$ peut être calculée selon la formule suivante:

$$d = \sum_{i=1}^{|\hat{S}|-1} d_{v_i, v_{i+1}} + d_{v_{\hat{S}}, v_1}$$

Nous pouvons également calculer la longueur de la séquence partielle $S_p = S \bigcap v$ lorsque le nœud v est ajouté à S comme suit,

$$g(v) = \sum_{i=1}^{|\hat{S}^p|-1} d_{v_i, v_{i+1}}$$

Nous définissons la fonction de récompense $r(s, v)$ à l'état s comme la longueur de la séquence partielle ordonnée de $\hat{S}$ où le nœud de départ est v .C'est-à-dire

$$r(s, v) = d - g(v)$$

— Politique : Sur la base de la fonction de valeur h du réseau neurone, nous utilisons la recherche arborescente de Monte Carlo comme politique par défaut pour sélectionner l'action v suivante. Après avoir répété t fois la lecture, nous choisissons une action v parmi toutes les actions valides de l'état racine s par la la formulation suivante,

$$v = arg \max_{v_i \in V_c} \hat{Q}_{v_i}$$

où V_c est l'ensemble de toutes les actions valides de l'état racine s, et $\hat{Q}_{v_i}$ est la récompense de l'état, qui est obtenue en effectuant l'action v_i de l'état racine.

6.3.2 Algorithme d'apprentissage

Tout d'abord, les paramètres du réseau neuronal sont initialisés à des poids aléatoires θ_0 . Lorsqu'un épisode se termine et que tous les nœuds ont été parcourus, les données pour chaque étape temporelle t sont stockées sous forme de (s_t, v_t, r_t), où r_t peut être calculé selon $r(s, v)$. Le réseau neuronal est formé à partir d'un échantillonnage uniforme de tous les pas de temps (s, v, t). En particulier, les paramètres θ sont appris par descente de gradient sur une fonction de perte I sur l'erreur quadratique moyenne:

$$I = (r - h)^2 + c||\theta||^2$$

où c est un paramètre qui contrôle le niveau.
Notre algorithme de formation, décrit dans l'algorithme 1,

Algorithm 1 Training Algorithm

Initialize experience replay memory M to capacity N
for $i = 0 \rightarrow MaxEpisode$ **do**
  Draw graph G from distribution D
  Initialize the state to empty S=()
  for $step = 1 \rightarrow EndStep$ **do**
    $v_t = \arg\max_{v_i \in \bar{S}} \hat{Q}_{v_i}$
    Add v_t to partial solution: $S_{t+1} := (S_t, v_t)$
  end for
  Add tuple (S_i, v_i, r_i) to M, $i = 1, 2, ..., EndStep$
  Sample random batch from B $\overset{\text{i.i.d}}{\sim}$ M
  Update Θ by Adam over (10) for B
end for
return Θ

FIGURE 6 – Algorithme d'apprentissage

7 Les résultats

7.1 Méthode MCTS directement

Pour le MCTS directement, nous utilisons le nombre d'itérations pour contrôler l'exécution du programme. Nous pouvons obtenir les résultats suivants.

Instance	a280	a280	berlin52	berlin52	bier127	bier127	bier127
Nombre d'itération	10000	100000	100000	1000000	200000	500000	5000000
Temps(s)	38s	369s	11s	64s	67s	134s	463s
Résultat k-opt+MCTS	3050	3022	13447	12670	266002	231796	216236
Résultat réel	2579	2579	7542	7542	118282	118282	118282

TABLE 1 – Résultat TSPLib MCTS

Nous pouvons constater que les résultats convergent en un certain temps, mais qu'ils sont loin de la solution optimale réelle. Dans le premier exemple, la différence n'est pas trop importante, mais lorsque la distance entre les villes est grande, comme dans le troisième instance, la différence par rapport à la solution optimale réelle est importante. Par conséquent, nous ne la considérons pas comme efficace.

7.2 Méthode utilisant k-opt et MCTS résultats

7.2.1 Données aléatoires en petite dimension

Tout d'abord, nous testons les données dans des dimensions plus petites. Nous supposons qu'il y a 5 villes, chacune avec des distances entre 1 et 10, et nous générons ces distances de manière aléatoire. Nous supposons que nous partons de la ville numéro 0.

```
Dis: 1000000000Dis: 4Dis: 7Dis: 8Dis: 6
Dis: 4Dis: 1000000000Dis: 4Dis: 6Dis: 7
Dis: 7Dis: 4Dis: 1000000000Dis: 3Dis: 10
Dis: 8Dis: 6Dis: 3Dis: 1000000000Dis: 2
Dis: 6Dis: 7Dis: 10Dis: 2Dis: 1000000000
```

FIGURE 7 – Distances 5 villes

Nous effectuons ensuite le test et les résultats sont les suivants.

```
CHANGE 19
City: 0City_Pre: 4City_Next: 1
City: 1City_Pre: 0City_Next: 2
City: 2City_Pre: 1City_Next: 3
City: 3City_Pre: 2City_Next: 4
City: 4City_Pre: 3City_Next: 0
```

FIGURE 8 – Résultat 5 villes

Le plus court chemin est

$$0 \to 1 \to 2 \to 3 \to 4$$

Enfin, nous revenons de 4 à 0. Et le distance de chemin est 19. Nous utilisons Cplex pour la vérification. Les résultats sont les suivants.

```
MIP - Integer optimal solution:  Objective =  1.9000000000e+01
Solution time =    0.01 sec.  Iterations = 9  Nodes = 0
Deterministic time = 0.20 ticks  (20.49 ticks/sec)

CPLEX> d so v -
Incumbent solution
Variable Name           Solution Value
x1_5                          1.000000
x2_1                          1.000000
x3_2                          1.000000
x4_3                          1.000000
x5_4                          1.000000
All other variables in the range 1-20 are 0.
CPLEX>
```

FIGURE 9 – Résultat 5 villes Cplex

On constate que le résultat est exactement le même que celui obtenu par notre algorithme, parfait.

7.2.2 Les instances de TSPLib

Nous l'avons ensuite testé en utilisant les instances de TSPLib. Nous avons choisi trois ensembles de données différents pour nos tests.

Pour le fichier "a280.tsp", nous avons un total de 280 villes et nous le laissons fonctionner pendant 280 secondes. On constate que le résultat diminue rapidement depuis la distance obtenue au début de la première simulation jusqu'au résultat final. Les résultats sont les suivants

```
CHANGE 32907
CHANGE 32683
CHANGE 32567
```

(a) Distance de chemin au début

```
CHANGE 2625
```

(b) Distance de chemin fin

FIGURE 10 – Résultat « a280.tsp »

Le résultat que nous avons trouvés sur internet pour ce fichier est 2579. Nous pouvons constater qu'ils ne sont pas très différents. Une période plus longue aurait permis d'obtenir ce résultat. Cependant, les résultats évolueront très lentement à partir de ce résultat, nous prendrons donc ce résultat comme celui que nous obtenons.

Pour les autres cas, nous laissons le programme s'exécuter pendant autant de secondes qu'il y a de villes. Étant donné que la durée d'exécution doit augmenter en fonction du nombre de villes, il n'est pas logique que nous fixions la même durée d'exécution. Les résultats sont présentés dans le tableau ci-dessous

Instance	a280	a280	berlin52	berlin52	bier127	bier127
Temps(s)	280s	5*280s	52s	5*52s	127s	5*127s
Résultat k-opt+MCTS	2625	2611	7542	7542	118431	118431
Résultat réel	2579	2579	7542	7542	118282	118282

TABLE 2 – Résultat TSPLib k-opt+MCTS

Nous pouvons constater que plus le temps est long, plus la convergence est lente. Nous savons qu'elle continuera à converger, mais il est encore difficile d'obtenir une solution pratique.

7.3 Comparaison de 2 méthodes

Instance	a280	berlin52	bier127
Temps MCTS	369s	64s	134s
Temps k-opt+MCTS	280s	52s	127s
Résultat MCTS	3022	12670	231796
Résultat k-opt+MCTS	2625	7542	118431
Résultat réel	2579	7542	118282

TABLE 3 – Comparaison de 2 méthodes

Avec les résultats des deux méthodes à des moments proches, nous pouvons voir que l'utilisation de la méthode k-opt + MCTS est beaucoup plus performante que l'utilisation directe de MCTS, en particulier pour la troisième instance (nous ne pouvons pas contrôler le même temps parce que la méthode MCTS est contrôlée à l'aide du nombre d'itérations). Ce résultat est justifié par le fait que l'utilisation directe de MCTS est plus aléatoire, alors qu'avec la combinaison de la méthode k-opt, nous sommes équivalents à l'extension des meilleurs résultats à partir d'une partie des bons résultats, ce qui est plus efficace.

7.4 Résultat pour le cas probabiliste avec MCTS directement

On utilise l'instance a280.tsp pour tester. Pour probabilité d'occurrence égale 0.8 sauf que la probabilité de le ville(0) de départ égale 1.

nombre de itération	temps	coût/chemin
1000	3.81s	2870 0 248 251 252 253 254 255 256 257 258 259 260 261 262 263 264 265 139 140 147 146 145 144 143 199 200 195 196 193 194 197 198 142 141 138 137 136 135 267 268 269 133 134 266 148 149 177 176 150 151 155 152 154 153 128 127 126 125 123 122 121 120 119 118 156 157 158 159 174 160 161 162 163 164 165 166 167 168 100 99 98 97 92 93 96 95 94 77 76 74 75 73 72 71 70 66 69 68 67 57 58 43 44 55 56 54 53 52 51 50 49 48 47 46 45 40 39 38 34 35 36 37 33 32 30 31 28 27 26 25 21 24 22 23 13 12 11 10 9 7 6 8 275 274 273 272 271 270 15 14 16 17 18 131 132 130 19 20 129 124 29 41 42 59 60 117 61 62 63 64 65 84 85 115 114 116 113 112 86 83 82 81 80 88 108 107 103 102 101 90 91 89 79 78 87 111 110 109 104 105 172 106 173 171 170 169 187 188 189 190 191 192 185 186 184 183 182 181 180 179 178 175 201 202 203 204 205 206 207 208 209 210 213 214 217 218 221 220 219 216 215 212 211 228 229 250 249 246 244 239 238 245 230 231 232 235 234 233 226 225 224 223 222 227 236 237 243 240 241 242 247 277 278 3 276 5 4 2 279 1
10000	34.02 s	2838 0 272 7 9 10 11 12 14 13 23 22 24 21 25 26 27 28 30 31 32 33 34 37 35 36 38 39 40 41 42 59 60 117 116 114 115 85 84 64 65 63 62 61 58 57 56 55 44 45 46 54 53 52 51 50 49 48 47 43 67 68 66 69 70 71 72 73 75 74 76 77 78 80 79 89 108 88 81 82 83 86 112 111 87 110 113 109 107 103 104 105 106 172 173 161 162 163 164 165 166 167 168 169 171 170 102 101 100 99 98 97 92 93 96 95 94 91 90 160 174 159 158 157 156 118 119 120 121 122 123 124 29 125 126 127 20 19 130 131 132 17 18 129 128 153 154 152 155 151 150 176 175 180 181 182 183 184 186 185 188 187 189 190 191 192 193 194 195 196 197 200 199 143 142 141 140 139 265 137 136 266 264 263 262 261 260 259 258 257 256 253 252 251 208 207 206 209 210 211 212 213 214 215 216 219 218 221 220 217 222 223 224 225 226 227 228 229 250 249 246 244 239 238 237 230 231 236 232 233 234 235 245 243 240 241 242 247 248 255 254 205 204 203 202 201 198 144 145 146 147 138 148 149 177 178 179 135 267 268 269 133 134 16 15 270 271 273 274 275 276 3 278 277 2 279 1 4 5 6 8

(a)

nombre de itération	temps	coût/chemin
100000	353s	2846 0 255 227 228 229 250 249 246 244 239 240 241 242 243 245 238 237 230 231 236 235 232 233 234 226 225 224 223 222 218 221 220 219 216 215 212 211 210 213 214 217 209 208 251 254 253 256 257 258 259 260 261 262 263 264 265 137 136 266 267 268 269 133 134 135 138 147 146 145 142 141 140 139 148 149 177 150 176 175 180 179 178 181 182 183 184 186 185 189 188 187 164 163 162 161 160 174 159 158 157 156 118 119 120 121 122 123 124 29 30 31 28 27 26 25 21 24 22 23 13 14 12 11 10 9 7 6 8 275 274 273 272 271 270 15 16 17 132 131 18 19 20 129 130 128 127 126 125 153 154 152 155 151 173 172 105 104 103 102 101 100 99 98 97 92 93 96 95 94 77 76 74 73 72 71 70 69 66 65 64 63 62 61 117 60 59 42 41 40 39 38 37 36 35 34 33 32 48 47 52 51 50 49 46 53 45 54 55 44 43 56 57 58 67 68 84 85 115 114 116 113 112 86 83 82 81 88 80 79 78 75 87 111 110 109 107 108 89 90 91 168 169 170 171 166 167 165 106 190 191 192 193 196 195 194 200 199 143 144 198 197 201 202 203 204 205 206 207 252 248 247 278 277 3 276 5 4 2 279 1

(b)

FIGURE 11 – Résultat « a280.tsp » cas probabiliste

Les résultats expérimentaux montrent que l'utilisation des MCTS en combinaison avec rollout pour résoudre le problème PTSP n'est pas aussi efficace. Elle prend plus de temps et ne permet pas

d'atteindre la solution optimale de manière récursive aussi rapidement. Cependant, avec moins de temps, les MCTS peuvent donner un résultat plus satisfaisant.

8 Conclusion

Dans ce projet, nous avons travaillé sur la résolution du problème du voyageur de commerce et sur l'application de l'algorithme de recherche arborescente de Monte Carlo. Nous en avons tiré beaucoup d'enseignements.

Premièrement, nous en avons appris davantage sur le problème du voyageur de commerce. Il s'agit d'un problème classique d'optimisation combinatoire avec un large éventail d'applications dans la pratique, telles que la planification logistique et l'optimisation des itinéraires. Cependant, en pratique, lorsqu'il y a de nombreuses villes, trouver la solution optimale peut être une tâche difficile en raison de la complexité de leur explosion combinatoire. La situation est encore plus complexe si l'on tient compte de la probabilité d'annulation des villes. C'est pourquoi nous devons nous concentrer davantage sur les algorithmes de recherche heuristiques tels que l'algorithme de recherche arborescente de Monte Carlo. Il est préférable de combiner les connaissances en matière de statistiques et de probabilités avec les puissantes capacités de calcul et de simulation des ordinateurs pour obtenir la solution optimale. Dans notre rapport, nous avons utilisé deux méthodes concernant le MCTS pour traiter le problème du voyageur de commerce. Les résultats obtenus pour une période donnée sont tous deux très proches des résultats réels. Cela prouve que l'utilisation d'algorithmes de recherche heuristique fonctionne très bien pour résoudre de tels problèmes.

Deuxièmement, nous avons appris à utiliser l'algorithme de recherche arborescente de Monte Carlo. Il s'agit d'une méthode de recherche heuristique qui guide le processus de recherche par la simulation et le hasard afin de trouver une solution de haute qualité. L'algorithme combine les idées de la simulation de Monte Carlo et de la recherche arborescente en sélectionnant et en simulant des chemins de manière aléatoire et en mettant continuellement à jour les nœuds de l'arbre de recherche afin d'optimiser progressivement la solution. L'algorithme de recherche arborescente de Monte Carlo excelle dans le traitement des problèmes d'optimisation combinatoire et d'espace d'état à grande échelle, avec efficacité et évolutivité. Par exemple, dans notre cas, nous pouvons effectuer 10 000 itérations de MCTS en quelques dizaines de secondes et obtenir un résultat qui n'est pas très différent de la solution optimale réelle. Cependant, nous avons constaté que pour cette méthode, la convergence est moins stable et qu'il est très difficile d'obtenir une meilleure solution optimale.

Pour le cas probabiliste, dans le présent document, MCTS sont d'abord proposés et développés pour résoudre le PTSP. Bien que les résultats numériques ne confirment pas la supériorité prédominante de le méthode MCTS, l'incorporation de la notion de diversification toutefois améliore de ses performances(rollout), Elle a permet au chercheur d'être conscient de la possibilité de combiner le MCTS avec d'autre méthode par exmaple (deep networks) ou améliore le étape simulaiton, dans notre cadre , les ville est distribué par la loi formé par le softmax qui restreins la l'aléatoirété parce que le différence de distance entre les villes est très grande.donc le probailité de choisir le ville plus proche est très grande , ce qui ne permets pas de sortir de la solution optimale locale.

Finalement, nous avons découvert quelques-unes des façons dont la recherche arborescente de Monte Carlo peut être combinée avec d'autres méthodes. Outre l'algorithme de recherche arborescente de Monte Carlo, il existe d'autres méthodes pour résoudre le problème du voyageur, telles que k-opt et branch and bound. Ces méthodes utilisent différentes stratégies et techniques afin de trouver la solution optimale avec plus de précision au cours du processus de recherche. Dans la pratique, la combinaison des avantages de la recherche arborescente de Monte Carlo et de ces méthodes peut encore améliorer l'efficacité et la qualité de la résolution des problèmes. Dans notre projet, par exemple, nous combinons la méthode k-opt avec la méthode MCTS. Dans cette méthode, nous n'utilisons pas une structure arborescente pour étendre la situation, mais nous utilisons la méthode k-opt pour former un nouvel permutation des villes. Les arêtes entre chaque ville se voient attribuer un score. Cette méthode peut être utilisée pour obtenir de meilleures solutions approximatives, mais elle prendra plus de temps.

Bien entendu, il est pratiquement impossible de converger vers la solution optimale réelle à l'aide des méthodes liées aux MCTS, si ce n'est pendant très longtemps. C'est un inconvénient, mais dans

la pratique, il suffit d'obtenir des solutions optimales approximatives qui ne sont pas très différentes.

Références

[1] https ://fr.wikipedia.org/wiki/Probl%C3%A8me_du_voyageur_de_commerce

[2] Soufia Benhida, Arnaud Knippel, and Ahmed Mir On the probabilistic traveling salesman problem

[3] Zhang-Hua Fu, Kai-Bin Qiu, Meng Qiu, Hongyuan Zha, Targeted sampling of enlarged neighborhood via Monte Carlo tree search for TSP https ://openreview.net/attachment ?id=ByxtHCVKwB&name=original_pdf

[4] https ://fr.wikipedia.org/wiki/S%C3%A9paration_et_%C3%A9valuation

[5] Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., ... & Dieleman, S. (2016). Mastering the game of Go with deep neural networks and tree search. nature, 529(7587), 484-489.

[6] Bravi, L., & Succi, S. (2018). Quantum Monte Carlo tree search. Physical Review E, 98(4), 043305.

[7] https ://en.wikipedia.org/wiki/Monte_Carlo_tree_search

[8] Jaillet, P.: Probabilistic Traveling Salesman Problems. Ph.D. thesis, Massachusetts, Institute of Technology (1985)

[9] Birattari M., Balaprakash P., Stützle T., and Dorigo M. (2007): "Estimation-based local search for stochastic combinatorial optimization". Technical Report [TR/IRIDIA/2007-003]. IRIDIA, Université Libre de Bruxelles, Brussels, Belgium.

[10] Solve Traveling Salesman Problem by Monte Carlo Tree Search and Deep Neural Network Zhihao Xing, Shikui Tu, Lei Xu . Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai, China