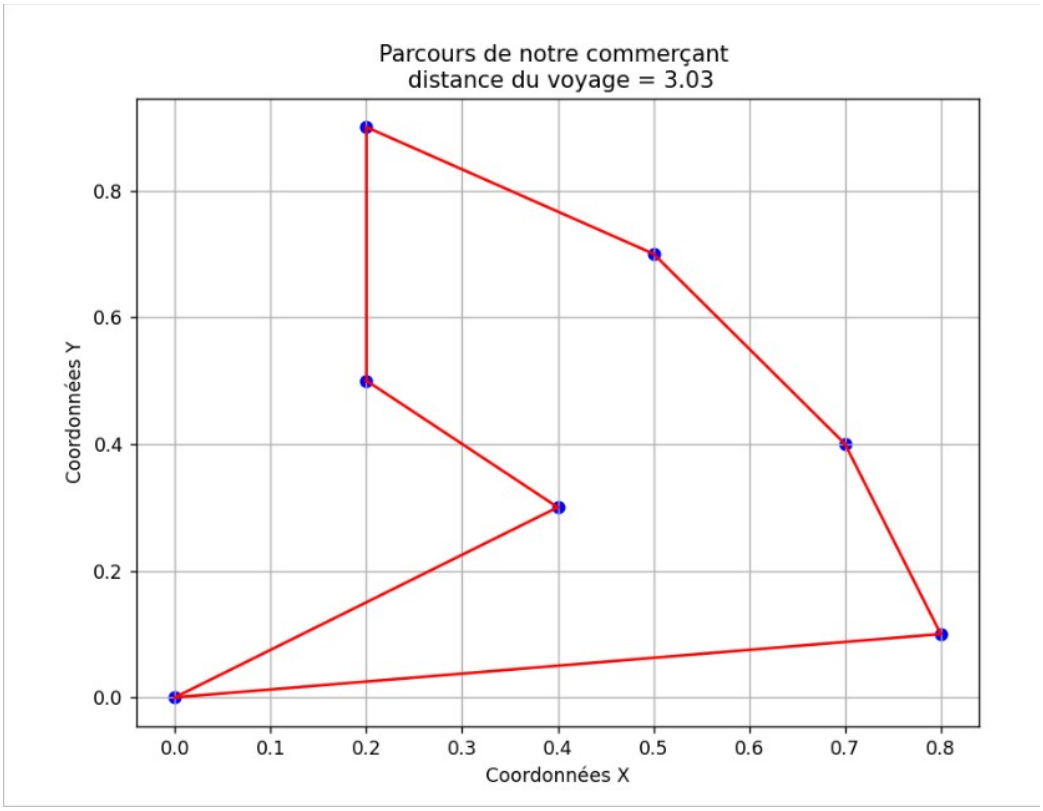
INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES ROUEN

Projet Scientifique Encadré

STPI/P6/2024 – 030

PROBLÈME DU VOYAGEUR DE COMMERCE : approches exactes et heuristiques



Étudiants :

Jules FLAMENT **Raphaël ROGER**

Dylan SANSON **Ambroise ARRIGONI**

Mathis BÔLE-FEYSOT **Amélie MATHIEU**

Enseignant-responsable du projet :

Julien SAUNIER

Cette page est laissée intentionnellement vierge.

Date de remise du rapport : ***13/06/2024***

Référence du projet : ***STPI/P6/2024 – 030***

Intitulé du projet : ***PROBLÈME DU VOYAGEUR DE COMMERCE : approches exactes et heuristiques***

Type de projet : ***Bibliographie et Expérimental***

Objectifs du projet :

Notre projet vise à comparer les différentes méthodes algorithmiques de résolution du problème de voyageur de commerce. Le principal axe de travail est de, à partir d'un échantillon donné de villes, retrouver le chemin le plus court (en terme de distance) qui les relie. Il existe pour cela diverses méthodes algorithmiques déjà implémentées mais toutes imparfaites. On implémentera deux méthodes exactes (le résultat sera assurément le bon, au détriment de la vitesse de calcul) et trois méthodes heuristiques (on aura une approximation du résultat seulement mais avec des méthodes plus rapides). En considérant différents échantillons de villes (de tailles croissantes) nous comparerons l'efficacité des algorithmes : la vitesse de calcul et la précision.

Au-delà du problème de voyageur de commerce à proprement parlé, notre objectif est aussi d'améliorer notre capacité à travailler en groupe et à s'adapter au travail en mode projet sur la période d'un semestre complet.

Mots-clefs du projet : ***algorithme, optimisation, recherche***

TABLE DES MATIÈRES

1. Introduction....5
2. Méthodologie, Organisation du travail et organigramme....6
3. Travail réalisé et résultats : étude des différents algorithmes implémentés et tests....7
3.1. Méthode brute....7
3.2. Méthode Branch and Bound....7
3.3. Méthode du plus proche voisin....8
3.4. Méthode du recuit-simulé....10
3.5. Méthode des colonies de fourmis....13
3.6. Affichages et comparaisons des résultats pour les différents algorithmes....16
4. Conclusions et perspectives....17
4.1. Conclusions sur le travail réalisé....17
4.2. Conclusions sur l'apport personnel de cet E.C. projet....17
4.3. Perspectives pour la poursuite de ce projet....17
5. Bibliographie....18
6. Annexes....19
6.1. Cahier des charges initial....19

1. INTRODUCTION

Le problème du voyageur de commerce, ou TSP en anglais (Travelling Salesman Problem), s'ancre dans la branche des mathématiques qui traite l'optimisation, la combinatoire, et l'informatique théorique. Il s'agit d'un problème, comme celui du sac à dos, dont on ne connaît pas de solution qui soit exacte et qui s'exécute en un temps raisonnable. On doit donc systématiquement faire un compromis, choisir entre rapidité de l'exécution ou exactitude du résultat. Notre objectif est d'explorer, en les programmant, plusieurs méthodes algorithmiques déjà existantes qui visent à résoudre ce problème et d'étudier leurs avantages et inconvénients. Pour cela, nous avons choisi d'utiliser le langage de programmation Python.

Ce problème a un énoncé qui est simple en apparence : étant donné un nombre donné de villes et leurs positions respectives (indiquées par un système de coordonnées), nous devons trouver le trajet qui passe par chacune d'elles, une et une seule fois, en trouvant le chemin le plus court possible. On retrouve alors la notion de graphe hamiltonien : graphe orienté dont on relie chacun des sommets une unique fois.

Nous avons choisi d'implémenter les méthodes de résolution suivantes :

Méthodes exactes :

- Méthode brute
- Branch and Bound

Méthodes heuristiques :

- Plus proches voisins
- Recuit-simulé
- Colonie de fourmis

Notons que le problème du voyageur de commerce fait partie de la catégorie des problèmes mathématiques dits « NP-complet » ou « NPC », c'est-à-dire que :

- on peut vérifier qu'une solution donnée est vraie en un temps polynomial (donc raisonnable). La catégorie de ces problèmes est la catégorie NP.
- On ne peut pas trouver une solution en un temps polynomial.

Étant donné que l'on n'est pas encore capable aujourd'hui de savoir si P = NP (avec P qui désigne l'ensemble des les problèmes mathématiques que l'on peut résoudre en un temps polynomial), alors on sait d'ores-et-déjà que l'on ne peut pas trouver de solution au problème du voyageur de commerce qui s'exécute en un temps acceptable. Cependant, il sera aisé pour nous de vérifier qu'une solution générée par un de nos programmes est correcte.

Il existe des tas d'algorithmes qui ont été développés au fil des années pour répondre à notre énoncé, qui prend son point de départ dans un problème concret du quotidien, mais nous n'en avons sélectionnés que cinq car la plupart des autres solutions ne sont que des combinaisons d'autres algorithmes. Il est par exemple possible de combiner deux méthodes pour en obtenir une troisième. Nous nous sommes concentrés sur des méthodes qui ne sont pas « mixées ».

2. MÉTHODOLOGIE, ORGANISATION DU TRAVAIL ET ORGANIGRAMME

Afin de réaliser ce projet, nous avons divisé notre groupe de six étudiants en deux sous-groupes de trois et nous nous sommes réparti les algorithmes à coder. En plus de la création des programmes informatiques avec les méthodes de résolution, nous nous sommes occupés de la génération des villes avec leurs positions (sous forme de matrices symétriques contenant les distances deux à deux des villes sur le graphe), de la création d'une unité de données (« data ») qui contient les matrices de base et les fonctions qui calculent les distances, ainsi que de l'implémentation d'un programme de test.

La répartition des tâches a donc été telle que :

- Ambroise, Jules et Amélie : création des échantillons de ville, méthode brute, Branch and Bound, colonie de fourmis ;
- Dylan, Mathis, Raphaël : algorithme des plus proches voisins, méthode du recuit-simulé, programme de tests et comparaison des résultats.

Sur l'organigramme : la répartition des algorithmes

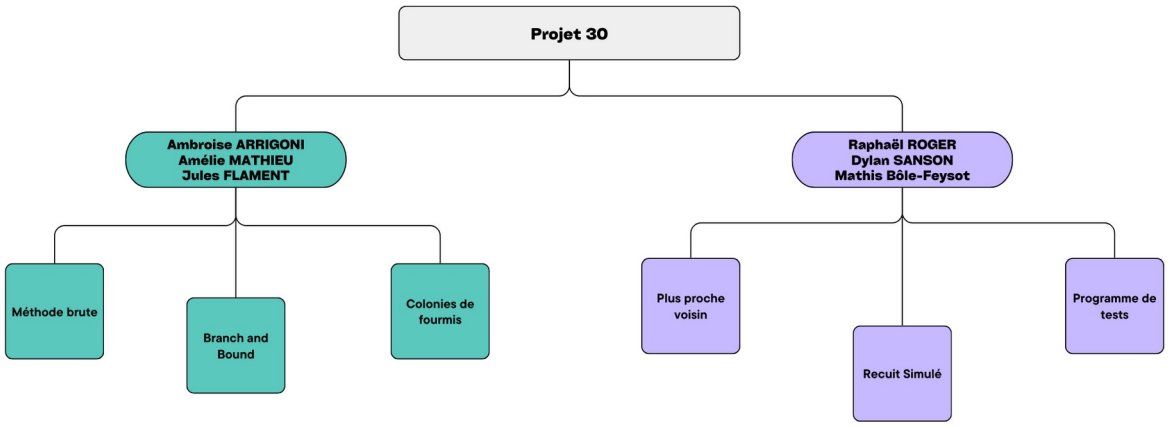


Figure 1: Organigramme pour la répartition des tâches

3. TRAVAIL RÉALISÉ ET RÉSULTATS : ÉTUDE DES DIFFÉRENTS ALGORITHMES IMPLÉMENTÉS ET TESTS

3.1. Méthode brute

La méthode brute est une méthode de résolution exacte, c'est à dire qu'elle trouve la meilleure solution à chaque fois qu'on l'utilise, et à coup sûr. Pour trouver le chemin le plus optimisé, cette méthode parcourt simplement tous les chemins possibles et sélectionne le chemin le plus court. Pour cela, elle commence par créer un arbre de solutions dont chaque branche est un chemin potentiel. Le point de départ du parcours du voyageur est le point culminant de l'arbre, puis on descend au fur et à mesure pour connaître les villes à arpenter. Cette méthode a comme principal avantage de trouver la meilleure solution à chaque fois. Néanmoins, elle a pour désavantage que le temps de calcul augmente de manière exponentielle en fonction du nombre de villes. Cela rend la méthode très coûteuse niveau temps d'exécution.

Il est important de remarquer que le temps mis par le programme pour s'exécuter dépend beaucoup des objets que l'on manipule dans nos fonctions. En effet, une deuxième version du programme nous a permis d'obtenir des résultats plus rapides. Mais, la méthode brute restera toujours, quoi qu'il arrive, la méthode la plus lente.

3.2. Méthode Branch and Bound

La méthode Branch and Bound est, elle aussi, une méthode exacte. Dans cette méthode, la distance totale d'un chemin est décomposée en une somme de distances entre les villes successives.

Voici comment cette méthode fonctionne :

- Calcul initial : Tout d'abord, on calcule la distance d'un chemin aléatoire entre les villes et on retient cette valeur comme une limite supérieure.
- Exploration des chemins : Ensuite, on explore d'autres chemins possibles. Pendant ce processus, si on constate qu'au milieu du calcul, la distance déjà parcourue dépasse la limite supérieure retenue, on abandonne immédiatement ce chemin. Cela permet de ne pas gaspiller du temps de calcul sur des chemins qui ne peuvent pas être optimaux.

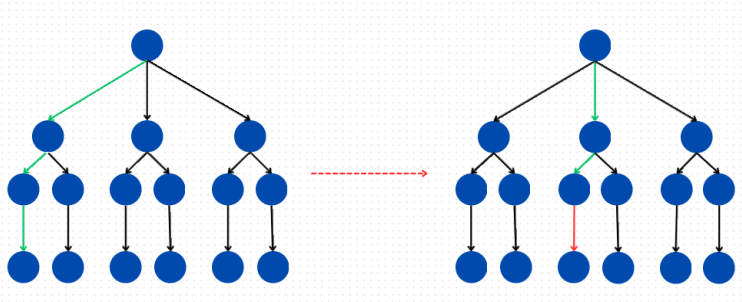


Figure 2: Arbre de décision pour l'algorithme Branch and Bound

Cette méthode a pour principal avantage de toujours trouver la bonne solution et d'être plus rapide que la méthode brute. C'est une version améliorée de cette dernière qui impose une borne supérieure et qui permet d'accélérer grandement les calculs. Malgré tout elle reste encore très lente quand le nombre de villes augmente. S'il s'agit bien d'une amélioration de la méthode brute (qui porte bien son nom), elle n'est cependant pas parfaite.

Dans l'exemple ci-dessus (voir le graphe), on peut voir que le programme calcule la distance d'un chemin à trois villes et le garde en mémoire. Après cela il veut calculer la distance d'un autre chemin mais il s'aperçoit que la distance du deuxième chemin est plus grande ,et ce, dès le parcours de la deuxième ville, il ne fait donc pas le troisième calcul et passe directement à un autre chemin.

Pour la méthode Branch and Bound nous avons implémenté deux approches. Une itérative et une récursive. La différence entre ces deux approches réside dans la gestion des sous-problèmes : l'approche itérative utilise des structures de données explicites, tandis que l'approche récursive utilise la pile d'appels du langage de programmation. Le choix entre les deux dépend souvent du contexte spécifique du problème à résoudre. On peut voir dans le tableau ci-dessous que dans notre cas le Branch and Bound récursif est beaucoup plus rapide que l'approche itérative. La différence se fait vraiment ressentir aux environs de 10 villes.

Nombre de villes :	6	
Méthode :	Branch & Bound Classique	Branch & Bound Récursif
Distance obtenue :	2.34047204616616	2.34047204616616
Temps nécessaire : (en s)	0.007458782196045	0.005049157142639
Écart type sur le temps :	0.00247556567553	0.001847652350122

Nombre de villes :	7	
Méthode :	Branch & Bound Classique	Branch & Bound Récursif
Distance obtenue :	2.31497479603227	2.31497479603227
Temps nécessaire : (en s)	0.054853296279907	0.023373627662659
Écart type sur le temps :	0.016365035896736	0.004266714328196

Nombre de villes :	8	
Méthode :	Branch & Bound Classique	Branch & Bound Récursif
Distance obtenue :	2.95727236934946	2.95727236934946
Temps nécessaire : (en s)	0.363874840736389	0.113714218139648
Écart type sur le temps :	0.013947296301057	0.01930207353523

Nombre de villes :	9	
Méthode :	Branch & Bound Classique	Branch & Bound Récursif
Distance obtenue :	3.02744066406025	3.02744066406025
Temps nécessaire : (en s)	3.18133845329285	0.493498778343201
Écart type sur le temps :	0.262723097360408	0.160355763992331

Nombre de villes :	10	
Méthode :	Branch & Bound Classique	Branch & Bound Récursif
Distance obtenue :	3.15165678429464	3.15165678429464
Temps nécessaire : (en s)	34.2395143508911	3.48181474208832
Écart type sur le temps :	2.1176920614868	1.4929478019784

Figure 3: Comparaison des méthodes itératives et récursives pour le Branch and Bound

3.3. Méthode du plus proche voisin

L'algorithme du plus proche voisin consiste à prendre un point de départ, qu'on appellera « point courant », parmi l'échantillon qui lui est fourni en entrée, et à répéter le processus suivant :

Premièrement, il va calculer la distance entre le point courant et tous les autres points de l'échantillon un par un. Le point se situant à la distance la plus courte devient le nouveau point courant et celui-ci est ajouté au chemin emprunté.

Nous répétons ce processus en changeant le point de départ initial à chaque fois et nous stockons les valeurs des distances de tous les chemins calculés, puis le meilleur résultat c'est-à-dire celui qui a la distance de parcours de tous les points la plus courte, répondant ainsi au problème est gardé et renvoyé.

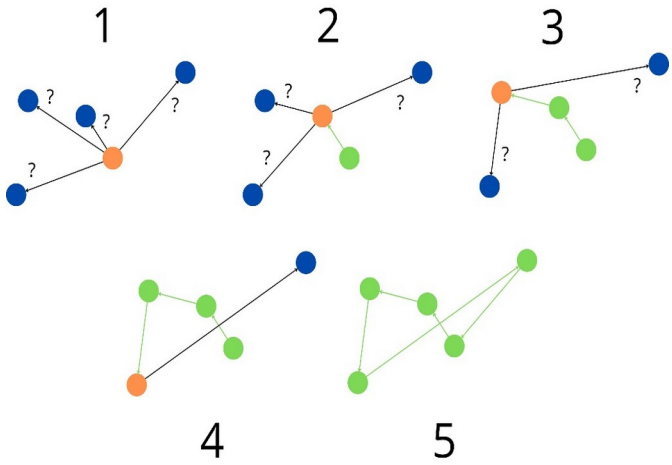


Figure 4: Exemple pour le Plus Proche Voisin

1. Le point courant (en orange) est choisi initialement et les distances aux autres sont calculées.
2. Le point avec la distance la plus courte est sélectionné et ajouté au chemin (symbolisé par la couleur verte) puis devient le nouveau point courant.
3. On répète l'étape 2.
4. Le dernier point est relié.
5. On revient au point de départ.

Finalement, on répète ce processus en prenant chaque point de l'échantillon comme nouveau point de départ. Le chemin le plus court parmi ces derniers ainsi que sa distance est renvoyée.

Lors de l'exécution du programme « resultat_moyenne.py » en utilisant la méthode du plus proche voisin avec les paramètres suivants :

nombre_de_point_min=10 ;
nombre_de_point_max=150 ;
nombre_de_fois=10 ;
nombre_de_pas=10 ;

c'est-à-dire, en calculant la moyenne sur 10 échantillons pour des ensembles allant de 10 à 150 villes, par paliers de 10, nous obtenons le graphique ci-dessous :

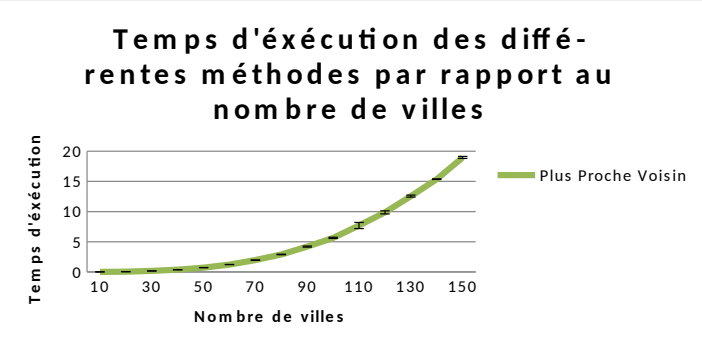


Figure 5: Temps d'exécution du Plus Proche Voisin / nombre de villes

Cette méthode du Plus Proche Voisin est donc préférable lorsque nous avons un très grand nombre de villes car nous pouvons obtenir un très bon résultat en très peu de temps (exprimé ici en secondes).

3.4. Méthode du recuit-simulé

Le recuit simulé est une méthode d'optimisation inspirée du processus physique de recuit en métallurgie, où un matériau est chauffé puis refroidi lentement pour minimiser les défauts de sa structure. Nous avons implémenté cette technique pour résoudre le problème du voyageur de commerce. Le recuit simulé applique un processus de recherche locale par des probabilités pour explorer l'espace des solutions possibles.

Voici les étapes principales de la méthode :

- Commencer avec une solution initiale (dans notre cas, nous appelons l'algorithme du plus proche voisin)
- Définir une température initiale élevée qui permet de diversifier les solutions explorées
- Répétition de l'algorithme :
  - Génération de voisinage : Générer une nouvelle solution aléatoire en modifiant légèrement la solution actuelle, en inversant l'ordre d'une séquence de villes
  - Évaluation : Calcul la différence de coût entre la nouvelle solution et la solution actuelle.
  - Décision d'acceptation :
    - Si la nouvelle solution est meilleure (réduction du coût), on l'accepte
    - Si la nouvelle solution est pire (augmentation du coût), on l'accepte avec une certaine probabilité p (critère de Metropolis) définie par la fonction d'acceptation : $p = e^{-\frac{\Delta E}{T}}$, où ΔE est la variation du coût et T est la température actuelle
  - Mise à jour de la température : Diminuer lentement la température selon une fonction de refroidissement, $T_{new} = alpha \times T$, avec $(0 < alpha < 1)$
- Arrêt : Répéter les étapes jusqu'à ce que la température atteigne un seuil minimal $T < epsilon$

Exemple de génération et acceptation d'une solution voisine :

- Inversion d'un sous-ensemble de 7 points, solution moins coûteuse donc acceptée :

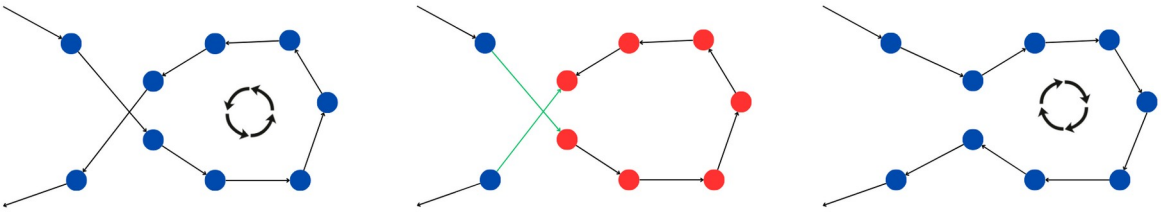


Figure 6: Exemple 1 du recuit simulé

- Inversion d'un sous-ensemble de 3 points, solution plus coûteuse donc accepté avec une probabilité p :

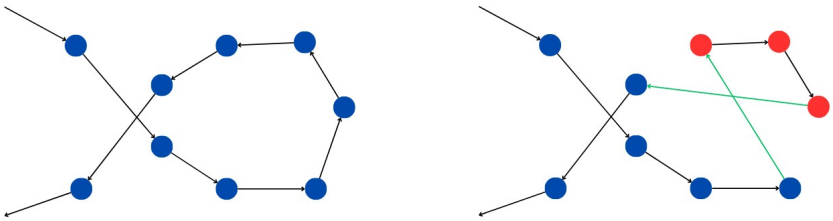


Figure 7: Exemple 2 du recuit simulé

Différentes méthodes implémentées :

Nous avons utilisé différentes méthodes pour générer des solutions voisines. Dans l'exemple précédent, nous avons appliqué l'inversion d'un sous-ensemble de villes. Nous avons également expérimenté avec la permutation de deux villes aléatoires et la permutation de deux villes adjacentes. Après plusieurs tests, il s'est avéré que l'inversion de sous-ensembles est la méthode la plus efficace.

Décision d'acceptation :

Le critère de Metropolis permet d'accepter parfois des solutions moins bonnes, ce qui favorise l'exploration d'un éventail plus large de solutions. Ainsi, il aide à éviter de se retrouver piégé dans des minima locaux qui ne correspondent pas à la solution optimale, mais dont les solutions voisines sont moins bonnes.

Analyse des paramètres de l'algorithme de recuit simulé :

- Solution initiale

La solution initiale dans l'algorithme de recuit simulé peut grandement influencer l'efficacité et la qualité des résultats obtenus. Une méthode couramment utilisée pour générer cette solution initiale est l'algorithme du plus proche voisin. Cette approche simple et rapide fournit une solution initiale raisonnable, souvent meilleure que des solutions générées aléatoirement. Cependant, elle peut parfois conduire à des parcours sous-optimaux. Utiliser cette méthode comme point de départ permet à l'algorithme de recuit simulé de partir d'une base relativement bonne et d'améliorer progressivement la solution en explorant des permutations plus avantageuses.

- Température Initiale

La température initiale est un paramètre crucial dans l'algorithme de recuit simulé. Une température trop élevée peut rendre l'algorithme inefficace en acceptant trop de solutions non optimales, tandis qu'une température trop basse peut limiter l'exploration de l'espace de solutions et entraîner une convergence prématurée vers des minima locaux. Idéalement, la température initiale doit être suffisamment élevée pour permettre une exploration large mais contrôlée.

- Facteur de Refroidissement

Le facteur de refroidissement détermine la vitesse à laquelle la température diminue au cours du temps. Un facteur de refroidissement trop rapide (α proche de 0) peut conduire à une exploration insuffisante, tandis qu'un refroidissement trop lent (α proche de 1) peut prolonger inutilement le processus.

- Seuil de Température

Le seuil de température est le point auquel l'algorithme s'arrête. Une température trop élevée comme seuil peut arrêter l'algorithme avant qu'il ait trouvé une solution de qualité, tandis qu'un seuil trop bas peut prolonger le calcul sans amélioration significative. Le seuil doit être fixé suffisamment bas pour garantir une bonne solution tout en évitant un temps de calcul excessif.

Nous avons donc réglé les paramètres, après plusieurs tests, comme suit :

- Température initiale : 10000
- Facteur de refroidissement : 0,99
- Seuil de température : 0,00001

Il est important de noter que les valeurs optimales pour ces paramètres dépendent souvent du problème spécifique et de la taille de l'instance du TSP. Par conséquent, des tests expérimentaux et des ajustements itératifs sont nécessaires pour déterminer les meilleurs paramètres pour un cas donné. En résumé, la combinaison de la température initiale, du facteur de refroidissement et du seuil de température doit être soigneusement équilibrée pour assurer une exploration efficace de l'espace de solutions, tout en convergeant vers une solution de haute qualité dans un temps (à peu près) raisonnable.

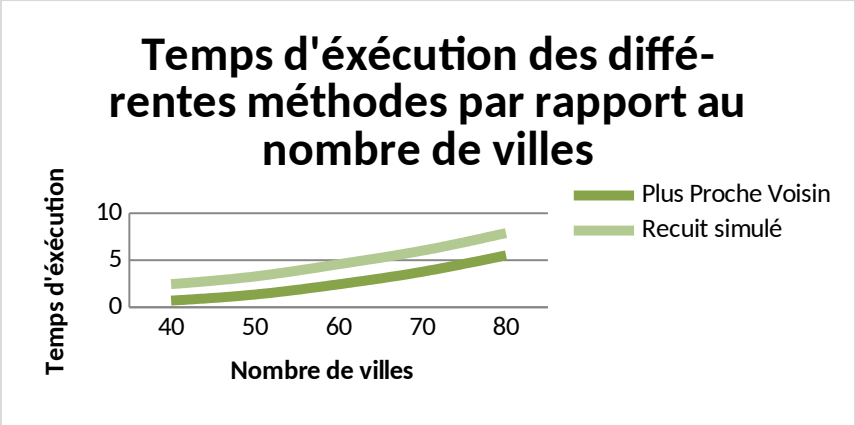


Figure 8: Temps d'exécution du recuit simulé et du plus proche voisin / nombre de villes

On remarque que la méthode du recuit simulé prend plus de temps que le plus proche voisin, et ce temps supplémentaire semble constant. Cela s'explique par le fait que la solution initiale du recuit simulé est le chemin renvoyé par le plus proche voisin, mais aussi par les nombreuses itérations et l'exploration exhaustive de l'espace de solutions. En prenant plus de temps, il tend à produire des solutions de meilleure qualité grâce à sa capacité à accepter temporairement des solutions moins bonnes pour échapper aux minima locaux.

3.5. Méthode des colonies de fourmis

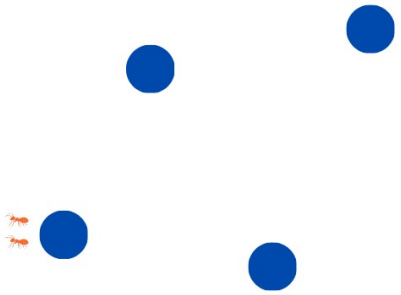
L'algorithme *colonie de fourmis* est une méthode heuristique qui s'inspire du comportement des fourmis recherchant un chemin entre leur colonie et une source de nourriture.

Un modèle expliquant ce comportement est le suivant :

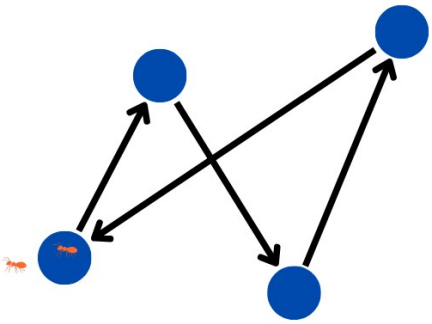
1. une fourmi parcourt plus ou moins au hasard l'environnement autour de la colonie ;
2. si celle-ci découvre une source de nourriture, elle rentre plus ou moins directement au nid, en laissant sur son chemin une piste de phéromones ;
3. ces phéromones étant attractives, les fourmis passant à proximité vont avoir tendance à suivre, de façon plus ou moins directe, cette piste ;
4. en revenant au nid, ces mêmes fourmis vont *renforcer* la piste ;
5. si deux pistes sont possibles pour atteindre la même source de nourriture, celle étant la plus courte sera, dans le même temps, parcourue par plus de fourmis que la longue piste ;
6. la piste courte sera donc de plus en plus renforcée, et donc de plus en plus attractive ;
7. la longue piste, elle, finira par disparaître, les phéromones étant volatiles ;
8. à terme, l'ensemble des fourmis a donc déterminé et «choisi» la piste la plus courte.

Ainsi, notre algorithme s'inspire de ce modèle en implémentant les phéromones pour le problème du voyageur de commerce. Pour expliquer notre algorithme, nous allons l'illustrer avec un exemple.

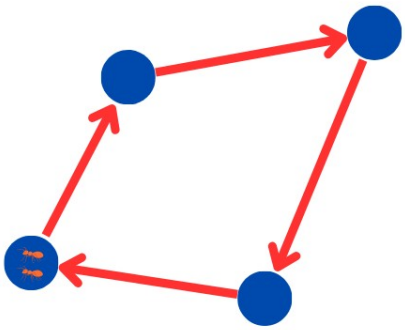
Prenons une première génération de fourmis, constitué de deux fourmis, qui doivent parcourir 4 villes représentées en bleu sur le schéma ci-dessous.



Nous avons une première fourmi qui va aléatoirement proposer une solution :



La deuxième fourmi parcourir aussi aléatoirement les 4 villes, et propose une autre solution:



La première génération a proposé ses solutions (parcours rouge/noir), maintenant les fourmis vont poser leur phéromones. La quantité de phéromones dépend de la qualité de la solution, dès lors nous implémentons la relation suivante:

$$Phéromones\,déposés = \frac{1}{Distance\,du\,parcours}$$

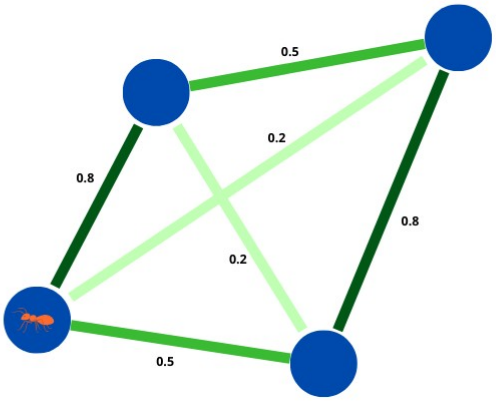
Ainsi, plus le chemin réalisé est long, moins de phéromones seront déposés (fonction décroissante). En reprenant notre exemple, on remarque que le parcours rouge est plus court que le parcourt noir, donc :

Phéromones déposés < **Phéromones déposés**

Chacune des deux fourmis dépose ainsi ses phéromones sur chaque arête de son parcours, les arêtes qu'elles auront en commun auront donc une quantité plus importante de phéromones.

Maintenant, la prochaine génération va parcourir à nouveau les 4 villes, mais cette fois, ses choix seront influencés par la présence plus ou moins importante de phéromones. Son parcours restera aléatoire, mais la fourmi aura plus de chance d'emprunter une arête avec beaucoup de phéromones.

Ainsi, grâce à l'aléatoire, la fourmi peut continuer à proposer de nouvelles solutions, sans forcément prendre exactement le même parcours rouge que son prédécesseur.



Pour éviter que les fourmis empruntent systématiquement les mêmes arêtes que la première génération, sans forcément chercher à explorer d'autres possibilités sur des arêtes qui n'avaient pas été explorées, nous avons ajouté une évaporation d'une partie des phéromones entre chaque génération, pour inciter à l'exploration.

Ainsi, nous avons créé un programme qui applique **2 X nombre_villes** de fourmis par génération sur **500** générations par défaut. Nous nous sommes formés sur les bases de la programmation orienté objet, pour coder les fourmis (sous forme de classe) et leur comportement.

L'avantage de cette algorithme est qu'il a un plutôt bon rapport temps/qualité, de plus, nous pouvons choisir si nous préférons privilégier le temps de calcul ou bien la qualité de la solution, en ayant la main sur le nombre de fourmis par génération, et le nombre de générations. Nous pouvons le constater sur le graphe suivant :

Nombre de villes :	**15**		
Méthode :	**500 génération, 2 fourmis**	**250 génération, 2 fourmis**	**250 génération, 4 fourmis**
Distance obtenue :	3,76813923286714	3,75394007583379	3,73844571216615
Temps nécessaire : (en s)	6,3625433921814	3,18717844486237	6,13661847114563
Écart type sur le temps :	0,0766580846083755	0,0457290534810313	0,0906624883727709

Nombre de villes :	**20**		
Méthode :	**500 génération, 2 fourmis**	**250 génération, 2 fourmis**	**250 génération, 4 fourmis**
Distance obtenue :	4,68408572184083	4,72357165319461	4,64610159233876
Temps nécessaire : (en s)	12,8720802307129	6,4325261592865	12,5493963718414
Écart type sur le temps :	0,117035399811038	0,0817780387192142	0,0993167610543637

Figure 9: Quelques résultats pour notre colonie de fourmis

3.6. Affichages et comparaisons des résultats pour les différents algorithmes

Le programme « resultat_moyenne.py » permet d'afficher les résultats des différentes méthodes. En utilisant la bibliothèque « openpyxl » disponible sur Python, nous créons un fichier Excel qui affiche les différents résultats en fonction de diverses variables.

- « nombre_de_point_min » et « nombre_de_point_max » déterminent le nombre de villes générées aléatoirement sur lesquelles tester les méthodes.

- « nombre_de_fois » indique le nombre d'échantillons différents utilisés pour calculer les moyennes.

- « nombre_de_pas » représente l'écart entre chaque itération.

- 'fonction' répertorie les différentes fonctions utilisées pour la comparaison.

La bibliothèque « openpyxl » permet également de styliser la page Excel pour améliorer la lisibilité des résultats. Ainsi, la mise en page a été effectuée directement via Python, sans avoir besoin d'ouvrir Excel.

Le programme créée ensuite un graphique Excel qui trace le temps d'exécution en fonction du nombre de villes.

Calculer l'écart type des temps d'exécution des différentes méthodes permet d'évaluer la variabilité et la stabilité des performances des algorithmes. « Openpyxl » ne permettant pas d'ajouter des barres d'erreurs à un graphique, celles-ci ont été ajoutés manuellement.

Voici un exemple d'un tableau Excel crée par ce programme :

Nombre de villes :	6				
Méthode :	Méthode Brute	Branch And Bound	Plus Proche Voisin	Recuit Simulé	Colonie de Fourmis
Distance obtenue :	2,322163576	2,322163576	2,335729443	2,322163576	2,339531795
Temps nécessaire : (en s)	0,002050853	0,003958273	0,001702237	0,080752158	0,394234133
Écart type sur le temps :	0,000568023	0,00048321	0,000600634	0,008731881	0,014122247
Nombre de villes :	7				
Méthode :	Méthode Brute	Branch And Bound	Plus Proche Voisin	Recuit Simulé	Colonie de Fourmis
Distance obtenue :	2,937396925	2,937396925	3,025837997	2,937396925	2,965167753
Temps nécessaire : (en s)	0,015470004	0,021948719	0,002360487	0,078986216	0,5597615
Écart type sur le temps :	0,001910242	0,00296171	0,000560983	0,003443676	0,018642244
Nombre de villes :	8				
Méthode :	Méthode Brute	Branch And Bound	Plus Proche Voisin	Recuit Simulé	Colonie de Fourmis
Distance obtenue :	3,006322206	3,006322206	3,094684183	3,006322206	3,052185065
Temps nécessaire : (en s)	0,141819501	0,112785888	0,003832841	0,082154417	0,850626659
Écart type sur le temps :	0,015017574	0,02756908	0,000850797	0,004763749	0,064926711
Nombre de villes :	9				
Méthode :	Méthode Brute	Branch And Bound	Plus Proche Voisin	Recuit Simulé	Colonie de Fourmis
Distance obtenue :	3,201677266	3,201677266	3,282948321	3,204406182	3,227476603
Temps nécessaire : (en s)	1,357689071	0,620363951	0,005457258	0,083726883	1,055830479
Écart type sur le temps :	0,026009476	0,12233275	0,001020581	0,005523807	0,041146711
Nombre de villes :	10				
Méthode :	Méthode Brute	Branch And Bound	Plus Proche Voisin	Recuit Simulé	Colonie de Fourmis
Distance obtenue :	3,310311464	3,310311464	3,358900218	3,324758989	3,32372459
Temps nécessaire : (en s)	14,58923404	2,901302195	0,007388663	0,085651064	1,417453527
Écart type sur le temps :	0,161037901	1,448123449	0,001298132	0,005409584	0,130943163

Figure 10: Comparaison des résultats pour tous les algorithmes

4. CONCLUSIONS ET PERSPECTIVES

4.1. Conclusions sur le travail réalisé

En conclusion de ce projet, on peut tout d'abord souligner le fait que le cahier des charges que nous avions fixé au début de ce semestre a été totalement complété (hormis concernant les fonctionnalités optionnelles), et nous avons en plus de cela correctement respecté les délais que nous nous étions fixés. Toutes les méthodes implémentées ont abouties à des résultats satisfaisants. La comparaison des résultats au fur et à mesure de l'avancement nous a permis d'améliorer nos méthodes en conséquence, tout au long du projet.

4.2. Conclusions sur l'apport personnel de cet E.C. projet

Le fait d'avoir été confronté à cet énoncé connu des problèmes d'informatique et d'optimisation, nous a beaucoup apporté dans le cadre de notre cursus scolaire à l'INSA car, par exemple, nous avons pu entendre parler du voyageur de commerce en cours de l'EC de i4 de ce semestre. Cela nous a également permis de découvrir une multitude de méthodes pour répondre à un même problème et nous avons pu nous rendre compte de la richesse de la recherche actuelle sur ce sujet.

En plus de cela, certains algorithmes que nous avons utilisés étaient assez complexes et une grande étape de compréhension de ceux-ci a été nécessaire, nous apportant ainsi des nouvelles façon d'appréhender les problèmes et de la réflexion vis-à-vis de ceux-ci à travers un travail bibliographique notamment.

4.3. Perspectives pour la poursuite de ce projet

Comme tous les problèmes NP-complet actuels tels que celui que nous avons étudié durant ce semestre avec le voyageur de commerce, il reste un grand travail de recherche à faire pour répondre à la question plus précisément. Savoir si P est effectivement égal à NP nous indiquerait que nous pouvons trouver une solution en un temps rapide, et nous assurerait par la même occasion que nous ne l'avons pas encore trouvée. Cependant, si P est différent de NP alors c'est que nous devrons nous contenter de solutions soit trop longues à implémenter soit approximatives.

En attendant, la recherche de l'algorithme le plus exact et le moins coûteux en temps de calcul reste encore à trouver. On peut améliorer les résultats en combinant des algorithmes qui existent déjà, ou en en inventant d'autres encore.

On a vu au cours de ce projet que, à nombre de villes changeant, le meilleur algorithme est changeant lui aussi. C'est-à-dire que l'on ne peut pas, à l'heure actuelle, désigner un seul algorithme comme étant le « meilleur de tous ». En effet, chacun a ses avantages et ses inconvénients, et l'important est de bien comprendre chaque méthode afin de déterminer comment les utiliser au mieux.

5. BIBLIOGRAPHIE

[1] https://interstices.info/le-probleme-du-voyageur-de-commerce/ (valide à la date du 10/06/2024)

[2] https://fr.wikipedia.org/wiki/Probl%C3%A8me_du_voyageur_de_commerce#Heuristiques_gloutonnes (valide à la date du 10/06/2024)

[3] https://fr.vikidia.org/wiki/Probl%C3%A8me_du_voyageur_de_commerce (idem)

[4] https://www.cari-info.org/actes2006/144.pdf (idem)

[5] https://vixra.org/pdf/1910.0534v1.pdf (idem)

[6] http://www.malaspinas.academy/prog_seq/exercices/09_voyageur_commerce/index.html (idem)

[7] https://antoinevastel.com/algorithme/python/algorithmes%20g%C3%A9n%C3%A9tiques/2016/04/30/probleme-voyageur-commerce.html (idem)

[8] https://www.researchgate.net/publication/358600281_Probleme_du_Voyageur_de_Commerce_PVC_Etude_theorique_du_probleme_Etude_Experimentale (idem)

[9] https://perso.eleves.ens-rennes.fr/people/julie.parreaux/fichiers_agreg/info_dev/ApproximationTSP.pdf (idem)

[10] https://wikimemoires.net/2014/02/la-probleme-de-voyageurs-de-commerce-bi-objectifs-et-la-metaheuristique/ (idem)

[11] https://archipel.uqam.ca/4339/1/M12267.pdf (idem)

[12] https://webia.lip6.fr/~fouilhoux/MAOA/Files/MAOA_ROOC_branch_slides.pdf (idem)

6. ANNEXES

6.1. Cahier des charges initial

Le problème du voyageur de commerce

cahier des charges

Description:

En informatique théorique, le problème du voyageur de commerce, est un problème d'optimisation qui consiste à déterminer, étant donné un ensemble de villes, le plus court circuit passant par chaque ville une seule fois.

Ainsi, on va chercher à utiliser deux types d'approches: exacte et heuristique.

La méthode exacte consistera à fournir le meilleur résultat au dépit du temps de calcul. Tandis que l'approche heuristique mettra en avant l'optimisation du temps de calcul tout en fournissant un résultat approché.

L'objectif de notre projet est donc de comparer les méthodes que nous allons implémenter sur différents aspects comme : la complexité et la précision du résultat.

Fonctionnalités attendues:

On souhaite implémenter ces différentes fonctionnalités dans notre projet :

Méthodes exactes :

-- la méthode brute

-- la méthode Branch and Bound

Méthodes heuristiques :

-- algorithme glouton (plus proche voisin)

-- algorithme recuit simulé

-- algorithme colonie de fourmis

Fonctionnalités optionnelles :

-- algorithme génétique

-- algorithme Lin-Kernighan

Versions:

Version 1: 11/03/2024

-- Échantillons de villes (Ambroise / Amélie / Jules)
-- algorithme glouton (plus proche voisin) (Dylan / Raphaël / Mathis)
-- la méthode brute (Ambroise / Amélie / Jules)

Version 2: 08/04/2024

-- la méthode Branch and Bound (Ambroise / Amélie / Jules)
-- algorithme de tests (Raphaël / Dylan / Mathis)
-- génération des villes aléatoires (Ambroise / Amélie / Jules)

Version 3: 13/05/2024

-- algorithme recuit-simulé (Raphaël / Dylan / Mathis)
-- algorithme colonie de fourmis (Ambroise / Amélie / Jules)

Version finale : 27 mai ou 3 juin

-- Exploitations des résultats / test