1 Introduction

En informatique, le **problème du voyageur de commerce**, ou **problème du commis voyageur**, 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-circuit qui passe par chaque ville une et une seule fois.

Malgré la simplicité de l'énoncé, on ne connaît pas d'algorithme permettant de trouver une solution exacte rapidement dans tous les cas. Plus précisément, on ne connaît pas d'algorithme en temps polynomial, et la version décisionnelle du problème du voyageur de commerce (*pour une distance D, existe-t-il un chemin plus court que D passant par toutes les villes et qui termine dans la ville de départ ?*) est un problème NP-complet, ce qui est un indice de sa difficulté.

C'est un problème algorithmique célèbre, qui a donné lieu à de nombreuses recherches et qui est souvent utilisé comme introduction à l'algorithmique ou à la théorie de la complexité. Il présente de nombreuses applications, que ce soit en planification, en logistique ou dans des domaines plus éloignés, comme la génétique (en remplaçant les villes par des gènes et la distance par la similarité).

1.1 Etat de l'art

Le problème du voyageur de commerce (TSP : Travelling Salesperson Problem) est l'un des problèmes d'optimisation combinatoire les plus répandus. Étant donné un graphe complet

G= (V, E) et une fonction de poids w:E→N.Lebut est de trouver un cycle Hamiltonien dans

G (également appelé un tour) de poids minimum. C'est l'un des problèmes centraux de l'informatique et de la recherche opérationnelle. Il est bien connu d'être NP-difficile et a fait l'objet

de recherches selon différentes perspectives, notamment par approximation, algorithmes de temps exponentiel et heuristiques.

En pratique, le TSP est souvent résolu au moyen d'heuristiques de recherche locales, dans lesquelles on part d'un cycle Hamiltonien arbitraire en G, puis on modifie le cycle au moyen de modifications locales en une série d'étapes. Après chaque étape, le poids du cycle devrait s'améliorer; lorsque l'algorithme ne trouve aucune amélioration, il s'arrête. L'un des exemples les plus réussis de cette approche est l'heuristique k−opt, dans laquelle un k− mouvement amélioré est effectué à chaque étape. Soit un cycle Hamiltonien H dans un graphe G= (V, E) un k−mouvement est une opération qui supprime k arêtes de H et ajoute k arêtes de G de sorte que l'ensemble des arêtes résultant H0 est un nouveau cycle Hamiltonien. Le k−mouvement s'améliore si le poids de H0est plus petit que le poids de H.

Les lignes d'attaque traditionnelles pour les problèmes NP-difficiles sont les suivantes:

•Concevoir des algorithmes exacts, qui fonctionnent raisonnablement vite que pour des problèmes de petite taille.

• Concevoir des algorithmes "sous-optimaux" ou heuristiques, c'est-à-dire des algorithmes qui fournissent des solutions approchées dans un délai raisonnable.

• Recherche de cas spéciaux pour le problème ("sous-problèmes") pour lesquels des heuristiques optimales ou exactes sont possibles.

Définition (Cycle Hamiltonien)

Un cycle Hamiltonien est un cycle dans un graphe passant par tous les sommets une fois .

1.2 Problème formel

On peut décrire formellement le problème TSP comme suit :

TRAVELING SALESMAN PROBLEM
Entrée : Un ensemble de villes $V = \{v_1, \cdots, v_n\}$, une distance $d(i,j)$ pour toute paire $i, j \in \{1, \cdots, n\}, i \neq j$
Sortie : Une permutation ϕ de $\{1, \cdots, n\}$
But : Minimiser $\left(\sum_{i=1}^{n-1} d(\phi(i), \phi(i+1))\right) + d(\phi(n), \phi(1))$

1.3 Complexités

• Complexité de TSP naïf : O(n!)

• Complexité de TSP DP : O(n2·2n)

•

Comparaison de temps d'exécution selon la complexité de l'algorithme, pour une machine effectuant un million d'opérations par seconde :

	Taille de l'instance (n)				
Fonction	20	30	40	50	60
n	0,00002 sec.	0,00003 sec.	0,00004 sec.	0,00005 sec.	0,00006 sec.
n^2	0,0004 sec.	0,0009 sec.	0,0016 sec.	0,0025 sec.	0,0036 sec.
n^3	0,008 sec.	0,27 sec.	0,064 sec.	0,125 sec.	0,216 sec.
n^5	3,2 sec.	24,3 sec.	1,7 min.	5,2 min.	13 min.
2^n	1 sec.	17,9 min.	12,7 jours	35,7 jours	366 siècles
3^n	58 min.	6,5 années	3855 siècles	2×10^8 siècles	$1{,}3 \times 10^{13}$ siècles

2 Algorithmes

2.1 Algorithme TSP Naïf

Algorithm 1 TSP Naif

Require: Nombre de villes n et une liste de coûts $c(i,j)_{i,j=1\cdots n}$ (Nous commençons par la ville numéro 1).
Ensure: Vecteur de villes et coût total

(* Valeurs de départ *)
$C = 0$
$Cost = 0$
$Visites = 0$
$e = 1$ (* e = pointeur de la ville visitée)
(* détermination du tour et du coût)
for $r = 1$ to $n - 1$ **do**
  choisir le pointeur j avec :
  minimum $= c(e,j) = \min\{c(e,k); visites(k) = 0\, et\, k = 1,\ldots,n\}$
  $cost = cost + minimum$
  $e = j$
  $C(r) = j$
end for
$C(n) = 1$
$cost = cost + c(e,1)$

2.2 Performances : TSP_Naif_CPP

```
temps_cpp<-function(f,n){
  s0 <- Sys.time()
  f(n,distance(n))
  s1 <- Sys.time()
  print(c("Nombre de Villes:",n,"Time (s)=",s1 - s0))
  }

for (n in 5:14){temps_cpp(TSP_Naif_CPP,n)}

>>>
[1] "Nombre de Villes:"   "5"              "Time (s)="           "0.00399494171142578"
[1] "Nombre de Villes:"   "6"              "Time (s)="           "0.00300312042236328"
[1] "Nombre de Villes:"   "7"              "Time (s)="           "0.0019960403442382 8"
[1] "Nombre de Villes:"   "8"              "Time (s)="           "0.00299787521362305"
[1] "Nombre de Villes:"   "9"              "Time (s)="           "0.00399899482727051"
[1] "Nombre de Villes:"  "10"              "Time (s)="          "0.0159909725189209"
[1] "Nombre de Villes:" "11"              "Time (s)="         "0.129922866821289"
[1] "Nombre de Villes:" "12"              "Time (s)="         "1.44419407844543"
[1] "Nombre de Villes:" "13"              "Time (s)="         "18.9941418170929"
[1] "Nombre de Villes:" "14"              "Time (min)="         "4.72061628500621"
```

3 - Algorithme du plus proche voisin

Cet article traite d'un algorithme d'approximation pour résoudre le problème du vendeur itinérant. Pour d'autres utilisations, voir Voisin le plus proche. Algorithme du plus proche voisin

L'algorithme du plus proche voisin a été l'un des premiers algorithmes utilisés pour résoudre approximativement le problème du vendeur itinérant. Dans ce problème, le vendeur commence dans une ville aléatoire et visite à plusieurs reprises la ville la plus proche jusqu'à ce que toutes aient été visitées. L'algorithme donne rapidement un court tour, mais généralement pas optimal.

Voici les étapes de l'algorithme :

La séquence des sommets visités est la sortie de l'algorithme.

L'algorithme du plus proche voisin est facile à mettre en œuvre et s'exécute rapidement, mais il peut parfois manquer des itinéraires plus courts qui sont facilement remarqués avec la perspicacité humaine, en raison de sa nature « gourmande ». En tant que guide général, si les dernières étapes de la visite sont comparables en longueur aux premières étapes, alors la visite est raisonnable ; s'ils sont beaucoup plus grands, il est probable qu'il existe de bien meilleures visites. Une autre vérification consiste à utiliser un algorithme tel que l'algorithme de la limite inférieure pour estimer si cette visite est assez bonne.

Dans le pire des cas, l'algorithme donne une visite beaucoup plus longue que la visite optimale. Pour être précis, pour chaque constante **r**, il existe une instance du problème du vendeur itinérant telle que la durée de la visite calculée par l'algorithme du voisin le plus proche est supérieure **à r** fois la durée de la visite optimale. De plus, pour chaque nombre de villes, il existe une assignation de distances entre les villes pour lesquelles l'heuristique du voisin le plus proche produit le pire tour possible. (Si l'algorithme est appliqué sur chaque sommet comme sommet de départ, le meilleur chemin trouvé sera meilleur qu'au moins N/2-1 autres tours, où N est le nombre de sommets.)

L'algorithme du voisin le plus proche peut ne pas trouver de visite réalisable du tout, même lorsqu'elle existe.

3.1 Implémentation avec Python

```python
def GenerateSolutions(self, numberOfSolutions):
    solutions = []
    for i in range(numberOfSolutions):
        solution = []
        isCityAdded = []
        firstCityIndex = 0
        lastCityAdded = None
        #les villes ajoutees =>false
        self.InitTabWithValue(isCityAdded, self.numberOfCities, False)
        #Renvoie n'importe quel entier aléatoire de 0 à self.numberOfCities - 1
        firstCityIndex = random.randint(0, self.numberOfCities - 1)

        solution.append(firstCityIndex)
        lastCityAdded = self.cities[firstCityIndex]
        isCityAdded[firstCityIndex] = True

        while self.AllCitiesAdded(isCityAdded) == False:
            bestDistance = sys.maxsize #donne une taille maximum
            tmpDistance = 0
            bestCityIndex = 0

            for i in range(self.numberOfCities):
                if isCityAdded[i] == False:
                    tmpDistance = self.cities[i].GetDistanceToCity(lastCityAdded)
                    if tmpDistance < bestDistance:
                        bestDistance = tmpDistance
                        bestCityIndex = i


            solution.append(bestCityIndex)
            lastCityAdded = self.cities[bestCityIndex]
            isCityAdded[bestCityIndex] = True


        solutions.append(solution)

    return solutions
```

4 - Algorithme d'insertion heuristique de plus proche voisin :

Toutes les heuristiques de cette classe, initier le processus de construction de la tournée avec un

« seed tour » composé soit d'un seul nœud avec une boucle automatique, soit d'une boucle

Impliquant seulement deux nœuds. (La méthode de sélection de ces deux nœuds initiaux est la suivante :

C'est en soi une question de choix.) Il est utile de définir le coût, c(i, k, j) de l'insertion d'un nouveau nœud, k, dans une visite entre deux nœuds, i et j, qui étaient auparavant adjacents dans la tournée :

c(i, k, j) = d(i, k) + d(k, j) - d(i, j).

Nous avons ensuite :

4-a Heuristique d'insertion aléatoire : le nœud suivant pour rejoindre la visite, T, est sélectionné aléatoirement parmi les nœuds encore dans (N - T) où N est l'ensemble des nœuds du réseau. L'emplacement où le nœud sélectionné est inséré est celui qui minimise c(i, k, j). La procédure est répétée jusqu'à ce que tous les nœuds aient été insérés dans T.

4-b Heuristique d'insertion la plus proche : le nœud suivant à rejoindre la visite, T, est le

Celui qui minimise d (k, T) parmi tous les nœuds k E (N-T). (Nous utilisons la notation d (k, T) pour la distance entre un nœud k et le nœud dans l'ensemble T qui est

Le plus proche de k.) L'emplacement où le nœud sélectionné est inséré est celui qui Minimise c (i, k, j). La procédure est répétée jusqu'à ce que tous les nœuds aient été insérés dans T.

4-c Heuristique d'insertion la plus éloignée : le nœud suivant à rejoindre la tournée, T, est le celui qui maximise d (k, T) parmi tous les nœuds k E (N-T). L'emplacement où le nœud sélectionné est inséré est celui qui minimise c (i, k, j). La procédure est répétée jusqu'à ce que tous les nœuds aient été insérés dans T.

4-d Heuristique d'insertion la moins chère : Le prochain nœud à rejoindre la tournée, T, est le un qui minimise c (i, k, j) parmi tous les nœuds k E (N-T) et pour tous les nœuds consécutifs paires de nœuds (i, j) E T. L'emplacement où le nœud sélectionné est inséré est, bien sûr, celui qui minimise c (i, k, j). La procédure est répétée jusqu'à ce que tous les nœuds ont été insérés dans T.

4.1 Implémentation avec Python :

```
def GenerateSolutions(self, numberOfSolutions):
    solutions = []
    for i in range(numberOfSolutions):
        solution = []
        isCityAdded = []
        numberOfCitiesInSolution = 3
        self.InitTabWithValue(isCityAdded, self.numberOfCities, False)
        self.PutThreeRandomCitiesToTheBack()
        self.InitSolutionWithThreeCitiesFromTheBack(solution, isCityAdded)
        while self.AllCitiesAdded(isCityAdded) == False:
            bestDistance = sys.maxsize
            tmpDistance = 0
            cityToAddIndex = 0
            for i in range(self.numberOfCities):
                if isCityAdded[i] == False:
                    for j in range(numberOfCitiesInSolution):
                        tmpDistance = self.cities[i].GetDistanceToCity(solution[j])
                        if tmpDistance < bestDistance:
                            bestDistance = tmpDistance
                            cityToAddIndex = i
            indexToInsert = 0
            bestGrowth = sys.maxsize
            tmpBestGrowth = 0
            for i in range(numberOfCitiesInSolution):
                tmpBestGrowth =self.cities[cityToAddIndex].GetDistanceToCity(solution[i - 1])+
                        self.cities[cityToAddIndex].GetDistanceToCity(solution[i])-solution[i - 1].GetDistanceToCity(solution[i])
                if tmpBestGrowth < bestGrowth:
                    indexToInsert = i
                    bestGrowth = tmpBestGrowth
            solution.insert(indexToInsert, self.cities[cityToAddIndex])
            isCityAdded[cityToAddIndex] = True
            numberOfCitiesInSolution = numberOfCitiesInSolution + 1
        solutions.append(solution)
    return solutions
```

5 Algorithme firefly :

Les lucioles (en anglais FireFly) sont de petits coléoptères ailés capables de produire uneLumière clignotante. Il existe environ deux mille espèces de lucioles, et la plupart des lucioles produisent des flashs courts et rythmés. Le motif des flashes est souvent unique pour une espèce particulière. La lumière clignotante est produite par un processus de bioluminescence, et les vraies fonctions de ces systèmes de signalisation continuent de débattre.Ces insectes sont capables de produire de la lumière à l'intérieur de leur corps grâce à desOrganes spéciaux situés très près de la surface de la peau. Cette production de lumière est due à un type de réaction chimique appelée bioluminescence. Les femelles peuvent imiter les signaux lumineux des autres espèces afin d'attirer des mâlesAfin de les capturer et les dévorer. Les lucioles ont un mécanisme de type condensateur, qui se décharge lentement jusqu'à ce qu'un certain seuil soit atteint, ils libèrent l'énergie sous forme de lumière. Le phénomène se répète de façon cyclique

L'algorithme Firefly (FA) a été développé pour la première fois par Yang en 2007 (Yang, 2008, 2009) qui était basé sur les modèles de clignotement et le comportement des lucioles. En substance, FA utilise les trois règles idéalisées suivantes :

Les lucioles sont unisexuées de sorte qu'une luciole sera attirée par d'autres lucioles, quel que soit leur sexe.

L'attractivité est proportionnelle à la luminosité et ils diminuent tous les deux à mesure que leur distance augmente. Ainsi, pour deux lucioles clignotantes, moins il y en aura une plus brillante, plus elle sera plus brillante. S'il n'y en a pas de plus brillant qu'une luciole particulière, elle se déplacera au hasard

La luminosité d'une luciole est déterminée par le paysage de la fonction objective.

Le mouvement d'une luciole *i* est attiré par une autre, plus attrayante (plus brillante) luciole *j* est déterminé par

$$x_i^{t+1} = x_i^t + \beta_0 \, \mathrm{e}^{-\gamma r_{ij}^2} (x_j^t - x_i^t) + \alpha \varepsilon_i^t$$

où est l'attractivité à la distance , et le deuxième terme est dû à l'attraction. Le troisième terme est la randomisation avec étant le paramètre de randomisation, et est un vecteur de nombres aléatoires tirés d'une distribution gaussienne ou d'une distribution uniforme au temps t. Si , cela devient une simple marche aléatoire. De plus, la randomisation peut facilement être étendue à d'autres distributions telles que les vols de Lévy.

Le vol de Lévy fournit essentiellement une marche aléatoire dont la longueur de pas aléatoire est tirée d'une distribution de Lévy

$$L(s, \lambda) = s^{-(1+\lambda)}, \quad (0<\lambda \leq 2)$$

qui a une variance infinie avec une moyenne infinie. Ici, les étapes forment essentiellement un processus de marche aléatoire avec une distribution de longueur de pas de loi de puissance avec une queue lourde. Certaines des nouvelles solutions devraient être générées par Lévy se promener autour de la meilleure solution obtenue jusqu'à présent; cela accélérera souvent la recherche locale (Pavlyukevich, 2007).

Une version de démonstration de l'implémentation de l'algorithme firefly, sans vols Lévy, est disponible sur le site Web d'échange de fichiers Mathworks. L'algorithme Firefly a attiré beaucoup d'attention (Apostolopoulos et Vlachos, 2011; Gandomi et coll., 2011b; Sayadi et coll., 2010). Une version discrète de FA peut résoudre efficacement les problèmes de planification NP-hard (Sayadi et al., 2010), tandis qu'une analyse détaillée a démontré l'efficacité de FA sur un large éventail de problèmes de test, y compris les problèmes de répartition de charge multiobjectifs (Apostolopoulos et Vlachos, 2011). Un algorithme de luciole amélioré par le chaos avec une méthode de base pour le réglage automatique des paramètres est également développé (Yang, 2011b).

où est la position d'une luciole dans l'itération t, définit l'attraction entre la luciole *j* et la luciole *i*, et est un vecteur de nombres aléatoires avec un paramètre de randomisation défini par . Ce paramètre est le facteur de mise à l'échelle aléatoire initial, défini comme suit :

$$\alpha_t = \alpha_t \delta^t,$$

où δ a une valeur comprise entre 0 et 1. Les valeurs des α, β et δ appliquées dans ce travail sont présentées dans le tableau

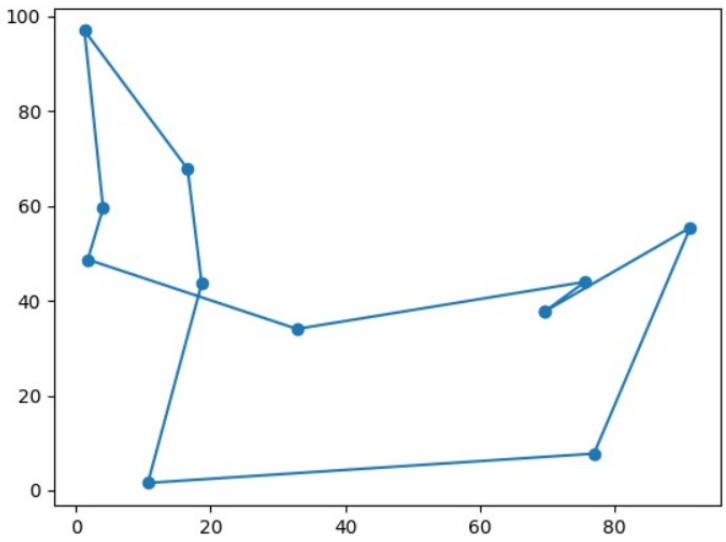
Paramètres	Valeur
Lucioles (*n*)	10
Nombre maximal d'itérations (*t*)	30
α	0.015
β	1
δ	0.95

5.1 Implementation avec Python firefly :

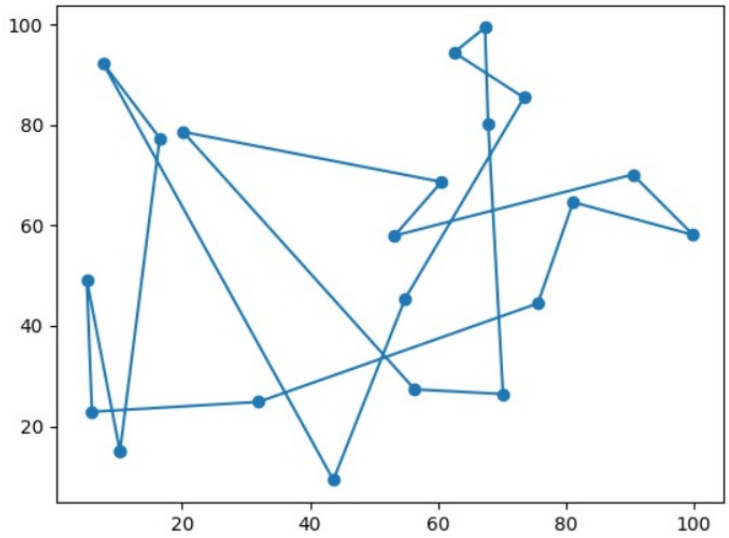
```python
def run(self, number_of_individuals=100, iterations=200, heuristics_percents=(0.0, 0.0, 1.0), beta=0.7):
    "gamma est le paramètre des intensités lumineuses, beta est la taille du voisinage en fonction de la distance de Hamming"
    # hotfix, will rewrite later
    self.best_solution = random_permutation(self.indexes)
    self.best_solution_cost = single_path_cost(self.best_solution, self.weights)

    self.generate_initial_population(number_of_individuals, heuristics_percents)
    value_of_reference = self.population[0][0]
    self.rotate_solutions(value_of_reference)
    self.determine_initial_light_intensities()
    self.find_global_optimum()
    print(self.best_solution_cost)

    individuals_indexes = range(number_of_individuals)
    self.n = 0
    neighbourhood = beta * len(individuals_indexes)
    while self.n < iterations:
        for j in individuals_indexes:
            for i in individuals_indexes:
                r = hamming_distance(self.population[i], self.population[j])
                if self.I(i, r) > self.I(j, r) and r < neighbourhood:
                    self.move_firefly(i, j, r)
                    self.check_if_best_solution(i)

        self.n += 1
        if self.n % 100 == 0:
            print(self.n)
            print(self.best_solution_cost)


    return self.best solution
```

6 – Résultats du problème de Firefly

```
PS C:\Users\HP\Desktop\tspFa2022> & C:/python/python.exe c:/Users/HP/Desktop/tspFa2022/main.py
Number of points: 11
408.11842818273874
100
379.93248191238234
200
379.93248191238234
```




```
PS C:\Users\HP\Desktop\tspFa2022> & C:/python/python.exe c:/Users/HP/Desktop/tspFa2022/main.py
Number of points: 21
892.8615271365268
100
707.7635251127344
200
707.7635251127344
```



7 – Conclusion

La méthode de l'algorithme firefly a été émergée récemment ; il a été fait référence à un nombre important de recherches. Compte tenu du problème des systèmes photovoltaïques en termes de cadre d'optimisation, le suivi MPP sur la caractéristique puissance-tension est un problème très familier. De plus, ce problème devient de plus en plus compliqué dans des conditions d'ensoleillement direct en raison de l'observation de multiples points maxima locaux sur la courbe. À cet égard, Shi et al. Ont proposé un algorithme de luciole modifié pour le problème connexe dans des conditions d'ombrage. Ils ont conclu que l'algorithme proposé permettait de faire converger le point maximal global efficacement et rapidement par rapport aux techniques conventionnelles. Un autre problème d'optimisation pour le système solaire est la prédiction du rayonnement solaire global en utilisant les données météorologiques de l'ensoleillement. Pour ce numéro, Olatomiwa et al. Ont abordé une méthode développée comme hybridation de machines à vecteurs de support et d'algorithme de luciole pour prédire le rayonnement solaire mensuel. L'efficacité de la méthode a été validée comme comparaison avec les réseaux de neurones artificiels et les modèles de programmation génétique. De la même manière, en considérant l'effet néfaste des conditions météorologiques variables pour produire de l'électricité à partir du système solaire, Sulaiman IS et al. Ont présenté des systèmes d'énergie photovoltaïque connectés au réseau. Ils ont proposé une approche de réseau neuronal artificiel pour modéliser la tension de sortie du système adopté et ont utilisé l'algorithme firefly pour rechercher le nombre optimal de neurones et d'autres critères de performance dans le processus d'entraînement

A travers ce chapitre, nous avons présenté les méthodes de résolution que nous allons utiliser durant la réalisation de ce travail, nous nous sommes focalisés sur une méthode de résolution exacte en donnant son principe, ensuite nous avons expliqué les méthodes de résolution approchée précisément les métaheuristiques en donnant une définition claire et précise des deux algorithmes qui seront utilisés par la suite (algorithme du plus proche voisin et algorithme d'insertion heuristique du plus proche voisin).