Université de Montréal

PHY3075 – Modélisation numérique

Optimisation par colonies de fourmis

par :
Ilya Iakoub
20226476

20 juin 2024[3cm]

1 Introduction

Les problèmes non-polynomiaux (ou NP) sont, par leur nature, difficiles à résoudre et à vérifier leur solution. Vérifier les solutions de ces problèmes est, au meilleur de nos connaissances, impossible en un temps polynomial (c'est-à-dire qu'avec une entrée de taille N, vérifier la solution pourrait prendre $N!$ étapes !). La résolution de ce genre de problème utilise aujourd'hui des algorithmes dits « métaheuristiques ». Sans trop entrer dans les détails, ce sont des algorithmes basés plus sur l'intuition que sur un fondement mathématique rigoureux, qui permettent d'estimer une solution quasi-optimale à un problème. Dans ce travail, nous allons explorer la résolution d'un problème non-polynomial, celui du commis-voyageur, en utilisant un algorithme méta-heuristique, celui d'optimisation par colonie de fourmis.

Le problème du commis voyageur est le suivant : étant donné la position de N points, il faut trouver le chemin optimal reliant les N points qui passe exactement une fois par chacun. Avec N points, il y a $(N-1)$ étapes de chemin à trouver pour compléter un parcours (chaque étape de chemin étant $(x_i, y_i) \to (x_j, y_j)$ où i et j sont les étiquettes de points quelconques). À la première étape, il y a $N-1$ options, à la deuxième, $N-2$ (puisqu'on ne peut pas repasser par le même chemin deux fois), etc. Donc, le nombre total de chemins possibles est $(N-1) \times (N-2) \times \ldots \times (N-1-(N-1)) = (N-1)!$. Toutefois, pour chaque chemin, il y a un chemin inverse identique qui prend la même distance, donc il y a $N_{\text{ch.}} = \frac{(N-1)!}{2}$ chemins indépendants. Cela signifie que seulement avec 12 points, le nombre de chemins possibles dépasse le double de la population du Québec, avec 13 la population du Brésil, avec 14 on a la moitié de la population humaine et avec 60 on a autant de chemins qu'il y a d'atomes dans l'univers connu ! (Le « connu » n'est pas factoriel, c'est bien une exclamation.)

L'approche d'optimisation par colonie de fourmis s'inspire évidemment du comportement des fourmis dans la nature. Lorsque les fourmis cherchent de la nourriture, une première vague de fourmis va explorer les environnements du nid. Lorsqu'elles trouvent une source de nourriture, elles laissent une trace de phéromones le long de leur trajet, qui guide les autres fourmis vers la source de nourriture. L'idée est que les trajets qui atteignent un certain but (ici la recherche de nourriture), vont contenir plus de phéromones, et les phéromones des anciens trajets non-optimaux vont s'évaporer avec le temps. Même si le comportement des fourmis n'est pas nécessairement déterministe à 100%, la trace de phéromones permettra à la colonie de fourmis de converger vers une solution quasi-optimale. Malheureusement, l'optimisation par colonie de fourmis ne peut pas rivaliser avec les meilleurs algorithmes pour le problème du commis voyageur[1], mais l'exemple du commis voyageur fournit une application facile et intuitive de l'algorithme d'optimisation par colonie de fourmis qui peut être appliqué à une grande variété de problèmes où sa performance est la meilleure[1] (par exemple des problèmes de classement). Dans ce contexte, il est intéressant d'explorer l'algorithme en tant que tel, et le problème du commis voyageur fournit une plateforme pour explorer ce dernier.

Finalement, nous allons modifier le problème du commis voyageur pour le rendre plus compliqué et voir quelle sera alors la performance de l'algorithme. En particulier, nous allons interdire certains chemins aux fourmis et/ou en rendre certains plus coûteux pour voir quelle sera la réaction et si la colonie de fourmis arrive toujours à converger vers une solution.

2 Méthodologie

Dans cette section, nous allons d'abord discuter de l'algorithme dans sa vue d'ensemble, puis parler des définitions concrètes nécessaires pour l'algorithme.

Il existe un grand nombre de variantes d'algorithmes d'optimisation par colonies de fourmis, il est donc nécessaire de décrire exactement de quelle variante a été implémentée. Remarquez que l'algorithme de base laisse une grande liberté face au choix possible d'extensions de celui-ci, que ce soit de la manière dont les phéromones sont appliquées, s'évaporent ou comment les fourmis explorent le graphe. Cela dit, faisons le tour du fonctionnement de l'algorithme que nous avons utilisé :

— Les fourmis explorent le graphe en vagues. Un certain nombre de fourmis vont explorer le graphe de N points à chaque vague. Lors d'une vague, toutes les fourmis vont faire un trajet complet en passant par chaque point une fois. À la fin de chaque vague, une trace de phéromones est laissée le long de chaque trajet. La trace de phéromones est plus forte sur les trajets plus courts. Avant d'appliquer les nouveaux phéromones, on laisse les anciens s'évaporer un peu. L'idée est que la trace de phéromones serve de mémoire collective pour la colonie de fourmis.
— Lors de chaque vague, toutes les fourmis commencent leur exploration à partir du même point.
— D'abord, une première vague de fourmis explore le graphe aléatoirement. Lors de cette vague, la probabilité de visiter chaque point est égale.
— Ensuite, les vagues suivantes vont choisir de manière probabiliste le prochain point de chaque trajet selon la trace de phéromones laissée entre les points. Une trace de phéromones plus forte résultera en une plus haute probabilité d'emprunter ce chemin.
— À chaque pas (de chaque vague), chaque fourmi a une probabilité de ne pas suivre la trace de phéromones, mais plutôt de choisir un point aléatoirement avec une probabilité égale.
— À chaque vague, un classement des fourmis est fait selon la longueur des chemins qui ont été empruntés. Les meilleures fourmis laisseront plus de phéromones, les pires n'en laisseront pas.
— Si la meilleure fourmi trouve un chemin plus court que le plus court qui n'a jamais été trouvé, elle sera encore plus récompensée.
— Lorsqu'un critère de convergence a été atteint, le meilleur chemin est retenu.

L'influence de chaque extension à l'algorithme de base sera discutée dans la section 5 : « Analyse ». En raison de son approche probabiliste, on peut considérer cet algorithme comme un algorithme Monte-Carlo. Maintenant qu'on a une idée générale de l'algorithme, on peut expliquer chaque partie plus en détail. Dans ce qui suit, chaque étape de l'algorithme sera décrite dans un langage mathématique, mais une explication plus intuitive est aussi fournie.

2.1 Le graphe

Le graphe, G, consiste en un ensemble de N positions.

$$G(L, N) \equiv \{(x_i, y_i) | x_i, y_i \leq L, 1 \leq i \leq N\}$$

où L est la dimension de la boîte contenant les points et i l'étiquette de chaque point. Dans l'implémentation de l'algorithme, nous avons fait en sorte qu'aucun point ne se répète $(x_i, y_i) \neq (x_j, y_j) \forall i \neq j$. Dans ce cas particulier, on peut référer de manière non ambiguë à chaque point par son indice. Nous travaillerons donc plutôt avec l'ensemble des indices

$$I \equiv \{1 \leq i \leq N\}$$

et un trajet est défini comme une permutation de cet ensemble pour laquelle on ajoute le point de départ à la fin (pour que le trajet soit fermé). Par exemple, pour un graphe de 4 points, on pourrait avoir les trajets

$$T_1 = \{1, 2, 3, 4, 1\}, \quad T_2 = \{2, 4, 1, 3, 2\}, \quad T_3 = \{2, 3, 4, 1, 2\}, \ldots$$

Pour référer à un segment reliant deux points d'un graphe $G(L, N)$, il nous faut donc deux nombres naturels entre 1 et N. Un premier pour référer au site de départ et un deuxième pour référer au site d'arrivée. Lors de la première vague, chaque trajet est initialisé comme une permutation aléatoire de I.

En termes de Python, on peut définir G comme

```
x,y=np.random.random(N),np.random.random(N)
```

(où `np.` fait référence au module `numpy`) et ayant une série d'indices dans un tableau `numpy` `indices`, on peut leur faire référence dans l'ordre avec `x[indices],y[indices]`.

2.2 Les distances

Soit un trajet T

$$T = \{a_n | 1 \leq n \leq N+1\}$$

On défini la distance d'un trajet, $d(T)$ comme suit :

$$d(T) = \sqrt{\sum_{n=1}^{N} (x_{a_{n+1}} - x_{a_n})^2 + (y_{a_{n+1}} - y_{a_n})^2}$$

En terme de numpy, étant donné un trajet `T` sous la forme d'une série d'indices dans un tableau `numpy`, cela s'écrit comme

```
d = np.sqrt(np.sum((x[T[1 :]]-x[T[ :-1]])**2 + (y[T[1 :]]-y[T[ :-1]])**2 ))
```

En pratique, on va vouloir tenir compte des trajets de chacune des N_f fourmies lors d'une itération dans un tableau $N_f \times N$, alors on calcul la distance comme

```
d=np.sqrt(np.sum((x[X[ :,1 :]]-x[X[ :, :-1]])**2
+(y[X[ :,1 :]]-y[X[, : :-1]])**2,axis=1))
```

2.3 Les phéromones

Compte tenu de la définition présentée ci-haut, à chaque itération i on peut tenir compte du nombre de phéromones entre chaque points avec une matrice $N \times N$ qu'on définira comme P_i, la matrice de phéromones. Les phéromones entre le site m et le site n sont défini comme un nombre réel position p_{nm}. Pour que les fourmis ne tiennent pas compte de la direction du trajet (ça ne fait pas partie du problème), il on a la condition $p_{n,m} = p_{m,n}$. De plus, une fourmis ne peut pas laisser de phéromones entre un point et lui-même, donc on a aussi la condition $p_{n,n} = 0$. Compte tenu de ces conditions, la matrice de phéromones, P_i est de la forme

$$P_i = \begin{pmatrix} 0 & p_{0,1} & p_{0,2} & \cdots & p_{0,N} \\ p_{0,1} & 0 & p_{1,2} & & \\ \vdots & & \ddots & p_{m,n} & \\ & & p_{m,n} & & \\ & & & p_{N-1,N} & 0 \end{pmatrix}$$

C'est une matrice symmétrique avec des 0 sur la diagonale.

On définit la fonction de phéromones, $\delta P(d_i^k, T_i^k)$ où d_i^k est la distance totale parcourue le long d'un trajet par une fourmi k lors de l'itération i et T_i^k est le trajet emprunté par cette fourmi lors de cette itération. La fonction de phéromones $\delta P(d_i^k, T_i^k)$ prend en entrée un trajet T_i^k et retourne une matrice de la même forme que P_i. Pour que cette fonction permette de résoudre le problème, il faut que δP_i^k soit une fonction monotone croissante. δP_i^k aura seulement des entrées sur les segments qui ont été visités par la fourmie. Donc en notation Dirac, δP_i^k est de la forme

$$\delta P_i^k = \sum_{n=1}^{N} f(d(T_i^k))(|a_n\rangle\langle a_{n+1}| + |a_{n+1}\rangle\langle a_n|)$$

où les a_n proviennent de la définition fournie dans la sous-section 2.2. En autre mots, δP_i^k a seulement des entrées non-nulles sur les cases correspondant aux points visités par la fourmie k lors de l'itération i et est symétrique. Remarquez qu'ici c'est la fonction $f(d(T))$ qui doit être monotone et croissante. Il y a

une liberté face au choix de la fonction exacte qui sera discutée dans la section 5 (Analyse). Finalement on peut définir une règle d'application des phéromones :

$$P_{i+1} = (1 - c_{\acute{e}})P_i + c_{\acute{e}} \sum_{k=1}^{N_f} C_k \delta P_i^k$$

où $c_{\acute{e}}$ est le coéfficient d'évaportation et C_k est un coéfficient correspondant au classement de la fourmie dans la vague et N_f est le nombre de fourmis. Le coéfficient d'évaporation est un nombre réel entre 0 et 1 qui correspond à la proportion des phéromones que vont occuper les nouvelles phéromones. Plus que $c_{\acute{e}}$ est grand, le plus rapidement que les fourmis vont oublier leurs anciens trajets. C_k permet d'appliquer plus de phéromones pour les fourmis qui ont le mieux performés lors d'une vague et/ou à celles qui ont trouvé un nouveau chemin optimal.

À cette étape, il est possible d'implémenter la contrainte d'une transition plus couteuse. Pour ce faire, on peut simplement forcer un élément m,n (et n,m) de la matrice P_i à s'évaporer plus rapidemetn (il suffirait de remplacer le coéfficient $c_{\acute{e}}$ par une matrice).

En somme, à chaque vague, on fait une moyenne pondérée entre les anciennes phéromones et les nouvelles phéromones. On applique les phéromones en sommant les nouvelles phéromones le long de chaque trajets emprunté par une fourmie lors de la vague. Le nombre de phéromones appliqué est grand si le trajet est court et petit si le trajet est long.

En terme de Python, étant donné la fonction f, on peut définir P_i comme un tableau $N \times N$ et étant donné le trajet T (sous forme d'un tableau) d'une fourmis et la distance $d(T)$, on pourrait calculer δP comme

```
delta_P=np.zeros((N,N),dtype='float')
delta_P[T[1 :],T[ :-1]]=f(d(T))
```

en itérant sur chaque fourmis et tenant compte du classement, on pourrait calculer P_i de cette manière.

2.4 Les probabilitées

La plus grande difficultée de l'algorithme est d'implémenter des probailitées conditionnelles. La probabilitée de visiter chaque site pour une fourmie est dépend des sites qu'elle a déjà visitée (la probabilitée de revisiter un site doit être nulle) et du site sur lequel elle se trouve à un pas donné. Le fait que la probabilitée de «transition» d'un site à l'autre dépend du site présent justifie l'utilisation d'une matrice, analogue à l'Hamliltonien en mécanique quantique, dont la case m,n indique la probabilitée de transition du site d'indice m au site d'indice n. On définit H_{i0} comme la matrice P_i dont chaque ligne a été normalisée à 1. Ainsi, dans cette matrice, la probabilitée de transition d'un site m vers tout les autres sites somme à $1 : \sum_{n=1}^{N} H_{i0}^{m,n} = 1$. De plus il y a une probabilitée nulle de transitionner d'un site vers lui-même puisque la matrice P_i n'a pas de termes diagonaux. Maintenant, dans chaque vague, il y a N pas p. Pour chaque fourmi k, pour chaque pas p on définit la matrice

$$H_i(p+1,k) = \sum_{m=1}^{N} \sum_{n=1}^{N} C_n(1 - \delta_{m,a_p}) H_i^{m,n}(p,k) |m\rangle\langle n|$$

où C_n normalise chaque ligne à 1, a_p est la position de la fourmie au pas p et on initialise $H_i(0,k) = H_{i0}$. Cette matrice aura des 0 sur les site visités (puisque le δ_{m,a_p} enlève les colonnes des sites déjà visités) et chacune de ses lignes sont normalisées à 1 (à cause de C_n). $H_i(p,k)$ donne les probabilités de transition tenant compte des sites visités par la fourmie k au pas p lors de la vague i (tout ce qu'on a fait c'est

enlever les sites déjà visités de H_{i0} et renormaliser). Concrètement, pour choisir le prochain pas d'une fourmie située à un site $|a_p\rangle$, il suffit de procéder comme suit :

— La probabilitée de transition vers les autres sites est donnée par

$$prob(n|i,p,k) = \langle n|H_i(p,k)|a_p\rangle$$

— On calcule la fonction de répartition

$$F(m) = \sum_{n=1}^{m} prob(n|i,p,k)$$

où m est un nombre entier entre 0 et N. Cette fonction est inversible.

— On choisis un nombre réel aléatoire, r, entre 0 et 1 selon une distribution uniforme. Le prochain site pour la fourmie est donné par $F^{-1}(r)$

Pour implémenter le fait que chaque fourmi peut aussi visiter des sites indépendamment des phéromones, on peut remplacer la matrice $H_{i,0}$ par

$$H'_{i,0} = (1+Nb)^{-1}\left[H_{i,0} + b\sum_{m=1}^{N}\sum_{n\neq m}^{N}|m\rangle\langle n|\right]$$

où b est un nombre réel entre 0 et $1/N$ correspondant à la probabilitée de visiter un site peu importe s'il y a des phéromones ou non. En autre mot, peu importe les phéromones, il y a une probabilitée de b de visiter un site aléatoirement. Une autre manière de l'implémenter est d'ajouter une quantitée de phéromones nette correspondant à b à chaque segments après chaque vague.

À cette étape, on peut aussi implémenter les contraintes. Si on veut interdire la transition entre un site m et un site n, il suffit de forcer l'élement m,n de la matrice H à zéro.

En somme, on calcul initialement les probabilités en ajoutant du bruit aux phéromones et en normalisant la matrice de phéromones. Ensuite, lors de chaque pas d'une fourmis, on met la probabilitée correpondant au site visité à 0. On calcul les probabilitées subséquente de cette manière pour chaque pas.

En terme de Python, étant donné une matrice H_i sous forme d'un tableau `numpy`, et l'indice de la position initiale de la fourmie, `pos`, on peut trouve les prochains pas comme suit :

```
for i in range(N) :
          H[ :,pos]=0
          H=normalize_lines(H)
          prob=H[pos, :]
          repartition=np.array([np.sum(prob[i]) for i in range(N)])
          r=random.random()
          pos=np.amax((r>=np.array(prob)).nonzero()[0])
```

Évidemment, ça ne serait pas tout ce qu'il y a à faire, mais avec un peu de chance, vous saisissez l'idée.

2.5 La convergence

Il y a deux critères de convergence utilisés : le premier est juste de prendre le trajet le plus court trouvé après un nombre maximal de vagues. Le second critère consiste à considérer un trajet comme la solution si ce trajet est le meilleur trouvé dans un certain nombre (on utilise 50) de vagues précédentes. Le second critère garanti que la solution est au moins un minimum local puisque les fourmis suivrons généralement des chemins qui dévient peu de la trace de phéromones la plus forte car la probabilitée de

transition d'un site vers un autre reste beaucoup plus haute s'il y a beaucoup de phéromones entre les deux et les fourmis vont donc dévier que légèrement du chemin le plus «phéromoneu».

3 Validation

Afin de vérifier que l'algorithme fonctionne correctement, il faut commencer par tester chaque outil que l'on utilise, comme ça il est clair que le bon algorithme a été implémenté. Pour ce faire, l'idée est de créer un graphe artificiel pour lequel il est évident de calculer les distances et les phéromones à appliquer. Par exemple, on peut définir le graphe suivant :

`x,y=np.array([0,0,1,1]),np.array([0,1,1,0])`

qui correspond aux sommets d'un carré de longueur 1. Pour valider que les calcul de distance est bon, on devrait vérifier par exemple que `d(np.array([0,1,2,3,0]))` donne 4 (trajet le long de l'extérieur du carré), et que `d(np.array([0,2,1,3,0]))` donne bien $2 \times 2 + 2 \times \sqrt{2}$.

Ensuite, pour vérifier que les phéromones se font bien appliquer, on pourrait tester avec $f = d$. Alors le long du premier trajet, on devrait appliquer 4 phéromones et le long du second $4 + 2\sqrt{2}$. On s'attend donc à ce que le tableau de phéromones soit

`[[0,4,4+2*sqrt(2),8+2*sqrt(2)],[...],[...],[...]]`

Ensuite on peut tester que les phéromones s'évaporent bien en claculant quelles devraient être les phéromones à la prochaine itération, etc. L'idée est simplement de vérifier manuellement les calculs pour un système suffisament simple.

D'autres tests sont de vérifier que tout les matrices utilisées sont bien symétriques et que pour les matrices de probabilitée, les lignes somment bien à 1. Une manière pratique de voir si les tableaux sont bien symétriques est de les visualiser en utilisant `plt.imshow()` (où plt. fait référence au modeul matplotlib de Python). Vous pouvez en voir un exemple dans la figure 1

Ensuite, le problème étant non-polynomial, il est difficile d'évaluer si la bonne solution a été trouvée. Ainsi, pour valider le bon fonctionnement de l'algorithme, la seule option est de lui donner une série de points pour laquelle le chemin le plus court est connu. Par exemple, pour tester l'algorithme sur une série de N points, il est possible de donner à l'algorithme N points équidistants sur la circonférence d'un cercle. Il est évident que le chemin le plus court est celui qui passe par la circonférence. Toutefois, il est à noter que ce test peut être particulièrement compliqué puisque ce problème appliqué à la circonférence d'un cercle contient beaucoup de minimums locaux par le biais du fait que tout les points sur le cercle sont équidistants.

Finalement, si on veut tester si l'optimisation avec contrainte fonctionne, on peut aussi réutiliser l'exemple du cercle. Si on empêche la transition directe entre deux points adjacents, alors le chemin le plus court est celui qui fait deux fois le tour du cercle en sautant au deuxième voisin. Vous pouvez voir l'exemple d'un tel chemin dans la figure 2. Toutefois il y a un problème substantiel avec cette approche : il y a trop de contraintes ! Premièrement, un chemin fermé qui relie chaque point une fois dans un tel système n'existe que pour un nombre de point de la forme a^n $(n > 2, a > 1)$, mais même dans ces cas, la solution est unique ! En général, on remarque qu'interdire complètement des transitions peut mener à des situations où les fourmis peuvent rester bloquer en explorant le graphe. Étant donner que ce n'était qu'un but secondaire du projet, nous allons nous contenter de rendre des transitions coûteuse/improbable.

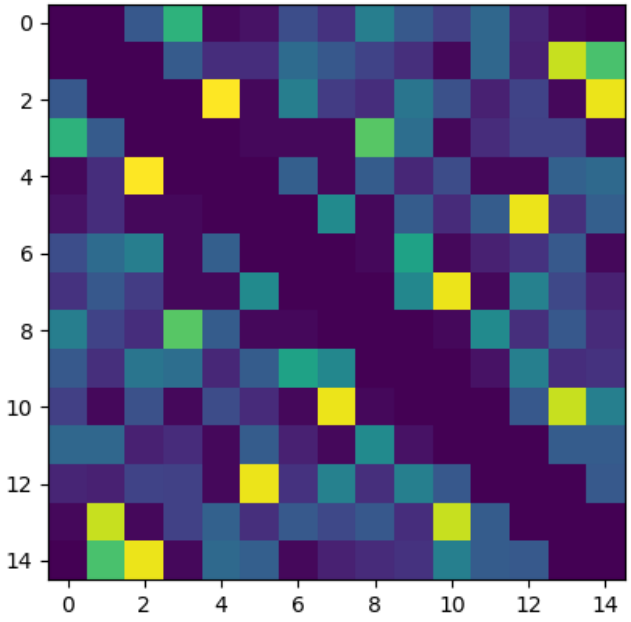

FIGURE 1 – Exemple de visualisation d'un tableau pour voir s'il est bien symétrique. Comme vous pouvez constater, celui-ci l'es, puisque les cases i, j ont visiblement la même couleur que les cases j, i.

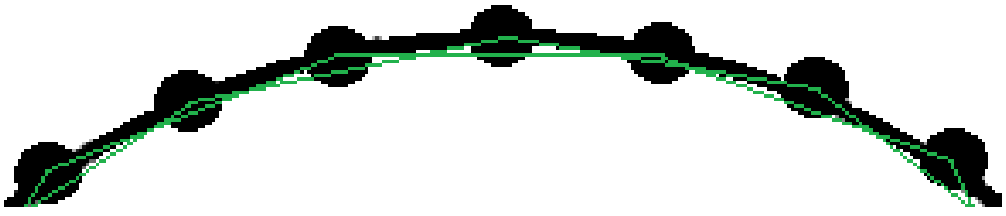

FIGURE 2 – Exemple de chemin le plus court le long d'un cercle si les transitions au premier voisin sont interdites.

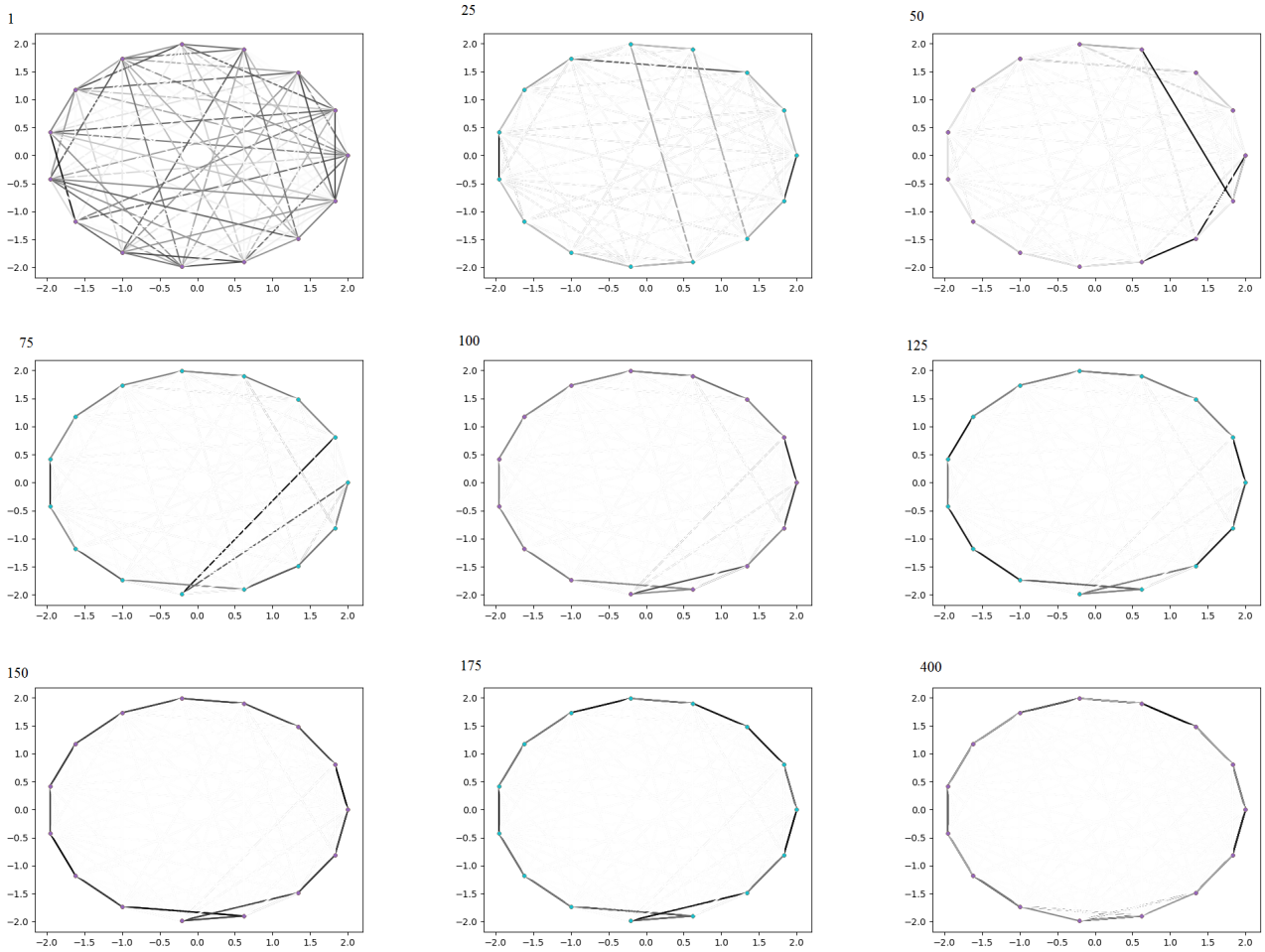

FIGURE 3 – La trace de phéromones selon la vague pour une simulation sur un cercle de rayon 1 de 15 points. Le numéro de la vague est indiqué en haut à gauche. Entre la vague 175 et 400 il n'y a essentiellement aucun changement, et après la vague 400 nonplus.

4 Résultats

Avant de commencer, il faut dire que dans tout les résultats qui suivent, le nombre d'itération maximal permis est de 1000 vagues de 10 fourmis (donc 10000 pas).

On peut vérifier que le code fonctionne correctement en utilisant les méthodes de validation discutés précédemment, c'est-à-dire vérifier que l'algorithme est capable de trouver le chemin le plus court sur la circonférence d'un cercle. Dans le cadre de ce projet, un cercle de rayon 1 consistant de 15 points a été utilisé. Ce nombre de points a été choisi puisqu'il est relativement grand sans toutefois prendre trop de temps à converger, permettant ainsi de faire un grand nombre de tests. Sur 20 essais avec 15 points, l'algorithme arrive à la bonne solution 19 fois. Chaque simulation a pris en moyenne 975 vagues de 10 fourmis. Toutefois, vous pouvez voir dans la figure 3 que la trace de phéromone commence rapidement à suivre la bonne solution. La signifiation de ce résultat est discutée dans la section Analyse.

Ensuite, on peut tester faire le test discuté dans la section précédente pour vérifier que l'optimisation avec contrainte a fonctionnée. On test ainsi sur un cercle de 16 points où les transitions au premier voisin sont interdites. Ce nombre de points a été choisi car c'est un des seuls où une solution existe. Vous pouvez voir les résultats de 3 simulation dans la figure 4. La signification de ce résultat et la validité du test est discutée dans la section Analyse, mais vous pouvez déjà voir que ce n'est pas le résultat attendu.

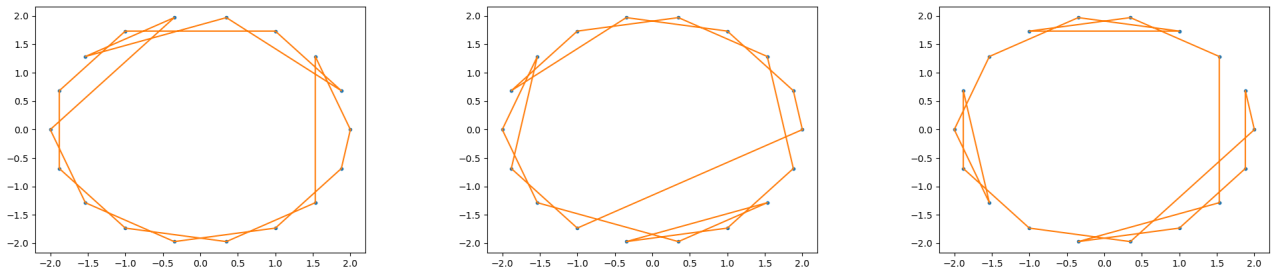

FIGURE 4 – Résultat de trois simulations de colonies de fourmies sur une série de 16 points sur la circonférence d'un cercle de rayon 1 avec la contrainte que les transitions au premier voisin sont interdites. Les trois simulations ont pris 1000 vagues de 10 fourmis.

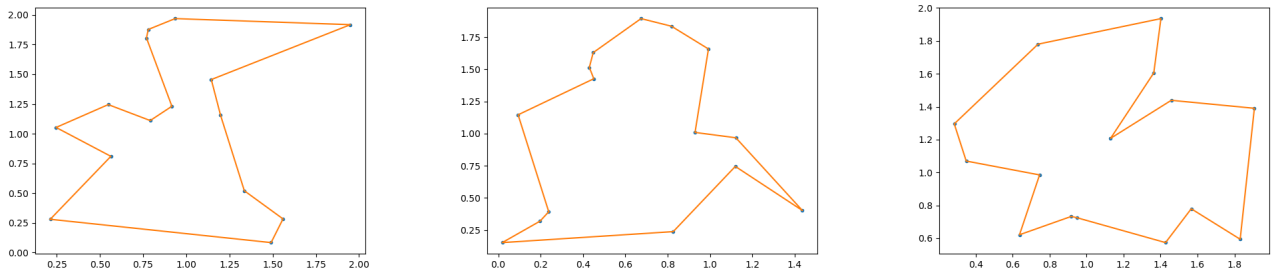

FIGURE 5 – Résultat de trois simulations de colonies de fourmies sur une série de 15 points aléatoires, avec des vagues de 10 fourmis.

On peut continuer en testant sur des distributions de points aléatoires. Vous pouvez voir dans la figure 5 une série de tests qui ont été fait sur 15 points aléatoires. La signification de ce résultat est discutée dans la section Analyse.

Un autre test qui est intéressant de faire est d'essayer de trouver une solution pour une série de points aléatoires avec contrainte. Le cas qui a été utilisé a été obtenu en choisissant une série de points aléatoires et interdisant manuellement 3 à 5 transitions (aussi aléatoires). Vous pouvez voir le résultat dans la figre **??**. Les lignes rouges montrent les transitions interdites.

Maintenant que des tests de base ont été fait, il est intéressant d'analyser l'influence de chaque paramètre sur la qualité de la solution. Nous allons donc, dans ce qui suit, faire varier le coefficient d'évaporation, le pourcentage de bruit dans les probabilités, l'absence ou présence de système de classement et les différentes fonctions phéromones utilisés et voir si l'algorithme arrive à converger vers une solution et en combien de pas. Remarquez que tout les tests sont fait sur une série de points sur un cercle car c'est une des seules manières non-ambiguë de tester la qualité des solutions. L'interprétation de ces résultats sera fait dans la section Analyse. Pour chacun de ces tests, vous pouvez retrouver un tableau indiquant le nombre de fois que l'algorithme a réussi à trouver la bonne solution sur un cercle après 10 essais.

Lorsqu'on fait varier tout les paramètres indiqués plus hauts, on les varies par rapport aux paramètres : $c_{\text{é}} = 0.5$, $f(d) = |d - 0.65 * \bar{l}|^{-1}$ (où $\bar{l}$ est la longueur moyenne d'un trajet $\approx Nl/2$ où l est la longueur de la boîte contenant les points), $b = 0.1$ (donc 10% de bruit dans les probabilités).

Finalement, avec le même test que précédemment, on remarque qu'avec un système de classement de

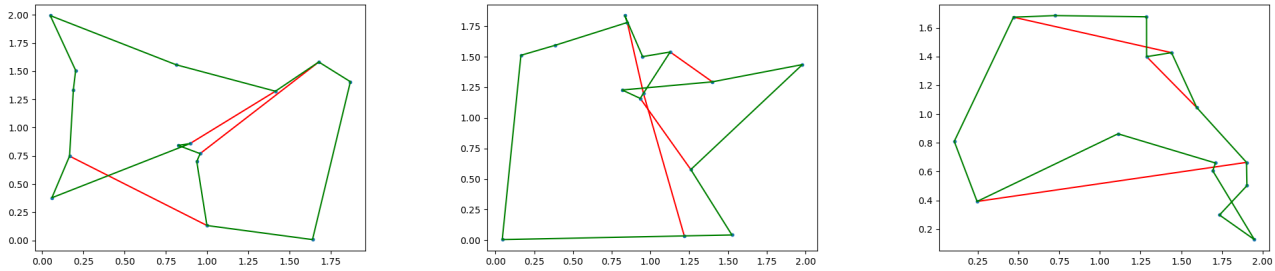

FIGURE 6 – Résultat de trois simulations de colonies de fourmies sur une série de 15 points aléatoires, avec des vagues de 10 fourmis mais où 3 à 5 transitions aléatoires sont interdites. Vous pouvez voir en rouge les transitions qui sont interdites et en vert la solution obtenue

b=0.01	0.025	0.05	0.075	0.1	0.125
0/10	2/10	5/10	8/10	10/10	10/10

TABLEAU 1 – Variation du bruit dans le probabilitées, b. Le /10 signifie que 10 essais ont été fait et indique le nombre de succès lorsque testé sur une distribution de points sur la surface d'un cercle.

fourmis, l'algorithme converge 10/10 fois vers la bonne solution, alors que sans, il converge 0/10 fois.

5 Analyse

Avec un taux de succès de 19 sur 20 essais lorsque testé sur 15 points sur la circonférence d'un cercle (sans contrainte), on peut juger que l'algorithme est efficace à accomplir son but. De plus, la seule fois où la solution obtenue n'était pas la bonne, elle restait proche de l'optimale. Le problème du commis voyageur appliqué sur des points sur la circonférence d'un cercle est particulièrement difficile car il possède énormément de minimums secondaires puisque pour chaque distance possible à partir d'un point (sauf 2×rayon), il y a deux points equidistants. C'est donc un bon test de performance, en plus qu'il est possible de vérifier la solution de manière non ambiguë.

En regardant la figure 4, vous pouvez constater que la performance de l'algorithme en ce qui concerne l'optimisation avec contrainte, lorsqu'il est testé sur un ensemble de points sur la circonférence d'un cercle, n'est pas convaincante. Cela semble être dû (au moins partiellement) au fait que ce problème est assez difficile en général. En effet, comme il a été mentionné précédemment, ce problème n'a pas toujours de solution, et même lorsqu'une solution existe, cette contrainte est très forte. En particulier, avec autant de contraintes, très rapidement les fourmis n'auront qu'une option pour le prochain point. Ainsi, les phéromones au début du trajet détermineront le trajet au complet. Les fourmis n'ont donc que quelques pas où elles peuvent explorer, ce qui explique que la solution obtenue n'est pas généralement la solution optimale. Toutefois, on remarque souvent que les fourmis vont former un motif de tressage dans plusieurs régions, qui est la solution optimale. On juge donc que même si l'algorithme n'atteint pas son but entièrement, il l'atteint partiellement, dans le sens où les solutions obtenues restent parmi les meilleures possibles.

$c_{\text{é}}$=0.1	0.25	0.5	0.75	0.9
10/10	10/10	10/10	7/10	8/10

TABLEAU 2 – Éffet de la variation du coéfficient d'évaporation $c_{\text{é}}$ sur la convergence. Le /10 signifie que 10 essais ont été fait et indique le nombre de succès lorsque testé sur une distribution de points sur la surface d'un cercle.

Par la complexité du problème, il est difficile d'interpréter de manière objective les résultats des figures 6 et 5 puisqu'il est presque impossible de vérifier si cette solution est la bonne. Toutefois, intuitivement, on voit que la solution donnée est au moins proche de la bonne. Dans tout les simulations qui ont été faites, le nombre de pas est inférieur à 1000 vagues $\times 10$ fourmis, il y a donc au plus 10000 pas. Or, la probabilitée de trouver aléatoirement la bonne solution en 10000 pas, en utilisant la formule de l'introduction, est de l'ordre de $10^{-16}\%$. On peut donc au moins dire objectivement que la méthode est meilleure que juste piger des solutions aléatoirement.

Les résultats du tableau 1 montrent qu'en augmentant la probabilitée des fourmis d'explorer le graphe indépendamment de la présence de phéromones, on augmente l'efficacitée de l'algorithme. Étrangement, et sans explication claire, le comportement du taux de succès selon le facteur b est presque linéaire avec une pente d'environ 10. Il est évident qu'en permettant aux fourmis d'explorer plus, on ne peut qu'améliorer l'algorithme, puisqu'en trouvant des nouveaux chemins, les fourmis ont plus de chances de sortir d'un minimum secondaire. Toutefois, il y a aussi un coût associé à cela : le nombre d'itérations nécessaires avant de converger vers une solution augmente beaucoup puisqu'il y a moins de chances que les fourmis retournent sur les bonnes solutions. Pour cette raison, un $b \approx 0.1$ semble optimal.

Il semblerait qu'un faible taux d'évaporation, donc une mémoire à plus long terme, semble être généralement meilleure qu'un haut taux d'évaporation comme vous pouvez voir dans le tableau 2. Cela semble être dû au fait qu'avec une mémoire à court terme (donc un haut taux d'évaporation), les fourmis ont plus de chances de suivre les meilleures solutions des dernières vagues, même si elles sont mauvaises (dû au fait que leurs traces de phéromones sont généralement plus puissantes). Toutefois, c'est aussi plus dur de converger vers une solution avec une trop grande mémoire, puisque les nouvelles solutions, même si elles sont bonnes, contribuent moins à la trace de phéromones. On estime donc qu'un taux d'évaporation de 0.5 est optimal (après $c_{\acute{e}} = 0.5$, on voit dans le tableau que la performance diminue).

La fonction $f(d) = |d - 0.65 * \bar{l}|^{-1}$ (où $\bar{l}$ est la longueur moyenne d'un trajet $\approx Nl/2$ où l est la longueur de la boîte contenant les points et d est la distance) a été choisi comme fonction phéromones car elle encourage énormement les fourmis qui trouvent un chemin 0.65 fois le chemin moyen en terme de longueur (par le biais du fait que f diverge vers l'infini autour de cette valeur). Ce nombre a été choisis par essais-erreurs. Cette analyse étant loin de rigoureuse, elle n'est pas décrite en détail dans ce travail.

Sans surprise, on voit qu'un système de classement améliore les performances. Cela est visiblement dû au fait que le système de classement élimine les mauvaises solutions (en diminuant leurs phéromones) à l'intérieur d'une vague en plus de récompenser les meilleures. De plus, une récompense additionnelle aux fourmis qui ont trouvé de nouvelles solutions optimales permet de chercher plus autour de ces solutions plutôt qu'autour des anciennes.

Finalement, il faut aussi mentionner que même dans des cas d'échec, la solution trouvée n'est généralement pas trop loin de la solution optimale (elle varie, dans la majorité des cas, d'une ou deux transpositions de deux points, comme vous pouvez le voir dans l'exemple ci-dessous).

6 Conclusion

En conclusion, on constate que l'algorithme d'optimisation par colonie de fourmis fonctionne pour résoudre le problème du commis voyageur et pourrait être appliqué à une série de problèmes connexes où sa puissance réelle pourrait être mieux mise en valeur. L'analyse faite dans ce travail n'arrive pas à une conclusion claire face à l'application de cet algorithme sur le problème du commis voyageur avec contraintes. Une question intéressante dans ce contexte serait de comprendre comment mieux implémenter

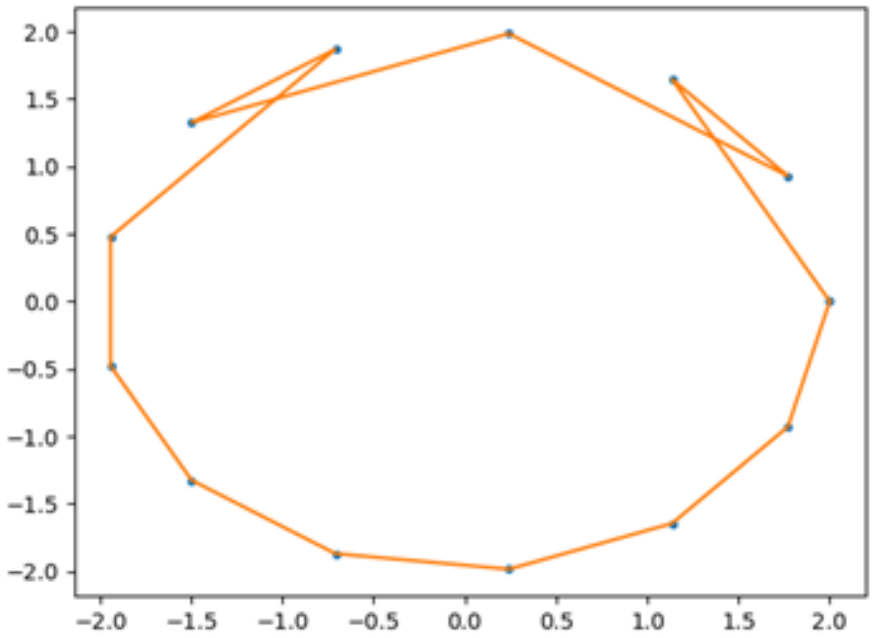

FIGURE 7 – Exemple typique de chemin érronné trouvé par l'algorithme lorsqu'appliqué à une série de points sur la circonférence d'un cercle.

des contraintes dans le problème du commis voyageur et d'avoir une condition exacte sur les contraintes qui sont légitimes.

Références

[1] TSAI, Chun-Wei ; CHIANG Ming-Chao. Handbook of Metaheuristic Algorithms : Chapter 8 - Ant Colony Optimization (DOI : https ://doi.org/10.1016/B978-0-44-319108-4.00021-6

[2] DORIGO, Marco : *Ant colony optimization* (doi :10.4249/scholarpedia.1461)

[3] DORIGO, Marco ; STÜZLE, Thomas. *Ant colony optimization* (ISBN 0-262-04219-3)