TP 3 : Voyageur de commerce dans le plan

Ce sujet est constitué de trois exercices, tous portant sur le problème du voyageur de commerce dans le plan. Le troisième exercice est optionnel (et plus difficile).

Bibliothèques Python utilisées. Pour celles et ceux effectuant le TP sur leur propre machine, vous aurez besoin de la bibliothèque `matplotlib` qu'il faut installer[1]. On utilise également les bibliothèques `math` et `random` qui font partie de la bibliothèque standard Python.

Rappel et notations. L'entrée du problème du voyageur de commerce dans le plan est une liste de points $p_0, \dots, p_{n-1}$ où $p_i = (x_i, y_i) \in \mathbb{R}^2$ est représenté par ses coordonnées dans le plan. En sortie, l'algorithme produit une permutation des sommets $[i_0, \dots, i_{n-1}]$ tel que le chemin $p_{i_0} \to p_{i_1} \to \cdots p_{i_{n-1}} \to p_{i_0}$ soit le plus court possible. Informatiquement, les points sont donnés comme une `list` de couples de `float`. La sortie est une `list` d'`int`.

Affichage graphique. Le fichier `dessins.py` fournit trois fonctions d'affichage graphique, basées sur `matplotlib`, qu'on doit importer en utilisant `from dessins import *` :

- `dessinPoint(Points)` affiche dans le plan les `Points` ;
- `dessinGraphe(Points, Adj)` affiche le graphe représenté par les listes d'adjacence `Adj` ;
- `dessinParcours(Points, Perm, Adj = {})` affiche le cycle représenté par la `Perm`utation, ainsi que le graphe représenté par `Adj` (en grisé) si `Adj` est fourni.

Exercice 1. *Fonctions préliminaires*

1. Écrire une fonction `distance(A, B)` qui prend en entrée deux points $A = (x_A, y_A)$ et $B = (x_B, y_B)$ et renvoie la distance euclidienne entre A et B. *Vous pouvez utiliser* `sqrt` *de la bibliothèque* `math`.

```
>>> A, B, C = (121,77), (48,70), (12,72)
>>> distance(A,B), distance(A,C), distance(B,C)
(73.33484846919642, 109.1146186356347, 36.05551275463989)
```

2. Écrire une fonction `aretes(P)` qui prend en entrée une liste de n points dans le plan, et renvoie la liste des $n(n-1)/2$ arêtes entre eux, sous forme de triplet (i, j, d) où d est la distance entre $P_{[i]}$ et $P_{[j]}$.

```
>>> P = [(6,20),(67,18),(96,4),(32,45)]
>>> aretes(P)
[(0, 1, 61.032778078668514),
 (0, 2, 91.41115905621152),
 (0, 3, 36.069377593742864),
 (1, 2, 32.202484376209235),
 (1, 3, 44.204072210600685),
 (2, 3, 76.00657866263946)]
```

3. **i.** Écrire une fonction `pointsAleatoires(n, xmax, ymax)` qui prend en entrée un entier n et deux réels positifs $x_{\max}$ et $y_{\max}$ et renvoie une liste de n points aléatoires (x, y) tels que $0 \leq x \leq x_{\max}$ et $0 \leq y \leq y_{\max}$. *Vous pouvez utiliser la fonction* `random()` *de la bibliothèque* `random` *qui renvoie un flottant entre* 0 *et* 1.

```
>>> pointsAleatoire(3, 10, 20)
[(9.763924343503144, 8.99578036522515),
 (7.760339550026788, 15.407684676706966),
 (3.2738998168257307, 2.0022671172103346)]
```

   ii. Représenter graphiquement les points obtenus avec `dessinPoints` pour vérifier qu'ils semblent bien aléatoires, et dans les bornes prévues.

4. **i.** Écrire une fonction `listesAdjacence(n, A)` qui prend en entrée un nombre n de sommets et une liste A d'arêtes (i, j), et qui renvoie un dictionnaire qui à chaque sommet $i \in \{0, \dots, n-1\}$ associe sa liste d'adjacence.

1. Voir `https://matplotlib.org/stable/users/installing/index.html`.

```
>>> A = [(0,1),(0,2),(0,3),(1,2),(1,3),(2,3)]
>>> listesAdjacence(4, A)
{0: [1, 2, 3], 1: [0, 2, 3], 2: [0, 1, 3], 3: [0, 1, 2]}
```

ii. Représenter graphiquement le graphe obtenu avec `dessinGraphe` et vérifier qu'il est correct.

Exercice 2. *Algorithme de 2-approximation*

On rappelle que l'algorithme de 2-approximation pour le voyageur de commerce dans le plan consiste à calculer un *arbre couvrant de poids minimal* des points en entrée, puis à effectuer un *parcours en profondeur* de l'arbre couvrant, afin d'en déduire une permutation des sommets.

1. **Arbre couvrant de poids minimal.** Le but de cette question est d'implanter (une version simplifiée de) l'algorithme de Kruskal vu en L2.

   1. $A \leftarrow \emptyset$ (*arbre vide*)
   2. $C \leftarrow [0, \dots, n-1]$ (*tableau des composantes*)
   3. $m \leftarrow n$ (*nombre de composantes*)
   4. Pour chaque arête (u, v), par poids croissants :
   5. Si $C_{[u]} \neq C_{[v]}$: *(u et v dans des composantes distinctes $\rightarrow$ l'arête ne crée pas de cycle)*
   6. Ajouter (u, v) à A
   7. Pour chaque sommet s tel que $C_{[s]} = C_{[v]}$: $C_{[s]} \leftarrow C_{[u]}$
   8. $m \leftarrow m-1$ et sortir de la boucle si $m = 1$
   9. Renvoyer A

   i. Écrire une fonction `arbreCouvrant(Aretes)` qui prend en entrée une liste d'arêtes renvoyées par la fonction `aretes`, et renvoie un arbre couvrant sous forme de liste d'adjacence. *Indication. Calculer la liste d'arêtes de l'arbre couvrant comme dans l'algorithme ci-dessus, puis sa représentation sous forme de listes d'adjacence à l'aide de* `listesAdjacence`.

   ```
   >>> A = aretes([(6,20),(67,18),(96,4),(32,45)])
   >>> arbreCouvrant(4, A)
   {0: [3], 1: [2, 3], 2: [1], 3: [0, 1]}
   ```

   ii. **Tester** votre fonction avec des points aléatoires, en vérifiant le résultat graphiquement. Trouver le nombre maximal [2] de points que vous arrivez à traiter en moins de 10 secondes.

2. **Parcours de l'arbre couvrant.** On implante maintenant un parcours en profondeur de l'arbre couvrant.

   1. $P \leftarrow [0]$ (*pile contenant le sommet* 0)
   2. $C \leftarrow \emptyset$ (*chemin du parcours, vide initialement*)
   3. Tant que P est non vide:
   4. $s \leftarrow$ DÉPILER(P)
   5. Ajouter s à C
   6. Pour chaque voisin v de s :
   7. Si v n'est pas dans C :
   8. EMPILER v sur P
   9. Renvoyer C

   i. Écrire une fonction `parcoursArbre(n, Adj)` qui prend en entrée le nombre n de sommets et la représentation de l'arbre couvrant sous forme de liste d'adjacence, et renvoie une permutation des sommets correspondants au parcours en profondeur de l'arbre couvrant.

   ii. Tester graphiquement votre fonction, à l'aide de la fonction `dessinParcours`.

3. **Algorithme complet.**

   i. Écrire une fonction `VdC(Points)` qui prend en entrée une liste de points et renvoie une permutation des sommets et la longueur du chemin correspondant ; tester votre fonction graphiquement. La figure 1 montre le résultat attendu sur l'entrée (longueur du chemin $\simeq 426$) : `[(6,60),(67,62),(96,76),(32,35),(70,39),(98,24),(129,30),(121,3),(48,10),(12,8)]`.

   ii. Appliquer votre algorithme sur des points aléatoires. Jusqu'à quel nombre de points votre algorithme trouve-t-il la solution en moins de 10 secondes ?

2. On cherche une valeur approchée, par exemple de de la forme $c \cdot 10^k$ où $0 < c < 10$.

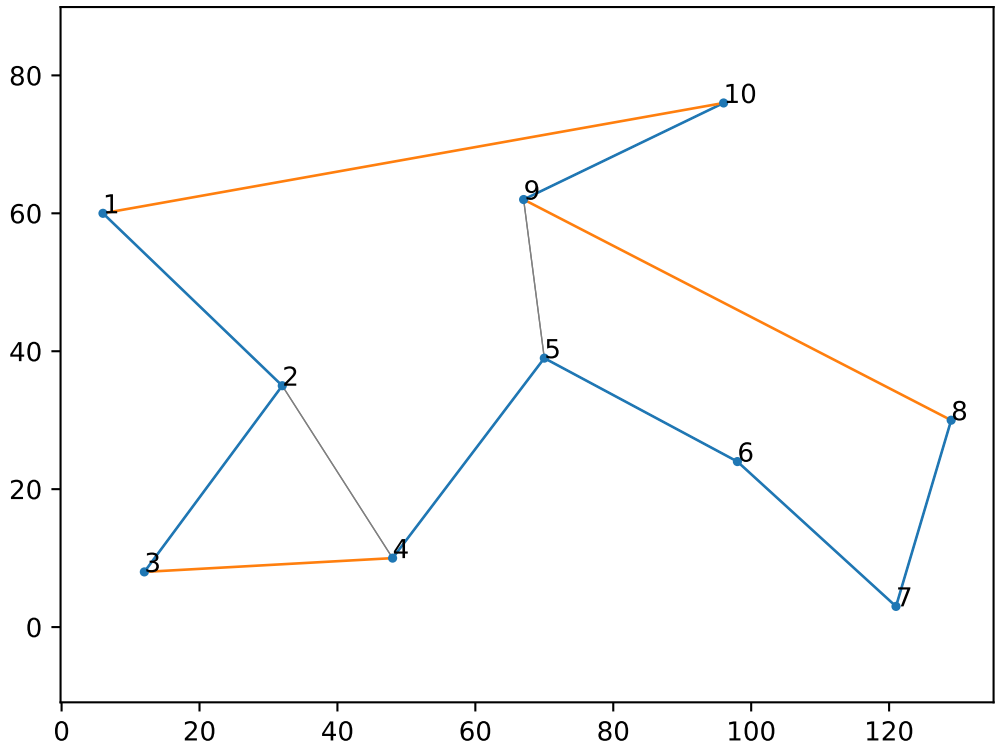

FIGURE 1 – Exemple de voyageur de commerce, avec trois type d'arêtes représentées : en bleu, les arêtes de l'arbre couvrant utilisées dans le parcours, en orange les *raccourcis* et en grisé les arêtes de l'arbre non utilisées. Les numéros représentent l'ordre de parcours.

Exercice 3. *Algorithme de Christofides*

Le but est d'implanter l'algorithme complet de Christofides. Pour cela, on a besoin de trois ingrédients supplémentaires : étant donné l'arbre couvrant, il faut calculer l'ensemble des sommets de degré impair ; ensuite, il faut calculer un *couplage de poids minimal* sur ces sommets ; enfin, il faut effectuer un *parcours eulérien* du multigraphe obtenu en combinant l'arbre couvrant et le couplage.

1. Écrire une fonction `sommetsImpairs(Adj)` qui prend en entrée les listes d'adjacence et renvoie la liste des sommets de degrés impairs.
2. Un *couplage* de l'ensemble S des sommets impairs est un ensemble d'arêtes tel que chaque sommet appartient exactement à une des arêtes de l'ensemble. Calculer un couplage de poids minimal est ardu. On va remplacer ce calcul exact par une *heuristique*, c'est-à-dire un algorithme qui ne fournit pas toujours la solution optimale mais qui n'est pas trop mauvais en pratique : on tire un sommet aléatoire et on choisit de le *coupler* avec le sommet dont il est le plus proche ; on recommence tant qu'il reste des sommets.
   Écrire une fonction `couplage(Points, S)` qui implante cet algorithme et renvoie la liste des arêtes du couplage.
3. Pour finir, il faut créer le multigraphe constitué de l'arbre couvrant et du couplage, et effectuer un *parcours eulérien* de ce multigraphe, c'est-à-dire un chemin qui passe par chaque arête une et une seule fois. L'algorithme est esquissé ci-dessous. La *fusion* de deux cycles consiste à les *recoller* au niveau d'un sommet qu'ils possèdent en commun.

   ```
   Choisir un sommet initial s
   Créer un cycle partiel C à partir de s : suivre des arêtes jusqu'à retourner à s, sans emprunter deux fois la même arête
   Tant qu'il existe des sommets v dans C avec des arêtes non visitées :
       Créer un cycle partiel C' à partir de v
       Remplacer C par la fusion de C et C'
   ```

   i. Écrire une fonction `multigraphe(Adj, C)` qui prend en entrée l'arbre couvrant (listes d'adjacence) et le couple (liste d'arêtes) et *modifie* `Adj` pour y ajouter les arêtes de C.
   ii. Écrire une fonction `cycle(Adj, s)` qui prend en entrée les listes d'adjacence et un sommet S et renvoie un cycle partiel à partir de s. *La fonction* `cycle` *modifie* `Adj` *en supprimant les arêtes utilisées dans le cycle partiel.*
   iii. Écrire une fonction `fusion(C1, C2)` qui fusionne les deux cycles C_1 et C_2 en un unique cycle C. *En fonction de la façon dont* `cycle` *a été codée, vous pouvez supposer que le sommet commun à C_1 et C_2 est le premier ou le dernier de C_2 : à déterminer en fonction de vos choix.*
   iv. Écrire une fonction `parcoursEulerien(n, Adj)` qui prend en entrée le nombre n de sommets et les listes d'adjacence du multigraphe, et renvoie un parcours eulérien du multigraphe.

4. Écrire une fonction `Christofides(P)` qui prend en entrée une liste de points et renvoie une permutation des sommets, en appliquant l'algorithme de Christofides. Tester et comparer la longueur des chemins obtenus avec cet algorithme et avec l'algorithme de l'exercice précédent.