TP4: Algorithmes d'approximation, 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 but de ces exercices est d'implémenter la 2-approximation du Voyageur de Commerce vue en cours.

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, ..., 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 $[i_0, \ldots, i_{n-1}]$ 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 quatre 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` ;
- `dessinArbre(Point, AdjArbre, AdjGraphe)` affiche le graphe dont la liste d'adjacence est `AdjGraphe` ainsi que son arbre couvrant dont la liste d'adjacence est `AdjArbre` ;
- `dessinParcours(Points, Perm, Adj = {})` affiche le cycle représenté par la `Permutation`, ainsi que le graphe représenté par `Adj` (en grisé) si `Adj` est fourni.

Exercice 1. *Routines utiles*

Implémenter les fonctions suivantes.

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)
print(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)]
print(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. 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.

```
print(pointsAleatoires(3, 10, 20))

>>>[(9.763924343503144, 8.99578036522515),
>>>(7.760339550026788, 15.407684676706966),
>>>(3.2738998168257307, 2.0022671172103346)]
```

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

4. Représenter graphiquement les points obtenus avec `dessinPoints` pour vérifier qu'ils semblent bien aléatoires, et dans les bornes prévues.
5. 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 (c'est-à-dire, pour tout sommet du graphe, on lui associe la liste des sommets qui lui sont adjacents).

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

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

Exercice 2. *Algorithme de* KRUSKAL

1. Écrire une fonction `arbreCouvrant(n, ListePoints)` qui prend en entrée un entier n et une liste des coordonnées de n points du plan comme renvoyée par la fonction `pointsAleatoires`, et renvoie la liste d'arêtes d'un arbre couvrant du graphe complet dont les sommets sont les points passés en paramètres et les arêtes sont pondérés par leur longueur dans le plan (comme retournées par la fonction `aretes`). *Pour cela, implémenter l'algorithme de Kruskal vu en cours*

```
1 ListePoints =([(6,20),(67,18),(96,4),(32,45)])
2 print(arbreCouvrant(4, ListePoints))
3
4 >>>[(0,3), (1,2), (1,3)]
```

2. Vérifier que vous obtenez bien les structures voulues en représentant graphiquement les points, le graphe complet calculé sur ceux-ci et l'arbre couvrant calculé. Utilisez pour cela les appels `DessinGraphe` et `DessinArbre`. *Attention, pour la liste d'arête du graphe à passer en paramètre, les primitives de dessin veulent un tableau d'arêtes non-pondérées. Il faut donc enlever les dernières coordonnées des éléments du tableau retourné par* `arête` *avant de le passer en paramètre à* `DessinGraphe` *et* `DessinArbre`.

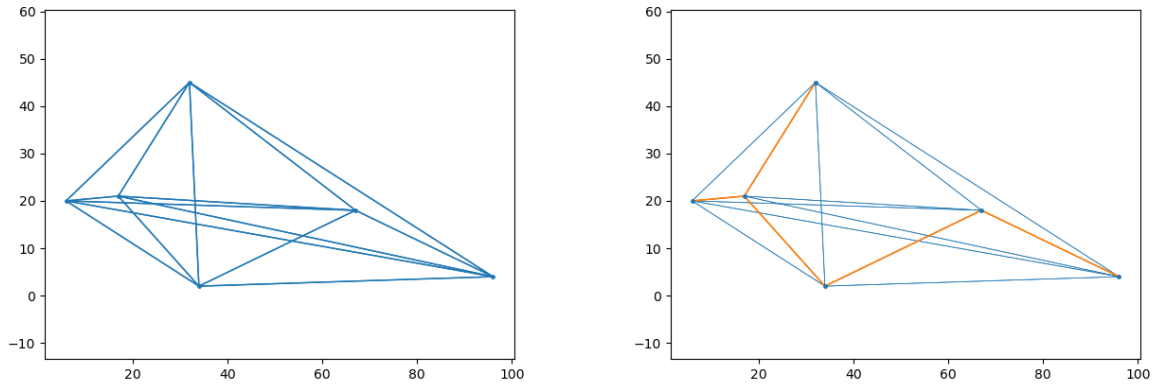

FIGURE 1 – Le graphe complet construit sur un nuage de points du plan et un arbre couvrant de longueur minimale de ce graphe.

3. Chercher le nombre maximum de points que votre programme peut traiter en moins de 10s.

Exercice 3. *2-Approximation du Voyageur de Commerce*

Dans ce dernier exercice, on implémente la 2-approximation du Voyageur de Commerce.

1. La première chose à faire est d'implémenter un parcours d'arbre en profondeur, qui est redonné ci-dessous pour mémoire. Votre algorithme prendra en entrée un graphe donné par *liste d'adjacence*, et retournera l'ordre de découverte des sommets au cours du parcours (ce que contient C ci-dessus).

    1. $P \leftarrow [0]$ (*pile contenant le sommet* 0)

2. $C \leftarrow \emptyset$ (*ordre 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

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

3. Écrire une fonction `VdC(Points)` qui prend en entrée une liste de points et renvoie une permutation des sommets correspondant à une tournée de la 2-approximation du Voyageur de Commerce et la longueur du chemin correspondant ; tester votre fonction graphiquement. La figure suivante 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)]
```

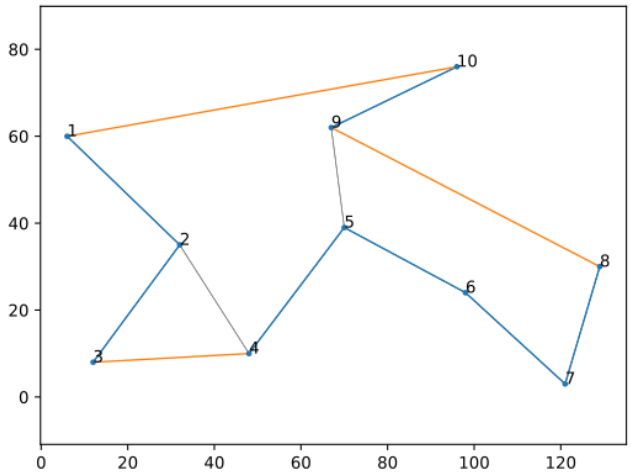

FIGURE 2 – Exemple de voyageur de commerce, avec trois type d'arêtes représentées : en bleu, les arêtes de l'arbre couvrant utilisées dans le parcours, en orange les *raccourcis* et en grisé les arêtes de l'arbre non utilisées. Les numéros représentent l'ordre de parcours.

4. 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 ?