TP5: Gloutons

MP2I Option SI: Informatique tronc commun

Voyageur de commerce

On considère n villes $V_0, \dots, V_{n-1}$, de coordonnées respectives $(x_0, y_0), \dots, (x_{n-1}, y_{n-1})$. Un voyageur de commerce souhaite visiter les n villes pour y vendre ses marchandises. Son objectif est de partir d'une des villes V_i, de visiter chaque ville une et une seule fois, puis de revenir en V_i. Afin d'être rentable, son trajet doit être le plus court possible. On considèrera qu'il se déplace en ligne droite, de telle sorte que la distance qu'il parcourt pour aller d'une ville V_i à une ville V_j est la distance euclidienne entre les deux.

On représentera les villes en Python par une liste de couples. L'archive du TP contient un fichier définissant :

— Une fonction `gen_villes(n)` générant une liste de n points aléatoires

— Une fonction `dessiner_chemin(villes, chemin)` prenant en entrée une liste de villes, ainsi qu'une liste d'indices de villes $[i_0, i_1, \dots, i_{p-1}]$ formant un chemin. La fonction trace le chemin entre les différentes villes. Le chemin est considéré comme cyclique, de telle sorte que les villes i_{p-1} et i_0 sont reliées.

Un exemple d'utilisation est donné. L'objectif de la première partie est d'implémenter un algorithme glouton de résolution du problème du voyageur de commerce, et de vérifier graphiquement que les chemins tracés sont raisonnablement bons.

Q1. Écrire une fonction prenant en entrée deux points P_1, P_2 dans le plan et renvoyant la distance euclidienne entre P_1 et P_2.

Q2. Écrire une fonction `matrice_distances(villes)` prenant en entrée une liste de points $[(x_0, y_0), \dots (x_{n-1}, y_{n-1})]$, et renvoyant une matrice numpy M de taille $n \times n$, telle que M_{ij} est la distance entre les points i et j.

Un algorithme glouton simple pour résoudre le problème du voyageur de commerce est de partir d'une ville aléatoire V_i et, tant qu'il reste une ville non-visitée, de se rendre vers la ville non-visitée la plus proche.

Q3. Trouver un contre-exemple montrant que cet algorithme n'est pas toujours optimal.

Q4. Écrire une fonction `plus_proche_voisin(M, vu, i)` prenant en entrée :

— `M` la matrice $n \times n$ renvoyée par la fonction `matrice_distances`,

— un tableau `vu` de n booléens tel que `vu[j]` vaut `True` si et seulement si la ville j a déjà été visitée,

— un indice de ville i.

Cette fonction renverra l'indice de la ville non-visitée la plus proche de la ville i.

Q5. En déduire une fonction `voyageur_glouton(villes)` prenant en entrée une liste de villes, et renvoyant un couple (C, d) où C est le chemin obtenu en appliquant l'algorithme glouton décrit plus haut, et d la distance totale de ce chemin.

Q6. Donner les complexités en fonction de n des fonctions `matrice_distances`, `plus_proche_voisin`, et `voyageur_glouton`.

Couverture de graphe

On considère une ville constituée de rues et d'intersections. La mairie souhaite construire des monuments sur certaines intersections, de façon à ce que sur chaque tronçon de rue, on puisse voir au moins un monument. L'objectif est de réaliser cet objectif en construisant le moins de monuments possibles.

On peut modéliser ce problème comme un problème d'optimisation sur les graphes : Étant donné un graphe $G = (S, A)$, on cherche un ensemble de sommets $S' \subseteq S$ de taille minimale tel que pour toute arête $(u, v) \in A$, on a soit $u \in S$ soit $v \in S$.

On représentera les graphes par tableaux de listes d'adjacence : un graphe G sur n sommets sera représenté en python par un tableau `G` de taille n tel que pour $i \in [\![0, n-1]\!]$, `G[i]` contient la liste des sommets voisins du sommet i.

On rappelle les opérations des ensembles Python :

```python
# créer un ensemble vide
s = set()

# ajouter un élément à un ensemble
s.add((1, 2))

# supprimer un élément d'un ensemble
s.remove((1, 2))

# retirer un élément arbitraire d'un ensemble et le renvoyer
x = s.pop(s)

# savoir si un ensemble est non-vide
if s:
  ... # code exécuté uniquement si s non vide
```

Q7. Écrire une fonction `aretes(G)` renvoyant l'ensemble des arêtes d'un graphe. Pour une arête (i, j), on ajoutera à l'ensemble uniquement le couple $(\min(i, j), \max(i, j))$, et pas le couple symétrique, afin d'éviter les ambiguïtés.

On propose un premier algorithme glouton : tant qu'il existe une arête non couverte, choisir un de ses sommets au hasard, le rajouter à la solution.

Q8. Compléter le code suivant pour implémenter l'algorithme glouton proposé :

```python
def glouton_VC(G):
  solution = []
  aretes_restantes = aretes(G)
  while ...:
    # prendre une arête non couverte au hasard, choisir un de ses sommets
    a = aretes_restantes.pop()
    i = a[0] # on pourrait aussi prendre a[1], ou choisir au hasard
    solution.append(i)
    ... # supprimer les autres arêtes incidentes à i

  return solution
```

Q9. Montrer que cet algorithme n'est pas optimal, en exhibant un contre-exemple.

On considère maintenant un deuxième algorithme glouton, qui va consister à sélectionner en priorité les sommets ayant un grand degré, puisqu'il couvriront plus d'arêtes :

Algorithme 1 : `glouton_degre`(G)

Entrée(s) : $G = (S, A)$ un graphe non-orienté
Sortie(s) : L une liste de sommets couvrant toutes les arêtes de G

```
1 S ← [] // liste solution
2 tant que G contient un sommet de degré > 0 faire
3     u ← sommet de G de degré maximal;
4     L.append(u);
5     Supprimer s (et ses arêtes incidentes) de G;
6 retourner L
```

Q10. Appliquer l'algorithme à la main sur quelques graphes. Est-il optimal ?

Afin d'éviter de modifier le graphe en entrée, nous allons utiliser une structure simple permettant de stocker l'état actuel du graphe au cours de l'exécution : on stocke dans un tableau le degré **actuel** de chaque sommet : pour supprimer un sommet, on fixe son degré à 0, et on baisse le degré de tous ses voisins de 1.

Q11. Écrire une fonction `degres(G)` renvoyant un couple (T, d), où T est le tableau des degrés de G, et d la somme de tous les degrés.

Q12. Écrire une fonction `sommet_max(T)` prenant en entrée le tableau des degrés, et renvoyant l'indice i d'un sommet de degré maximal.

Q13. Écrire une fonction `supprimer_sommet(G, T, i, d)` prenant en entrée un graphe G, le tableau des degrés actuels T, l'indice d'un sommet i à supprimer, et d la somme des degrés de T. Cette fonction modifiera T pour supprimer i, et renverra la valeur mise à jour de d.

Q14. Implémenter l'algorithme `glouton_degre` en Python, en utilisant les fonctions précédentes.