Séance 9: Calculer des itinéraires routiers

Université Paris-Diderot

Objectifs:

- Travailler avec des listes de différents types et des listes de listes
- Utiliser les boucles `for` et `while`

Dans ce TP nous allons travailler avec des listes de listes qui représentent une carte routière (simplifiée) de la France. À l'aide des boucle `while` et `for` nous allons implémenter un simple algorithme permettant de calculer un itinéraire (sous-)optimal entre deux villes.

Rédiger les réponses aux questions et les tests pour chaque réponse dans le fichier `Itineraire.py` qui vous est fourni.

1 Chargement de la carte routière

Des données décrivant une carte routière sont disponibles dans les fichiers *cities.csv* et *routes.csv*. La procédure `loadData()` qui vous est fournie dans `Itineraire.py`, s'occupe de charger ces données dans les listes `cityName`, `cityCoord`, `adjCities`, `route` et `nRoutes`. Ces variables sont accessibles par toutes les fonctions que vous aller écrire dans `Itineraire.py`.

Voici ce que ces listes contiennent après l'exécution de `loadData()` :

- `cityName` décrit les codes et les noms des villes. Chaque ville a un code entier unique compris entre 1 et 232 (inclus). Pour une ville de code `i`, `cityName[i]` est le nom de la ville `i`.
- `cityCoord` contient les coordonnées des villes dans le plan (en km). Pour une ville de code `i`, `cityCoord[i][0]` et `cityCoord[i][1]` représentent les coordonnées X et Y de la ville `i`, respectivement.
- `adjCities` est une liste de listes qui décrit les routes entre villes. S'il existe une route entre deux villes, on dit qu'elles sont *voisines*. Chaque ville a au plus 10 villes voisines. Pour une ville de code `i`, `adjCities[i]` est une liste contenant les villes voisines de `i` (l'ordre des voisines dans cette liste ne sera pas important pour nous). Donc `adjCities[i][j]` c'est le code du `j`-eme ville voisine de `i`.
- `route` est une liste de listes qui décrit les longueurs des routes. Plus précisément, `route[i][j]` est la longueur (en km) de la route qui lie la ville de code `i` à sa `j`-eme ville voisine (remarquez que cette voisine est la ville de code `adjCities[i][j]`).

Chaque route peut être parcourue dans les deux sens donc, dans les listes ci-dessus, si la ville `k` est parmi les voisines de `i`, alors la ville `i` est parmi les voisines de `k`.

2 Trouver les itinéraires

Pour trouver le "meilleur" itinéraire d'une ville de départ D à une ville d'arrivée A, on va utiliser la stratégie suivante.

1. Se placer dans la ville de départ D ;
2. Depuis la ville courante se déplacer dans sa ville voisine la plus "proche" de A, si une ville voisine existe ;
3. Répéter l'étape précédente jusqu'à se placer sur A (ou jusqu'à ce qu'il ne soit plus possible de se déplacer).

Ici pour mesurer combien une ville V est "proche" de A on va utiliser les coordonnées de V et A pour calculer leur distance (à vol d'oiseau) sur le plan à deux dimensions. Attention à ne pas confondre cette distance avec la longueur d'une route. En effet V et A ne sont pas forcement connectées par une route.

De plus on va chercher uniquement des itinéraires simples (c'est à dire qui ne passent jamais deux fois par la même ville). On restreindra donc la recherche de l'étape 2 aux villes qui ne font pas déjà partie de l'itinéraire.

La stratégie plus haut ne garantit pas de trouver l'itinéraire le plus court (ni de trouver un itinéraire !), mais peut trouver des itinéraires la plupart du temps "satisfaisants" sur des réseaux routiers suffisamment connectés.

Suivre les étapes suivantes.

Exercice 1 (Calculer la distance à vol d'oiseau, $\star$)
Écrire une fonction `distCoord` *qui attend en paramètre le code d'une ville de départ et le code d'une ville d'arrivée et renvoie leur distance (entière) calculée à partir des coordonnées des deux villes. Pour calculer la partie entière de la racine carrée, utilisez la fonction* `sqrt` *comme exemplifiée en bas :*

```
from math import sqrt
int(sqrt(x)) # partie entiere de la racine carree de x
```

□

Exercice 2 (Calculer une étape de l'itinéraire, $\star\star$)
Écrire une fonction `findClosest` *pour la recherche de la ville la plus proche d'une ville d'arrivée. Cette fonction reçoit le code d'une ville de départ V, le code d'une ville d'arrivée A, et une liste de booléens* `except`*. Pour une ville de code* i*,* `except[i]` *est* `True` *si* i *doit être exclue de la recherche, et* `False` *sinon.*

La fonction `findClosest` *renvoie l'indice de la ville voisine de V la plus proche de A (selon* `distCoord`*), parmi les voisines de V marquées* `False` *dans la liste* `except`*. La fonction renvoie -1 si une telle voisine n'existe pas.*

Remarquer que la valeur de retour n'est pas le code de la ville voisine trouvée, mais l'indice de cette ville dans la liste des voisines de V.

Par exemple si V à 3 voisines `[V1, V2, V3]`*, si* `distCoord(V, V1) == 34`*,* `distCoord(V, V2) == 45`*,* `distCoord(V, V3) == 40`*,* `except[V1] == True`*,* `except[V2] == except[V3] == False`*, alors la fonction* `findClosest` *renvoie l'indice de* `V3`*, c'est-à-dire 2.*

□

Exercice 3 (Calculer un itinéraire, $\star\star\star$)
Écrire une fonction `findRoute` *qui reçoit le code d'une ville de départ, le code d'une ville d'arrivée et renvoie la longueur d'un itinéraire trouvé selon la stratégie décrite au début de la section 2. La longueur d'un itinéraire est la somme des longueurs des routes dont il est composé.*

Suggestion. Pendant le calcul de l'itinéraire, maintenir une liste de booléens `visited`*, tel que* `visited[i] == True` *si la ville* `i` *a déjà été ajoutée à l'itinéraire. Consulter cette information à chaque étape de la stratégie pour éviter de se déplacer dans une ville déjà rencontrée.*

Prévoir de renvoyer -1 dans le cas où la stratégie ne trouve aucun itinéraire.

Tester la fonction, après avoir initialisé les listes avec `loadData`.

> ***Contrat:***
> ***Entrée*** *: ville de départ : 5 (Paris), ville d'arrivée : 59 (Bordeaux)*
> ***Sortie*** *: 631*
> ***Entrée*** *: ville de départ : 230 (Arles), ville d'arrivée : 154 (Auch)*
> ***Sortie*** *: 453*

□

Exercice 4 (Afficher l'itinéraire, ⋆)
Modifier la fonction de l'exercice 3 pour qu'elle affiche sur la console les noms de toutes les villes de l'itinéraire calculé, y compris la ville de départ et d'arrivée.

Tester la fonction, après avoir initialisé les listes avec `loadData`.

> ***Contrat:***
> ***Entrée*** *: ville de départ : 5 (Paris), ville d'arrivée : 59 (Bordeaux)*
> ***Sortie*** *:*
> *Paris Chartres Le Mans Angers Chôlet Parthenay Niort Saintes Bordeaux*
> *631*
>
> ***Entrée*** *: ville de départ : 230 (Arles), ville d'arrivée : 154 (Auch)*
> ***Sortie*** *:*
> *Arles Montpellier Millau Albi Toulouse Auch*
> *453*

□

Exercice 5 (Itinéraire par nom, ⋆)
Écrire une fonction `findRouteByName` *qui est identique à* `findRoute` *mais reçoit les noms des deux villes au lieu de leur code. Tester cette fonction comme à l'exercice 4, sur des couples de villes françaises de votre choix. Sur une carte de la France, vérifier la plausibilité de l'itinéraire calculé.*

Vous pouvez aussi vérifier la "qualité" de l'itinéraire en comparant sa longueur à celle d'un itinéraire trouvé sur le Web.

□