Outils numériques et programmation

Examen final, Janvier 2015

Consignes :
— Vous avez accès à tout l'internet « statique » (hors mail, tchat, forum, etc.), y compris donc au cours en ligne.
— Ne soumettez pas de codes non-fonctionnels (i.e. provoquant une exception à l'interprétation, avant même l'exécution) : les erreurs de syntaxe seront lourdement sanctionnées.
— Respectez scrupuleusement les directives de l'énoncé (nom des variables, des méthodes, des fichiers, etc.), en particulier concernant le nom des fichiers à renvoyer aux correcteurs.
— Envoyez vos codes aux *deux* adresses suivantes : elise.dumont@ens-lyon.fr *et* ycopin@ipnl.in2p3.fr

1 Exercice

Un appareil de vélocimétrie a mesuré une vitesse à intervalle de temps régulier puis à sorti le fichier texte « `sample.txt` » ci-joint (attention à l'entête). Vous écrirez un script python « `nom_prénom.py` » (sans accent) utilisant `matplotlib` qui générera, affichera et sauvegardera sous le nom « `nom_prénom.pdf` » une figure composée de trois sous-figures, l'une au dessus de l'autre :

1. la vitesse en mm/s mesurée en fonction du temps,
2. le déplacement en mètres en fonction du temps. On utilisera volontairement une intégration naïve à partir de zéro via la fonction `numpy.cumsum`,
3. l'accélération en m/s^2 en fonction du temps. On utilisera volontairement une dérivation naïve à deux points :
$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$
via la fonction `numpy.diff`. Attention, si l'entrée de cette fonction est un tableau de taille *N*, sa sortie est un tableau de taille *N-1*.

Le script doit lire le fichier `sample.txt` dans le répertoire courant. On prendra soin des noms des axes et des unités physiques. Si les trois axes des abscisses sont identiques, seul celui de la troisième sous-figure peut être nommé.

2 Le problème du voyageur de commerce

2.1 Introduction

Le problème du voyageur de commerce est un problème d'optimisation consistant à déterminer le plus court chemin reliant un ensemble de destinations. Il n'existe pas d'algorithme donnant la solution optimale en un temps raisonnable (problème NP-complet), mais l'on peut chercher à déterminer des solutions approchées.

On va se placer ici dans le cas d'un livreur devant desservir une seule fois chacune des n destinations d'une ville américaine où les rues sont agencées en réseau carré (Fig. 1). On utilise la « distance de Manhattan » (norme L1) entre deux points $A(x_A, y_A)$ et $B(x_B, y_B)$:

$$d(A,B) = |x_B - x_A| + |y_B - y_A|.$$

En outre, on se place dans le cas où les coordonnées des destinations sont *entières*, comprises entre `0` (inclus) et `TAILLE = 50` (exclus). Deux destinations peuvent éventuellement avoir les mêmes coordonnées.

Les instructions suivantes doivent permettre de définir les classes nécessaires (`Ville` et `Trajet`) et de développer deux algorithmes approchés (heuristiques) : l'algorithme du plus proche voisin, et l'optimisation 2-opt. Seules la librairie standard et la librairie `numpy` sont utilisables si nécessaire.

Un squelette du code, définissant l'interface de programmation et incluant des tests unitaires (à utiliser avec `py.test`), vous est fourni : `exam.py`. Après l'avoir renommé « `exam_nom_prénom.py` » (sans accent), l'objectif est donc de compléter ce code progressivement, en suivant les instructions suivantes.

Une ville-test de 20 destinations est fournie : `ville.dat` (Fig. 1), sur laquelle des tests de lecture et d'optimisation seront réalisés.

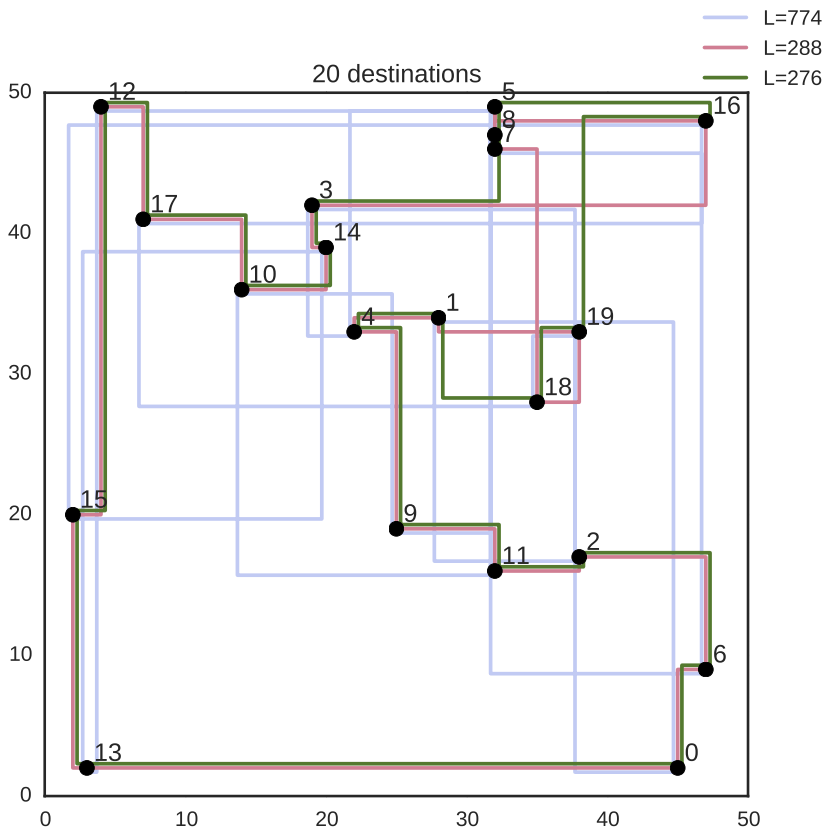

FIGURE 1 – Ville-test, avec 20 destinations et trois trajets de longueurs différentes : un trajet aléatoire (*L=774*), un trajet *plus proche voisins* (*L=288*), et un trajet après optimisation *opt-2* (*L=276*).

2.2 Classe `Ville`

Les n coordonnées des destinations sont stockées dans l'attribut `destinations`, un tableau `numpy` d'entiers de format `(n, 2)`.

1. `__init__()` : initialisation d'une ville sans destination (déjà implémenté, ne pas modifier).
2. `aleatoire(n)` : création de n destinations aléatoires (utiliser `numpy.random.randint`).
3. `lecture(nomfichier)` : lecture d'un fichier ASCII donnant les coordonnées des destinations.
4. `ecriture(nomfichier)` : écriture d'un fichier ASCII avec les coordonnées des destinations.
5. `nb_trajet()` : retourne le nombre total (entier) de trajets : $(n-1)!/2$ (utiliser `math.factorial`).
6. `distance(i, j)` : retourne la distance (Manhattan-L1) entre les deux destinations de numéro i et j.

2.3 Classe `Trajet`

L'ordre des destinations suivi au cours du trajet est stocké dans l'attribut `etapes`, un tableau `numpy` d'entiers de format `(n,)`.

1. `__init__(ville, etapes=None)` : initialisation sur une `ville`. Si la liste `etapes` n'est pas spécifiée, le trajet par défaut est celui suivant les destinations de `ville`.
2. `longueur()` : retourne la longueur totale du trajet *bouclé* (i.e. revenant à son point de départ).

2.4 Heuristique *Plus proche voisin*

1. `Ville.plus_proche(i, exclus=[])` : retourne la destination la plus proche de la destination i (au sens de `Ville.distance()`), *hors* les destinations de la liste `exclus`.
2. `Ville.trajet_voisins(depart=0)` : retourne un `Trajet` déterminé selon l'heuristique des plus proches voisins (i.e. l'étape suivante est la destination la plus proche hors les destinations déjà visitées) en partant de l'étape initiale `depart`.

2.5 Heuristique *Opt-2*

1. `Trajet.interversion(i, j)` : retourne un *nouveau* `Trajet` résultant de l'interversion des 2 étapes i et j.
2. `Ville.optimisation_trajet(trajet)` : retourne le `Trajet` le plus court de tous les trajets « voisins » à `trajet` (i.e. résultant d'une simple interversion de 2 étapes), ou `trajet` lui-même s'il est le plus court.
3. `Ville.trajet_opt2(trajet=None, maxiter=100)` : à partir d'un `trajet` initial (par défaut le trajet des plus proches voisins), retourne un `Trajet` optimisé de façon itérative par interversion successive de 2 étapes. Le nombre maximum d'itération est `maxiter`.

2.6 Questions hors-barême

À l'aide de la librairie `matplotlib` :

1. `Ville.figure()` : trace la figure représentant les destinations de la ville (similaire à la Fig. 1).
2. `Ville.figure(trajet=None)` : compléter la méthode précédente pour ajouter un trajet au plan de la ville (utiliser `matplotlib.step` pour des trajets de type « Manhattan »).