TP2 : Aide à la navigation d'un véhicule

Dans ce TP, nous allons mettre en œuvre des algorithmes de tri et de recherche pour implémenter en langage Python, un assistant à la navigation d'un véhicule.

Quelques consignes :
Ne pas modifier les noms des fonctions demandés
Commenter votre code pour expliquer votre intention de traitement

1. Étude de la taille du problème

Un véhicule se déplace sur un plan d'une surface de 100 km².
Nous avons à disposition une base de données contenant un million de points d'intérêt dont les coordonnées sont exprimées dans ce plan avec une précision au mètre, voici un extrait du fichier de base de données :

```
Mall|1528|2103
Gaz-station|3266|6986
Restaurant|9909|7229
Hospital|5062|6148
Gaz-station|886|3048
Parking|5221|7227
```

Le fournisseur de la base de données indique la description suivante :

```
File lines content : <category>|<x_coordinate>|<y_coordinate>
Line description :
<category> : 1 to 16 alphanumeric characters
<x_coordinate> & <y_coordinate> : interger coords of POI in meters
Available category :
"Gaz-station",
"Mini-Market",
"Mall",
"Restaurant",
"Cinema",
"Hospital",
"Hotel",
"Parking",
"Bar"
```

a. Calculer la taille qu'occuperait la base de données si elle était chargée entièrement en mémoire.
b. En considérant que le plan sur lequel le véhicule se déplace, est un plan cartésien carré dont l'unité de mesure est en mètre, calculer sa largeur.
c. En considérant que chaque point d'intérêt occupe 1m² et que deux points ne peuvent être confondus, combien de points d'intérêt faudrait-il pour recouvrir entièrement le plan ?

2. Mise en place du modèle de données

a. Créer une fonction

```
def readDB(dbFileName)
```

prenant un nom de fichier de base de données et qui retourne la liste globale contenant les points d'intérêt de la base et le nombre de points trouvés.

3. Étude des fonctionnalités

A l'aide de notre système de navigation, l'utilisateur doit être capable de :

1- rechercher le point d'intérêt le plus proche

2- afficher le nombre de points d'intérêt par catégorie dans la base de données

3- afficher le nombre de points d'intérêt par catégorie dans un rayon de 25, 50, 100, 250, 500m

4- rechercher les 30 points d'intérêt les plus proches, triés du plus proche au plus éloigné

5- rechercher tous les points d'intérêt dans un rayon de 25, 50, 100, 250, 500m, triés du plus proche au plus éloigné

Notez que pour les fonctionnalités de recherche, nous souhaitons pouvoir préciser une catégorie de point ou non.

a. Pour chacun de ces traitements identifier les algorithmes à mettre en œuvre (parcours, tri, recherche…)

b. Pour chacun des algorithmes identifier leur complexité

c. En déduire pour chaque fonctionnalité, « la somme des complexités »

4. Développement des fonctions de recherche de points d'intérêt

a. Développer une fonction

```
getDistance( [xA,yA], [yA,yB] )
```

prenant en paramètre les coordonnées de 2 points d'intérêt et qui calcule la distance euclidienne entre ces deux points.

Pour rappel, la distance euclidienne se calcule avec la formule suivante :

$$AB = \sqrt{(x_B - x_A)^2 + (y_B - y_A)^2}.$$

b. Développer une fonction prenant en paramètre une liste de points d'intérêt et une position (x, y) de référence et qui retourne le point d'intérêt le plus proche de cette position.

```
getPoiClosest(tab,x, y, categorie)
```

c. Développer une fonction

```
getPoiInRange(tab, x, y, threshold, categorie )
```

prenant en paramètre une liste de points d'intérêt, une position (x, y) de référence et un seuil de distance et qui retourne une nouvelle liste de tous les points d'intérêt dont la distance par rapport au point de référence est inférieure ou égale au seuil de distance.

5. Développement du tri

a. Développer une fonction

```
void sort(Poi *tab, int size, int x, int y)
```

prenant en paramètre un tableau de points d'intérêts et une position (x, y) de référence et qui

retourne un tableau de points d'intérêt triés du plus proche au plus éloigné en fonction de leur distance par rapport à la position de référence. Vous implémenterez l'algorithme de tri de votre choix.

6. Assemblage des fonctionnalités

Avec les fonctions de base développées aux points 4 et 5. Développer les 6 fonctionnalités décrites dans la question 3.

Les fonctions devront porter les noms suivants :

```
1- getPoiClosest
2- printDbStats
3- printDbStats
4- getPoiThirtyClosest
5- getPoiClosestInRange
```

7. Étude d'une fréquence de mise à jour

Suite à une recherche, pour que le véhicule soit capable de suivre correctement un itinéraire, la liste des points d'intérêt doit être fréquemment mise à jour.

a. Pour chaque recherche, estimer le temps moyen de réponse en considérant 10^6 opérations par secondes et une base de donnée de 1 million de points d'intérêt.

b. Calculer la densité que représente 1 million de points d'intérêt sur un plan de 100km².

c. Estimer une distance moyenne entre le véhicule et son point d'intérêt le plus proche quelque soit la position du véhicule sur le plan.

d. Sachant que le véhicule se déplace à 20 km/h, proposer une fréquence de rafraîchissement de l'itinéraire.

8. Découpage du plan

A la vue des résultats de l'étude précédente, nous proposons de découper le problème en problèmes de taille inférieure en découpant le plan en carrés plus petits.

a. En comptant le temps de recherche et de tri, avec quelle surface maximale Σ, pourrait-on avoir un temps de réponse « raisonnable » ?

Comme la répartition n'est certainement pas équitable nous proposons de diviser par deux cette surface maximale, et nous proposons de ne lancer les recherches que dans la surface $S = \Sigma / 2$ dans laquelle se trouve le véhicule.

b. Dans ce cas de figure, déterminer à quel(s) moment(s) il devient nécessaire de calculer une nouvelle liste de points d'intérêt.

Nous proposons de découper le plan en N tuiles T de surface S.

c. Créer une nouvelle fonction

```
getPoiInTile(tab,  ul_x, ul_y, br_x, br_y)
```

prenant en paramètre la liste globale des points d'intérêt, la position du véhicule et les deux points de la diagonale de S et qui retourne une liste des points d'intérêt contenus dans la surface donnée.

d. Implémenter un tableau contenant N tableaux de points d'intérêts correspondant chacun à une tuile T_i avec $i \in [0, N-1]$

```
Tiles[][] // contient les points d'intérêt répartis dans N tableaux où N représente le nombre de surfaces S nécessaires pour recouvrir tout le plan
```

e. Modifier la fonction de lecture de la base de données (développée en 2.c) afin qu'elle charge tous les points d'intérêt en mémoire et les réparti dans les N tableaux

Lorsque le véhicule se déplace dans une tuile T_i il faut considérer également les 8 tuiles voisines pour connaître les points d'intérêt les plus proches. On comprend alors que s'il faut rechercher les points les plus proches dans 9 tuiles de surface S, nous allons de nouveau dépasser le temps de traitement souhaité. Pour pallier à ce problème nous proposons de découper chaque tuile T en 9 autres tuiles plus petites. Le plan se découpe alors en 9 x N tuiles t de surface s.

f. Étudier le nombre minimal et maximal de tuiles t qu'il faudrait charger lorsque le véhicule se déplace et qu'il fait appel à une des 5 fonctionnalités du système de navigation.

g. Créer la structure Tile qui représente une tuile T contenant 9 sous-tuiles :

```
class Tile :
    poiInTile ; // tableau contenant les points d'intérêts de la tuile
    // ci-dessous délimitation de la tuile
    upperLeftX ; // coordonnée en X du point le plus haut à gauche
    upperLeftY ; // coordonnée en Y du point le plus haut à gauche
    downRightX ; // coordonnée en X du point le plus bas à droite
    downRightY ; // coordonnée en Y du point le plus bas à droite
    subtiles ; // tableau contenant les 9 sous-tuiles
```

Lorsque Tile est une tuile *t*, subtiles est NULL

h. Créer une fonction

```
cut(tab, width)
```

prenant en paramètre la liste globale des points d'intérêt et une largeur de carré et retourne un tableau de *N* Tile

i. Créer une fonction

```
subcut(tile)
```

prenant en paramètre une tuile de type Tile et fabrique ses 9 sous-tuiles.

j. Créer une fonction

```
getPoiToLookAt(tiles, x, y)
```

prenant en paramètre le tableau des tuiles *T* et la position du véhicule et qui retourne un tableau contenant les points d'intérêt de la sous-tuile dans laquelle le véhicule est positionné ainsi que les points d'intérêt des sous-tuiles voisines. C'est ce tableau de points d'intérêt que l'on fournira aux 5 fonctions de recherche.

Pour vos tests vous avez à disposition 5 jeux de données :

```
map_mini_100m2_100_poi.db : mini plan de 100m² avec 100 points d'intérêt
map_1km2_10k_poi.db       : plan de 1 km² avec 10.000 points d'intérêt
map_10km2_100k_poi.db     : plan de 10 km² avec 100.000 points d'intérêt
map_100km2_500k_poi.db    : plan de 100 km² avec 500.000 points d'intérêt
map_100km2_1M_poi.db      : plan de 100 km² avec 1.000.000 points d'intérêt
```

Les images associées donnent une idée de la répartition des points sur le plan.

Pour vos tests sur le plan de 100km² vous pouvez utiliser les positions de véhicule suivantes

```
A (0 ;0)
B (4999 ; 4999)
C (3456 ; 7652)
D (9532 ; 8671)
E (9999 ; 9999)
F (9999 ; 0)
G (0 ; 9999)
H (1234 ; 4567)
I (2889 ; 6324)
```