Université d'Ottawa
Faculté de génie

École de science
d'informatique
et de génie électrique

uOttawa
L'Université canadienne
Canada's university

University of Ottawa
Faculty of Engineering

School of Electrical
Engineering
and Computer Science

Object detection with the DBScan algorithm
Détection d'objets avec l'algorithme DBScan
Exercice de programmation
Partie 2
(10%)
CSI2510 Algorithmes et Structure de Données

Automne 2022

Ce projet est à réaliser de façon individuelle

Partie 2 due le 5 Décembre avant 23:59

Pénalité de retard : -30% pour un retard de moins de 24 heures

Description du problème

In Part 1 of this assignment, you created a program that detects objects (clusters of points in fact) in a scene captured by a LiDAR (a laser scanner). This has been done using the DBSCAN algorithm. In the second part of this programming assignment, we will try to improve the efficiency of this algorithm.

But what is the runtime complexity of DBSCAN? Basically, you have to go over all N points of the set and for each point you must search its neighborhood; the current neighborhood search is O(N) since we have to go over all points to find the ones at a distance less than eps. Therefore the global complexity of DBSCAN is $O(N^2)$; note that this is true only if the average number of neighbors per point is small compared to N.

Dans la première partie de cet exercice de programmation, vous avez créé un programme permettant de détecter des objets (des groupements de points en fait) dans une scène captée par un LiDAR (un scanner laser). Ceci a été réalisé en utilisant l'algorithme DBSCAN. Dans la seconde partie de cet exercice, nous voulons améliorer l'efficacité de cet algorithme.

Mais quelle est la complexité en temps d'exécution de DBSCAN? Lors de son exécution, vous devez simplement parcourir tous les N points de l'ensemble et pour chacun de ces points vous devez identifier les points de son voisinage; l'algorithme de recherche de voisinage que nous vous avons proposé a une complexité de O(N) puisque vous devez itérer sur tous les points afin de trouver ceux se situant à une distance inférieure à eps. Conséquemment, la complexité globale de l'algorithme DBSCAN est de $O(N^2)$; ceci est toutefois vrai seulement si le nombre moyen de voisins d'un point est bien inférieur à N.

Votre tâche

To improve the efficiency of the DBSCAN algorithm, we will create a new NearestNeighbors class with a new rangeQuery method.

Our objective is to use an abstract data type called k-d tree in order to obtain a better runtime complexity for the neighborhood search. The k-d tree partitions a point set using a binary tree representation. There will be two programming steps:

1. *Building the k-d tree by inserting all points in this binary tree. This will be done in the constructor of the new NearestNeighborsKD class.*
2. *Finding the neighbors of a given point by searching in the binary k-d tree that should have a depth of O(logN). This will be done in the rangeQuery method.*

Afin de rendre notre algorithme DBSCAN plus efficace, nous allons maintenant créer une nouvelle version de classe `NearestNeighbors` qui aura une nouvelle méthode `rangeQuery`.

Notre objectif est d'utiliser un type abstrait de données appelé k-d tree afin d'obtenir une meilleure complexité pour la recherche des voisins d'un point. Le k-d tree partionne un ensemble de points en utilisant une représentation en arbre binaire. Il y aura deux étapes à programmer :

1. Construire un k-d tree en y insérant tous les points. Ceci se fera dans le constructeur de la nouvelleclasse `NearestNeighborsKD`.
2. Trouver les voisins d'un point en recherchant dans l'arbre binaire k-d qui devrait avoir une profondeur de O(logN). Ceci se fera dans la méthode `rangeQuery`.

1. Construire un k-d tree

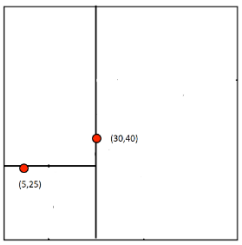
Votre k-d tree sera construit à partir de nuages de points contenus dans les fichiers qui vous ont été donnés. Vous devez lire les points du fichier csv et les stocker dans une `List<Point3D>` comme pour la partie 1. Vous devrez par la suite construire l'instance de `NearestNeighborsKD` en donnant cette liste de points à son constructeur. C'est dans ce constructeur que l'arbre binaire k-d est construit.

Les k-d trees ont été inventés par Jon Bentley au cours des années 70s. Ils sont utilisés afin de stocker des données spatiales sur lesquelles des opérations de recherche doivent être effectuées. Un k-d tree est un arbre binaire de recherche qui subdivise l'espace de points en effectuant des comparaisons selon une seule coordonnée (dimension), en alternance, à chaque niveau. Par exemple, considérant le cas simple à 2 dimensions (x,y), le premier niveau de l'arbre utilisera la coordonnée x pour les comparaisons, puis la coordonnée y pour le second niveau, et la coordonnée x au troisième niveau, et y pour le quatrième niveau etc.

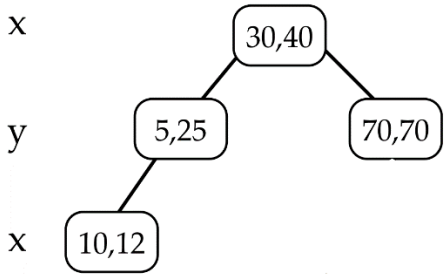
Comme pour les arbres de recherche réguliers, l'insertion du premier élément va simplement créer la racine de l'arbre k-d tree. Par exemple, supposons que le point (30,40) est inséré dans l'arbre initialement vide. Le prochain point qui sera inséré sera comparé à ce point en utilisant seulement sa coordonnée x. Tous les points dont la coordonnée x est plus petite ou égale à ce point se retrouveront du côté gauche de l'arbre alors que tous ceux avec une coordonnée x plus grande seront placés dans le sous-arbre de droite. Donc si nous insérons le point (5,25), celui-ci deviendra le sous-arbre de gauche de notre arbre k-d.

Quand des points à être insérés atteindront ce nouveau nœud, ils seront alors comparés à la coordonnée y de celui-ci. Donc, si le nouveau point a une coordonnée x inférieure à 30 et une coordonnée y inférieure à 25, il se retrouvera tout à gauche de notre arbre.

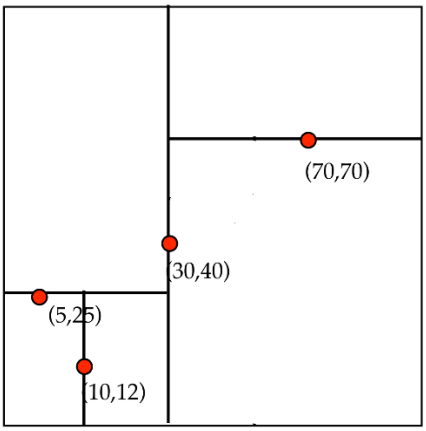
Si nous regardons maintenant l'espace de points, nous pouvons observer que notre arbre k-d a, dans les faits, séparés l'espace en deux partitions : l'ensemble des points ayant un x inférieur ou égale à 30 et ceux ayant un x supérieur à 30. En insérant le deuxième point, nous avons alors créé une nouvelle séparation de la partition de gauche. Ceux-ci sont divisés en une sous-partition contenant les points ayant un y inférieur ou égale à 25 et ceux ayant un y supérieur à 25. Cette division de l'espace est illustrée ici :



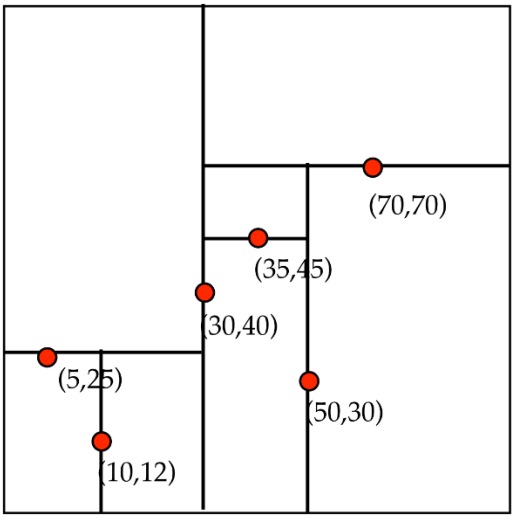
Chaque fois qu'un nouveau point est inséré, la partition contenant ce point est à nouveau subdivisée le long de la dimension correspondant à son niveau dans l'arbre. Voyons maintenant ce qui se produit si nous insérons les points (10,12) et (70,70). L'arbre sera comme suit :

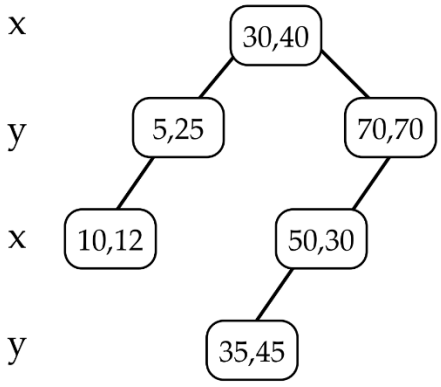


Ce qui divise l'espace de la façon suivante :



Ce processus se poursuit jusqu'à ce que tous les points aient été insérés. Complétons notre exemple en insérant deux autres points.





Rappelez vous que dans le cas de points 3D, les niveaux devront alterner entre les coordonnées x, y et z.

1. *Building the k-d tree*

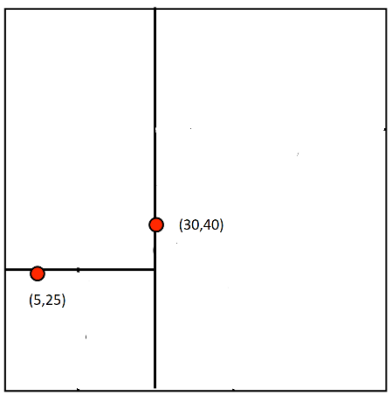
The k-d tree will be built from the point cloud contained in the files given to you. Since these are 3D point, your k-d tree will be a 3-d tree. You will read the point cloud csv file and store the point in a `List<Point3D>` *as you did before. You will then construct the* `NearestNeighborsKD` *instance by passing the list of points to the constructor inside which the binary tree will be built.*

K-d trees have been invented by Jon Bentley in the 1970s. They are used to store spatial data on which quick search operations can be done. A k-d tree is a kind of binary search tree that splits a point set through comparisons done against one dimension at each level. For simplicity, we show an example that uses 2-dimensional points (x,y). In this case, the first level of the tree will use the x-dimension, the second level uses the y-dimension, the third level the x-dimension, the fourth the y-dimension etc.

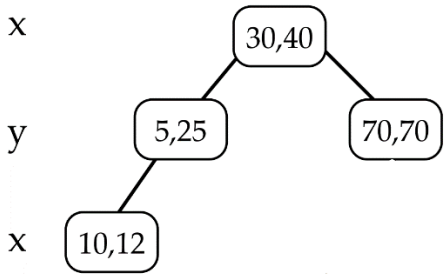
Like for the case of binary search tree, the first element that is inserted becomes the root of the tree. For example, let's say that the point (30,40) is inserted in our 2-d tree. The next points to be inserted will be compared to this point at the root but using only the x-dimension. All points with an x-coordinate smaller or equal to 30 will go to the left of the tree, the ones greater than 30 will go to the right. Therefore, if we insert point (5,25), this one will become the left child of the tree.

When points to be inserted will reach this second node, they will be compared using, this time, the y-coordinate, if this one is smaller or equal to 25, then the point will go to the left, if not it will go to the right.

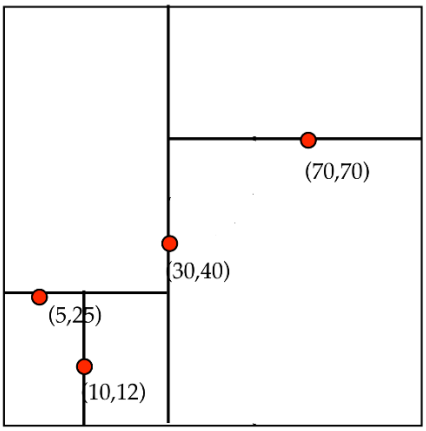
Now, if we look at the space of points, this means that with the first node we have split the space of points into 2 partitions: the points with an x-coordinate smaller than 30, and the ones with an x-coordinate greater than 30. Because of the second point inserted, the left partition has been further split into 2 sub-partitions, along the vertical axis this time, with 25 as the splitting value. This is illustrated by the following 2D space of points.



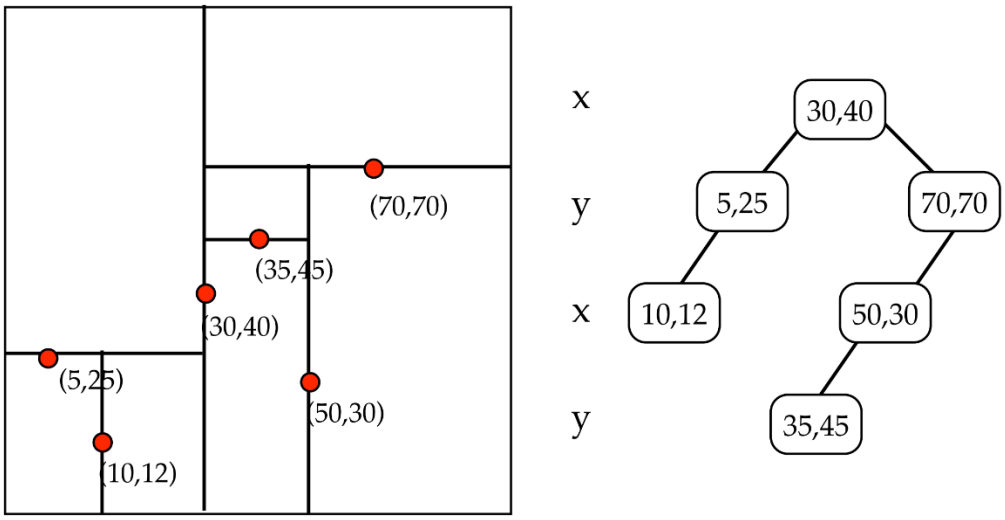
Each time we insert a new point, the space of point, one of the partitions is split along the axis that corresponds to the tree level where the point has been inserted (always alternating between x and y). Let's see what happen if we insert points (10,12) and (70,70). The 2-d tree will be as follows:



which partitions the 2D space as follows:



And the process continues until all points have been inserted. Here we add two more points:



Remember that in the case of 3D points, you will alternate between x, y and z-coordinates.

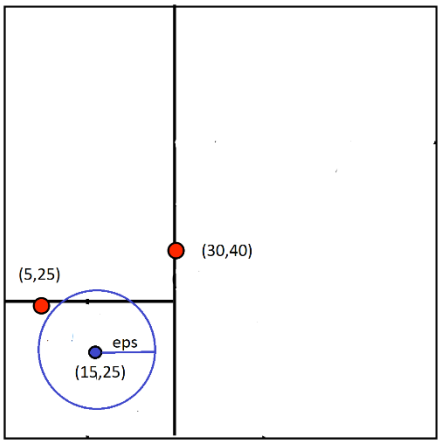
2. Trouver les voisins d'un point

Vous devez créer une nouvelle méthode `rangeQuery` qui recherchera à l'intérieur de votre k-d tree. Cette méthode sera appelée par votre méthode `rangeQuery` initiale.

Encore une fois, nous vous expliquons comment cette recherche s'effectue pour un espace à 2 dimensions. Reprenant l'exemple de la section précédente, nous savons que l'espace de points a été divisé en deux partitions, les points ayant une coordonnée x plus petite ou égale à 30 et ceux ayant une coordonnée x supérieure à 30. Maintenant supposons que nous recherchons les voisins d'un point requête (15,20) avec une valeur eps de 10.

- La première chose à vérifier est de déterminer si le point à la racine est un voisin de (15,20). Puisque la distance entre (15,20) et (30,40) est plus grande que 10, alors ce point ne fera pas partie du voisinage de (15,20).
- Nous continuons ensuite la recherche dans l'arbre k-d. Cela se fait en comparant la coordonnée x du point requête (15,20) au point du nœud courant (30,40). Puisque 15+10=25 est plus petit que la valeur de division 30, nous savons que tous les points du voisinage de (15,20) se trouvent du côté gauche de l'arbre. Nous pouvons donc ignorer tous les points du sous-arbre de droite et effectuer la recherche des voisins, récursivement, à partir du sous-arbre de gauche. Ceci nous mène au nœud (5,25).
- La distance entre le point requête (15,20) et le point (5,25) est plus grande que 10, alors ce point ne fait pas partie du voisinage de (15,20).
- Comme nous l'avons fait pour le nœud parent, nous devons maintenant rechercher dans quelle partition se trouve les voisins du point requête. A ce niveau, la valeur de division correspond à la coordonnée y du point (5,25). Puisque 20-10=10 est plus petit que la valeur 25, nous devrons chercher des voisins dans le sous-arbre de gauche. Mais nous avons aussi que 20+10=30 est plus grand que 25, alors il est aussi possible que des voisins de notre point requête soient présents dans le sous-arbre de droite. Dans ce cas, nous devons donc rechercher dans les deux sous-arbres.

Ce processus est illustré à la figure ci-dessous ou nous observons que tous les voisins du point requête (15,20) doivent se trouver à l'intérieur du cercle montré. Il faudra donc chercher des voisins dans toutes les partitions qui intersectent ce cercle.



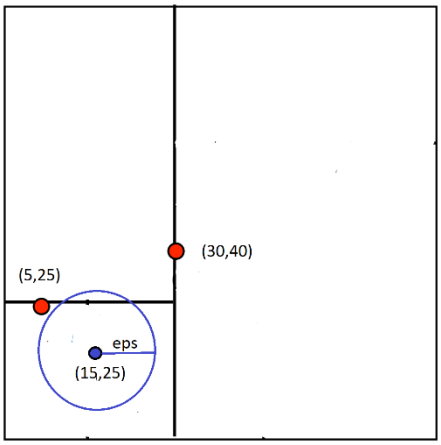
2. *Finding the neighbors of a point*

You will have to create a new `rangeQuery` *method that will be called by your initial* `rangeQuery` *method.*

Again, for simplicity, we now explain how to do the search using a 2-d tree example. Going back to the example we show, we know that, at the root of the tree, the space of points has been divided into two partitions, the ones having an x-coordinate smaller and greater than 30. Now suppose you are looking for the neighbors of a query point (15, 20) with an eps value of 10.

- *The first thing to do is to check if the point stored at this node is a neighbor: the distance between (15,20) and (30,40) is greater than 10 so (30,40) is not a neighbor of query point (15,20) so we do not add it to the list of neighbors.*
- *The next step is to continue the search inside the k-d tree. We do this by comparing the x-coordinates of the query point and the node point. Since 15+10=25 is smaller than the cut value of 30, we know that all neighbors of point (15,20) have to be on the left part of the tree. We can therefore recursively search the neighbors from the left sub-tree which leads us to the (5,25) node.*
- *The distance between the query point (15,20) and the point (5,25) is also greater than 10, so this point is not a neighbor of point (15,20). At this level, the cut value is the y-coordinate of the node point (5,25). Since 20-10=10 is smaller than the cut value 25, we have to search for neighbors in the left sub-tree. But we also have 20+10=30 greater than 25, so neighbors can also be present in the right subtree. In this case, we must therefore search in both sub-trees.*
- *As we did before, we must now check in which sub-partition we should continue to search for the neighbors of (15,20). At this level, the cut value is the y-coordinate of the node point (5,25). Since 20-10=10 is smaller than the cut value 25, we have to search for neighbors in the left sub-tree. But we also have 20+10=30 greater than 25, so neighbors can also be present in the right subtree. In this case, we must therefore search in both sub-trees.*

This process is illustrated below where we see that the neighbors of the query point have to be inside the drawn circle. We must search inside all partitions that intersects with this circle.



Programmation du k-d tree

Vous devez créer la classe interne `KDnode` avec les attributs suivants :

- `point`: le `Point3D` associé à ce noeud
- `axis`: l'axe de division (x, y, or z) representé par les entiers 0, 1 or 2
- `value`: la valeur de division (la valeur de la coordonnée de l'axe de division)
- `left`: une référence au noeud du sous-arbre de gauche (ou `null`)
- `right`: une référence au noeud du sous-arbre de droite (ou `null`)

Afin de faciliter la programmation, vous devriez ajouter à la classe `Point3D` créée à la partie 1, une méthode `get` qui retourne la valeur d'une des coordonnées du point :

-
```
public double get(int axis)   // axis is 0 for x,
                              // 1 for y, 2 for z
```

You must create the `KDnode` inner class with the following attributes:

- *`point`: the `Point3D` associated with this node*
- *`axis`: the splitting axis (x, y, or z) represented by integers 0, 1 or 2*
- *`value`: the splitting value (the coordinate value of the splitting axis)*
- *`left`: a reference to the left node (or `null`)*
- *`right`: a reference to the right node (or `null`)*

To facilitate your programming, you should add to the `Point3D` class, created in Part 1, the following `get` method that returns the value of a specific coordinate

-
```
public double get(int axis)   // axis is 0 for x,
                              // 1 for y, 2 for z
```

La classe `KDtree` contenant l'arbre binaire. L'insertion dans un tel arbre se fait comme pour un arbre binaire de recherche régulier sauf que la valeur utilisée pour la division alterne entre les différentes coordonnées de niveau en niveau.

The `KDtree` class that contains the tree structure. Inserting in this tree is like inserting in a binary search tree except that the coordinate used for splitting alternates from level to level.

```
public class KDtree {

  public class KDnode {

     public Point3D point;
     public int axis;
     public double value;
     public KDnode left;
     public KDnode right;

     public KDnode(Point3D pt, int axis) {
       this.point= pt;
       this.axis= axis;
       this.value= pt.get(axis);
       left= right= null;
     }
  }

  private Node root;

  // construct empty tree
  public KDtree() {

     root= null;
  }
     …
```

La nouvelle classe `NearestNeighborsKD` aura les même méthodes que celle créée à la partie .

The new `NearestNeighborsKD` class will have the same basic methods as the one created in part 1, that is:

- un constructeur qui accepte une liste de `Point3D`
  a constructor that accepts a List of `Point3D`

```
NearestNeighborsKD(List<Point3D>) // that creates a k-d tree
                                  // from the list of points
```

- une méthode `rangeQuery` qui trouve les voisins d'un point 3D
  a `rangeQuery` method that finds the nearest neighbors of a 3D point

```
// this method will call a new method searching a the k-d tree
public List<Point3D> rangeQuery(Point3D p, double eps) {
     List<Point3D> neighbors;
     rangeQuery(p, eps, neighbors, kdtree.root());
     return neighbors;
}
```

- Une instance de `KDtree` sera embarquée dans cette classe. Le constructeur de `NearestNeighborsKD` va créer une instance de cet arbre and y ajoutera tous les points :
  A `KDtree` instance will be embedded into this class. The constructor of the `NearestNeighborsKD` class will create this tree instance and add all points to this tree:

```
NearestNeighborsKD(List<Point3D> list) {

     kdTree = new KDTree();
     List<Point3D> neighbors;
     for (Point3D p : list) {
          kdTree.add(p); // the add method should call the
                         // insert method given in pseudo-code
     }

     // plus possibly other initializations
}
```

- La recherche des voisins dans le k-d tree sera fait par la méthode suivante (appelée par la méthode `rangeQuery` originale). Le pseudo-code de cette méthode récursive est donnée à la section suivante.
  The search for neighbors in the k-d tree is done by the following method (called by the original `rangeQuery` method). The pseudo-code of this recursive method is given in the next section.

```
private void rangeQuery(Point3D p, double eps,
                        List<Point3D> neighbors, KDnode node)
```

Note:

Il existe plusieurs variantes de k-d tree. Celle présentée ici est l'une des plus simples mais elle ne garantie pas un arbre équilibré. Elle a donc un pire cas de complexité O(N).

There are several variants of the k-d tree representations. This one is among the simplest but does not guarantee to produce a balanced tree so it has a worst case search complexity of O(N).

Algorithmes

```
insert(P : Point, node : KDnode, axis : integer) {

     if (node == null)
          node = new KDNode(P, axis)
     else if (P.get(axis) <= node.value)
          node.left = insert(P, node.left, (axis+1) % DIM)
     else
          node.right = insert(P, node.right, (axis+1) % DIM)

     return node
}


rangeQuery(P : Point, eps : double, N : list, node : KDnode) {

     if (node == null)
          return

     if (distance(P, node.pt) < eps)
          N.add(node.pt)

     if (P.get(node.axis) - eps <= node.value)
          rangeQuery(P, eps, N, node.left)

     if (P.get(node.axis) + eps > node.value)
          rangeQuery(P, eps, N, node.right)

     return
}
```

Expérimentations

Afin de procéder à la phase d'expérimentation, vous devz avoir les deux classes suivantes :

- `NearestNeighbors`: qui trouve les voisins d'un point en utilisant une recherche linéaire (voir partie 1).
- `NearestNeighborsKD`: qui trouve les voisins d'un point en utilisant un k-d tree.

To perform the experimental part of this assignment, you must have the following 2 classes:

- *`NearestNeighbors`: that finds the nearest neighbors of a point though a simple linear search (see part 1)*
- *`NearestNeighborsKD`: that finds the nearest neighbors of a point using k-d trees.*

1. Expérience 1: Validation

En utilisant les points contenus dans le fichier `Point_Cloud1.csv`, trouver les voisins des points ci-dessous en utilisant une valeur eps de 0.05.

Using the point list contained in the file `Point_Cloud1.csv`, find the nearest neighbors of the following points using an eps value of `0.05`:

```
1. (-5.429850155, 0.807567048,   -0.398216823)
2. (-12.97637373, 5.09061138,    0.762238889)
3. (-36.10818686, 14.2416184,    4.293473762)
4. (3.107437007,  0.032869335,   0.428397562)
5. (11.58047393,  2.990601868,   1.865463342)
6. (14.15982089,  4.680702457,   -0.133791584)
```

Pour chacun de ces points requête, créer un fichier texte contenant les points voisins. Vous effectuez cette recherche pour les deux approches (linéaire et k-d tree). Nommer ces fichiers en utilisant le numéro attribué au point (1 à 6) et la méthode utilisée (lin vs kd). Par exemple, les fichiers contenant les voisins du premier point devraient être nommés :

`pt1_lin.txt` and `pt1_kd.txt`

Si vos deux clasees fonctionnent correctement, les deux méthodes devraient produire la même liste de voisins mais pas nécessairement dans le même ordre.

For each of these query points, create a text file containing the neighbors. You perform this search using both approaches (linear and k-d tree). Name the file using the point number (1 to 6) and the method used (lin vs kd). For example, the file containing the neighbors of the first query point should be named:

`pt1_lin.txt` and `pt1_kd.txt`

If your two classes work well, both methods should produce the same list of neighbors but not necessarily in the same order.

2. Expérience 2: Temps d'exécution

Pour chacun des trois nuages de points (1 à 3) qui vous ont été fournis, vous devez calculer le temps requis (en ms) afin de trouver les voisins des points aux positions multiples de 10 et avec une valeur de eps égale à 0.5 (i.e. dans le cas du fichier `Point_Cloud1.csv`, vous trouverez les voisins du 10ième , 20ième , 30ième points etc, jusqu'au 29630ième point, le premier point étant (0,0,0)). Important : vous calculez le temps d'exécution de la méthode `rangeQuery` et non le temps requis pour construire l'arbre.

Vous devez donner le temps moyen pour trouver les voisins d'un point pour les deux approches (lin et kd) pour chacun des fichiers. L'arbre k-d devrait en principe donner de meilleurs résultats…

For each of the three point cloud files (1 to 3) provided, you will compute the time required (in ms) to find the neighbors of every 10 points in a file, using the eps value `0.5` (i.e. in the case of `Point_Cloud1.csv`, you will find the neighbors of the 10th point in the file, the 20th , the 30th until the 29630th point, the 1st point being the point (0,0,0)). Important: you compute the time to execute the method `rangeQuery`, not the time required to build the k-d tree.

```
long startTime = System.nanoTime();
nn.rangeQuery(point, eps);
long endTime = System.nanoTime();

long duration = (endTime - startTime)/1000000 // in milliseconds.
```

We want the average time to find these neighbors for each file and for each method (linear and k-d tree). Hopefully, the k-d tree method should be faster…

3. Expérience 3: Intégration à DBScan

Nous voulons maintenant comparer la vitesse d'exécution de DBScan en utilisant la classe `NearestNeighborsKD` au lieu de la recherche linéaire réalisée à la partie 1. Le temps d'exécution devrait être inférieur.

Calculer le temps d'exécution de votre programme DBScan créé à la partie 1, puis remplacer la classe `NearestNeighbors` par la classe `NearestNeighborsKD` et calculer à nouveau ce temps d'exécution. Effectuer cette comparaison pour les trois fichiers de points.

Now we would like to compare the computational speed of DBScan using `NearestNeighborsKD` instead of the linear neighbor search implemented in P1; we expect the former to be faster than the latter.

Compute the runtime of your DBScan program created in part 1, then replace the `NearestNeighbors` class by the `NearestNeighborsKD` class and compute again the total runtime. Perform this comparison for the three point cloud files provided.

Soumission

Pour la seconde partie de ce projet, vous devez soumettre les éléments suivants :
For the second part of this assignment, you have to submit the following items:

- **All your Java classes in a zip file**, in particular
  - the `NearestNeighborsKD` class
  - the `KDTree` class
  - and three classes containing the main method required to perform experiment 1, 2 and 3.

- **A report in PDF format**
- For experiment 1, you must provide:
  - The `Exp1` class containing the `main` method for running experiment 1. This class should be run by specifying, the method used for searching (lin or kd), the parameter value (eps), the point cloud filename and the query point:

```
java Exp1 lin 0.05 Point_Cloud_1.csv -5.42985 0.80756 -0.39821
```

    The program should display number of neighbors found and the list of neighboring points (one per line).
  - In the report, a description of the results you obtained and the tests you made to confirm that both methods give the same results.
  - The 12 files `pt1_lin.txt`, `pt1_kd.txt` to `pt6_lin.txt`, `pt6_kd.txt` in a subdirectory called `exp1`.
- For experiment 2, you must provide:
  - The `Exp2` class containing the `main` method for running experiment 2. This class should be run by specifying, the method used for searching (lin or kd), the parameter value (eps), the point cloud filename and the step parameter for the query points.

```
java Exp2 kd 0.5 Point_Cloud1.csv 10
```

    The program should display the average compute time to find neighbors from the list of points.
  - In the report, a description of the results you obtained. Comment on the compute times you obtained, are they what you expected?
- For experiment 3, you must provide:
  - The `Exp3` class containing the `main` method for running experiment 3. This class should simply run the DBScan as in part 1 but, this time, using the new NearestNeighborsKD class. In addition, the program should display the total runtime (including all steps: file reading, tree construction, queries, etc.).
  - In the report, provide a comparison between the runtime of the DBScan program for the three provided files and for the two methods. Comment on the results you obtained (are they the same for both methods) and on the compute times you obtained, are they what you expected?

- Note that experiment 3 worth only 10%, so if you have not been able to complete part 1 of this assignment, simply explain in the report why you are not able to run this last experiment and your assignment will be marked on 90%.

Grille de correction

k-d tree construction	20%
k-d tree search	20%
NearestNeighborsKD class	10%
Quality of programming (structures, organisation, etc)	10%
Quality of documentation (report, comments and headers)	10%
Validation test	10%
Computational time experiment	10%
Integration to DBScan	10%

All your files must include a header that includes your name and student number. All the files must be submitted in a single zip file.

Tous vos fichiers doivent inclure un entête incluant votre nom et numéro d'étudiant. Tous ces fichiers doivent être soumis dans un fichier zip.