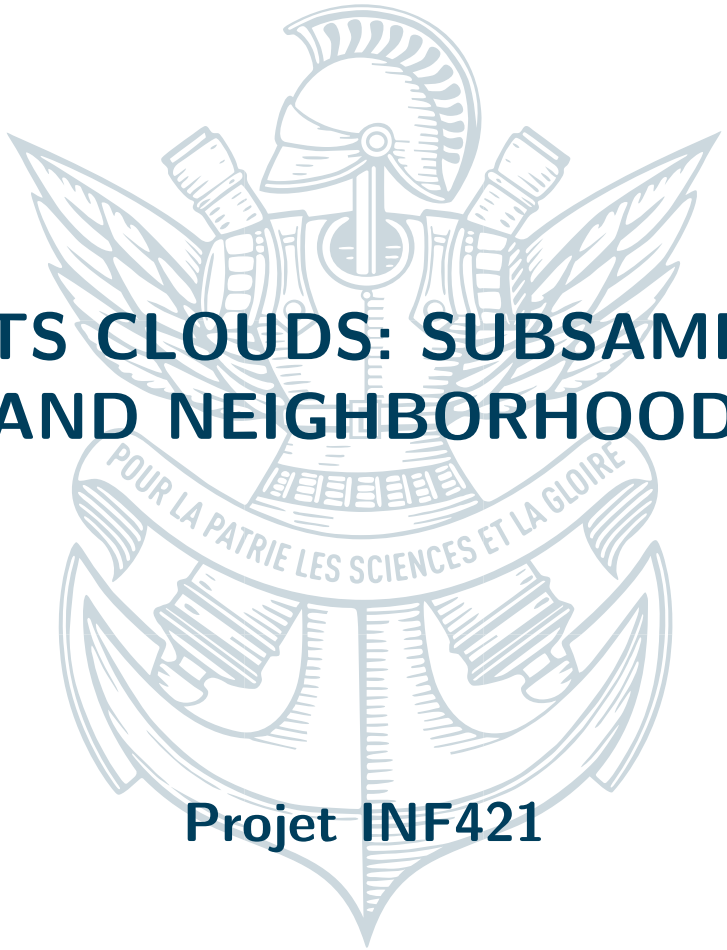


POINTS CLOUDS: SUBSAMPLING AND NEIGHBORHOOD

Projet INF421

11 février 2024

Vincent-Adam Alimi, Adib Mellah

1 SUBSAMPLING

1.1 Motivation et principe du Subsampling

Le subsampling consiste à déduire d'un nuage de points un autre nuage de cardinal plus faible préservant ses propriétés géométriques. Cette technique permet d'exécuter des algorithmes sur des sortes de résumé de jeux de données trop important, comme par exemple des graphes de réseaux routiers contenant plusieurs dizaines de millions de nœuds et d'arrêts.

On cherche dans cette partie à déterminer un algorithme pour *sampler* efficacement un nuage de points, tous suivant la même heuristique de prendre le point le plus éloigné de ceux déjà choisis. Le premier point étant choisi par l'utilisateur.

Pour toute cette partie, les variables d'entrée sont un ensemble `P` de points du plan, `p` un point source, et k le nombre de point à *sampler*. On note n le cardinal de `P`.

1.2 L'algorithme naïf

Description et analyse Commençons par un algorithme naïf. Nous partons d'une liste `S` contenant le point source que l'on retire de `P`, cela se fait en $O(n)$ opérations. Nous choissons alors le point le plus loin de `S` : pour tout point de `P` ($O(n)$), on calcule les distances aux points de `S` et on choisi le minimum ($O(k)$). On se souvenant à chaque itération du maximum actuel, on obtient le nouveau point *samplé*. On itère alors ce processus $k-1$ fois et on obtient un sample de k points en $O(nk^2)$.

1.3 Un raffinement quadratique

Description et analyse On optimise en gardant la trace de la distance des points de `P` à `S` lors de la construction de S. On introduit pour cela un dictionnaire `distP` qui contient en clé les points de `P` et en valeur leur distance à `S`. Ainsi, il suffit de trouver la clé avec la valeur maximum pour obtenir le point le plus loin de S, cela se fait en $O(n)$ opérations.
Après avoir décidé du nouveau point de `S`, il faut mettre à jour `distP`. Pour chaque clé il suffit de prendre pour nouvelle valeur le minimum de la précédente et de la distance au nouveau point, cela se fait donc en $O(n)$ opérations. On itère alors $k-1$ fois ce procédé pour obtenir une liste de k éléments. On a donc finalement une complexité de $O(nk)$.

1.4 Une optimisation par un argument géométrique

Description Tentons d'optimiser la structure de donnée employée et d'utiliser les propriétés géométriques de la distance.
Afin d'obtenir chaque nouveau centre en $O(k)$, nous allons maintenir un dictionnaire `regionS` dont les clés seront les points `c` de `S` et les valeurs associées seront : le rayon $r_{\texttt{c}}$ de la région de `c`, un point réalisant le rayon de `c`, et enfin la liste des points dans la région de `c`, stockée sous forme d'une file FIFO.
Pour trouver un nouveau centre qui sera donc le prochain point samplé, il suffit de trouver le centre dans le dictionnaire dont le rayon est le plus grand et alors de choisir le point réalisant le rayon de ce centre comme nouveau centre, `newC`.
Il faut maintenant mettre à jour `regionS`. Pour cela, il suffit d'observer les centres `c` vérifiant l'inégalité :

$$d(\texttt{newC}, \texttt{c}) \leq 2r_{\texttt{c}}$$

En effet, dans le cas contraire, pour un point `p` dans la région de `c`, on a, par inégalité triangualaire :

$$d(\texttt{newC}, \texttt{p}) \geq d(\texttt{newC}, \texttt{c}) - d(\texttt{c}, \texttt{p}) > 2r_{\texttt{c}} - r_{\texttt{c}} = r_{\texttt{c}}$$

Ainsi, un point de la région de `c` ne pourra pas être volé par `newC`, il n'est donc pas nécessaire d'examiner les points de cette région.
Pour les autres régions suceptibles de se faire voler des points, nous les examinons un par un en comparant leur distance à leur centre avec celle à `newC` pour construire la région de `newC` et mettre à jour celle des autres centres.

Analyse Récupérer le nouveau centre se fait en $O(k)$ opérations puisque l'on itère sur les clés du dictionnaire. Pour la mise à jour de la structure, nous n'avons pas réussi à obtenir une preuve formelle. Cependant, nous avons le raisonnement suivant. La contrainte géométrique expliquée dans la description de l'algorithme nous permet de n'avoir à consulter que les régions proche de `newC`. Avec k suffisament petit par rapport à n, il semblerait qeu cela contraigne fortement le nombre de centre qu'il peut y avoir dans une petite zone. Nous supposons alors que, pour un nouveau centre, il n'y a qu'un nombre borné par une constante C de régions à explorer. Lors du calcul de la région du $i+1$-ème centre, les i régions contiendront en moyenne $\frac{n}{i}$ points (car tout point de `P` est contenu dans une région. Ainsi, en itérant $k-1$ fois le processus, nous obtenons au plus, à une constante multiplicative près,

$$\underbrace{k + \cdot + k}_{k \text{ fois : recherche de } \texttt{newC+}} + \underbrace{C\frac{n}{1} + C\frac{n}{2} + \cdot + C\frac{n}{k-1}}_{\text{mise à jour de } \texttt{regionS+}}$$

Le développement asymptotique de la série harmonique nous donne $H_k \sim \ln k$ d'où $H_k = O(\ln k)$. Ainsi, nous obtenons que cet alogorithme s'execute, pour de petites valeurs de k comparées à n, en temps $O(k^2 + n \ln k)$ en moyenne.

1.5 Une piste pour un algorithme efficace

Commentaire Quand k devient trop grand, itérer sur S pour obtenir le nouveau centre peut s'avérer trop long. De même que pour chercher les régions à observer pour calculer celle de `newC`. On met en place pour cela une file de priorité [1] sous forme de dictionnaire ayant pour clé les centres `c` de S déjà calculés et en priorité `-r_c` (la priorité la plus faible sort en premier). Cette file de priorité utilise des tas binaires et renvoie la top-priorité en temps constant et nécessite un temps en $O(\text{len(file)})$ pour modifier la priorité d'un élément et pour insérer un nouvel élément ainsi que sa priorité.

On introduit également le graphe d'amitié `Gf` entre les centres comme introduit dans l'énoncé. On remarque que si `s` est le centre d'origine de `newC`, alors une condition nécessaire pour que la région de `newC` contienne des points de `s1` est

$$d(\texttt{s}, \texttt{s1}) \leq 2r_{\texttt{s1}} + r_{\texttt{s}}$$

En effet, par inégalité triangulaire et en réutilisant la condition nécessaire prouvée en section 2.4, on a, si `newC` vole des points de `s1` :

$$d(\texttt{s}, \texttt{s1}) \leq d(\texttt{s1}, \texttt{newC}) + d(\texttt{newC}, \texttt{s}) \leq 2r_{\texttt{s1}} + r_{\texttt{s}}$$

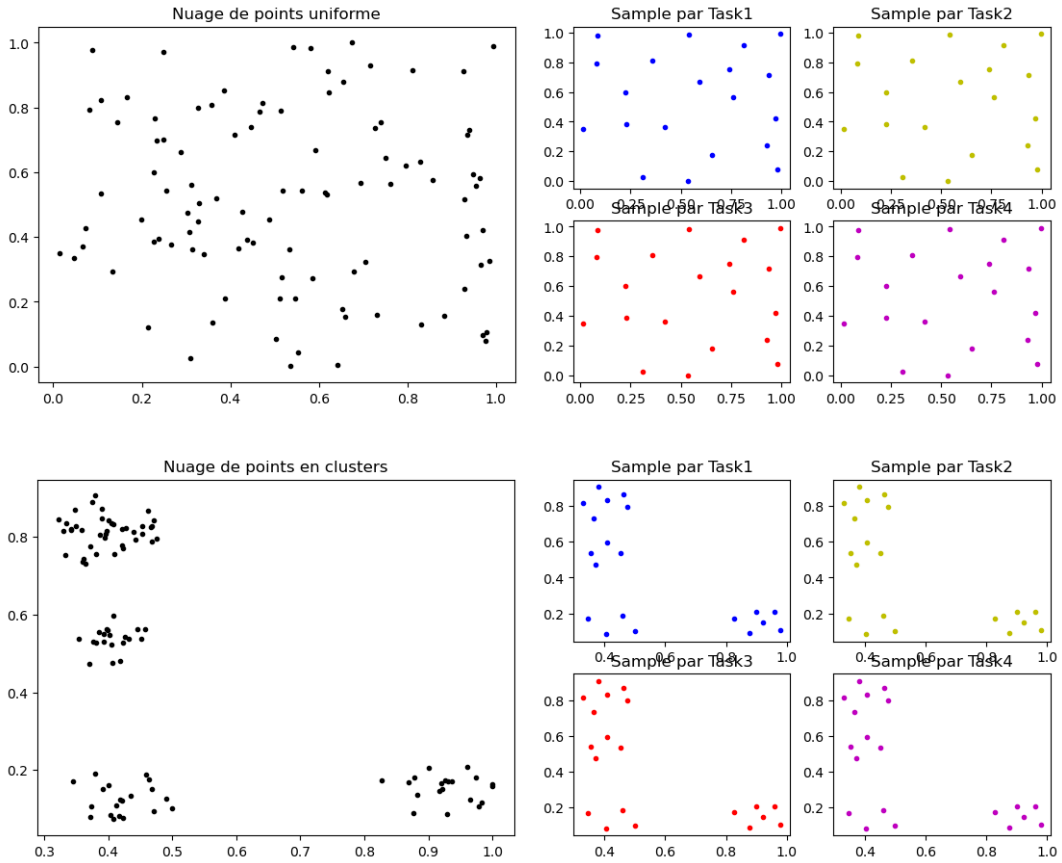
Ce graphe (non orienté *a priori*), implémenté sous forme d'un dictionnaire ayant en clé les centres de `S` et en valeur la liste des voisins, donne donc les régions à consulter lors de la mise à jour de `regionS`.
Il reste donc à maintenir ces deux nouvelles structures. Pour la file de piorité, on ajoute `newC` et son rayon et on modifie le rayon des autres centres dans la file, après la mise à jour de `regionS`. Cependant, nous n'avons pas réussi à trouver un moyen efficace de maintenir le graphe. Il semblerait que la condition précédente pour être voisin dans le graphe ne permette pas de contraindre suffisament les voisins potentiels du nouveau centre. Cela nous oblige à parcourir tout les centres de `Gf` pour savoir lesquels ont pour voisin `newC` et lesquels sont voisins de `c`.

Nous pensons qu'une condition mieux choisie pour que deux centres soient voisins dans le graphe pourrait imposer que les voisins de `newC` ainsi que les centres dont `newC` est voisin soient tous contenu dans les voisins du centre d'origine de `newC` (voire les voisins de ces voisins). En supposant alors que la géométrie du nuage de point impose que les sommets du graphe soit de degré borné par une constante, nous obtiendrons une mise à jour du graphe en temps constant. Par ailleurs, le nombre de région à étudier pour déterminer la région de `newC` serait alors aussi borné par une constante. Par un raisonnement similaire à celui de la section 2.4., nous obtiendrons alors un algorithme en $O(n \ln k)$.

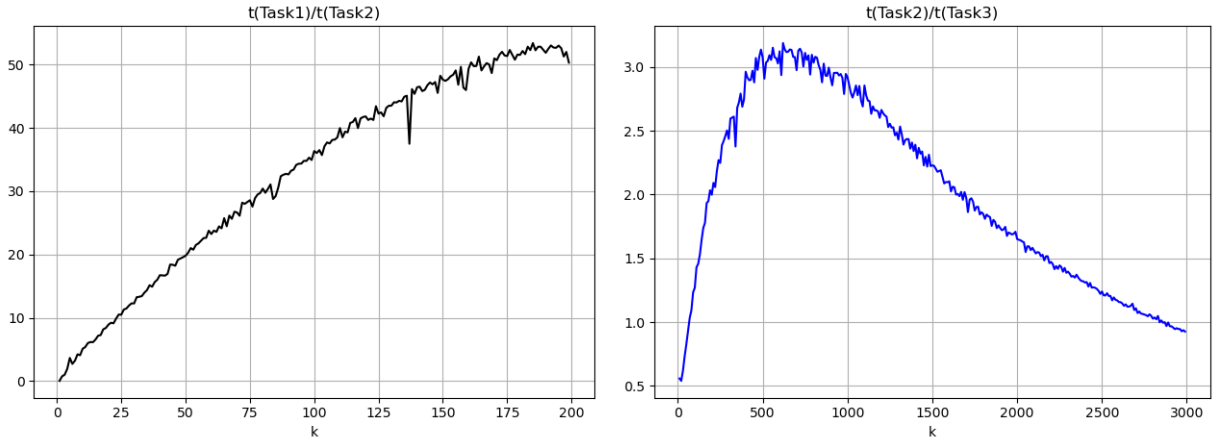
1. Cette structure de donnée pqdict, codée Nezar Abdennur est en opensource : https ://github.com/nvictus/priority-queue-dictionary

1.6 Comparaison des algorithmes

Correction des algorithmes Pour expériemeter les algorotithmes, nous avons utilisés deux types de datasets, un dataset uniforme, et un en cluster. Le premier génère `n` points uniformément dans le carré $[0,1] \times [0,1]$. Le second prend en entré deux paramètres : `c` et `k`. Alors, `c` points sont générés uniformément dans le carré unité, et sont choisis `c` rayons associés, compris entre $\frac{1}{3c}$ et $\frac{1}{2c}$. Ensuite, on choisis uniformément dans les `c` disques ainsi définis `k` points. En exécutant les algorithmes sur de telles instances, on obtient les nuages identiques suivant.



Comparaison de performances Afin de comparer les performances de nos algorithmes, nous les avons exécuter sur plusieurs nuages de points uniformes de taille `n` fixé et pour des `k` variables. Puis nous avons calculé le quotient des temps d'exécution des deux algorithmes comparés pour chaque `k` pour obtenir les courbes suivantes.



Comme attendu, l'algorithme `Task2` est significativement plus rapide que `Task1`. Nous observons également que `Task3` est quasiment tout le temps plus rapide que `Task2` mais l'est notamment plus lorsque `k` est dans la plage entre `n/4` et `n/2`.

2 NEIGHBORHOOD

2.1 Définition, utilités et cadre d'étude

Motivations Lorsque l'on dispose d'une base de données, il est crucial de savoir comment relier les différentes données afin d'en tirer le maximum d'informations. Il est alors nécessaire de pouvoir trouver ces liens de manière efficace parmi le nombre fréquemment immense de données.

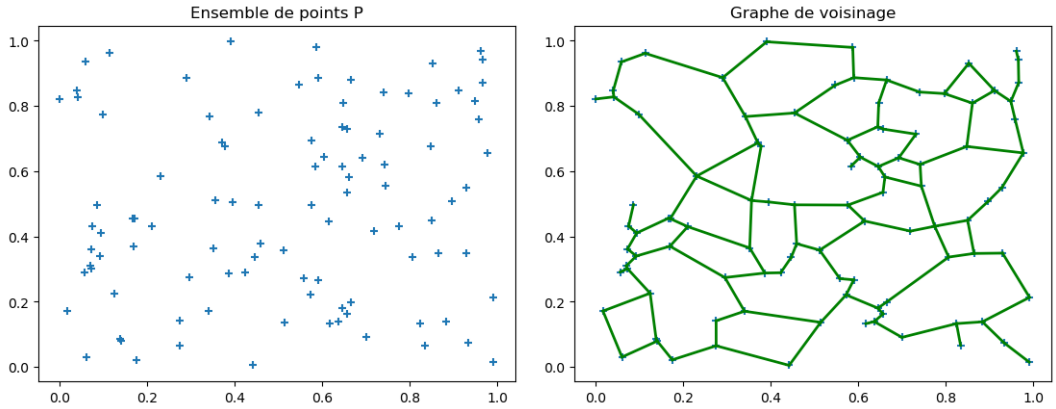
Objectif En vue de répondre aux problématiques évoquées, nous souhaitons donc pouvoir établir le "voisinage" d'un ensemble P de données, c'est-à-dire construire le graphe de P constitué des arêtes reliant tous les points voisins d'après la distance $d(.,.)$. Nous noterons alors G un tel graphe.

2.2 Premier algorithme - algorithme naïf

Description Prenons un ensemble de points P. Par boucles imbriquées, pour chaque couple $(p_i, p_j) \in P^d$, pour chaque point $p_k \in P$, avec p_i, p_j, p_k distincts, nous vérifions si $max(d(p_i, p_k), d(p_j, p_k)) \geq d(p_i, p_j)$, auquel cas nous ajoutons (p_i, p_j) au graphe G. G représente alors le *voisinage* de P.

Analyse Pour le premier niveau de boucle, nous avons $n(n-1)$ couples de points distincts (rappel : $n = |P|$). Pour chacun de ces couples, il faut vérifier la relation de voisinage, ce qui se fait en temps constant. La relation doit donc être vérifiée $n-2$ fois, d'où une complexité de $O(n^3)$ pour la création du graphe G. La solution trouvée est nécessairement correcte, puisque toutes les possibilités d'arêtes ont été étudiées (sans compter les cas triviaux de points identiques).

Voici ce que nous obtenons en dimension 2 pour un ensemble de 100 points :



2.3 Second algorithme

Description Nous allons mettre en place un algorithme $find\_neighbors(p)$ qui permet de trouver tous les voisins d'un point $p \in P$. Pour obtenir cela, nous utilisons la propriété suivante : **Soit $p \in P$. Si $q \in P$ est le point le plus proche de p, alors q empêche $r \in P$ d'être voisin de p si et seulement si r est plus près de q que de p.** (P1)
Démontrons-le : reprenons ces mêmes p, q, r et supposons que $d(r,q) < d(r,p)$. Par définition de q, $d(p,q) < d(p,r)$. Alors, $max(d(p,q), d(q,r)) < d(p,r)$. C'est exactement dire que q empêche p, r d'être voisins. La réciproque est triviale : si q empêche p, r d'être voisins, $max(d(p,q), d(q,r)) < d(p,r)$ d'où $d(q,r) < d(p,r)$.

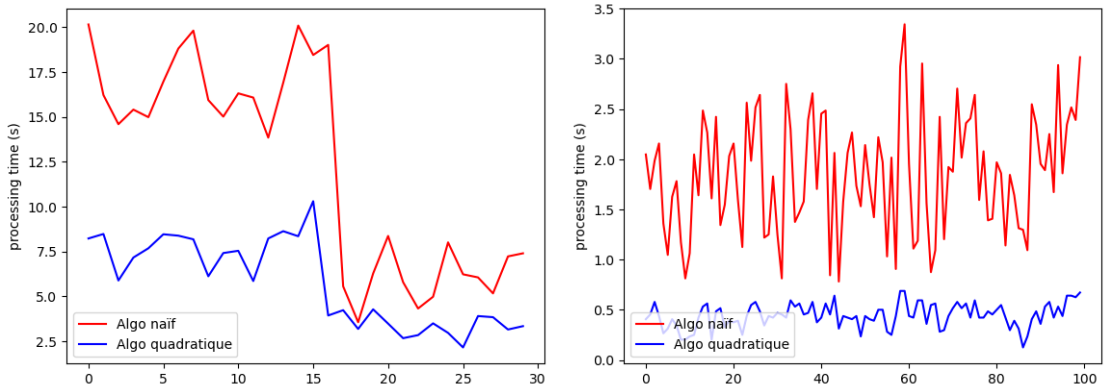
Nous allons maintenant pouvoir construire notre algorithme pour trouver les voisins de $p \in P$. Commençons avec une copie C de l'ensemble P et construisons un ensemble P_N de possibles voisins (*Potential neighbors*). Tant que C n'est pas vide :

- trouvons le point q le plus proche de p
- retirons les points r plus proches de q que de p de C (ils ne peuvent pas être dans P_N d'après (P1))

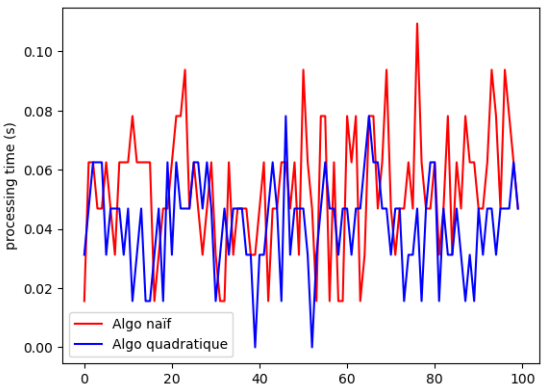
Alors, nous avons une liste réduite de voisins potentiels, i.e $P\_N = C$. Pour obtenir les voisins effectifs de p, il suffit de vérifier la relation de voisinage pour chaque voisin potentiel de p. Cela constitue donc l'algorithme $find\_neighbors(p)$, qu'il suffit d'utiliser pour chaque $p \in P$.

Analyse L'algorithme est correct. En effet, toutes les arêtes non considérées (i.e la relation de voisinage n'a pas été étudiée) ne peuvent pas être dans G d'après (P1), et on ne peut pas se tromper sur celles considérées.
Trouvons donc la complexité de cet algorithme. Nous savons qu'il faudra appeler $find\_neighbors(p)$ pour chaque élément de P, i.e n fois. Chacun de ces appels se décompose en établir la liste P_N et vérifier qui sont les voisins dans celle-ci. La première tâche est en $O(n)$, puisque trouver le point le plus proche est en $O(n)$ (revient à trouver le min d'une liste) et le retrait des points est aussi en $O(n)$ (il faut et suffit de passer par chaque élément restant de la liste). La seconde tâche dépend du cardinal de P_N, mais nous admettons que ce nombre est en O(1). Alors, il faut vérifier que les éléments de P_N sont bien des voisins, ce qui se fait en $O(n)$ pour chaque candidat, donc en $O(n)$ pour toute la liste.
Finalement, nous avons prouvé que cet algorithme trouve le graphe G de voisinage en $O(n^2)$, contre $O(n^3)$ auparavant, ce qui est une amélioration très importante (les ensembles en jeu sont souvent des bases de données énormes).

Analysons donc les performances des deux algorithmes en pratique. Le graphique de gauche repose sur des points générés uniformément dans l'espace, tandis que celui de droite utilise une génération plus ordonnée avec des clusters. Les performances ont été mesurées pour des ensembles de 1000 points à gauche et 300 à droite.



Si nous avons n faible, la différence s'estompe :



2.4 Mise à jour d'un graphe

Nous voulons un algorithme permettant de mettre à jour le graphe G de P lorsque l'on y ajoute un élément.

Description La mise à jour du graphe G peut se décomposer en deux morceaux : vérifier si le nouvel élément p détruit certains liens de voisinages et trouver les voisins de p.
Il suffit donc de :

- vérifier la relation de voisinage entre les nœuds de chaque arête avec p comme bloqueur potentiel
- trouver les voisins de p à l'aide de l'algorithme $find\_neighbors$ décrit dans la section précédente

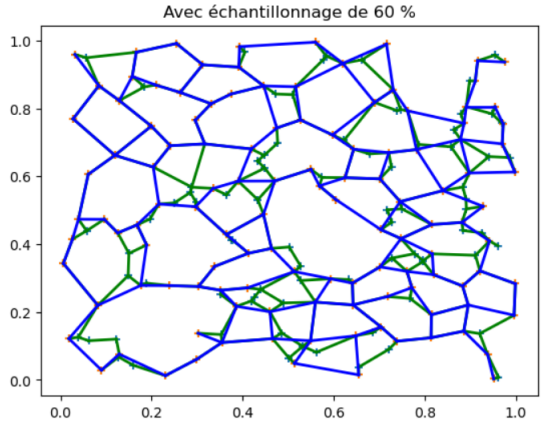
Nous obtenons alors le nouveau graphe de P avec l'ajout d'un nouvel élément p.

Analyse Pour chaque arête de G, il est évident que les nœuds ne sont plus voisins après l'ajout de p si et seulement si c'est p qui empêche les nœuds d'être voisinsLa validité de $find\_neighbors$ a déjà été établie, donc l'algorithme est correct.
Nous avons admis que le degré de chaque nœud de G est en $O(1)$, donc le cardinal de G est en $O(n)$. Puisque la relation de voisinage est vérifiée en temps constant, la première partie de l'algorithme est en $O(n)$. Or, la complexité de $find\_neighbors$ est aussi linéaire en n, donc la mise à jour du graphe se fait bien en complexité linéaire.

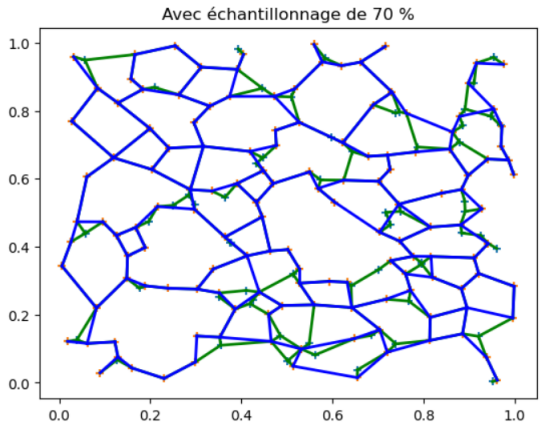
2.5 *Subsampling* et *neighborhood*

Nous avons maintenant établi des algorithmes efficaces pour le *subsampling* et la création des graphes de voisinage. Or, pour des ensembles P de taille conséquente, la création du graphe est longue : pour $|P| = 500$, cela prend déjà plus de deux secondes. Comme les algorithmes de sous-échantillonnage ont pour objectif le gain de temps, nous allons utiliser cet avantage.
Prenons un ensemble P et partons d'un point p. Le graphe est donc originellement vide. Alors, nous pouvons effectuer le subsampling point par point (cf description des algortihmes dans la partie I). A chaque nouveau point, nous pouvons mettre à jour le graphe (cf section précédente) avec ce nouveau point. En s'arrêtant à la précision désirée, nous aurons ainsi un gain de temps conséquent pour un graphe de voisinage proche de celui de P entier.

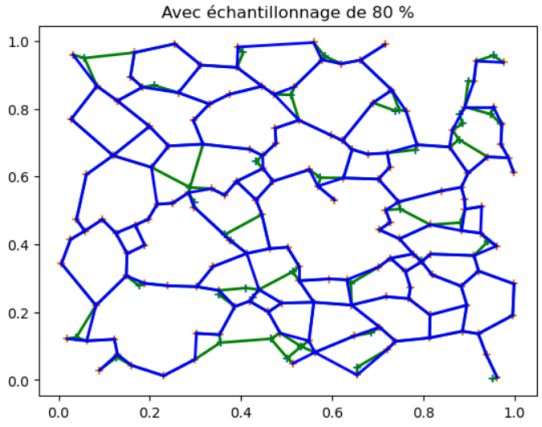
Voici quelques graphiques ainsi que le gain de temps gagné pour illustrer la pertinence du sous-échantillonage en amont de la création du graphe de voisinage. Les graphes complets sont représentés en vert tandis que ceux issu du sous-échantillonnage sont en bleu.




```
Durée sans échantillonnage:  0.5937500000000003
Durée avec échantillonnage de 60 % :  0.29687500000000017
Temps gagné :  0.29687500000000017
```




```
Durée sans échantillonnage:  0.4687500000000002
Durée avec échantillonnage de 70 % :  0.37500000000000017
Temps gagné :  0.09375000000000006
```




```
Durée sans échantillonnage:  0.6875000000000003
Durée avec échantillonnage de 80 % :  0.5156250000000002
Temps gagné :  0.1718750000000001
```