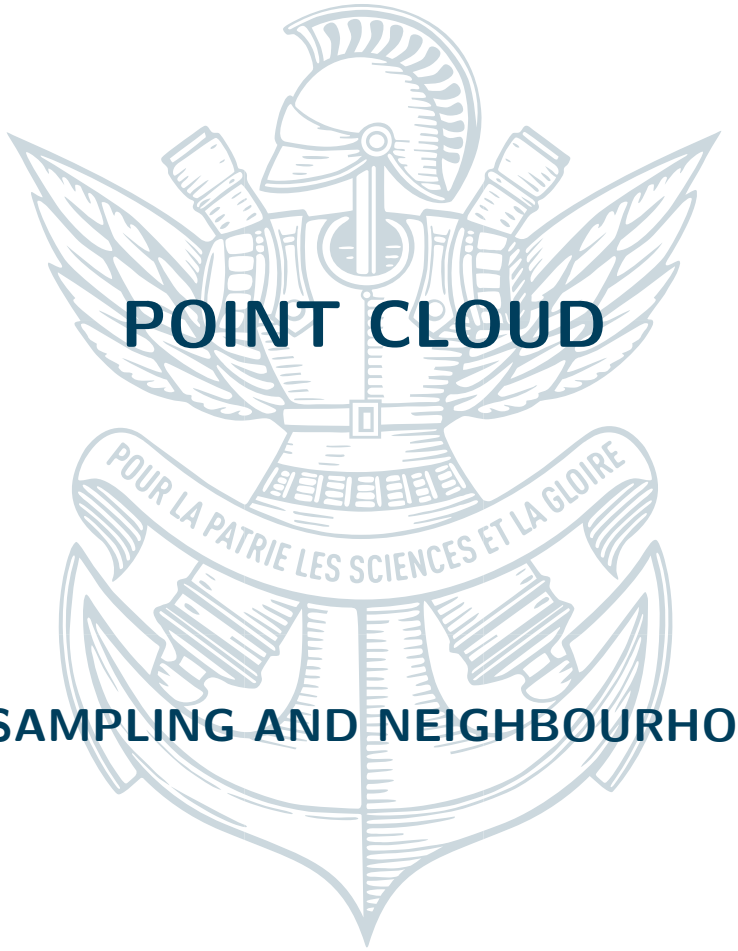

POINT CLOUD

SUBSAMPLING AND NEIGHBOURHOODS

11 février 2024

Loic Tabueu, Ulrich Totuo

ÉCOLE POLYTECHNIQUE

INSTITUT POLYTECHNIQUE DE PARIS

Inria

TABLE DES MATIÈRES

1 INTRODUCTION

Le traitement des nuages de points est essentiel dans divers secteurs, allant de la modélisation 3D à l'analyse de données géospatiales. Ce projet se penche sur des algorithmes destinés à améliorer notre compréhension et notre manipulation de ces collections de points, en mettant l'accent sur deux éléments clés : l'échantillonnage et la détermination des voisins.

L'objectif principal est de concevoir des algorithmes performants pour la réduction d'échantillonnage (subsampling) et la création du graphe des voisins (neighborhood graph) à partir de nuages de points bidimensionnels. Ces deux tâches sont vitales pour simplifier la manipulation des données, améliorer la visualisation et faciliter le traitement ultérieur.

Dans ce rapport, nous exposerons les choix fondamentaux concernant le langage de programmation, les structures de données, et les restrictions imposées à l'utilisation de bibliothèques existantes. Par la suite, nous décrirons en détail les algorithmes mis en œuvre pour chaque tâche, en mettant l'accent sur les approches naïves et les optimisations progressives. Des expériences seront réalisées sur différents ensembles de données pour évaluer la performance des algorithmes et assurer leur validité.

Ce projet offre une occasion précieuse d'explorer des problèmes réels liés à la manipulation de données géométriques, tout en favorisant une approche algorithmique créative et efficace. Les résultats obtenus contribueront à éclairer les implications pratiques de ces algorithmes dans des contextes réels, tout en offrant des perspectives pour des développements futurs.

Enfin, ce rapport traitera chaque aspect du projet en détail, en décrivant les choix de conception, les méthodes d'implémentation, les expériences réalisées, et les conclusions tirées de ces expériences.

2
POINTS GÉNÉRAUX

Dans cette section, nous explorons les choix fondamentaux liés au langage de programmation et aux structures de données, ainsi que les directives sur la lisibilité du code et les restrictions liées à l'utilisation de bibliothèques existantes, en mettant particulièrement en lumière notre utilisation du langage Python.

2.1 Choix du langage de programmation et des structures de données

Le choix du langage de programmation constitue une étape cruciale dans le développement de notre projet. Nous avons opté pour Python en raison de sa polyvalence, de sa lisibilité, et de sa vaste communauté de développeurs. La simplicité syntaxique de Python a facilité la mise en œuvre des algorithmes, tout en permettant une évolution rapide du code. Nous avons travaillé sur Jupyter Notebook afin d'obtenir une structure de code cohérente et bien organisée. L'aspect syntaxique très maniable de Python a également été un gage de succès pour ce langage dans notre processus de choix, nous permettant ainsi d'implémenter facilement les structures de données que nous considérions comme les plus adaptées à la solution recherchée.

Parlant des structures de données, elles jouent un rôle central dans la mise en œuvre des algorithmes. Dans notre projet, nous avons utilisé des structures de données telles que les tableaux, les dictionnaires, les listes imbriquées, etc., pour optimiser l'efficacité des algorithmes. Ces choix ont été faits après une analyse approfondie des exigences spécifiques du projet. Nous avons notamment remarqué avec ce projet que l'on peut réduire la complexité d'un algorithme en changeant la structure de données utilisée, ce qui fut particulièrement intéressant, car nous en avons testé plusieurs pour avoir un réel aperçu de leur impact.

2.2 Consignes sur la lisibilité du code

La lisibilité du code revêt une importance particulière pour favoriser la compréhension et la collaboration au sein de l'équipe. Nous avons suivi des directives strictes pour assurer une lisibilité maximale, en adoptant une approche de codage propre, linéaire, avec des commentaires explicatifs et des conventions de nommage cohérentes. Ayant utilisé un notebook Jupyter, l'exécution de notre code se fait de manière linéaire. Comme conseillé par le sujet, nous avons commencé par la partie 3, ce qui s'est avéré très utile, car il a fallu effectuer des tests au fur et à mesure que nous écrivions le code, sans quoi nos algorithmes n'auraient pas été corrects.

2.3 Restrictions sur l'utilisation de bibliothèques existantes

Bien que Python offre une pléthore de bibliothèques, nous avons décidé de restreindre leur utilisation dans un souci pédagogique. Cette décision a encouragé une compréhension approfondie des mécanismes sous-jacents des algorithmes, même si elle a parfois impliqué un effort supplémentaire. Malgré tout, nous avons utilisé quelques bibliothèques Python. Cependant, nous avons conditionné l'usage d'une bibliothèque dans notre projet en maîtrisant précisément le coût d'utilisation de cette dernière en termes de complexité, ce qui est un point crucial de notre projet. Un exemple notable est l'utilisation de "np.linalg.norm()" pour le calcul de la distance entre deux points.

3
SUBSAMPLING (ÉCHANTILLONNAGE)

3.1 Basique - complexité O(n^3)

Dans cette section introductive dédiée à l'échantillonnage, nous nous penchons sur l'implémentation d'un algorithme naïf caractérisé par une complexité algorithmique de $O(n^3)$. Cette approche permet d'appréhender les bases de l'échantillonnage, mais elle révèle également ses limitations en termes de performance.

Objectifs

L'objectif principal de cette section est de fournir une mise en œuvre claire et compréhensible de l'algorithme de subsampling de base. En mettant l'accent sur la simplicité, nous posons les fondations nécessaires à la compréhension des versions plus avancées à venir.

Pseudo-code

Le pseudo code "Algorithme 1 : Fonction Task1 pour effectuer une tâche de regroupement" est fourni en annexe.

Contenu

1. Description de l'Algorithme

   L'algorithme ‘task1‘ effectue une tâche de regroupement en formant k clusters à partir d'un ensemble de points P, en utilisant une approche itérative. Voici une description détaillée de l'algorithme :

   Entrées : - ‘P‘ : Ensemble de points que l'on souhaite regrouper. - ‘p1‘ : Point de départ pour initialiser l'algorithme. - ‘k‘ : Nombre de clusters à former.

   Sorties : - ‘centers‘ : Liste des centres de cluster formés. - ‘S‘ : Ensemble de points dans chaque cluster.

   1. On initialise l'ensemble ‘S‘ avec le point de départ ‘p1‘ et la liste des ‘centers‘ comme étant une liste vide.

   2. On répète l'étape suivante ‘k‘ fois pour former ‘k‘ clusters.

   3. À chaque itération, on recherche dans l'ensemble de points ‘P‘ le point ‘Pmax‘ qui est le plus éloigné de l'ensemble ‘S‘. On utilise la distance euclidienne pour mesurer la distance entre les points.

   4. On recherche ensuite dans l'ensemble ‘S‘ le point ‘Smin‘ qui est le plus proche de ‘Pmax‘.

   5. On ajoute ‘Pmax‘ à l'ensemble ‘S‘ et ‘Smin‘ à la liste des ‘centers‘.

   6. On répète ces étapes jusqu'à former ‘k‘ clusters distincts.

   Remarques : - L'algorithme utilise la distance euclidienne pour mesurer la proximité entre les points.

   - Les points ‘Pmax‘ sont choisis de manière itérative pour maximiser la distance par rapport à l'ensemble ‘S‘.

   - Les points ‘Smin‘ sont choisis comme les points de ‘S‘ qui sont les plus proches de chaque ‘Pmax‘.

   - Les ‘centers‘ représentent les points dans ‘S‘ qui sont les plus proches des ‘Pmax‘ respectifs.

   - L'algorithme garantit la formation de ‘k‘ clusters distincts.

   Cet algorithme de regroupement est basé sur des distances spatiales pour former des clusters autour de points spécifiques.

2. Identification des structures de données essentielles utilisées dans l'implémentation.

   Le code implémente un algorithme de clustering (‘task1‘) qui forme k clusters à partir d'un ensemble de points (‘P‘). Les structures de données principales sont deux listes, ‘S‘ pour les points et ‘centers‘ pour les centres des clusters. À chaque itération, le code recherche le point le plus éloigné de l'ensemble actuel (‘S‘) et le point de cet ensemble le plus proche. Ces points sont ajoutés respectivement à ‘S‘ et ‘centers‘. L'algorithme se répète k fois pour former les clusters. Les distances entre les points sont calculées à l'aide de la fonction ‘euclidean-distance‘. La fonction retourne la liste des centres de clusters et l'ensemble final de points.

3. Analyse de Complexité Discussion sur la complexité algorithmique $O(n^3)$. Identification des points critiques qui contribuent à cette complexité.

Rôle dans le Projet

Bien que cet algorithme puisse sembler rudimentaire, il sert de point de départ crucial pour comprendre les défis de l'échantillonnage. Les apprentissages tirés de cette implémentation fourniront une base essentielle pour évaluer et améliorer les versions ultérieures de l'algorithme.

3.2 Plus efficace - complexité O(nk)

Cette section se penche sur une amélioration significative de l'algorithme d'échantillonnage en proposant une implémentation plus efficace, caractérisée par une complexité algorithmique de $O(nk)$. L'objectif est d'optimiser le processus d'échantillonnage pour des jeux de données de taille plus importante.

Pseudo-code

Le pseudo-code de la tache 2 est constituer de deux algorithme a savoir : "Algorithme 2 : Task2 Function to Find Index of Maximum Value in a Column" et "Algorithme 3 : Task 2 Function" et ils sont en annexe.

Objectifs

L'objectif principal de cette section est de présenter une alternative plus efficace à l'algorithme de base, en introduisant des mécanismes qui réduisent la complexité tout en maintenant la qualité de l'échantillonnage.

Contenu

1. Optimisations Algorithmiques Description des modifications apportées à l'algorithme de base pour améliorer son efficacité.

   Désormais, l'état de notre système est décrit par un dictionnaire qui, à l'étape i, contient tous les points de P qui ne sont pas dans S. Chaque entrée du dictionnaire est une ligne représentée sous la forme d'un tableau de trois colonnes : [point de P priver de S, son centre dans S, sa distance à son centre].

   Cette approche est très pertinente, car elle permet de rechercher le point Pi en un temps linéaire et de répéter cette opération k fois, ce qui nous donne une complexité O(nk).

2. Mise en évidence des choix de conception visant à réduire la complexité. Le dictionnaire (‘dictionnaire‘) est une liste temporaire utilisée pour stocker des informations sur les distances entre les points. Voici un résumé de son rôle et de son utilisation dans l'algorithme ‘task2‘ :

- Rôle du dictionnaire : Il sert à stocker temporairement des informations sur les distances entre les points par rapport à un point spécifique.

- Composition : Chaque élément du dictionnaire est une liste contenant trois éléments : 1. Le point en question (‘dictionnaire[i][0]‘). 2. Le point de référence associé (‘dictionnaire[i][1]‘), initialement défini comme le premier point (‘p1‘) dans ‘task2‘. 3. La distance entre le point et son point de référence (‘dictionnaire[i][2]‘).

- Utilisation : Le dictionnaire est utilisé pour stocker temporairement les distances entre chaque point de l’ensemble ‘P‘ et le dernier point ajouté à l’ensemble ‘S‘. Ces distances sont calculées et mises à jour au fur et à mesure de l’algorithme.

- Opérations : - Ajout d’un élément : Lorsque de nouveaux points sont évalués, leurs informations sont ajoutées au dictionnaire. - Suppression d’un élément : Une fois le point le plus éloigné choisi, l’entrée correspondante est retirée du dictionnaire. - Accès aux informations : L’accès aux informations de distance se fait par index (‘dictionnaire[i][2]‘).

- Efficacité : Les opérations d’ajout et de suppression dans le dictionnaire sont généralement efficaces. Cependant, la recherche de l’indice maximum dans la colonne des distances (‘indice-max-colonne‘) peut être coûteuse en termes de complexité, en fonction de sa mise en œuvre spécifique, notons que cela est arrangeant pour nous dans ce cas.

En résumé, le dictionnaire est un composant essentiel pour stocker temporairement des informations sur les distances, contribuant ainsi au processus de sélection des points pour former les clusters.

3. Analyse de Complexité Comparaison détaillée avec l’algorithme de base $O(n^3)$ pour mettre en évidence les gains obtenus avec la nouvelle approche $O(nk)$. Discussion sur les facteurs qui influent sur la complexité.

Rôle dans le Projet

Cette section joue un rôle central dans l’évolution de l’algorithme d’échantillonnage. Elle offre une alternative optimisée pour répondre aux exigences de performances lors de l’échantillonnage de données plus volumineuses, tout en posant les bases pour les sections ultérieures qui explorent des scénarios plus complexes.

3.3 Cas Métrique : Faible k - Optimisations pour k relativement petit

Cette section se concentre sur les cas où la valeur de k, le paramètre déterminant le nombre de points à échantillonner, est relativement petit. L’objectif est d’explorer des optimisations spécifiques adaptées à ces situations pour améliorer l’efficacité de l’algorithme.

Pseudo-code

Le pseudo-code correspondant à cette partie est constitué des algorithmes suivants : "Algorithme 5 : Function finding" pour rechercher le point le plus eloigner de S, "Algorithme 6 : Function updating" pour mettre a jour l’etat de notre systeme a chaque etape et enfin "Algorithme 7 : Function task3" pour initialiser l’etat de notre et demarrer le clustering.

Objectifs

L’objectif principal de cette section est de proposer des optimisations ciblées visant à accélérer le processus d’échantillonnage lorsque k est faible, tout en maintenant une qualité d’échantillonnage acceptable.

Contenu

1. Stratégies d'Échantillonnage pour Faible k

   Ici, nous avons choisi de décrire notre système avec deux variables dont le contenu évolue au cours des étapes. Nous avons une première variable nommée "centres-rayons", qui est un tableau contenant les couples (rayons-max-i, centre-i) à la i-ème ligne. La deuxième variable, appelée "regions", contient à la ligne i les points ayant centre[i][1] comme centre.

   Le principal objectif ici est, au cours d'une étape, d'éviter au maximum les actions inutiles en modifiant uniquement les régions qui vérifient une condition spécifique déduite de l'inégalité triangulaire sur la distance utilisée. On se rend compte que, pour de faibles valeurs de k, cet algorithme est beaucoup plus rapide que les deux précédents.

2. Analyse de la complexite Boucle Principale (k itérations) : Finding : Appelle une fonction externe (indice-max-colonne), qui a une complexité de O(k) dans le pire des cas. Updating : Boucles sur le nombre de régions (nombre de centres) : O(k) Boucle interne sur le nombre de points dans chaque région : O(n) Tri des points voles par p : O(n * log(n)) Mise à jour des centres-rayons et regions : Opérations en temps constant.

   Initialisation (une seule itération) : Création des structures de données initiales (centres-rayons et regions) : O(n * log(n))

   La complexité totale de cet algorithme est donc dominée par les opérations dans la boucle principale, ce qui donne une complexité totale de O(k * n * log(n)).

Rôle dans le Projet

Cette section contribue à affiner l'algorithme d'échantillonnage en fournissant des ajustements spécifiques pour des situations où la demande de précision (représentée par k) est modeste. Ces optimisations renforcent la polyvalence de l'algorithme, le rendant adaptable à un large éventail de scénarios d'utilisation.

3.4 Cas Métrique (Difficile) - Optimisations avancées pour k plus grand

Cette section se penche sur les situations où la métrique de sélection (représentée par le paramètre k) devient plus exigeante, nécessitant des optimisations avancées pour garantir des performances satisfaisantes. De plus, elle se concentre sur l'analyse approfondie de l'algorithme développé, en particulier dans des conditions difficiles.

Pseudo-code

Alors la, nous avons toujours trois algorithmes avec les meme roles que precedement mais ayant neanmoins une modifications particuliere. les algorithmes sont en anexes en sont nommee : "Algorithme 8 : Function finding4", "Algorithme 9 : Function updating4" et "Algorithme 10 : Function task4"

Objectifs

L'objectif principal de cette section est de présenter des optimisations avancées pour l'échantillonnage dans des situations où k est significativement plus grand. De plus, elle vise à fournir une analyse approfondie de la complexité de l'algorithme dans ces conditions difficiles.

Contenu

1. Explication

   Ici aussi, nous avons choisi de décrire notre système avec trois variables dont le contenu évolue au cours des étapes. Nous avons une première variable nommée "centres-rayons", qui est un tableau contenant les couples (rayons-max-i, centre-i) à la i-ème ligne. La deuxième variable, appelée "regions", contient à la ligne i les points ayant centre[i][1] comme centre et un tableau nommé "max" qui contient le centre ayant la regions avec le rayon le plus grand.

   Le principal objectif ici est, au cours d'une étape, d'éviter au maximum les actions inutiles en modifiant uniquement les régions qui vérifient une condition spécifique déduite de l'inégalité triangulaire sur la distance utilisée. On se rend compte que, pour de faibles valeurs de k, cet algorithme est beaucoup plus rapide que les deux précédents.

2. Optimisations Avancées pour k Élevé

   Tout ce que nous avons fait, c'est réduire le coût de la fonction "funding", qui est passée de O(n) à O(1). Pour y parvenir, nous avons pris l'initiative, lors de la mise à jour des variables d'état "centres-rayons" et "regions", de stocker à chaque étape un tableau nommé "max", défini comme suit : [index du centre avec le rayon maximal, rayon maximal, centre ayant le rayon maximal]. Ainsi, la recherche de Pi devient une opération triviale d'accès à la valeur d'un tableau.

3. Analyse de la Complexité dans des Conditions Difficiles

   La complexité de cet algorithme dépend principalement des boucles itératives et des opérations effectuées à chaque étape. Analysons les parties cruciales de l'algorithme :

   Boucle Principale (k itérations) : Finding4 : Opérations en temps constant. Updating4 : Boucles sur le nombre de régions (nombre de centres) : O(k) Boucle interne sur le nombre de points dans chaque région : O(n) Tri des points voles par p : O(n * log(n)) Mise à jour des centres-rayons et regions : Opérations en temps constant.

   Initialisation (une seule itération) : Création des structures de données initiales (centres-rayons et regions) : O(n * log(n))

   La complexité totale de cet algorithme est donc dominée par les opérations dans la boucle principale, ce qui donne une complexité totale de O(k * n * log(n)). Notez que cette analyse suppose que les opérations de recherche, d'ajout et de suppression dans les listes sont des opérations en temps constant, ce qui est généralement vrai pour les listes Python.

Rôle dans le Projet

Cette section apporte des améliorations spécifiques à l'algorithme d'échantillonnage, les adaptant aux situations où une précision accrue (représentée par un k plus grand) est exigée. L'analyse approfondie de la complexité offre une vision critique des performances de l'algorithme dans des conditions difficiles, guidant ainsi les choix futurs d'optimisation et de développement.

4
VOISINAGE (NEIGHBORHOOD)

4.1 Basique - graphe G avec une complexité cubique

Dans cette section, on se consacre à la construction d'un graphe un algorithme naif de complexité $O(n^3)$.

Objectifs

L'objectif principal de cette section est de fournir une mise en œuvre claire et compréhensible de l'algorithme du Neighborhood basic. En mettant l'accent sur la simplicité, nous posons les fondations nécessaires à la compréhension des versions plus avancées à venir.

Pseudo-code

Le pseudo code "Algorithme 10 : Fonction Task4 pour générer ke graphe" est fourni en annexe.

Contenu

1. Description de l'Algorithme

   L'algorithme 'task5', pour chaque point determine tous ces voisins. Voici une description détaillée de l'algorithme :

   Entrées : - 'P' : Ensemble de points de l'espace.

   Sorties : - G : le Graph des points de l'espace associé

   . On initialise les sommets du graph par les points de P.

   . On répète l'étape suivante pour chaque point.

   . À chaque itération on choisit un nouveau point. on calcule ensuite la distance entre ces points et on se rassure qu'aucun autre point n'est entre ces derniers .

   . Si pour ce point choisit, il n'y a aucun point entre les deux alors on ajoute cet edge au graphe .

   . On répète ces étapes pour tous les points de P.

   Remarques : - L'algorithme utilise la distance euclidienne mais peux etre étendu pour tout autre distance due l'espace.
2. Identification des structures de données essentielles utilisées dans l'implémentation. Pour cet algorithme, Les structures de données en place sont les tableaux pour contenir les sommets et
3. Analyse de Complexité Discussion sur la complexité algorithmique $O(n^3)$. Identification des points critiques qui contribuent à cette complexité.

4.2 Plus efficace - graphe G en temps quadratique

Dans cette section, on se consacre à la construction d'un graphe un algorithme naif de complexité $O(n^2)$.

Objectifs

L'objectif principal de cette section est de fournir une mise en œuvre claire et compréhensible de l'algorithme du Neighborhood basic beaucoup plus efficient en se basant sur le fait que chaque nouveau voisin divise l'espace des possibles futures voisins en deux.

Pseudo-code

Le pseudo code "Algorithme 14 : Fonction Task5 pour générer le graphe" est fourni en annexe.

Contenu

1. Description de l'Algorithme

   L'algorithme 'task6', pour chaque point determine tous ces voisins. Voici une description détaillée de l'algorithme :

   Entrées : - 'P' : Ensemble de points de l'espace.

   Sorties : - G : le Graph des points de l'espace associé

   . On initialise les sommets du graph par les points de P.

   . On répète l'étape suivante pour chaque point.

   . À chaque itération, on détermine les points les plus proches de point initiale grace aux algorithmes 12 $\text{Select}_c loser et 13 r_n o_h idden_b y_q . Cette partie a une complexité de O(n)$ et produit $O(1)$ points.

   . Ensuite, à partir du reste des points on applique la methode brute force(algorithme naif) sur les $O(1)$ points pour déterminer quels de ces points sonyeffectivement des voisins. Cette partie a une complexité de $O((n - (O(1))*(O(1)))$

   . On répète ces étapes pour tous les points de P.

   Remarques : - L'algorithme utilise la distance euclidienne mais peux etre étendu pour tout autre distance due l'espace.

2. Identification des structures de données essentielles utilisées dans l'implémentation. Pour cet algorithme, Les structures de données en place sont les tableaux pour contenir les sommets et Le graphe pour contenir les somments et les aretes

3. Analyse de Complexité Discussion sur la complexité algorithmique $O(n^2)$. Identification des points critiques qui contribuent à cette complexité.

4.3 Incrémental - mise à jour dee graphe G de manière incrémentale

Dans cette section, on se consacre à la construction d'un graphe un algorithme naif de complexité $O(n^2)$.

Objectifs

L'objectif principal de cette section est de fournir une mise en œuvre claire et compréhensible de l'algorithme du Neighborhood basic beaucoup plus efficient de facon itératif.

Pseudo-code

Le pseudo code "Algorithme 14 : Fonction Task7 pour générer le graphe" est fourni en annexe.

Contenu

1. Description de l'Algorithme

   L'algorithme 'Incremental', pour chaque point l'ajoute à un graphe déjà existant. Voici une description détaillée de l'algorithme :

Entrées : - ‘P‘ : Ensemble de points de l’espace. - ‘p‘ : le point à ajouter - ‘R‘ : le graphe à l’entrée Sorties : - G : le Graph R avec le point ‘p‘ ajouté

. On détermine les points les plus proches de ‘p‘ et par brute force les voisins de ‘p‘ .

.Parmi les voisind de ‘p‘, on cherche tous les couples de points qui sont déjà voisins pour le graphe R. Cette partie a une complexité de $O(n)$ .Ensuite, l’ajoute de ces voisind de ‘p‘ ne brise pas le lien déjà existant si ets eulemnt si la distance ‘p‘ à l’unde ces points est égale à la distance de ces deux points. Cette partie a une complexité de $O((‰(1))^2)$ .Enfin, on ajoute les aretes concernéés.

2. Identification des structures de données essentielles utilisées dans l’implémentation. Pour cet algorithme, Les structures de données en place sont les tableaux pour contenir les sommets et Le graphe pour contenir les somments et les aretes
3. Analyse de Complexité Discussion sur la complexité algorithmique $O(n^2)$. Identification des points critiques qui contribuent à cette complexité.

5 EXPÉRIENCES

Dans cette section, nous détaillerons les expériences menées pour évaluer les performances de nos algorithmes d’échantillonnage et de construction du graphe de voisinage. Nous diviserons cette section en trois sous-sections distinctes, chacune se concentrant sur un aspect spécifique de nos expérimentations.

5.1 Jeux de données et Génération

Dans cette première sous-section, nous présentons les différents jeux de données que nous avons utilisés pour tester nos algorithmes. À l’aide de la fonction "random", nous avons généré des données de manière aléatoire que nous stockons dans un tableau. Nous expliquons la diversité de ces jeux de données, mettant en évidence des caractéristiques variées telles que la taille, la densité et la distribution spatiale des points. Notamment, certains points sont situés sur les bords d’un cercle, d’un rectangle, ou sont générés de manière uniforme dans un pavé en dimension 2.

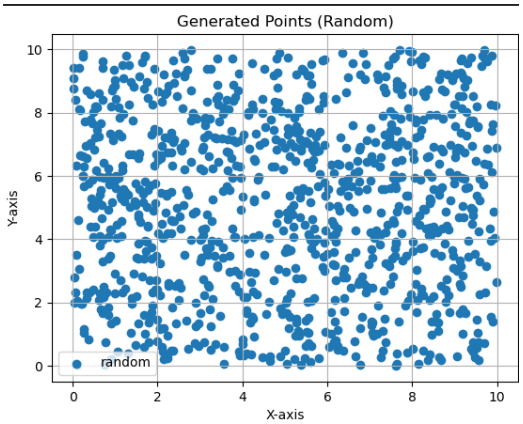

Figure 1 – points randoms

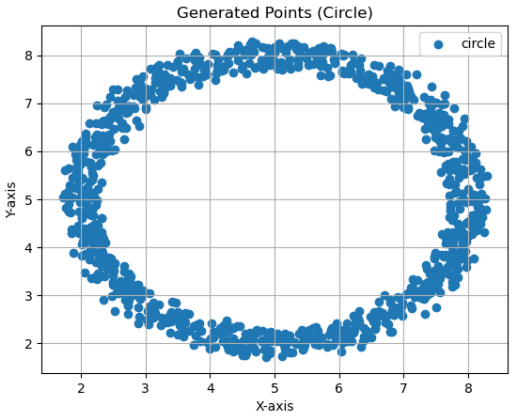

FIGURE 2 – points au bord du cercle

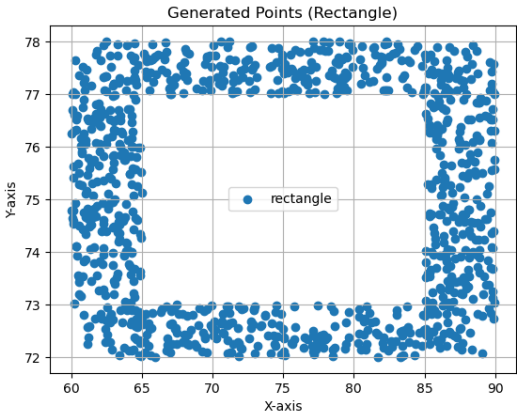

FIGURE 3 – points au bord du rectangle

5.2 VISUALISATION

La visualisation des résultats joue un rôle crucial dans la compréhension des performances de nos algorithmes. Dans notre projet nous avons juste afficher les tableau S et centres a la fin de l'obtention des k valeurs. Nous présenterons des graphiques et des représentations visuelles pour illustrer de manière efficace les caractéristiques des échantillons, du graphe de voisinage grace a la bibliotheque matplotlib, ainsi que les résultats de nos expériences. Cette visualisation contribuera à une meilleure interprétation des résultats obtenus.

```
Temps d'exécution de task4 :  1.704448938369751 pour k = 479 n = 1000
centers4 : [(-4.060183546423741, 6.89987855075946), (-4.060183546423741, 6.89987855075946), (-4.060183546423741, 6.89987855075946), (-4.0601
S4 : [(-4.060183546423741, 6.89987855075946), (9.915447518311936, -9.571304455583684), (-9.924499025979898, -9.810156700481942), (9.66049762

Temps d'exécution de task3 :  3.2817630767822266 pour k =  479 n =  1000
centers3 :  [(-4.060183546423741, 6.89987855075946), (-4.060183546423741, 6.89987855075946), (-4.060183546423741, 6.89987855075946), (-4.060
S3 :  [(-4.060183546423741, 6.89987855075946), (9.915447518311936, -9.571304455583684), (-9.924499025979898, -9.810156700481942), (9.6604976

Temps d'exécution de task2 :  2.2396562099456787 pour k =  479 n =  1000
centers2 :  [(-4.060183546423741, 6.89987855075946), (-4.060183546423741, 6.89987855075946), (-4.060183546423741, 6.89987855075946), (-4.060
S2 :  [(-4.060183546423741, 6.89987855075946), (9.915447518311936, -9.571304455583684), (-9.924499025979898, -9.810156700481942), (9.6604976

Temps d'exécution de task1 :  464.8885147571564 pour k =  479 n =  1000
centers1 :  [(-4.060183546423741, 6.89987855075946), (-4.060183546423741, 6.89987855075946), (-4.060183546423741, 6.89987855075946), (-4.060
S1 :  [(-4.060183546423741, 6.89987855075946), (9.915447518311936, -9.571304455583684), (-9.924499025979898, -9.810156700481942), (9.6604976

len(S1):  480  len(centers1):  479
len(S2):  480  len(centers2):  479
len(S3):  480  len(centers3):  479
len(S4):  480  len(centers4):  479
```

FIGURE 4 – Affichage et comparaison des resultats des taches

Dans le fichier networkanimation, vous allez trouver une animation montrant comme dans un plan on peut former le graphe

6 ALLER PLUS LOIN

Dans cette section, nous explorerons plusieurs pistes pour étendre et améliorer les fonctionnalités de notre projet au-delà des aspects fondamentaux abordés jusqu'à présent. Ces propositions, bien que considérées comme optionnelles, ouvrent la voie à des développements futurs et à une augmentation de la robustesse de notre solution.

GÉNÉRALISATION POUR DES DIMENSIONS SUPÉRIEURES

Une extension naturelle de notre projet serait la généralisation de nos algorithmes pour des dimensions supérieures.

La généralisation se fait intuitivement car la fonction que nous avons définie pour calculer la distance entre deux points est "np.linalg.norm". La différence que nous notons réside au niveau de la complexité requise pour le calcul de cette distance : elle passe de O(1) à O(d), où d est la dimension.

En considérant l'algorithme réalisé à la tâche 4, regardons comment varie le temps d'exécution en fonction de la dimension d, que nous ferons varier de 1 à 20. Nous effectuons des mesures pour différentes valeurs de d, générons des ensembles de points dans différentes dimensions, mesurons le temps d'exécution de l'algorithme pour chaque ensemble de points, puis traçons un graphique illustrant comment le temps d'exécution change en fonction de d.

En abordant ces aspects optionnels, notre projet s'ouvre à des opportunités d'amélioration et d'adaptation pour répondre à des besoins spécifiques ou des évolutions potentielles dans le domaine d'application.

7 CONCLUSION

En conclusion, ce projet, fruit d'une collaboration étroite entre mon collègue et moi-même, représente une exploration approfondie et collaborative des domaines de l'échantillonnage (subsampling) et de la construction de graphes de voisinage (neighborhood) dans des ensembles de points en dimensions variables.

Notre travail a abouti à la mise en œuvre réussie d'algorithmes efficaces pour l'échantillonnage, allant des solutions basiques à des approches plus optimisées, prenant en compte des considérations métriques cruciales pour des applications concrètes.

La construction de graphes de voisinage a également été abordée avec des approches diverses, depuis des solutions naïves jusqu'à des méthodes plus performantes, offrant une flexibilité pour répondre à différentes exigences de complexité. Cette collaboration nous a permis de relever divers défis liés à la métrique, à la gestion de la complexité, et à la maintenance incrémentale des structures de données.

Au-delà des succès obtenus, des pistes d'amélioration se dessinent naturellement. La généralisation de nos solutions à des dimensions supérieures ou l'optimisation d'heuristiques pour une sélection de points plus centraux sont autant de perspectives intéressantes que nous envisageons pour approfondir notre travail.

Bien que notre projet ait atteint ses objectifs, il n'est pas exempt de limites. Des résultats spécifiques à nos jeux de données pourraient influencer certaines conclusions, et des variations de performances pourraient être observées dans des contextes différents. Nous restons conscients de ces nuances dans l'interprétation de nos résultats.

Nous tenons à exprimer notre reconnaissance envers toutes les personnes qui ont contribué à la réussite de ce projet, que ce soit par des conseils avisés, des discussions fructueuses, ou un soutien technique précieux.

Cette expérience de collaboration et d'apprentissage mutuel a grandement enrichi notre parcours. En tant qu'équipe, nous sommes fiers des résultats obtenus et considérons ce projet comme le point de départ d'une exploration continue des problématiques algorithmiques dans le domaine de l'analyse de données.

Les compétences acquises et les leçons tirées, aussi bien sur le plan technique que collaboratif, seront des atouts précieux pour nos projets futurs et nos démarches de recherche plus approfondies.

8
ANNEXES

Algorithme 1 : Fonction Task1 pour effectuer une tâche de regroupement

Entrées : P - ensemble de points, p1 - point de départ, k - nombre de clusters
Sorties : Tuple contenant les centres de cluster (centers) et l'ensemble de points dans chaque cluster (S)

$S \leftarrow [p1]$ Initialiser l'ensemble de points avec le point de départ
$centers \leftarrow []$ Initialiser la liste des centres de cluster
pour u ***dans*** *plage(k)* **faire**
  $point_{max} \leftarrow []$;
  $distance_{max} \leftarrow 0$;
  pour p ***dans*** P **faire**
    si p *n'est pas dans* S **alors**
      $distance_{min} \leftarrow \infty$;
      pour s ***dans*** S **faire**
        $distance_{ps} \leftarrow$ distance_euclidienne(p, s) ;
        si $distance_{ps} < distance_{min}$ **alors**
          $distance_{min} \leftarrow distance_{ps}$;
        fin
      fin
      si $distance_{min} > distance_{max}$ **alors**
        $distance_{max} \leftarrow distance_{min}$;
        $point_{max} \leftarrow p$;
      fin
    fin
  fin
  $point_{min} \leftarrow []$ Rechercher S qui est le plus proche de $P_{\max}$ $distance_{min} \leftarrow \infty$;
  pour s ***dans*** S **faire**
    $distance_{ps} \leftarrow$ distance_euclidienne$(point_{max}, s)$;
    si $distance_{ps} < distance_{min}$ **alors**
      $distance_{min} \leftarrow distance_{ps}$;
      $point_{min} \leftarrow s$;
    fin
  fin
  S.ajouter$(point_{max})$;
  $centers$.ajouter$(point_{min})$;
fin
retourner $centers, S$

Algorithme 2 : Task2 Function to Find Index of Maximum Value in a Column

Input : liste - input list, numero_colonne - column number
Output : Index of the maximum value in the specified column
colonne_max ← max(range(len(liste)), key=lambda i : liste[i][numero_colonne]);
return *colonne_max*;

Algorithme 3 : Task 2 Function

Input : P - set of points, $p1$ - starting point, k - number of clusters
Output : Tuple containing the cluster centers (centers) and the set of points in each cluster (S)

```
S ← [p1];
centers ← [];
dictionnaire ← [];
numero_colonne ← 2;
foreach p in P do
    if p not in S then
        dictionnaire.append([p, p1, euclidean_distance(p, p1)]);
    end
end
indice_max ← IndiceMaxColonne(dictionnaire, numero_colonne);
point_max ← dictionnaire[indice_max][0];
point_min ← dictionnaire[indice_max][1];
S.append(point_max);
centers.append(point_min);
dictionnaire.pop(indice_max);
for u in range(k − 1) do
    for i in range(len(dictionnaire)) do
        distance_ps ← euclidean_distance(dictionnaire[i][0], S[−1]);
        if distance_ps < dictionnaire[i][2] then
            dictionnaire[i][1] ← S[−1];
            dictionnaire[i][2] ← distance_ps;
        end
    end
    indice_max ← IndiceMaxColonne(dictionnaire, numero_colonne);
    point_max ← dictionnaire[indice_max][0];
    point_min ← dictionnaire[indice_max][1];
    S.append(point_max);
    centers.append(point_min);
    dictionnaire.pop(indice_max);
end
return centers, S;
```

Algorithme 4 : Function to Find Index of Maximum Value in a Column

Input : liste - input list, numero_colonne - column number
Output : Index of the maximum value in the specified column

```
colonne_max ← max(range(len(liste)), key=lambda i : liste[i][numero_colonne]);
return colonne_max;
```

Algorithme 5 : Function `finding`

Input : S - set of points
Output : Index of the array `centres_rayons` with the largest radius

```
return indice_max_colonne(S, 0);
```

Algorithme 6 : Function `updating`

Input : `centres_rayons` - array of center-radius pairs, `regions` - array of distance-point pairs, `ligne_p` - line index
Output : Updated arrays `centres_rayons` and `regions`

```
r ← [];
distance_point ← regions[ligne_p].pop(len(regions[ligne_p]) − 1);
if len(regions[ligne_p]) > 0 then
    centres_rayons[ligne_p][0] ← regions[ligne_p][−1][0];
end
else
    centres_rayons[ligne_p][0] ← 0;
end
centres_rayons.append([0, distance_point[1]]);
tableau_temporaire ← [];
index_delete ← [];
for i in range(len(regions)) do
    distance_reference ← euclidean_distance(distance_point[1], centres_rayons[ligne_p][1]);
    if distance_reference < 2 × centres_rayons[i][0] then
        table ← [];
        for j in range(len(regions[i])) do
            if [i, j] not in index_delete then
                distance ← euclidean_distance(distance_point[1], regions[i][j][1]);
                if distance < regions[i][j][0] then
                    index_delete.append([i, j]);
                    temporaire ← regions[i][j];
                    tableau_temporaire.append((distance, temporaire[1]));
                end
                else
                    table.append(regions[i][j]);
                end
            end
        end
        r.append(table);
    end
    else
        r.append(regions[i]);
    end
    if len(r[i]) > 0 then
        centres_rayons[i][0] ← r[i][−1][0];
    end
    else
        centres_rayons[i][0] ← 0;
    end
end
regions ← r;
tableau_temporaire ← sorted(tableau_temporaire, key = λx : x[0]);
if len(tableau_temporaire) > 0 then
    centres_rayons[−1][0] ← tableau_temporaire[−1][0];
end
regions.append(tableau_temporaire);
return centres_rayons, regions;
```

Algorithme 7 : Function `task3`

Input : P - set of points, p1 - starting point, k - number of clusters
Output : Tuple containing the cluster centers (`centers`) and the set of points in each cluster (`S`)

```
S ← [p1];
centers ← [];
centres_rayons ← [[0, p1]];
regions ← [];
distance_max ← 0;
tableau ← [];
for p in P do
    if p ≠ p1 then
        distance ← euclidean_distance(p, p1);
        tableau.append((distance, p));
        if distance > distance_max then
            distance_max ← distance;
        end
    end
end
centres_rayons[0][0] ← distance_max;
tableau ← sorted(tableau, key = λx : x[0]);
regions.append(tableau);
for u in range(k) do
    ligne_p ← finding(centres_rayons);
    S.append(regions[ligne_p][−1][1]);
    centers.append(centres_rayons[ligne_p][1]);
    centres_rayons, regions ← updating(centres_rayons, regions, ligne_p);
end
return centers, S;
```

Algorithme 8 : Function `finding4`

Input : s - input array
Output : First element of the array s

```
return s[0];
```

Algorithme 9 : Function `updating4`

Input : `max` - input array, `centres_rayons` - array of center-radius pairs, `regions` - array of distance-point pairs
Output : Updated array `max`, array `centres_rayons`, and array `regions`

```
ligne_p ← max[0];
r ← [];
imax ← 0;
dmax ← 0;
distance_point ← regions[ligne_p].pop(len(regions[ligne_p]) − 1);
if len(regions[ligne_p]) > 0 then
    centres_rayons[ligne_p][0] ← regions[ligne_p][−1][0];
end
else
    centres_rayons[ligne_p][0] ← 0;
end
centres_rayons.append([0, distance_point[1]]);
tableau_temporaire ← [];
index_delete ← [];
for i in range(len(regions)) do
    distance_reference ← euclidean_distance(distance_point[1], centres_rayons[i][1]);
    if distance_reference ≤ 2 × centres_rayons[i][0] then
        table ← [];
        for j in range(len(regions[i])) do
            if [i, j] not in index_delete then
                distance ← euclidean_distance(distance_point[1], regions[i][j][1]);
                if distance < regions[i][j][0] then
                    index_delete.append([i, j]);
                    temporaire ← regions[i][j];
                    tableau_temporaire.append((distance, temporaire[1]));
                    if len(regions[i]) == 1 then
                        centres_rayons[i][0] ← 0;
                    end
                    else
                        centres_rayons[i][0] ← regions[i][−1][0];
                    end
                end
                else
                    table.append(regions[i][j]);
                end
            end
        end
        r.append(table);
    end
    else
        r.append(regions[i]);
    end
    if len(r[i]) > 0 then
        centres_rayons[i][0] ← r[i][−1][0];
    end
    else
        centres_rayons[i][0] ← 0;
    end
    if centres_rayons[i][0] > dmax then
        dmax ← centres_rayons[i][0];
        imax ← i;
    end
end
regions ← r;
tableau_temporaire ← sorted(tableau_temporaire, key = λx : x[0]);
if len(tableau_temporaire) > 0 then
```

Algorithme 10 : Function `task4`

Input : P - set of points, p1 - starting point, k - number of clusters
Output : Tuple containing the cluster centers (`centers`) and the set of points in each cluster (`S`)

$S \leftarrow$ [p1];
centers $\leftarrow$ [];
centres_rayons $\leftarrow$ [[0, p1]];
regions $\leftarrow$ [];
distance_max $\leftarrow 0$;
tableau $\leftarrow$ [];
for *p in P* **do**
  if $p \neq p1$ **then**
    distance $\leftarrow$ euclidean_distance(p, p1);
    tableau.append((distance, p));
    if *distance* $>$ *distance_max* **then**
      distance_max $\leftarrow$ distance;
    end
  end
end
centres_rayons[0][0] $\leftarrow$ distance_max;
tableau $\leftarrow$ sorted(*tableau*, key $= \lambda x : x[0]$);
regions.append(*tableau*);
max $\leftarrow$ [0, centres_rayons[0][0], centres_rayons[0][1]];
for *u in range(k)* **do**
  ligne_p $\leftarrow$ finding4(max);
  S.append(regions[ligne_p][-1][1]);
  centers.append(centres_rayons[ligne_p][1]);
  max, centres_rayons, regions $\leftarrow$ updating4(max, centres_rayons, regions);
end
return *centers*, S;

Algorithme 11 : Task5

Data : P : List of points
Result : G : Graph

```
G ← Create an empty graph;
S ← P;
Bool ← 1;
edges ← [];
for p in S do
    for s in S do
        a ← euclidean_distance(p, s);
        if a = 0 then
            continue;
        end
        Bool ← 1;
        for k in S do
            if max(euclidean_distance(p, k), euclidean_distance(s, k)) < a then
                Bool ← 0;
                break;
            end
        end
        if Bool = 1 then
            edges.append((p, s));
        end
    end
    G.add_edges_from(edges);
    edges ← [];
end
return G;
```

Algorithme 12 : Select Closer

Data : p : Point, P : List of points
Result : r : Closest point

```
r ← P[0];
a ← euclidean_distance(p, r);
for j ← 1 to len(P) do
    b ← euclidean_distance(p, P[j]);
    if b = 0 then
        continue;
    end
    if a > b or a = 0 then
        r ← P[j];
        a ← b;
    end
end
return r;
```

Algorithme 13 : *r_no_hidden_by_q*

Data : p : Point, q : Point, S : List of points
Result : T : List of points

```
T ← [];
for r in S do
    a ← euclidean_distance(p, r);
    if euclidean_distance(q, r) > a and a ≠ 0 then
        T.append(r);
    end
end
return T;
```

Algorithme 14 : Task6

Data : P : List of points
Result : G : Graph

```
G ← Create an empty graph;
G.add_nodes_from(P);
edges ← [];
Rest_P ← [];
Bool ← 1;
for p in P do
    S ← P;
    while len(S) ≠ 0 do
        q ← select_closer(p, S);
        Rest_P.append(q);
        S ← r_no_hidden_by_q(p, q, S);
    end
    Rest ← list(set(P) − set(Rest_P));
    for r in Rest_P do
        a ← euclidean_distance(p, r);
        if a = 0 then
            continue;
        end
        for k in Rest do
            if max(euclidean_distance(p, k), euclidean_distance(r, k)) < a then
                Bool ← 0;
                break;
            end
        end
        if Bool = 1 then
            edges.append((p, r));
        end
        Bool ← 1;
    end
    G.add_edges_from(edges);
    edges ← [];
    Rest_P ← [];
end
return G;
```

Algorithme 15 : Incremental

Data : p : Point, P : List of points, R : Graph
Result : G : Graph, $Edges$: List of edges

$S \leftarrow P$.copy();
$T \leftarrow []$;
$Edges \leftarrow []$;
$Rest\_T \leftarrow []$;
Bool $\leftarrow 1$;
$G \leftarrow R$;
while $len(S) \neq 0$ **do**
  $q \leftarrow$ select_closer(p, S);
  T.append(q);
  $S \leftarrow$ r_no_hidden_by_q(p, q, S);
end
if $len(T) = 1$ ***and*** $T[0] = p$ **then**
  $T \leftarrow []$;
end
$Rest \leftarrow$ list(set(P) − set(T));
for r ***in*** T **do**
  $a \leftarrow$ euclidean_distance(p, r);
  if $a = 0$ **then**
    continue;
  end
  for k ***in*** $Rest$ **do**
    if max$(euclidean_distance(p, k), euclidean_distance(r, k)) < a$ **then**
      Bool $\leftarrow 0$;
      break;
    end
  end
  if $Bool = 1$ **then**
    $Rest\_T$.append(r);
  end
  Bool $\leftarrow 1$;
end
for u ***in*** $Rest\_T$ **do**
  for v ***in*** $Rest\_T$ **do**
    if $G.has\_edge(u, v)$ **then**
      $a \leftarrow$ euclidean_distance(u, v);
      if $euclidean\_distance(p, u) \neq a$ ***or*** $euclidean\_distance(p, v) \neq a$ **then**
        G.remove_edge(u, v);
      end
    end
    $Edges$.append$((p, v))$;
  end
  $Edges$.append$((u, p))$;
end
return $G, Edges$;

Algorithme 16 : Task7

Data : P : List of points, Z : Graph
Result : Z : Graph

$edge \leftarrow []$;
for p ***in*** P **do**
    Z.add_nodes_from($[p]$);
    $Z, ed \leftarrow$ incremental(p, P, Z);
    $edge$.extend(ed);
end
Z.add_edges_from($edge$);
return Z;