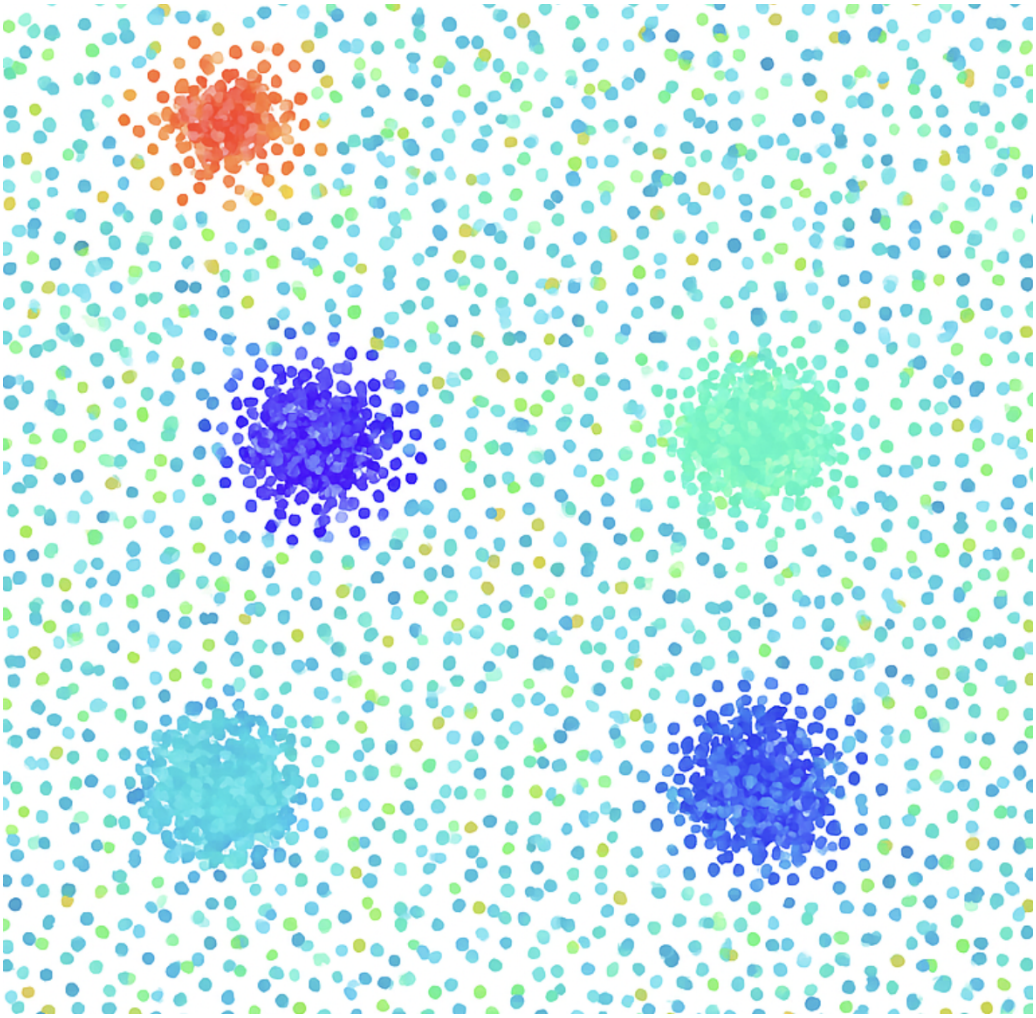
Rapport d'Algorithmie 2024-2025

Algorithme hybride DFS-Glouton pour la recherche de composantes connexes dans un ensemble de points

Dieynaba Dogue & Akram Mohammed Lrhorfi



Dans l'analyse de réseau, on cherche toujours à savoir quels éléments partagent une certaine relation. Leur degré de rapprochement est modélisé généralement par une sorte de seuil sur notre espace d'étude, notamment dans l'apprentissage automatique avec l'algorithme de K-means vu brièvement en prépa.

C'est ainsi qu'on peut comprendre l'intérêt d'identifier des composantes connexes dans un nuage de points. À ce sujet-là, nous avons implémenté deux algorithmes pour détecter ces composantes dans un graphe implicite où les arêtes relient les points à une distance $\leq s$. Afin d'aboutir à notre but, on va suivre le schéma suivant :

- **Le fil de notre pensée** : Modélisation des algorithmes et pourquoi/comment nous avons optimisé notre algorithme
- **Évaluation** : Recherche de paramètres optimaux et comparaison des performances sur différentes densités
- **Analyse théorique** : Complexités temporelles des algorithmes
- **Synthèse et perspectives d'amélioration**

1. Le fil de notre pensée

Modélisation du problème

Le nuage de points est modélisé comme un graphe $G = (V, E)$:

- V : Points avec coordonnées (x, y).
- E : Arêtes si la distance euclidienne entre deux points est $\leq s$.

Les composantes connexes représentent des groupes de points connectés directement ou indirectement (c'est-à-dire par une suite d'arêtes).

Première idée

Dans un premier temps, on cherche à savoir une composante connexe en partant d'un seul point. Cela nous amène à choisir notre algorithme de parcours. En effet, aller d'un premier point pour explorer ses voisins peut se faire généralement de deux manières dans un graphe : soit par un parcours en largeur où on parcourt les voisins les plus proches jusqu'à ceux les plus lointains, soit par un parcours en profondeur (DFS) où on va jusqu'au fond de notre parcours puis rebrousse chemin par récursion.

Mais

Le problème avec le parcours en largeur, c'est qu'il commence à mélanger des points éloignés dans notre liste de points déjà visités, réduisant l'efficacité de la localité spatiale. Alors qu'avec un parcours en profondeur ou DFS, on évite les vérifications redondantes et les sauts de mémoire inutiles.

DFS-classique

L'algorithme DFS classique parcourt le graphe en profondeur pour identifier chaque composante connexe, en marquant les points visités. Depuis un point non visité, il explore tous les voisins à distance $\leq s$, en utilisant une pile.

```
1 def print_components_sizes_dfs_classique(distance, points):
2     """
```

```
    affichage des tailles triees de chaque composante
    """

    # Dictionnaire pour stocker les composantes connexes
    c_connexes = {}

    def connexe(p1, points, sommets=None):
        if sommets is None:
            sommets = [p1]
        for p2 in points:
            # On recherche les voisins de p1
            if p1.distance_to(p2) <= distance and p2 not in sommets:
                sommets.append(p2)
                connexe(p2, points, sommets)  # On recherche les
                    voisins de voisins
        return sommets
    indice = 0
    copie = points.copy()
    # On construit chaque composante connexe
    while copie:
        p1 = copie[0]  # Prendre le premier point restant
        c_connexes[indice] = connexe(p1, points)  # Trouver sa
            composante connexe
        # Supprimer les points de cette composante de la copie
        for p2 in c_connexes[indice]:
            if p2 in copie:
                copie.remove(p2)
        indice += 1
    # Calculer les tailles des composantes
    tailles = []
    for composante in c_connexes.values():  # Utiliser .values() pour
        obtenir les listes de points
        tailles.append(len(composante))  # Ajouter la taille de la
            composante
    tailles.sort(reverse=True)
    sortie = "["
    for i in range(len(tailles)):
        sortie += str(tailles[i])
        if i < len(tailles) - 1:
            sortie += ", "
    sortie += "]"
    print(sortie)
    return tailles
```

Les premiers problèmes

Avec le DFS classique, on a rencontré certains défis d'optimisation :

- **Lent sur des graphes denses** : Pour chaque point, vérifier la distance à tous les autres points conduit à une complexité quadratique dans le pire des cas.
- **Redondance** : Les calculs de distances sont répétés inutilement pour des points déjà explorés (problème de mémorisation).
- **Scalabilité limitée** : Pour $n \geq 5000$, le temps d'exécution devient prohibitif, surtout avec des données clusterisées où les composantes sont grandes ou si, par exemple, on n'a

pas imposé la profondeur de récursion pour le code.

L'algorithme hybride glouton + DFS

Pour éviter ces petits problèmes, on s'est inspirés de l'idée du TD 3 d'algorithmique où on a mixé entre tri par insertion et récursivité pour créer un nouveau tri fusion rapide. Dans notre cas d'étude, nous n'avons pas une liste à trier mais une composante à déterminer. Nous avons mis en place un algorithme qui regroupe rapidement les points proches (glouton) jusqu'à une taille limite k, puis applique DFS pour les composantes plus grandes. Notre approche se décompose comme suit :

- **Phase gloutonne** : Construit des composantes partielles en ajoutant les voisins proches (cela evite l'exploration de tous le graphe) .
- **Phase DFS** : Complète l'exploration pour les composantes dépassant k, garantissant la précision.
- **Rôle de** k : Contrôle le compromis entre rapidité (petit k) et précision (grand k).

```python
def print_components_sizes_dfs_glouton(distance, points):
    """
    affichage des tailles triees de chaque composante
    """
    n = len(points)
    visites = [False] * n
    tailles = []
    k=8 # on a choisi k=8 car c est suppose etre le plus optimal ( voir pdf )

    def glouton(depart):
        composante = [depart]
        visites[depart] = True
        pile = [depart]
        while pile:
            if len(composante) > k:
                return composante, True
            courant = pile.pop()
            for voisin in range(n):
                if not visites[voisin] and points[courant].distance_to(points[voisin]) <= distance:
                    composante.append(voisin)
                    pile.append(voisin)
                    visites[voisin] = True
        return composante, False

    def dfs(composante_initiale): # pour ce dfs on adopte une approche iterative pour eviter les recurssions limit errors
        # On continue l exploration a partir des points deja dans composante initiale
        taille_composante = len(composante_initiale)
        pile = composante_initiale.copy()  # Commencer avec tous les points de la composante initiale
        while pile:
            courant = pile.pop()
            for voisin in range(n):
                if not visites[voisin] and
```

```
                    points[courant].distance_to(points[voisin]) <=
                    distance:
33                   pile.append(voisin)
34                   visites[voisin] = True
35                   taille_composante += 1
36          return taille_composante
37
38      for i in range(n):
39          if not visites[i]:
40              composante, basculer = glouton(i)
41              if basculer:
42                  taille = dfs(composante)  # Passer la composante
                        entiere a dfs
43              else:
44                  taille = len(composante)
45              tailles.append(taille)
46
47      tailles.sort(reverse=True)
48      sortie = "["
49      for i in range(len(tailles)):
50          sortie += str(tailles[i])
51          if i < len(tailles) - 1:
52              sortie += ", "
53      sortie += "]"
54      print(sortie)
55      return tailles
```

On peut aussi ajouter que dans les graphes denses, les points proches forment souvent des clusters naturels. En prétraitant ces clusters avec une approche gloutonne (plus rapide pour les composantes de petite taille), on réduit le nombre de calculs de distances avant d'appliquer DFS.

2. Évaluation

k optimal

Les tests ont été effectués pour l'algorithme hybride avec k allant de 1 à 10. Le graphe ci-dessous résume les temps :

On recherche un k qui équilibre la phase gloutonne (exploration limitée) et la phase DFS. Pour n petits, on a remarqué $k = 2$ ou 3 sont légèrement meilleurs (ex. : 1.06 ms pour $k = 2$, $n = 100$), car les composantes sont très petites. Par contre, pour n grand, on a constaté que $k = 6$ ou 8 sont bien meilleurs (19576.73 ms pour $k = 8$, $n = 1000000$). Cependant, même au alentour de $k = 8$ il y a de forte variations, ce qui explique que notre modèle dépend de la distribution auquel il est sujet.

Cluster vs Uniforme

Nous avons évalué l'algorithme hybride DFS-Glouton sur deux types de distributions : uniforme (les résultats déjà montré dans la section précédente) et clusterisée. La figure ci-dessous montre un exemple de distribution clusterisée pour $n = 1000$, où les points forment des groupes distincts, contrairement à une distribution uniforme où les points sont répartis de manière plus homogène.

En réalité, dans une distribution clusterisée, les composantes sont plus grandes en raison

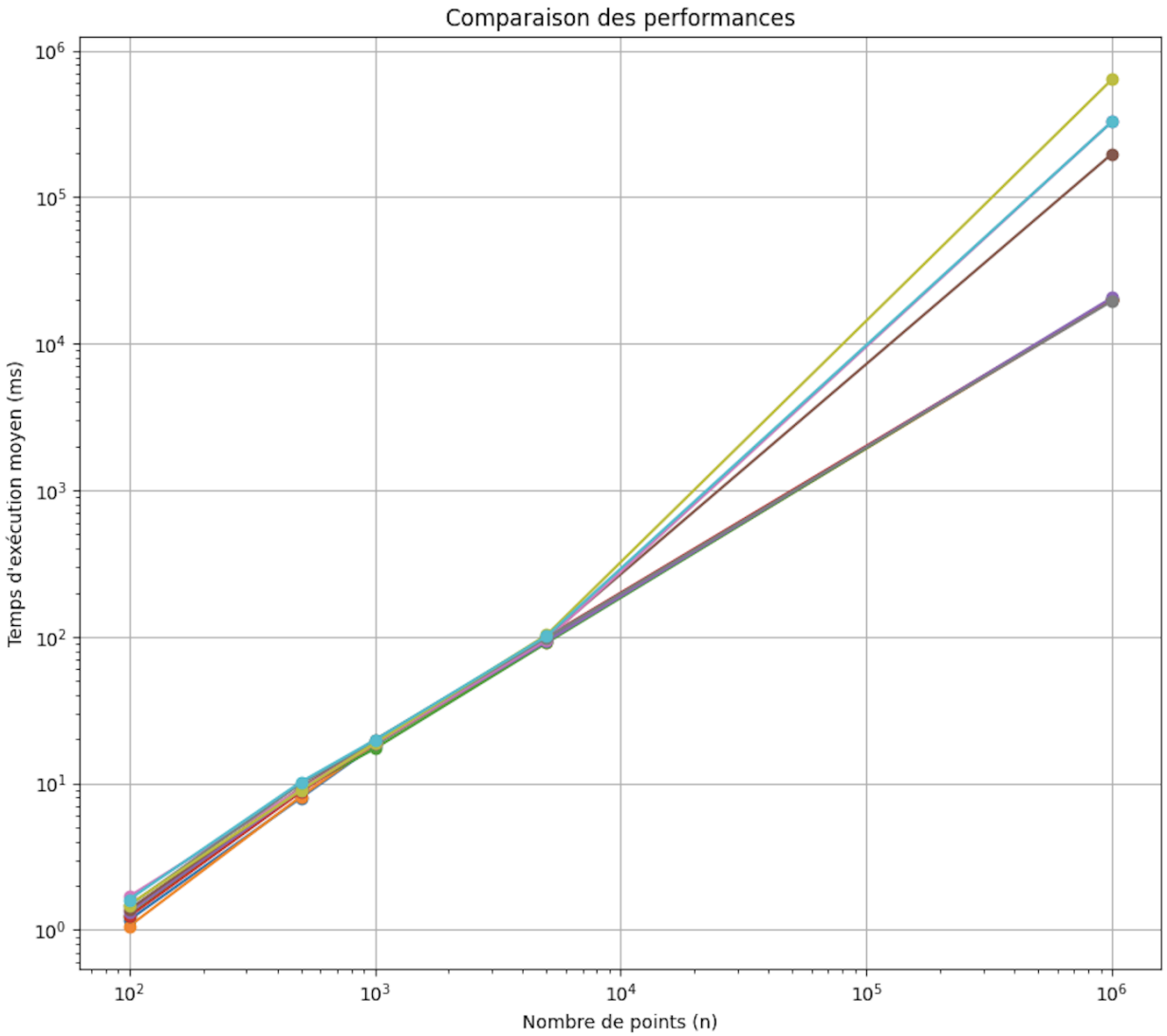


FIGURE 1 – Temps d'exécution en fonction de k pour différentes tailles n.

des regroupements naturels de points. Pour $n = 100$, $k = 6$ est optimal (0.59 ms), car la phase gloutonne peut rapidement regrouper les points au sein des clusters. À $n = 500$, $k = 7$ donne le meilleur temps (2.55 ms), et pour $n = 1000$, $k = 10$ est le plus rapide (5.09 ms), bien que $k = 6$ (5.98 ms) reste performant. Cela montre que des valeurs de k légèrement plus grandes sont préférables pour les données clusterisées, car elles permettent à la phase gloutonne de capturer efficacement les grandes composantes avant de passer à la phase DFS.

Ainsi, l'algorithme hybride s'adapte bien aux deux types de distributions : pour les données uniformes, un k (3–6) est idéal, tandis que pour les données clusterisées, un k plus grand (6–10) reduit le temps d'exécution global.

3. Analyse théorique

Comparaison des complexités

— **DFS Classique** :

Pour chaque point dans la composante C_i, on vérifie les voisins parmi tous les n points, mais seuls les points non encore visités (i.e., hors de la composante courante) sont ajoutés. Le coût par point est proportionnel au nombre moyen de voisins à une

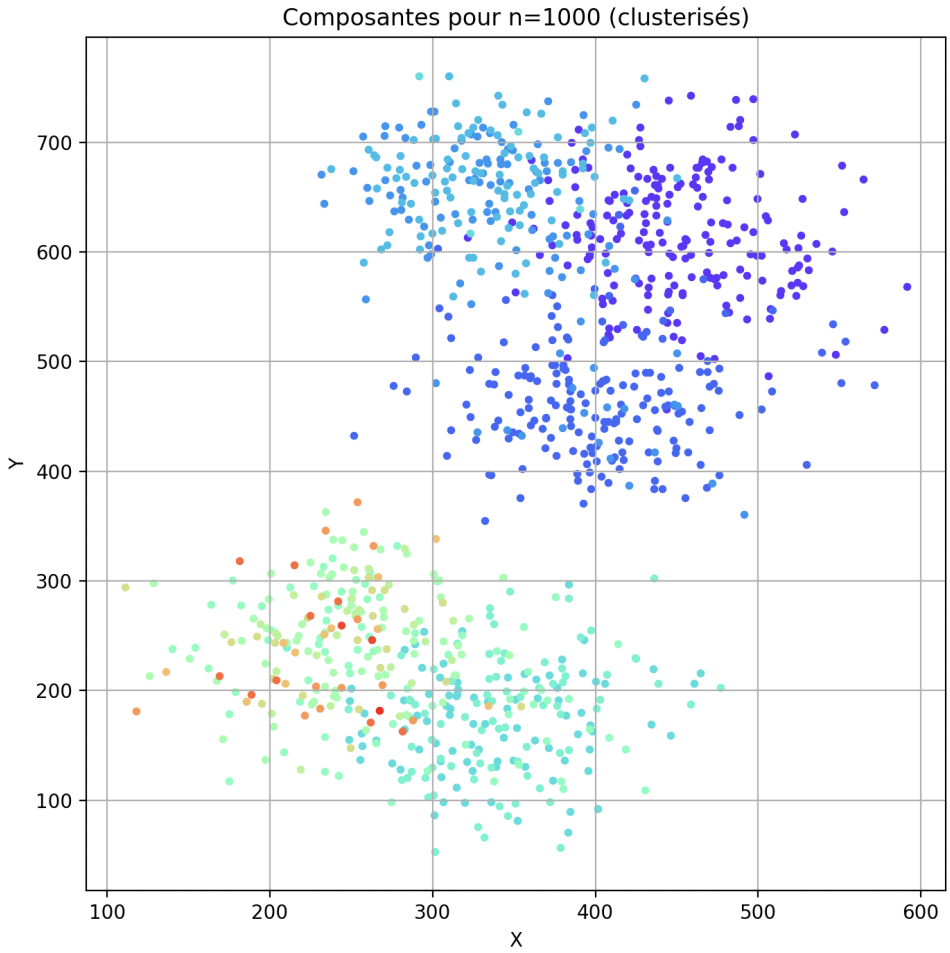


FIGURE 2 – Exemple de distribution clusterisée pour $n = 1000$.

distance $\leq d$, soit $\deg_{\text{moyen}}$. Ainsi, le coût total par composante est :

$$\text{Coût}_i = C_i \times \deg_{\text{moyen}}$$

en moyenne. Le coût total dépend du nombre de composantes, mais est proportionnel à $\deg_{\text{moyen}}$. Puisque chaque point est traité exactement une fois, le coût total est :

$$\text{Coût total} = \sum_i \text{Coût}_i = \left(\sum_i C_i\right) \times \deg_{\text{moyen}} = n \times \deg_{\text{moyen}}$$

La complexité est donc $O(n \cdot \deg_{\text{moyen}})$ en pratique. Si $\deg_{\text{moyen}}$ est constant, cela devient $O(n)$, mais dans le pire cas, $\deg_{\text{moyen}}$ peut croître avec n, menant à $O(n^2)$.

— **Hybride (Glouton + DFS)** :
  — L'algorithme se divise en deux phases :
    — **Phase gloutonne** : Pour chaque composante de taille C_i, on explore les voisins de manière gloutonne jusqu'à une taille $\min(C_i, k)$, coûtant :

$$\min(C_i, k) \times \deg_{\text{moyen}}$$

en moyenne, où $\deg_{\text{moyen}}$ est le degré moyen d'un point (nombre moyen de voisins à une distance $\leq d$).

— **Phase DFS** : Si $C_i > k$, DFS explore les $C_i - k$ sommets restants en continuant à partir de tous les points déjà trouvés dans la phase gloutonne, coûtant :

$$(C_i - k) \times \deg_{\text{moyen}}$$

Le coût total par composante est donc :

$$\mathbb{E}[\text{Coût}_i] = \mathbb{E}[\min(C_i, k)] \times \deg_{\text{moyen}} + P(C_i > k) \times \mathbb{E}[(C_i - k) \mid C_i > k] \times \deg_{\text{moyen}}$$

Le coût total dépend du nombre de composantes, mais est proportionnel à $\deg_{\text{moyen}}$, soit $O(n \cdot \deg_{\text{moyen}})$ en pratique. Si $\deg_{\text{moyen}}$ is constant, cela devient $O(n)$, mais dans le pire cas, $\deg_{\text{moyen}}$ peut croître avec n, menant à $O(n^2)$.

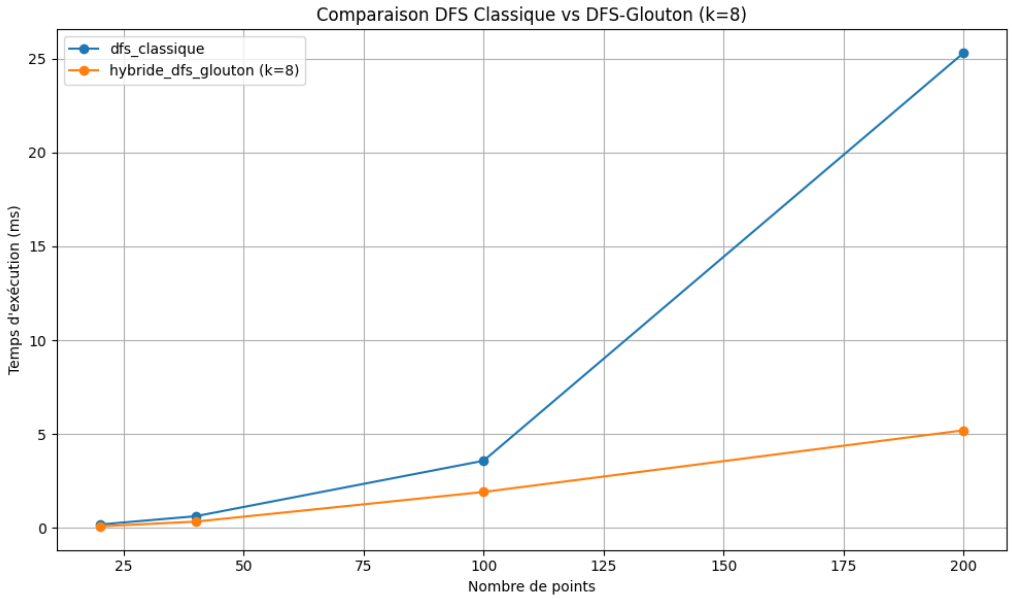


FIGURE 3 – Performances des algorithmes DFS classique et hybride sur les fichiers exemple_*.pts.

4. Synthèse

L'algorithme hybride DFS-Greedy surpasse le DFS classique en termes de performance, notamment pour les grandes tailles de données. Nos tests montrent que pour $n = 200$ points (exemple_4.pts), l'hybride (avec $k = 8$) est environ 7 fois plus rapide que le DFS classique (3.50 ms contre 25.27 ms). Cette amélioration est cohérente avec l'analyse théorique : le coût attendu par composante est réduit . La phase gloutonne capture efficacement les petites composantes, tandis que la phase DFS garantit la précision pour les rares grandes composantes, comme observé dans exemple_4.pts où une composante de taille 197 est correctement identifiée.

Perspectives

Des optimisations comme l'indexation spatiale (ex. : quad-trees) peuvent réduire les vérifications de distance de O(n) à O(degeff). La parallélisation permettrait de traiter plusieurs composantes simultanément sur des architectures multicœurs. Tester des distributions clusterisées pourrait confirmer l'optimalité de k=8 ou indiquer un meilleur choix. Varier le seuil de distance aiderait à évaluer l'impact de degmoyen. Enfin, ce type d'algorithme a des applications en géospatial, réseaux sociaux, et segmentation d'images.