Projet Algo 2021

18 avril 2021

Introduction :

Cette année, le projet était très intéressant, non seulement par sa simplicité extérieure, ainsi que par sa complexité. En effet, la problématique était de chercher l'ensemble des parties connexes d'un graphe. L'idée de l'algorithme n'était pas si difficile à saisir, et conceptualiser, la difficulté réside plutôt dans l'optimisation du temps d'exécution en variant les techniques. Nous mettrons par la suite l'accent sur les différents algorithmes et code utilisé pour résoudre cette problématique

I - 1er programme :

1. méthode suivi :

   Nous nous sommes basés principalement sur deux fonctions: *proches_voisins(points, distance)* qui cherche les proches voisins d'une manière très naïve, et la fonction *print_components_sizes(distance, points)* qui permet d'afficher les tailles des différentes composantes connexes du graphe.

```
27
28  def proches_voisins( points, distance):
29      M = {pt : [] for pt in points}
30      n = len(points)
31      for i in range(n):
32          for j in range(n):
33              pt1 , pt2 = points[i], points[j]
34              if pt1.distance_to(pt2) <= distance :
35                  M[pt1] += [pt2]
36      return M
```

```python

def print_components_sizes(distance, points):
    """
    affichage des tailles triees de chaque composante
    """
    visited = {}
    for i in points :
        visited[i] = False
    L =[]
    for e in points :
        if not visited[e] :
            q = queue.Queue()
            q.put(e)
            visited[e] = True
            taille = 1
            while not q.empty() :
                w = q.get()
                for k in proches_voisins(points, distance)[w] :
                    if not visited[k] :
                        visited[k] = True
                        q.put(k)
                        taille += 1
            L += [taille]
    L.sort(reverse= True)
    print(L)
```

2. inconvénients :

   La fonction proche_voisin était très brute, et de complexité très forte ($O(n^2)$) qui se multiplie par le coût de la boucle while et la boucle for au sein de la fonction print_compenents_sizes, d'où la nécessité de réimplémenter ces deux fonctions.

II - 2eme programme :

1. méthode suivi :

En s'inspirant de la méthode précédente, on s'est dit qu'il serait peut-être plus efficace d'implémenter une boucle imbriquée pour qu'on puisse profiter de la symétrie de la distance, nous avions également pensé à sortir la fonction proche_voisin du code et n'utiliser que la partie dont on avait besoin de.

```
27
28 def proches_voisins( points, distance):
29     M = {pt : [] for pt in points}
30     n = len(points)
31     for i in range(n):
32         for j in range(n):
33             pt1 , pt2 = points[i], points[j]
34             if pt1.distance_to(pt2) <= distance :
35                 M[pt1] += [pt2]
36     return M
```

```
52
53 def print_components_sizes(distance, points):
54     """
55     affichage des tailles triees de chaque composante
56     """
57     visited = {}
58     for i in points :
59         visited[i] = False
60     L =[]
61     voisin = proches_voisins(points, distance)
62     for e in points :
63         if not visited[e] :
64             q = queue.Queue()
65             q.put(e)
66             visited[e] = True
67             taille = 1
68             while not q.empty() :
69                 w = q.get()
70                 for k in voisin[w] :
71                     if not visited[k] :
72                         visited[k] = True
73                         q.put(k)
74                         taille += 1
75             L += [taille]
76     L.sort(reverse= True)
77     print(L)
```

2. inconvénients :

   Même après avoir essayer de diminuer la complexité de ces deux fonctions, elle a resté quand même énorme, parce qu'on n'a pas optimisé le parcours des points du graphe, ceci dit qu' à chaque fois on recalcule les points voisins dans tout l'espace des points sans éliminer les points dont les composantes connexes ont été déjà trouvé.

III - 3eme programme :

1. méthode suivi :

   A fin de plus optimiser, nous avions essayé de définir une fonction récursive *connected_componenets(pnt_0, points, distance)* qui retourne la partie connexe dont pnt_0 fait partie, et puis on itère sur l'ensemble des points.

2. inconvénients :

   Bien que cette solution apparait optimal, mais elle nous a pas permet d'atteindre le nombre de points que nous voulions atteindre, ainsi qu'elle time out sur 100.000 points.

IV - 4eme programme :

1. méthode suivi :

   La méthode de ce programme consiste à éliminer tout point faisant partie d'une partie connexe déjà établie. Pour cela, nous avions implémenté des fonctions supplémentaires dans un fichier nommé file :

```
28
29  def print_components_sizes(distance, points):
30      """
31      affichage des tailles triees de chaque composante
32      """
33      Liste = []
34      out_points = points
35      while out_points != []:
36          pnt = out_points[0]
37          queue = file.Queue()
38          queue.enqueue(pnt)
39          taille = 0
40          while  queue.isEmpty() == False :
41                  w = queue.dequeue()
42                  a = file.Points_within_radius(w, out_points, distance)
43                  list_points =a[0]
44                  out_points = a[1]
45                  for pt in list_points :
46                          queue.enqueue(pt)
47                          taille += 1
48          Liste += [taille]
49      Liste.sort(reverse= True)
50      print(Liste)
```