Rapport Algorithmique :

Groupe : Benjelloun Hamza
Ait lahmouch Nadir

Le projet d'algorithmique de cette année est tant intéressant par sa simplicité extérieure que par sa complexité sous-jacente.

L'idée en soi de l'algorithme n'est pas tellement difficile à conceptualiser. La difficulté réside dans l'optimisation du temps d'exécution en utilisant différentes techniques d'implémentation étudiées jusqu'ici.

Pour bien cerner le problème et s'assurer de notre compréhension du travail demandé, notre premier objectif était de faire tourner un algorithme fonctionnel ne se souciant pas du temps d'exécution ou de la complexité. Ainsi, et naturellement on s'est retrouvé avec un algorithme en temps quadratique testant toutes les combinaisons possibles entre tous les points, mais donnant quand même le résultat escompté.
S'en est suivi une série d'algorithmes implémentant tous genres d'idées dans l'optique d'optimiser peu à peu notre solution, faisons maintenant un petit point sur l'évolution de notre projet en détaillant un peu plus ces différents algorithmes avec leurs failles et avantages.

Algorithme 1 :

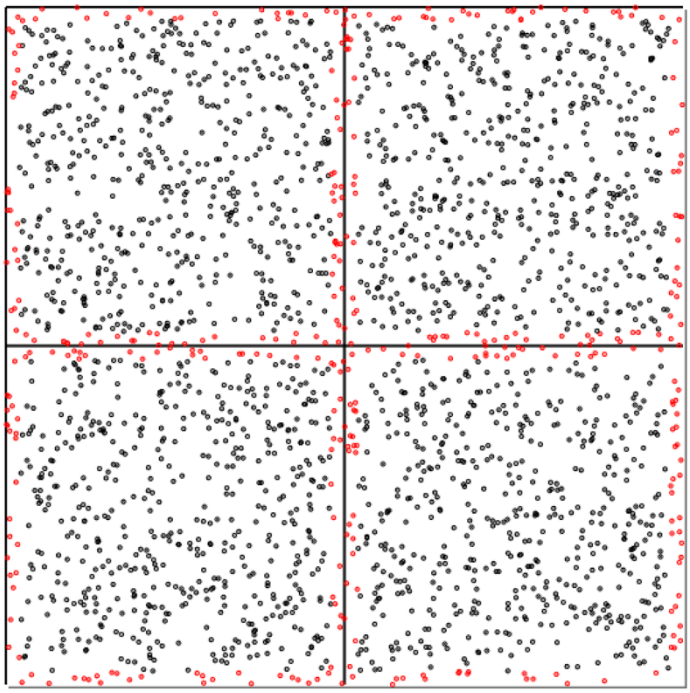
Algorithme naïf comme déjà susmentionné, en temps quadratique, mettant en place une boucle « for » pour tester toutes les combinaisons possibles de tous les points.
Cette algorithme comme le reste utilise une structure d'ensemble, avantageuse par rapport à un dictionnaire classique.
On relie donc chaque point à l'ensemble des points dont la distance avec est inférieure au seuil donné avant de mettre à jour tous les connexes en updatant (mélangeant) un ensemble de points dans l'autre pour en ressortir un connexe.

Algorithme2 :

Le deuxième algorithme, un peu moins naïf, utilise déjà le principe vu en cours « diviser pour régner » pour espérer limiter l'interaction entre les points, chose qui va permettre de réduire de manière significative le temps d'exécution.
Reste maintenant à savoir comment procéder pour arriver à cette fin.
L'idée la plus intuitive est de diviser à chaque appel le quadrant limitant ainsi le nombre de points à traiter à la fois.
Après avoir essayé de le diviser en quatre verticalement puis en trois verticalement, notre choix s'est porté sur une division en quatre classique **(north east, north west, south west, south east)** vu que les autres débouchaient toujours sur l'erreur « maximum recursion depth exceeded ».

Mais il restait toujours un problème à traiter, remonter dans la fusion tout en liant entre les connexes de chaque sous quadrant.
Une première idée est de définir des points critiques dans les marges de chaque sous quadrant dont l'emplacement est dans (les côtés – seuil) pour n'omettre aucune liaison entre deux connexes dans deux différents quadrants.
La fin donc de l'algorithme débouchait sur une boucle for sur l'ensemble des points critiques pour lier les connexes qui doivent être liés.
Deux points importants sont à souligner, nous utilisons à cette étape du projet les classes Point et Quadrant données dans le sujet. Le deuxième point est la condition d'arrêt, celle-ci est choisi dans l'optique de ranger tous les points restants dans le plus petit sous quadrant dans un seul connexe, ainsi la distance maximale que peuvent avoir deux points dans ce quadrant devait être le seuil ce qui revient à dire que le diamètre du quadrant était de seuil soit de côté seuil/racine(2).

Voici un schéma généré par un code python utilisant Svg montrant un exemples de sous quadrants avec chacun leur marge de points critiques.



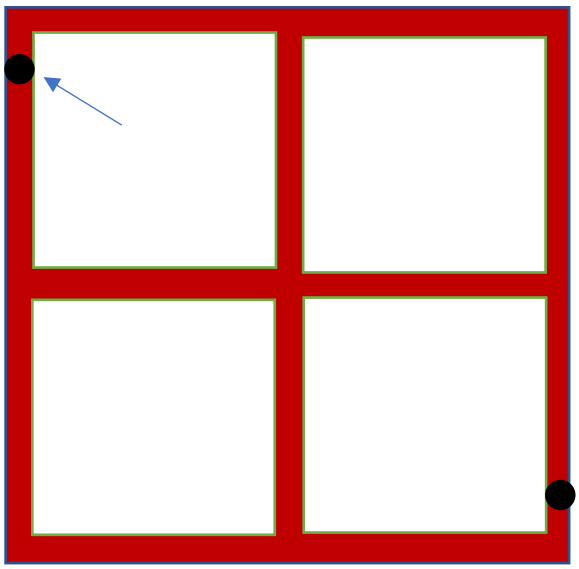
Algorithme 3 :

Dans cet algorithme, l'idée de supprimer totalement les classes utilisées nous est venue en voyant toutes les fonctions que mobilisait l'exécution de notre algorithme, commande ***(python3 cProfile tottime . /connectes.py fichier_points.pts).***
On a remarqué donc que l'appel constant à des classes extérieures, en l'occurrence ici Point et Quadrant consommait beaucoup de temps.

Ainsi dans le load instance, les points sont désormais stockés sous forme de tuples et les quadrants sous de liste de listes. Aussi, la fonction distance entre les points se fait directement à chaque fois et les appels aux fonctions sont minimisés car derniers consommaient aussi beaucoup de temps.

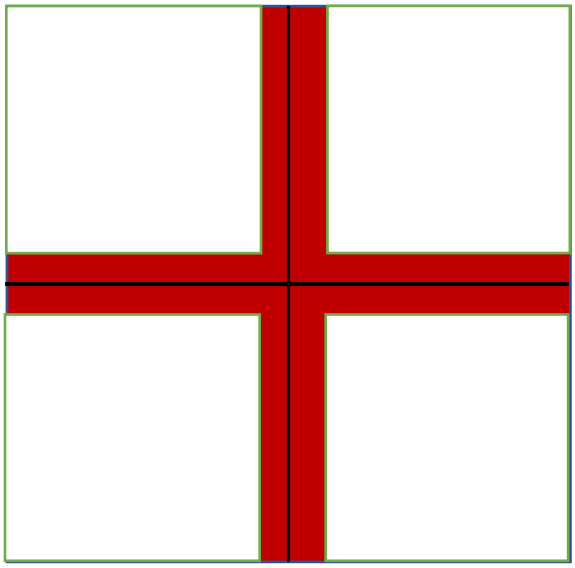
Algorithme 4 :

A cette étape de notre projet, et tenant une piste solide à nos yeux, notre objectif était d'optimiser notre algorithme précédent dans le but de minimiser le temps d'exécution.
Une première optimisation réussie était de changer la condition d'arrêt : on a passé de celle présentée dans l'algo2 à une nouvelle arrêtant l'exécution du programme dès que x_max – x_min était inférieure au seuil.
En effet, cette dernière condition est plus efficace vu notre définition des points critiques qui l'étaient si ils étaient dans la marge du quadrant de taille seuil. Ainsi pas besoin d'aller jusqu'à obtenir une différence d'abscisse de seuil/racine(2), une différence de seuil suffisait.
Une autre optimisation, la plus importante, résidait dans le traitement des points critiques entre eux, en effet avec la disposition des points critiques vue dans les algorithmes précédents des cas inutiles étaient traités dans la boucle for reliant les connexes lors de la fusion, en effet, des points critiques bien très loin entre eux étaient traités et ralentissaient donc l'exécution (voir schéma).
Notre idée était donc de supprimer ces interactions inutiles en définissant de nouveaux points critiques pour chaque sous quadrant suivant sa position dans le quadrant parent (ne, nw, se, sw), une image vaut mille mots …



Ceci étant notre quadrant dans l'algorithme précédent, le traitement entre les points critiques se faisaient entre tous les points étant dans les bandes rouges du quadrant ci-contre, or on a remarqué qu'il était inutile de traiter certains cas, comme les deux points montrés ci-contre car la distance entre eux était largement supérieure au seuil.

Nous sommes passés donc de la disposition ci-dessus à celle-ci-dessous qui rend les choses plus rapides.



Avec cette disposition, on enlève plusieurs cas dans le traitement pour le même résultat.

Voilà la dernière version de cet algorithme donc :

```
#!/usr/bin/env python3
"""
compute sizes of all connected components.
sort and display.
"""
from timeit import timeit
from sys import argv
from collections import defaultdict
from geo.point import Point
from geo.quadrant import Quadrant
from math import sqrt
from itertools import combinations, chain


def load_instance(filename):
    """
    loads .pts file.
    returns distance limit and points.
    """
    with open(filename, "r") as instance_file:
        lines = iter(instance_file)
        distance = float(next(lines))
        points = [tuple(float(f) for f in l.split(",")) for l in lines]
    return distance, points
```

```
def appartient_au_quadrant (point, quadrant):
    """
    verifie si le point appartient au quadrant ou pas.
    """
    x_min,y_min = quadrant[0][0], quadrant[0][1]
    x_max,y_max = quadrant[1][0], quadrant[1][1]
    x, y = point[0], point[1]
    return x_min <= x <= x_max and y_min <= y <= y_max


def critique(point, quadrant, d, n):
    """
    fonction qui définit si un point est critique selon le quadrant
    0 => south west
    1 => south east
    2 => north west
    3 => north east
    """
    x_min,y_min = quadrant[0][0], quadrant[0][1]
    x_max,y_max = quadrant[1][0], quadrant[1][1]
    x, y = point[0], point[1]
    if n == 0:
        return ( y_max - d <= y <= y_max ) or (x_max - d <= x <= x_max)
    elif n == 1:
        return ( y_max - d <= y <= y_max ) or (x_min <= x <= x_min + d)
    elif n == 2:
        return ( y_min <= y <= y_min + d ) or (x_max - d <= x <= x_max)
    else:
        return ( y_min <= y <= y_min + d ) or (x_min <= x <= x_min + d)

def pts_critiques(points, quadrant, d, n):
    """
    Fonction qui définit pour un quadrant l'ensemble de ses points
critiques.
    """
    pts_critiques_2 = set()
    for pt in points:
        if critique(pt, quadrant, d, n):
            pts_critiques_2.add(pt)
    return pts_critiques_2


def fonction_arret(points, quadrant, distance, dict):
    """
    Fonction qui s'execute après la condition d'arrêt de notre algorithme
récursif.
    Cette fonction relie en connexes les points en comparant leurs
distances mutuelles au seuil donné.
    """
    points_critique = set()
    for pt1 in points:
        points_critique.add(pt1)
        for pt2 in points:
            if ((pt1[0]-pt2[0])*(pt1[0]-pt2[0]))+((pt1[1]-pt2[1])*(pt1[1]-
pt2[1])) <= distance*distance:
                dict[pt1].add(pt2)
    return points_critique
```

```
def diviser_pour_regner (points, distance, quadrant, dict, n):
    """
    Algorithme récursif:
    Divise à chaque fois le quadrant en 4 sous quadrants et relance l'algo
sur les sous quadrants
    avec leurs points respectifs.
    Aussi, pour chaque sous quadrant, on définit ses points critiques selon
son type (ne, nw, se, sw)
    et on relie les sous quadrants avec une boucle testant les distances
des points critiques entre eux
    pour relier ou pas les connexes.

    Schéma:

            ----------------------
            |          |          |
            |   nw     |   ne     |
            |          |          |
            ----------------------
            |          |          |
            |    sw    |   se     |
            |          |          |
            ----------------------

            ----------------------------
            |           * | *            |
            |   nw      * | *   ne       |
    ==>     |*********** | **********   |
            ----------------------------
            |*********** | ************ |
            |    sw     * | *    se      |
            |           * | *            |
            ----------------------------

    Les '*' représentent les marges critiques pour chaque quadrant

    """
    x_min, y_min = quadrant[0][0], quadrant[0][1]
    x_max, y_max = quadrant[1][0], quadrant[1][1]
    """
    La condition d'arrêt est la suivante, elle est la plus optimale dans
notre cas.
    Nous en parlons un peu plus sur toutes les cdts d'arrêt testées dans le
rapport.
    """
    if (y_max - y_min) <= distance:
         return fonction_arret(points, quadrant, distance, dict)
    else:
        quadrants = [[[x_min,y_min],[(x_max+x_min)/2 ,(y_max+y_min)/2]],
                     [[(x_max+x_min)/2  ,y_min],[x_max,(y_max+y_min)/2]],
                     [[x_min,(y_max+y_min)/2],[(x_max+x_min)/2 ,y_max]],
                     [[(x_max+x_min)/2,(y_min+y_max)/2],[x_max,y_max]]]
        """
        On définit les points pour chaque quadrant avant d'appeler
diviser_pour_regner.
        """
        pointss = [[] for _ in range(4)]
        for pt in points:
            for indice, quadrant in enumerate(quadrants):
                if appartient_au_quadrant(pt,quadrant):
                    pointss[indice].append(pt)
```

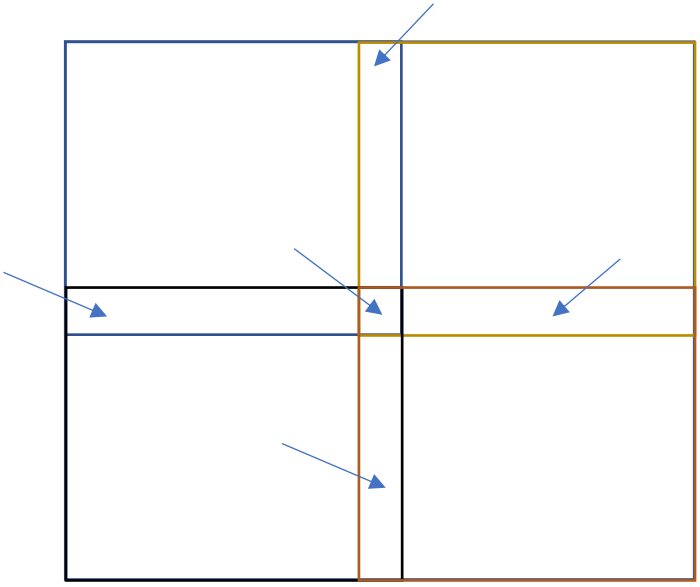
```
        """
        Notre fonction récursive retourne les points critiques du plus
grand quadrant pour fusionner
        les connexes tout en remontant.
        """
        pts_critiques1 =
diviser_pour_regner(pointss[0],distance,quadrants[0], dict, 0)
        pts_critiques2 =
diviser_pour_regner(pointss[1],distance,quadrants[1], dict, 1)
        pts_critiques3 =
diviser_pour_regner(pointss[2],distance,quadrants[2], dict, 2)
        pts_critiques4 =
diviser_pour_regner(pointss[3],distance,quadrants[3], dict, 3)
        """
        Nous effectuons ici le dit tour de boucle sur les points critiques
pour relier les connexes.
        """
        for pt1, pt2 in
combinations(chain(pts_critiques1,pts_critiques2,pts_critiques3,pts_critiqu
es4), 2): # probleme ici
            if ((pt1[0]-pt2[0])*(pt1[0]-pt2[0]))+((pt1[1]-pt2[1])*(pt1[1]-
pt2[1])) <= distance*distance:
                set_pt1, set_pt2 = dict[pt1], dict[pt2]
                if set_pt1 is not set_pt2:
                    if len(set_pt2) > len(set_pt1):
                        set_pt1, set_pt2 = set_pt2, set_pt1
                    set_pt1.update(set_pt2)
                    for pt in set_pt2:
                        dict[pt] = set_pt1
        return  pts_critiques(points, [quadrants[0][0], quadrants[3][1]],
distance, n)


def main():
    """
    ne pas modifier: on charge une instance et on affiche les tailles
    """
    for instance in argv[1:]:
        distance, points = load_instance(instance)
        x_min = 0
        y_min = 0
        x_max = 1
        y_max = 1
        quadrant = [[x_min,y_min],[x_max,y_max]]
        dict = defaultdict(lambda:set())
        pts_critiques = diviser_pour_regner(points, distance, quadrant,
dict, 0)
        composantes = {}
        for point in dict:
            composante = dict[point]
            composantes[id(composante)] = len(composante)
        print(sorted(composantes.values()))
main()
```

NB : Ceci n'est pas notre algorithme définitif.

Algorithme 5 :

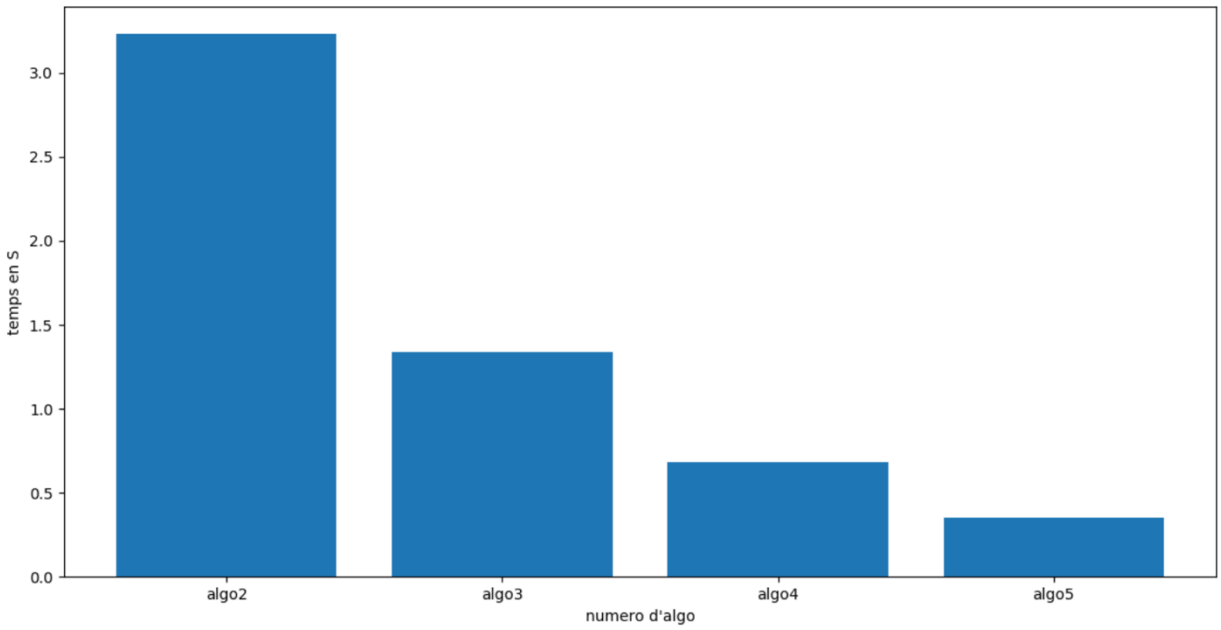
Arrivant à la limite du code utilisant les marges critiques pour relier entre les différents connexes des différents sous quadrants, une autre idée nous est venue :
Établir un chevauchement de Seuil/2 entre les sous quadrants pendant la division pour éviter le traitement des points critiques qui peut s'avérer être long.
Exemple de chevauchement.



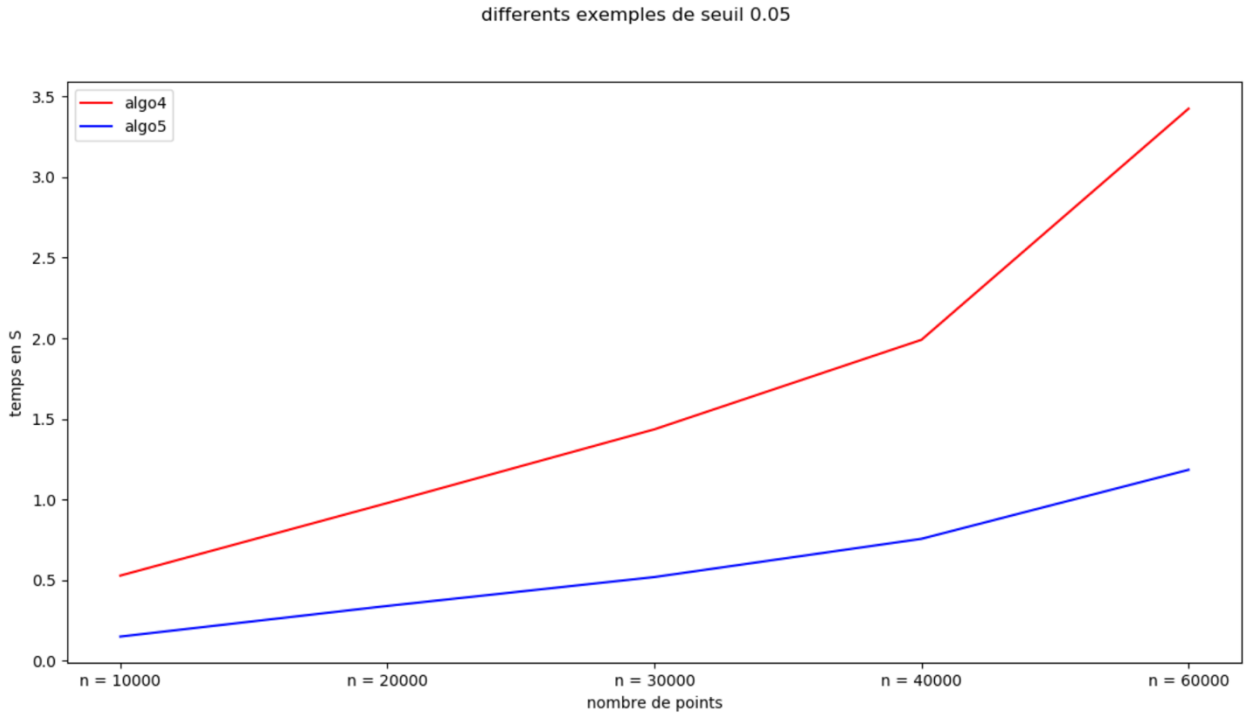
Dans ce cas, toutes les zones pointées par des flèches sont communes par deux ou quatre quadrants. Ainsi comment un point dans ces zones appartient à plus qu'un seul quadrant, la connexion entre les connexes se fait directement.

Passons maintenant à la comparaison entre le temps d'exécution de ces différents algorithmes sur un seul et même fichier de 20000 points.

Graphe de comparaison entre les algorithmes sur un seul fichier de points



Graphe de comparaison d'algorithme 4 et d'algorithme 5 sur différents fichiers de points.



On remarque donc que notre algorithme est grandement plus performant que le précédent.

Synthèse :

En définitif, nous pensons que ce projet a était d'une bien précieuse aide pour développer nos capacités dans la programmation, en nous poussant à se soucier d'avantage de la complexité et du temps d'exécution de nos implémentations. Chose qui va être obligatoire dans le futur lors de manipulation de banques de données bien plus importantes qu'un simple fichier de points.