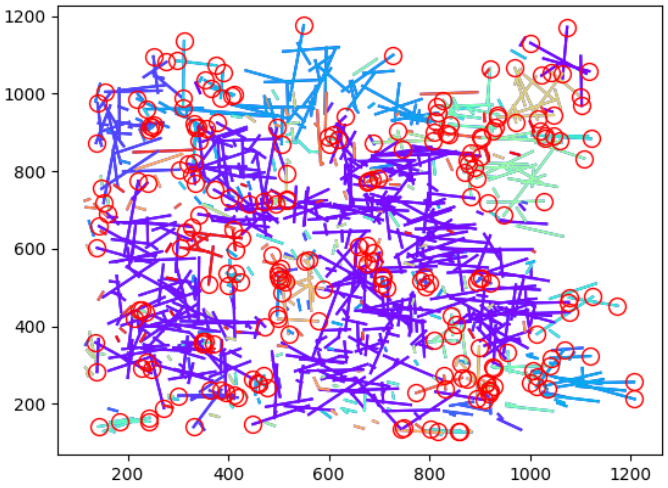
Práctica 3 voluntaria - GCOMP

Celia Rubio Madrigal

17 de mayo de 2022

Índice



1. Introducción

En esta práctica se nos propone encontrar las componentes conexas de un sistema de segmentos en el plano bidimensional. En concreto, los segmentos representan carriles de circulación, y el sistema es el mapa de una ciudad.

Si dos carriles se intersecan, entonces forman parte del mismo subsistema conexo de circulación, y sus transeúntes pueden llegar a todos sus puntos sin salirse de los carriles.

Nuestro objetivo es contar cuántos subsistemas conexos hay, es decir, cuántas componentes conexas tiene el sistema.

2. Material usado y metodología

Contamos con un sistema A de 1000 segmentos aleatorios que nos vienen dados de antemano. Sus coordenadas oscilan entre 110 y 1230. Son, por tanto, un subespacio del plano $\mathbb{R}^2$.

2.1. Apartado *i)*

En el primer apartado, vamos a crear un algoritmo para calcular el número de componentes conexas de N segmentos.

Para ello, vamos a hacer uso de la conocida estructura de datos de conjuntos disjuntos, y del algoritmo Union-Find. Consiste en un conjunto de árboles, es decir, un bosque, y cada árbol se corresponde con un conjunto disjunto distinto. La raíz de cada árbol es el representante canónico del conjunto.

Si tenemos dos elementos que están en el mismo conjunto, se unen con la operación Union. Si queremos saber a qué conjunto pertenece un elemento, la operación Find devuelve la raíz de su árbol.

Si la operación Union se implementa de la forma más ingenua, que es, por ejemplo, haciendo un nodo padre del otro, en el peor caso cada operación tarda un tiempo de $\mathcal{O}(n \log n)$ con n el número de nodos. Si siempre se añade el árbol más pequeño al más grande, se mejora a $\mathcal{O}(\log n)$.

Sin embargo, este tiempo se puede reducir aún más, a $\mathcal{O}(\alpha(n))$, donde $\alpha(n)$ es la inversa de la función de Ackermann, que es, a todos los efectos prácticos, constante —es menor que 5 para todos los valores utilizables de n. Se consigue mediante la compresión del árbol cada vez que se busca un nodo: todos los nodos por el camino se ponen como hijos de la raíz para que siempre estén cerca.

Gracias a este algoritmo, solo basta con encontrar las intersecciones de todos los segmentos, unirlos dentro de esta estructura, y contar cuántos árboles quedan.

Para encontrar las intersecciones de todos los segmentos, podemos utilizar la forma más ingenua, que es comparar todos los pares en tiempo $\mathcal{O}(n^2)$.

También podemos utilizar el conocido algoritmo Sweep-Line. Consiste en escanear todos los puntos en orden de sus abscisas, y comprobar si los puntos iniciales acaban cruzándose con otros segmentos en un orden distinto a los iniciales. Es del orden de $\mathcal{O}(n \log n)$, y se pueden encontrar más detalles en mi trabajo `https://github.com/celrm/geometria-computacional/blob/master/files/main.pdf`.

2.2. Apartado *ii*)

En el segundo apartado representaremos el espacio A y calcularemos con el algoritmo previo cuántas componentes conexas tiene (`n_comp`).

Para conseguir conectar todas las componentes, está claro que harían falta como máximo `n_comp`^{2} segmentos adicionales, que conectarían cada subsistema a pares. Sin embargo, esta cota se puede rebajar teniendo en cuenta las componentes colineales entre sí. De hecho, tres componentes son colineales si y solo si sus envolventes conexas lo son.

3. Resultados y conclusiones

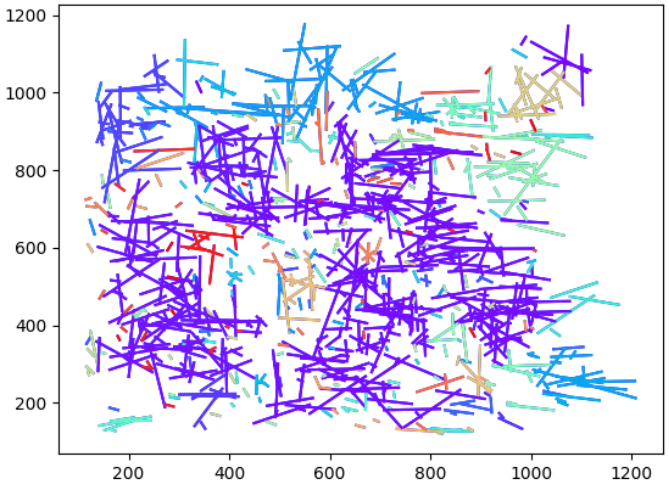
3.1. Apartado *i*)

El algoritmo que calcula las componentes conexas de un sistema de puntos $xy = (X, Y)$ se encuentra programado en la función `connected_components`. Devuelve el número de componentes, el árbol Union-Find que relaciona cada índice de segmento con su índice representante, y la lista de segmentos.

Se ha implementado la manera cuadrática de calcular todas las intersecciones, pero también podría implementarse la forma más eficiente, en la función `find_intersections`.

3.2. Apartado *ii*)

A continuación, se muestra el espacio A de segmentos sobre el plano. Están coloreados en función de la componente conexa a la que pertenecen (aunque los colores no son únicos).



El número de componentes conexas que devuelve el algoritmo es 338.

Con este algoritmo podremos, de manera general, calcular el número de componentes conexas de cualquier sistema de segmentos en el plano arbitrario que nos propongan, y en un tiempo muy eficiente.

4. Código

```
# -*- coding: utf-8 -*-
"""
Plantilla

"""
import random
import numpy as np
import matplotlib.pyplot as plt
from shapely.geometry import LineString, Point
from scipy.spatial import ConvexHull
from matplotlib import cm

# A class to represent a disjoint set
class DisjointSet:
    parent = {}
    rank = {}

    def makeSet(self, universe):
        for i in universe:
            self.parent[i] = i
            self.rank[i] = 0

    def Find(self, k):
        if self.parent[k] != k:
            self.parent[k] = self.Find(self.parent[k])
        return self.parent[k]

    def Union(self, a, b):
        x = self.Find(a)
        y = self.Find(b)

        if x == y:
            return
        if self.rank[x] > self.rank[y]:
            self.parent[y] = x
        elif self.rank[x] < self.rank[y]:
            self.parent[x] = y
        else:
            self.parent[x] = y
            self.rank[y] = self.rank[y] + 1


def find_intersections(segments):
    """
    Finds the intersections between segments.
    """
    intersections = []
```

```
    for i in range(len(segments)):
        for j in range(i + 1, len(segments)):
            s1 = segments[i]
            s2 = segments[j]
            if s1.intersects(s2):
                intersections.append((i, j))
    return intersections


def connected_components(xy=None, segments=None):
    """
    Finds the connected components of a system.
    """
    if xy is None and segments is None:
        return ()
    if segments is None:
        segments = [ LineString(
                [Point(xy[0][s][0], xy[1][s][0]), Point(xy[0][s][1], xy[1][s][1])
                    ]
            ) for s in range(len(xy[0])) ]

    ds = DisjointSet()
    ds.makeSet(range(len(segments)))
    all_intersections = find_intersections(segments)
    for i, j in all_intersections:
        ds.Union(i, j)
    return len(ds.parent), ds, segments


# ################################## PARTE 1
    ########################################

# Generamos 1000 segmentos aleatorios, pero siempre serán los mismos

# Usaremos primero el concepto de coordenadas
X = []
Y = []

# Fijamos el modo aleatorio con una versión prefijada. NO MODIFICAR!!
random.seed(a=1, version=2)

# Generamos subconjuntos cuadrados del plano R2 para determinar los rangos de X
    e Y
xrango1 = random.sample(range(100, 1000), 200)
xrango2 = list(np.add(xrango1, random.sample(range(10, 230), 200)))
yrango1 = random.sample(range(100, 950), 200)
yrango2 = list(np.add(yrango1, random.sample(range(10, 275), 200)))

for j in range(len(xrango1)):
```

```
    for i in range(5):
        random.seed(a=i, version=2)
        xrandomlist = random.sample(range(xrango1[j], xrango2[j]), 4)
        yrandomlist = random.sample(range(yrango1[j], yrango2[j]), 4)
        X.append(xrandomlist[0:2])
        Y.append(yrandomlist[2:4])

# Representamos el Espacio topológico representado por los 1000 segmentos
for i in range(len(X)):
    plt.plot(X[i], Y[i], "b")
plt.savefig("1")

n_comp, ds, segments = connected_components(xy=(X, Y))

# Coloreamos los segmentos conexos del mismo color
colours = cm.rainbow(np.linspace(0, 1, len(X)))
for i in range(len(X)):
    plt.plot(*segments[i].xy, color=colours[ds.Find(i)])
plt.savefig("2")

points_hull = {i: [] for i in ds.parent}
for i in range(len(X)):
    px, py = segments[i].xy
    points_hull[ds.Find(i)] += [[px[0], py[0]], [px[1], py[1]]]

for i in ds.parent:
    points = np.array(points_hull[i])
    if len(points) < 3:
        continue
    hull = ConvexHull(points)
    plt.plot(
        points[hull.vertices, 0],
        points[hull.vertices, 1],
        "o",
        mec="r",
        color="none",
        lw=1,
        markersize=10,
    )

plt.savefig("3")
```