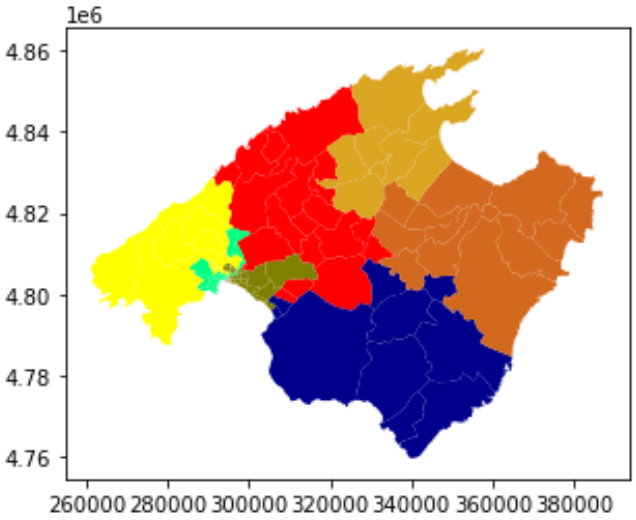
APLICANDO ALGORITMOS EVOLUTIVOS PARA EL DISEÑO DE ZONAS

Practica final Inteligencia Computacional



1 DE MAYO DE 2022

XAVIER FEMENIAS SEGUI
Universitat de les illes balears

Índice.

1.Introducción.

En este trabajo se presenta un algoritmo evolutivo para dividir cualquier conjunto de poblaciones o unidades administrativas en distritos, se pide como condición que estos distritos deben ser contiguos, es decir con todas las poblaciones adheridas a un mismo conglomerado. Además se debe procurar que entre ellos tengan el mismo número de habitantes y que sean lo más compactos posible.

Para ello se va a utilizar un algoritmo evolutivo el cual se basa en asociar una puntuación a cada solución e ir combinando las mejores de una población generación tras generación para así optimizar la búsqueda y llegar a una solución óptima mas rápido que con algoritmos de fuerza bruta. Se va a aplicar concretamente al caso de Mallorca

2.Tipos de datos utilizados.

UnidadArea: O el conjunto de poblaciones y barrios de palma de Mallorca. En esta clase vamos a guardar los habitantes, los datos geométricos para su representación, su id, un booleano *esCentro* que nos dirá si esa población es el centro del distrito o no y su centro geométrico.

Distrito: Las zonas en las que dividiremos el territorio. Aquí guardaremos un array de *UnidadArea,* su id, el numero de habitantes y la población que está en el centro.

Candidate: O las posibles soluciones, contendrá un array de distritos, la puntuación asociada a esta: *fitness* de aquí en adelante, la media de los habitantes por distrito así como métodos propios para representar la división *plot(),* o para calcular los atributos anteriores.

Evolutive: Conjunto de funciones de combinación, generación aleatoria de candidatos y otras que nos van a ser útiles.

3.Flujo del algoritmo.

3.1. Carga de datos e inicialización

Empezamos cargando los datos de dos ficheros, PAD2020.csv, que contiene los habitantes de cada unidad administrativa, y un fichero geoJSON que contendrá los datos geométricos para poder dibujar las poblaciones. Estos ficheros han sido cortesía del compañero de clase LLuis Bernat. Ambos tienen las poblaciones en el mismo orden con lo que con un recorrido podemos ir cogiendo los datos de uno y de otro usando el mismo índice. A continuación creamos un objeto GeoSeries de la librería GeoPandas con los datos obtenidos del geoJSON lo cual nos va a permitir realizar las operaciones de cálculo de distancias y de dibujado. Conjuntamente con los datos de los habitantes y los nombres de las poblaciones montamos una lista de objetos unidadArea.

Es hora de crear nuestro objeto Evolutive que es el que contendrá el modelo diseñado, le deberemos pasar la lista anteriormente citada, junto con los parámetros *population_size,* que establecerá el número máximo de individuos que deben pasar a la siguiente generación descartando los que tienen un fitness menor, *child_generation_rate* que será la cantidad de hijos que se van a crear en cada generación, *max_generations* para establecer cuantas generaciones queremos y *distritos* para indicar en cuantas zonas queremos dividir el territorio.

He observado que cuando se desean más distritos el algoritmo requiere de mas generaciones y/o hijos por generación, siendo suficientes unas 100 generaciones a 3 hijos/generación si se quieren de 2 a 10 distritos lo cual tardaría unos 7-10 minutos con rendimiento mononucleo de 2,4 GHz y a partir de ahí necesitaremos aumentar para que el algoritmo pueda encontrar algo. Probablemente este algoritmo pueda ser optimizado para aumentar la velocidad a la que se generan hijos.

Cuando iniciamos nuestro modelo con la función *.resultado()*, lo primero que se hace es crear una matriz de adyacencia de las poblaciones usando las geometrías y la función *.touches()* de geopandas, que básicamente mira si dos geometrías se tocan en la periferia, y posteriormente guardarla en una variable global de clase que nos servirá para en cualquier momento saber si dos poblaciones son contiguas sin tener que recalcular esta matriz. Luego a partir de esta matriz también crearemos un grafo usando la función *adjacency()* de la librería iGraph el cual nos va a ayudar a solucionar el problema de la continuidad. Este grafo también es almacenado en una variable para no tener que recalcularlo cada vez.

Inicializamos una población aleatoria de un tamaño *population_size*, típicamente se han testeado tamaños de 10 o 15 individuos. La función encargada de esto es *candidatoAleatorio().*

Esta función, primero asigna aleatoriamente una población a cada distrito y después con la subfunción *listaPueblos_a_distritos()* a partir de una lista de poblaciones no asignadas selecciona aleatoriamente un distrito y le asigna la primera población de la lista de poblaciones no asignadas hasta que la lista de poblaciones no asignadas se vacíe. Así nos aseguramos que todos los distritos de todos los candidatos aleatorios sean contiguos.

3.3. Fitness de los candidatos.

A medida que se van creando los candidatos aleatorios, es necesario calcular su fitness y guardarlo dentro del mismo objeto. El fitness es calculado en función de dos parámetros de calidad: la diferencia de las poblaciones de cada distrito con la media, y lo compactos que sean estos distritos.

Para la diferencia de las poblaciones hemos usado la siguiente formula:

$$\min \sum_{j} |P_j - \mu|$$

Para la medida de la compacidad:

$$\sum_{j} \sum_{i=\in Z_j} d_{ij}$$

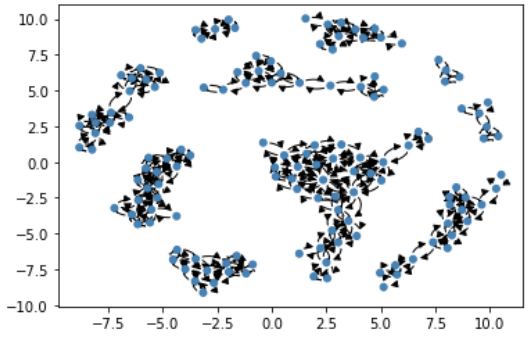
Si simplemente sumamos las dos expresiones obtendremos una puntuación que nos permitirá medir la calidad de los candidatos, donde un candidato con menor puntuación es mejor. El fitness óptimo va a ser diferente en función de los distritos que le pidamos al algoritmo, por ejemplo, para el caso de Mallorca y 3 distritos un fitness de 1.450.000 puede ser considerado bueno, mientras que si le pedimos 8 distritos el óptimo baja hasta unos 1.150.000.

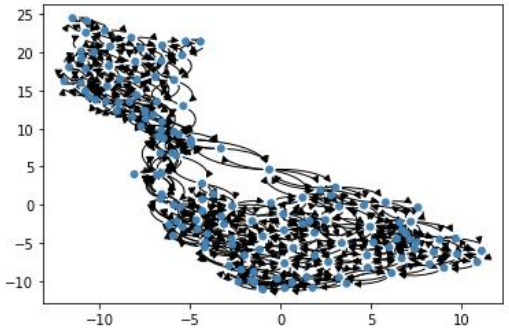
Solo queda por resolver el tema de la contigüidad de los distritos, que aunque hayamos inicializado la población contigüa, durante el proceso de combinación pueden aparecer “islotes” o distritos separados. Para ello voy a hacer uso del grafo que he creado en la inicialización.

En la variable global *grafo* tenemos un grafo donde cada vértice es un pueblo, y si dos pueblos son contiguos entonces tendrán un enlace entre ellos. Cada vez que se crea un candidato nuevo debemos coger este grafo y hacer un recorrido entre todos sus enlaces. Para cada enlace miramos las id’s de los vértices que

conecta, y si no pertenecen al mismo distrito, eliminamos el enlace. He escrito una función: *perteneAlDistrito(id,candidato)* que a partir de la id de una población nos busca la id del distrito que pertenece.

Ahora que tenemos el grafo cortado en distritos, si simplemente comprobamos que todos los vértices tienen un camino al centro de su distrito (función *shortest_paths* de iGraph), tendremos un candidato perfectamente contiguo,





mientras que si hay alguno que no, directamente penalizamos mucho y terminamos la comprobación para optimizar el tiempo de ejecución. En las siguientes figuras podemos ver los grafos, el cortado en distritos y el original respectivamente

La penalización suma 1.000.000 al fitness total para asegurar que ningún candidato no contiguo sobreviva hasta la última generación.

La función *shortest_paths* de iGraph devuelve un entero: el numero de nodos que hay entre dos vértices, mientras que si no existe ningún camino devuelve infinito. Python interpreta el número infinito como un float, por lo que para nuestro propósito nos es suficiente con hacer la siguiente verificación:

```
return type(grafo.shortest_paths(id1,id2)[0][0]) == int
```

3.4. La selección, combinación y mutación de candidatos.

Una vez tenemos nuestra población creada, es hora de empezar a combinar individuos. Primeramente deberemos seleccionar dos candidatos de nuestra población para ello recurrimos al método de la ruleta explicado en clase. Consiste

simplemente en que cada candidato tiene un probabilidad de ser seleccionado para ser padre igual a: *fitnessCandidato/fitnessTotal*. Como en nuestro caso un fitness menor es mejor, he calculado la inversa de cada candidato y el fitness total ha sido la suma de todas las inversas. Para la implementación de la ruleta he usado la función *numpy.random.choice,* que toma una lista de candidatos y una lista de igual longitud con las probabilidades de sus candidatos, después extraemos el seleccionado de la población y repetimos, para asegurar que no combinamos el mismo individuo lo cual no tendría sentido.

La combinación de candidatos se realiza tomando aleatoriamente la mitad de distritos de un progenitor y la mitad del otro, si son pares; o la mitad + 0,5 y la mitad -0.5 si son impares. Lógicamente cuando intentemos encajar el resultado nos van a quedar poblaciones "solapadas" es decir que estaban tanto en los cromosomas del padre como en los de la madre. Para arreglar esto primero ponemos en una lista todas las poblaciones que se han solapado:

```
listaSolapamientos =
list(set(chain(*[deepcopy(cromosoma.poblaciones) for cromosoma in cromosomasPadre])) &
set(chain(*[deepcopy(cromosoma.poblaciones) for cromosoma in cromosomasMadre])))
```

Luego realizamos un recorrido por todos los distritos y poblaciones de uno de los dos y borramos dichas poblaciones.

Todavía nos queda un problema por resolver, y es que nos pueden quedar huecos o poblaciones no asignadas a ningún distrito, o incluso distritos vacíos.

Esto lo hemos arreglado con la función *rellenarHuecos().* Para el caso de los distritos vacíos le asigna una población aleatoria de la lista de poblaciones no asignadas. Para estos casos también he pensado que lo mejor probablemente sea descartar el candidato asignándole un fitness alto para así no sesgar tanto el algoritmo.

A continuación recorremos la lista de poblaciones no asignadas y miramos a que distrito pertenecen las poblaciones colindantes, se le asignará el distrito que menos tenga como vecino hasta que no queden poblaciones no asignadas en la lista.

Finalmente nos queda la mutación que consiste en modificar un gen del candidato, con una probabilidad igual a *mutation_rate*. Esta parte nos a ayudar para cuando en la población tengamos individuos muy parecidos, en estos casos

la combinación siempre da lugar a un individuo muy parecido, entonces la mutación va a ser nuestra última carta para mejorar el fitness. En nuestro caso se selecciona un distrito aleatorio del candidato, se le extrae una población, y se introduce en otro distrito aleatorio diferente del primero.

Al final de la generación se descartan los candidatos con peor fitness dejando una población igual a *population_size.* Para ello ordenamos la lista de población de candidatos de menor a mayor según su fitness y la cortamos:

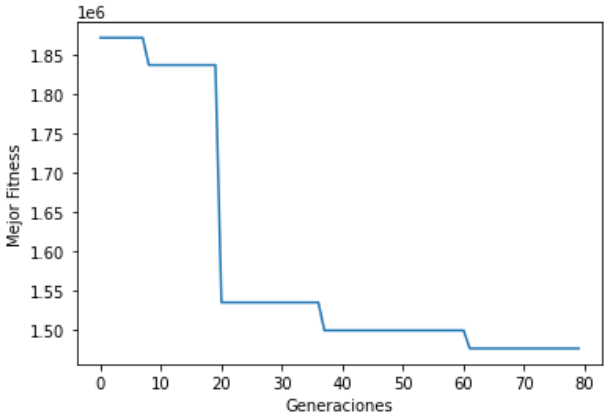
```
return population[:self.population_size]
```

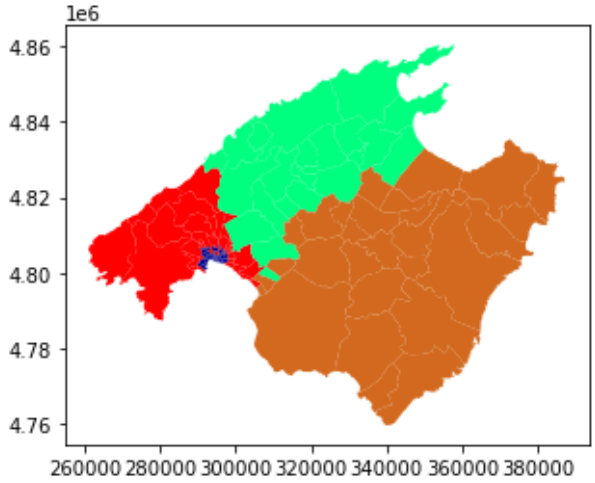
Al acabar la generación se exportan los resultados a distritos.txt.

4. Resultados.

A continuación adjuntamos el resultado y la gráfica del fitness para un par de casos.

- Para 4 distritos:





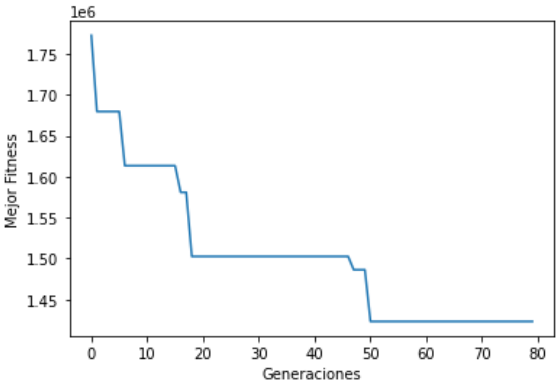
El distrito 0 (red) tiene: 251695 habitantes.

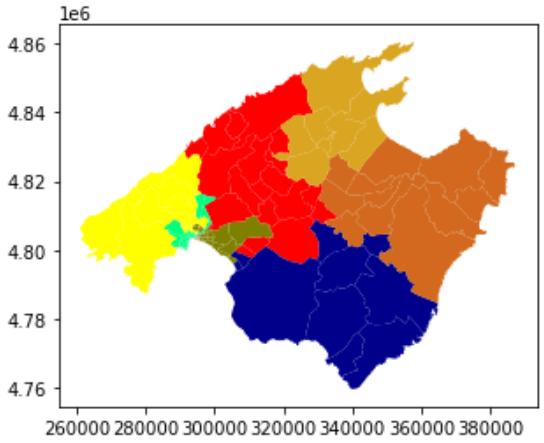
El distrito 1 (darkblue) tiene: 226201 habitantes.

El distrito 2 (springgreen) tiene: 197768 habitantes.

El distrito 3 (chocolate) tiene: 236507 habitantes.

- Para 8 distritos.





El distrito 0 (red) tiene: 121304 habitantes.

El distrito 1 (darkblue) tiene: 107135 habitantes.

El distrito 2 (springgreen) tiene: 128450 habitantes.

El distrito 3 (chocolate) tiene: 112587 habitantes.

El distrito 4 (goldenrod) tiene: 100129 habitantes.

El distrito 5 (yellow) tiene: 143251 habitantes.

El distrito 6 (olive) tiene: 275212 habitantes.

El distrito 7 (darksalmon) tiene: 33580 habitantes.

Nótese que se han usado 80 generaciones, si se utilizan mas generaciones el resultado sería aun mejor.