APLICANDO ALGORITMOS EVOLUTIVOS EN EL DISEÑO DE ZONAS

Practica final Inteligencia Computacional

7 DE JULIO DE 2022

XAVIER FEMENIAS SEGUI
Universitat de les Illes Balears

Índice.

1. Introducción.

En este escrito se presenta un algoritmo que nos va a permitir dividir cualquier conjunto de poblaciones o unidades administrativas en distritos bajo unas determinadas condiciones. Se pide como condición indispensable 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 un número de habitantes los más parecido posible y por último que sean lo más compactos posible. El objetivo de todo esto es evitar lo que se conoce como **gerrymandering,** o un amaño por parte de los presidentes de los distritos para hacerse con poblaciones y que esto genere un desequilibrio entre distritos.

Para solucionar esto se ha elaborado un algoritmo evolutivo, el cual consiste en ir combinando los candidatos de una población aleatoria inicial, de forma que aquellos que son mejor solución tienen una probabilidad mayor de reproducirse, y solo los mejores candidatos podrán pasar a la siguiente generación. Para saber si un candidato es mejor que otro necesitamos asociar a cada uno una puntuación, o *fitness* de ahora en adelante. Se comprobará que, de esta manera, las generaciones resultantes van siendo cada vez mejores, que consiguen evolucionar generación tras generación

De esta forma se consigue optimizar la búsqueda dentro del espacio de características y llegar a una solución óptima mucho más 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, un 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 número de habitantes y la población que está en el centro.

Candidate: O las posibles soluciones, contendrá un array de distritos, el *fitness*, así como la *compacidad* o *la suma de las diferencias de los habitantes entre*

distritos 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[1]. 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.

```
1.  datos = read_csv("PAD2020.csv", sep=";")
    datosGeometricos = gpd.read_file("districts_geometry.geojsonl")
    datosGeometricos = datosGeometricos.to_crs(crs='EPSG:3857')
```

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*.

```
1.  listaPueblos = []
2.
3.
4.  for i in range(len(datos)):
5.      #obtenemos los datos vectoriales de la poblacion que tenga el codigo de la fila
    por la que vamos
6.
7.      shape = gpd.GeoSeries(datosGeometricos['geometry'][i])
8.      listaPueblos.append(UnidadArea(i, datos.Habitantes[i], shape.centroid, da-
    tos.Poblacion[i], shape))
9.
```

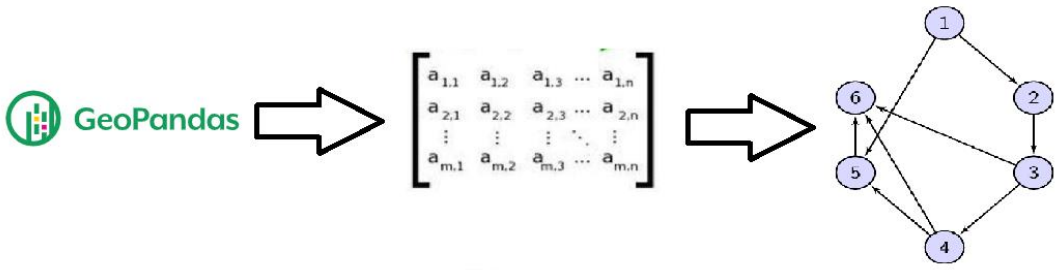
[1] Estos ficheros han sido cortesía del alumno Lluís Bernat

A continuación instanciamos un 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.

```
1.  modelo = Evolutive(population_size, mutation_rate, max_generations,
2.                     child_generation, distritos, listaPueblos =
    deepcopy(listaPueblos))
3.
```

He observado que cuando se desean más distritos el algoritmo requiere de más 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 mono núcleo 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 se guarda 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 6 a 10 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 que sea contigua a alguna de las poblaciones que ya están asignadas al distrito, y si no hay ninguna se pasa a la siguiente población. Reiniciamos este proceso hasta que la lista de poblaciones no asignadas se vacíe. De esta forma los candidatos iniciales quedaran 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 fórmula:

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

Donde P_j representa la población del distrito j y μ es la media de las poblaciones de los distritos.

Para la medida de la compacidad:

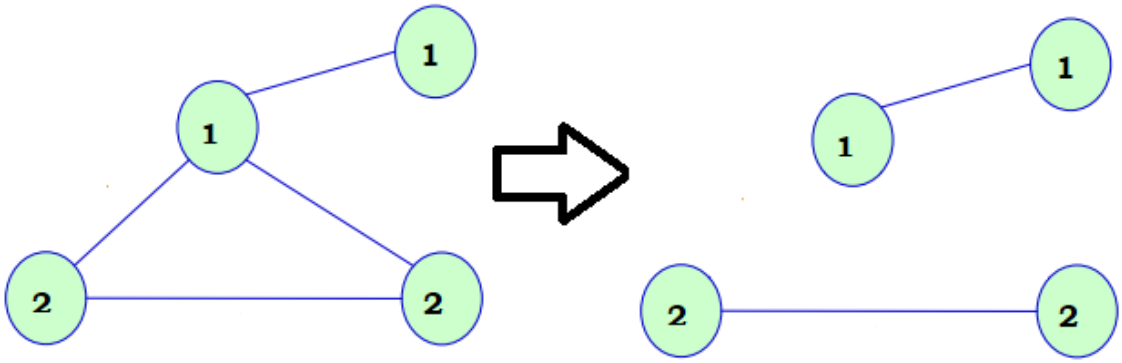
$$\sum_{j} \sum_{i=\in Z_j} d_{ij}$$

Donde d_{ij} es la distancia de la población i al centro de su distrito j, para todas las poblaciones pertenecientes a ese distrito j.

Si simplemente sumamos las dos expresiones obtendremos un fitness que nos permitirá medir la calidad de los candidatos, donde un candidato con menor fitness 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 contigua, durante el proceso de combinación pueden aparecer “islotes” o poblaciones de un mismo distrito separados de este. Para ello voy a hacer uso del grafo que he creado en la inicialización.

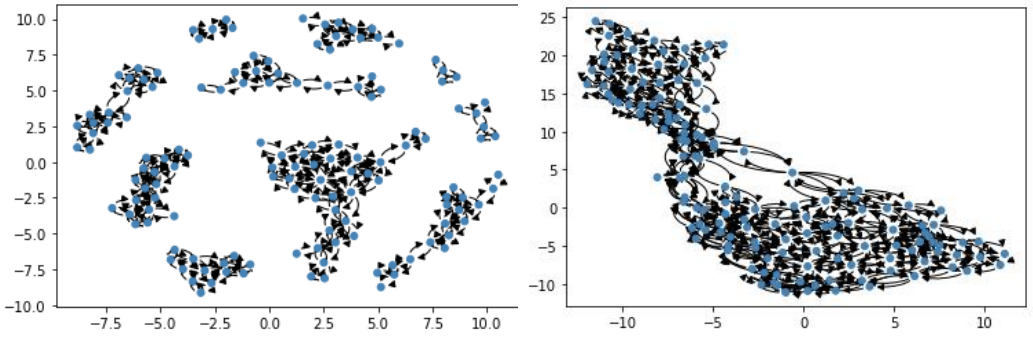
En la variable global *grafo* tenemos un grafo donde cada vértice es un pueblo, y si dos pueblos son contiguos entonces tendrán un enlace entre ellos. Cada vez que se crea un candidato nuevo debemos coger este grafo y hacer un recorrido entre todos sus enlaces. Para cada enlace miramos las id's de los vértices que conecta, y si no pertenecen al mismo distrito, eliminamos el enlace.



Para facilitar el proceso he escrito una función: *perteneAlDistrito(id,candidato)* que a partir de la id de una población nos busca la id del distrito que pertenece.

```
1.  def perteneceAlDistrito(i,candidate):
2.
3.      for distrito in candidate.distritos:
4.          for pueblo in distrito.poblaciones:
5.
6.              if pueblo.id == i:
7.
8.                  return distrito.id
9.
```

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 número 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:

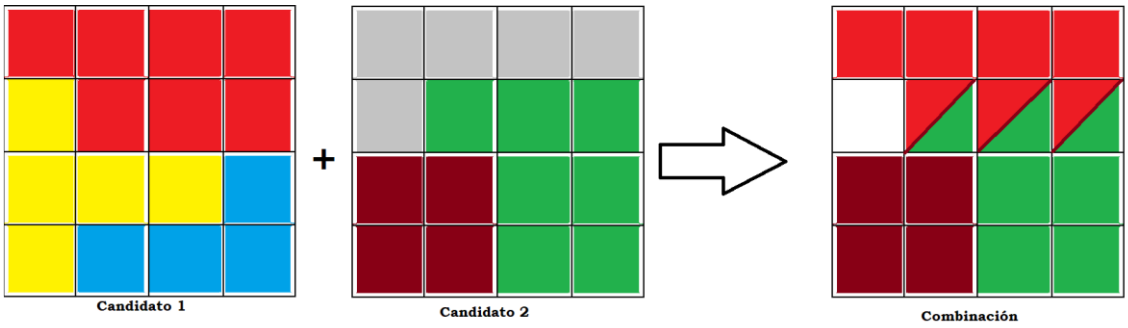
```
1.  return type(grafo.shortest_paths(id1,id2)[0][0]) == int
2.
```

3.4. La selección, combinación y mutación de candidatos.

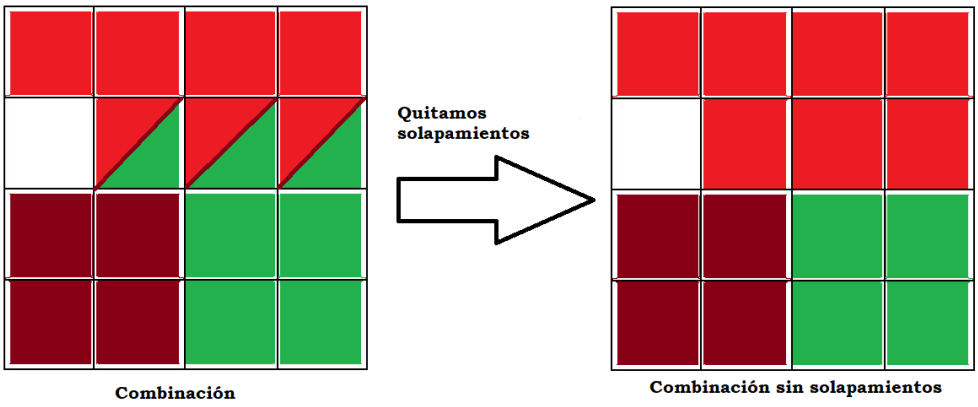
Una vez tenemos nuestra población creada, es hora de empezar a combinar individuos. Primeramente deberemos seleccionar dos candidatos de nuestra población para ello recurrimos al método de la ruleta explicado en clase. Consiste simplemente en que cada candidato tiene un probabilidad de ser seleccionado 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.

En el siguiente esquema se ve un ejemplo muy sencillo teniendo en cuenta 16 poblaciones divididas en 3 distritos. En el candidato 1 se selecciona el distrito rojo y en el candidato 2 el verde y el marrón. Se puede observar como la combinación resultante tiene tres poblaciones que pertenecen tanto al distrito rojo como al verde:



Pues bien eliminamos los solapamientos que pertenecen al candidato 2 (siempre será el 2), es decir el verde:



En el código lo que se ha hecho es primero poner en una lista todas las poblaciones que se han solapado:

Luego realizamos un recorrido por todos los distritos y poblaciones del candidato 2 y borramos dichas poblaciones.

```
1.  listaSolapamientos = list(set(chain(*[deepcopy(cromosoma.poblaciones) for cromosoma
    in cromosomasPadre])) &
2.                            set(chain(*[deepcopy(cromosoma.poblaciones) for cromosoma
    in cromosomasMadre])))
3.
4.
5.
6.
7.          copia = deepcopy(cromosomasPadre) # hacemos una copia asi cuando quitemos
    los solapamientos no nos afectara al recorrido
8.
9.          for i, cromosoma in enumerate(copia):
10.             for j,pueblo in enumerate(cromosoma.poblaciones):
11.
12.                 if pueblo in listaSolapamientos:
13.
14.                     cromosomasPadre[i].poblaciones.remove(pueblo)
15.
```

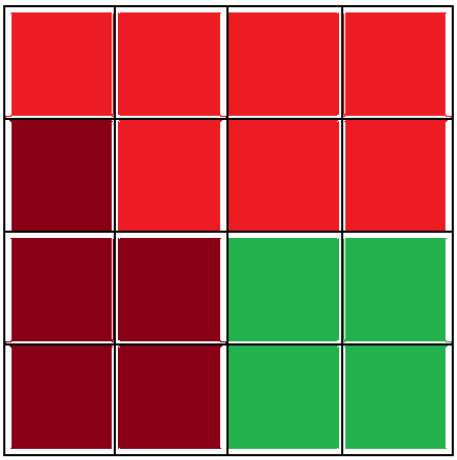
Entonces solo nos queda combinar los distritos del padre y de la madre en el del hijo.

```
1.  adnHijo = cromosomasMadre + cromosomasPadre
2.
```

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.

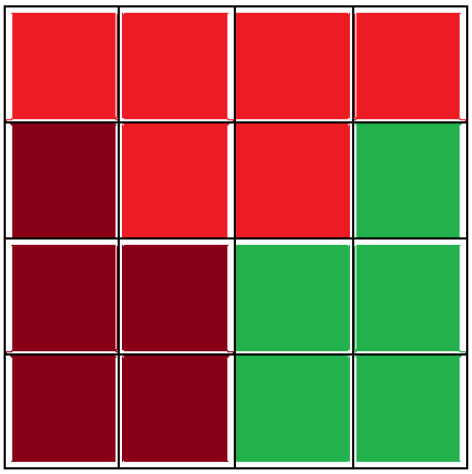
Esto lo hemos arreglado con la función *rellenarHuecos().* Para el caso de los distritos vacíos le asigna una población aleatoria de la lista de poblaciones no asignadas.

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.



Combinación final

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.



Combinación final mutada

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:

```
1.  def seleccionSupervivientes(self, population):
2.
3.          population.sort()#ordenamos de menor a mayor segun su fitness
4.
5.          return population[:self.population_size]
6.
```

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, usando esta configuración:

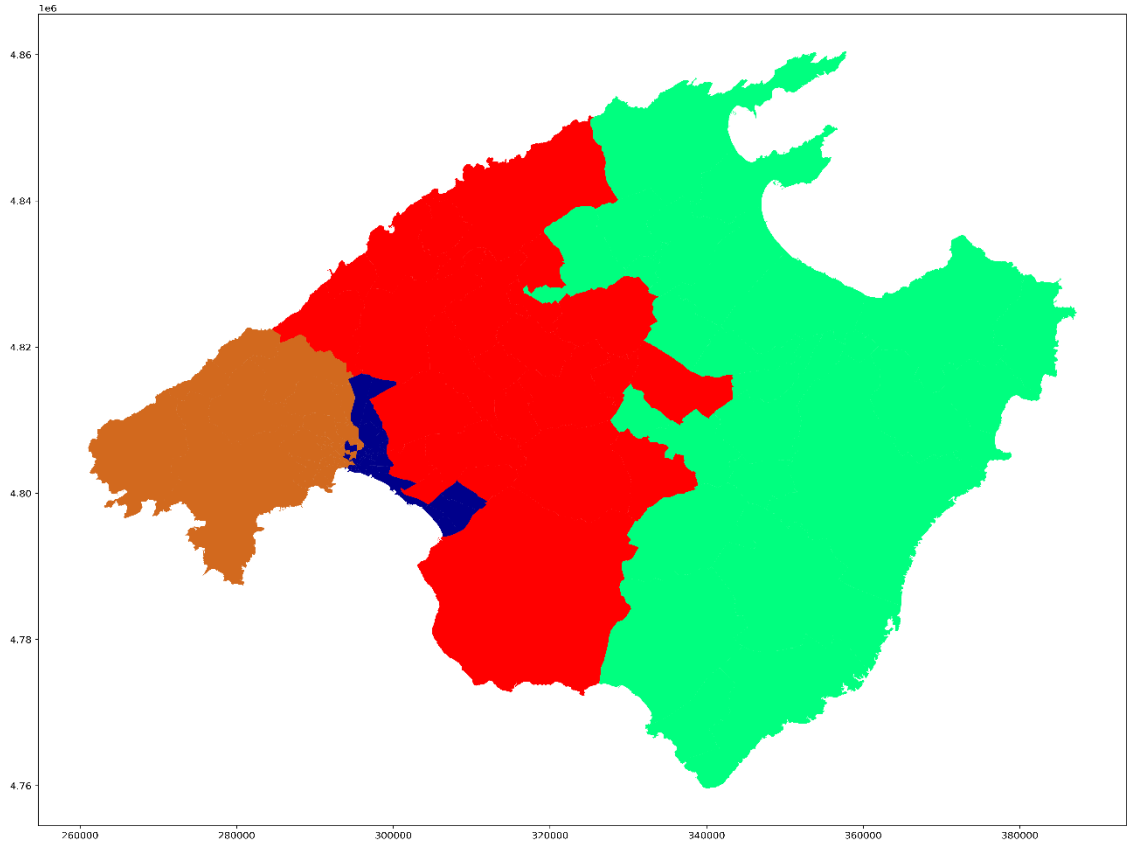
population_size = 10

mutation_rate = 0.1

child_generation = 4

max_generations = 100

distritos= 4



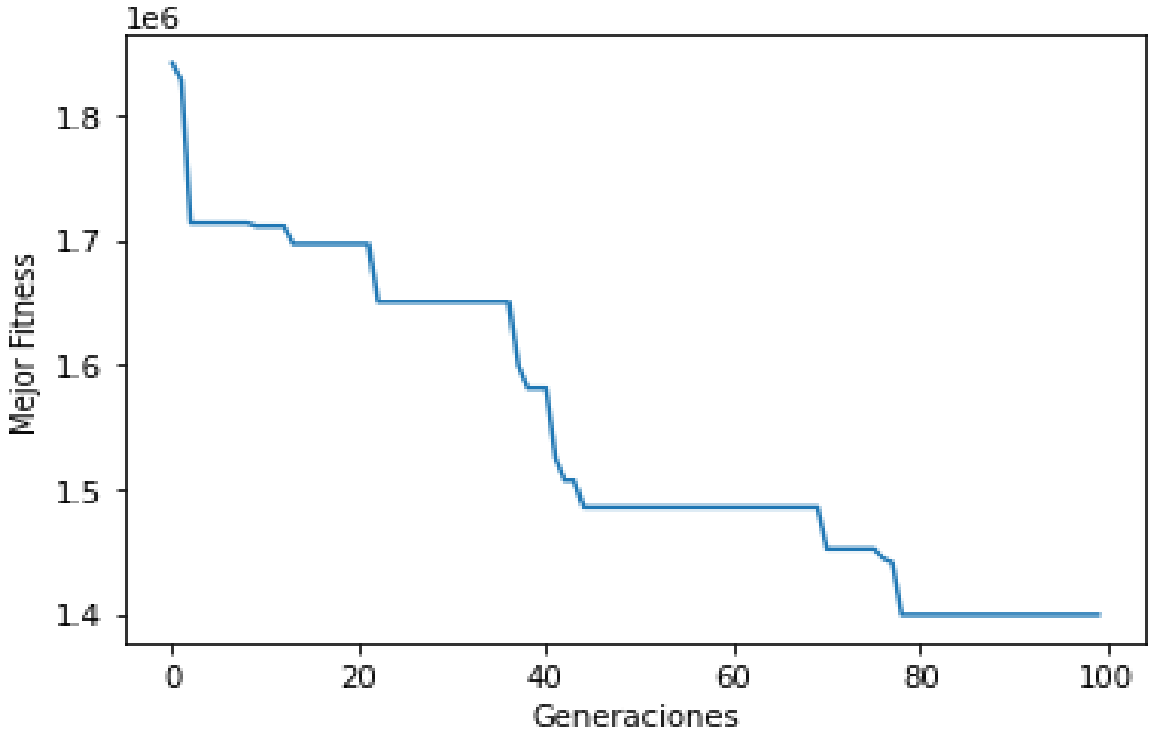
Obtenemos un fitness de 1401526 con una diferencia de habitantes de 16736 y una compacidad 1384790

El distrito 0 (red) tiene: 201056 habitantes.

El distrito 1 (darkblue) tiene: 252309 habitantes.

El distrito 2 (springgreen) tiene: 237139 habitantes.

El distrito 3 (chocolate) tiene: 226366 habitantes.


1e6
1.8
1.7
1.6
1.5
1.4
Mejor Fitness
0
20
40
60
80
100
Generaciones

- Para 8 distritos reduciremos el population_size por que ahora los mismos padres pueden hacer combinacionalmente mas hijos, y asi los individuos con mas fitness tendrán más probabilidades de ser seleccionados.

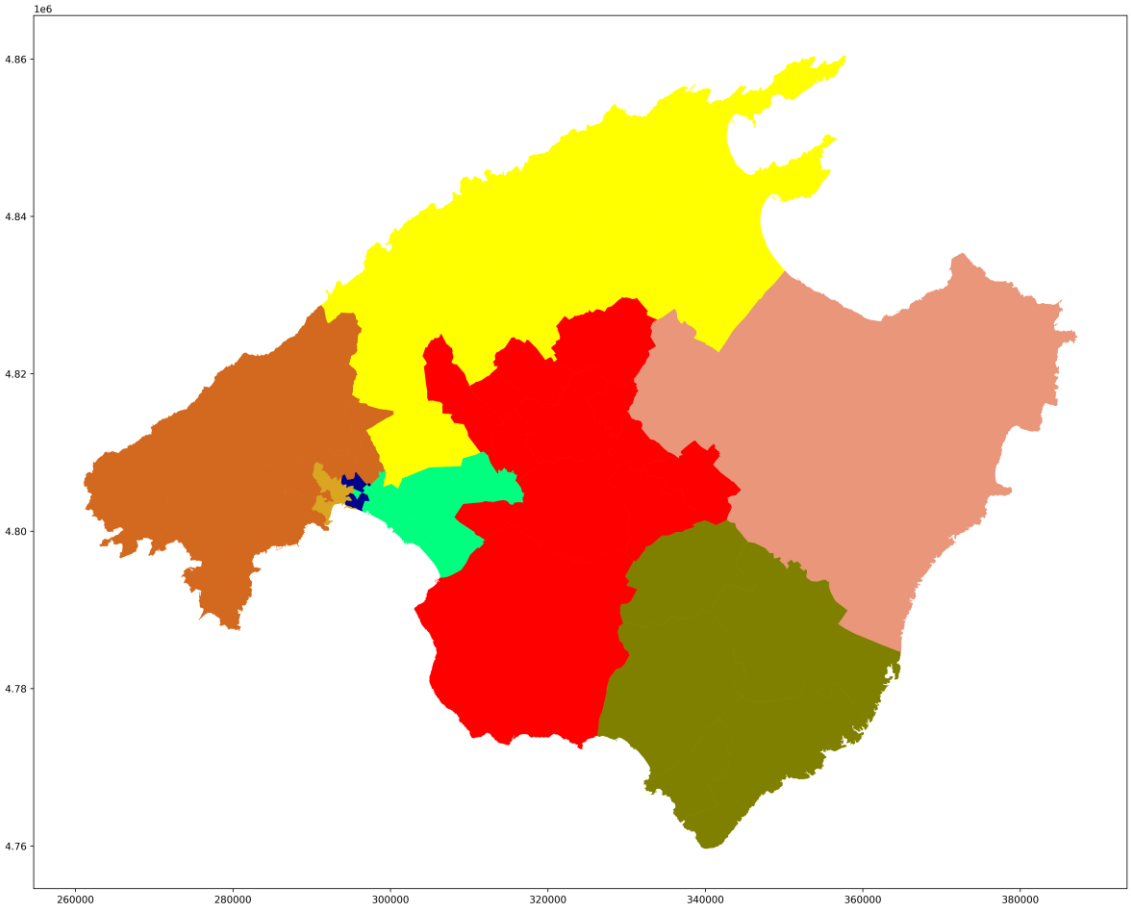
population_size = 5

mutation_rate = 0.1

child_generation = 4

max_generations = 200

distritos= 8



Se obtiene un fitness de 1038721 con una diferencia de poblaciones de 130877 y una compacidad de 907844

No se penaliza la población verde que está dentro de la azul, porque realmente es contigua, dado que toca la componente principal y por tanto tiene un camino al centro.

El distrito 0 (red) tiene: 113715 habitantes.

El distrito 1 (darkblue) tiene: 110214 habitantes.

El distrito 2 (springgreen) tiene: 135653 habitantes.

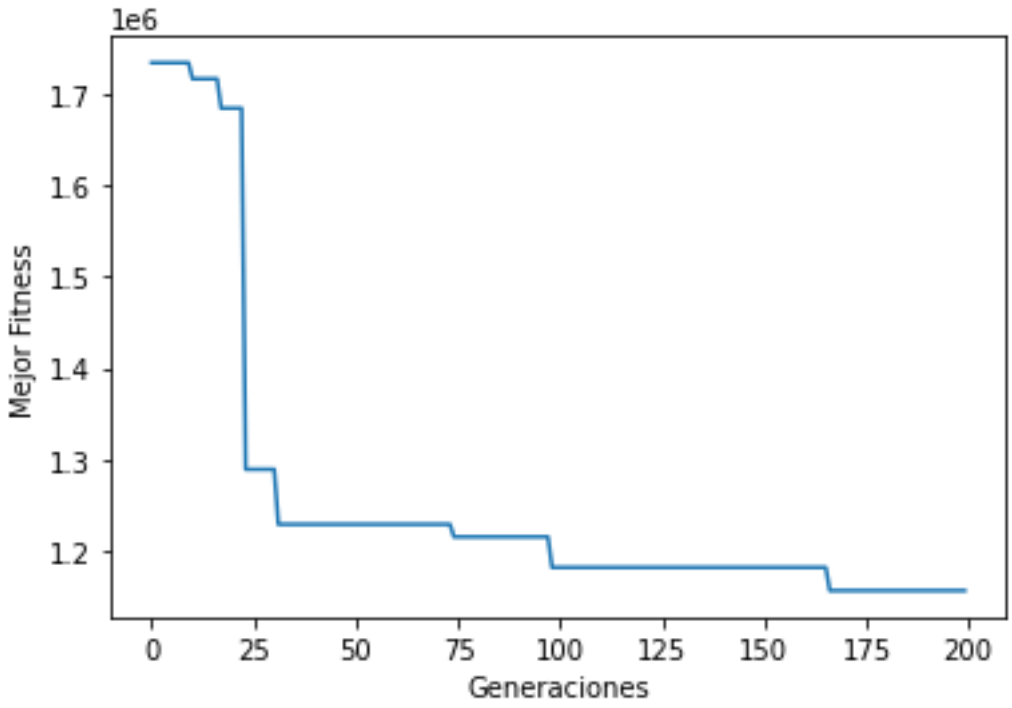
El distrito 3 (chocolate) tiene: 124127 habitantes.

El distrito 4 (goldenrod) tiene: 122537 habitantes.

El distrito 5 (yellow) tiene: 139487 habitantes.

El distrito 6 (olive) tiene: 52530 habitantes.

El distrito 7 (darksalmon) tiene: 113908 habitantes.



- Con 12 distritos:

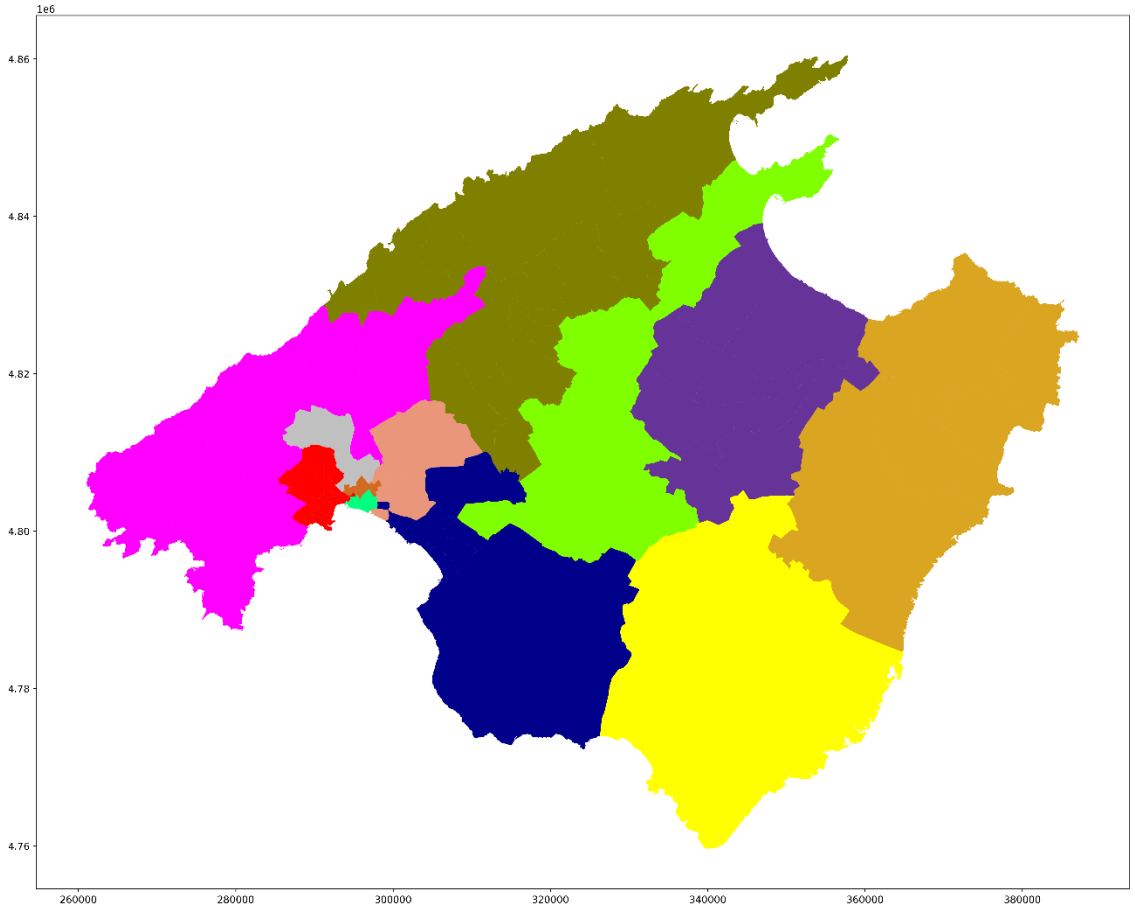
  population_size = 8

  mutation_rate = 0.1

  max_generations = 1000

  child_generation = 3

  distritos = 12



Se ha obtenido un fitness de 1004149 con una compacidad de 838937, y una suma de las diferencias de los habitantes del distrito con la media de habitantes de los distritos de 165212. Además, la distribución de poblaciones ha quedado de la siguiente manera:

El distrito 0 (red) tiene: 103628 habitantes.

El distrito 1 (darkblue) tiene: 71191 habitantes.

El distrito 2 (springgreen) tiene: 86816 habitantes.

El distrito 3 (chocolate) tiene: 98401 habitantes.

El distrito 4 (goldenrod) tiene: 85228 habitantes.

El distrito 5 (yellow) tiene: 101961 habitantes.

El distrito 6 (olive) tiene: 74899 habitantes.

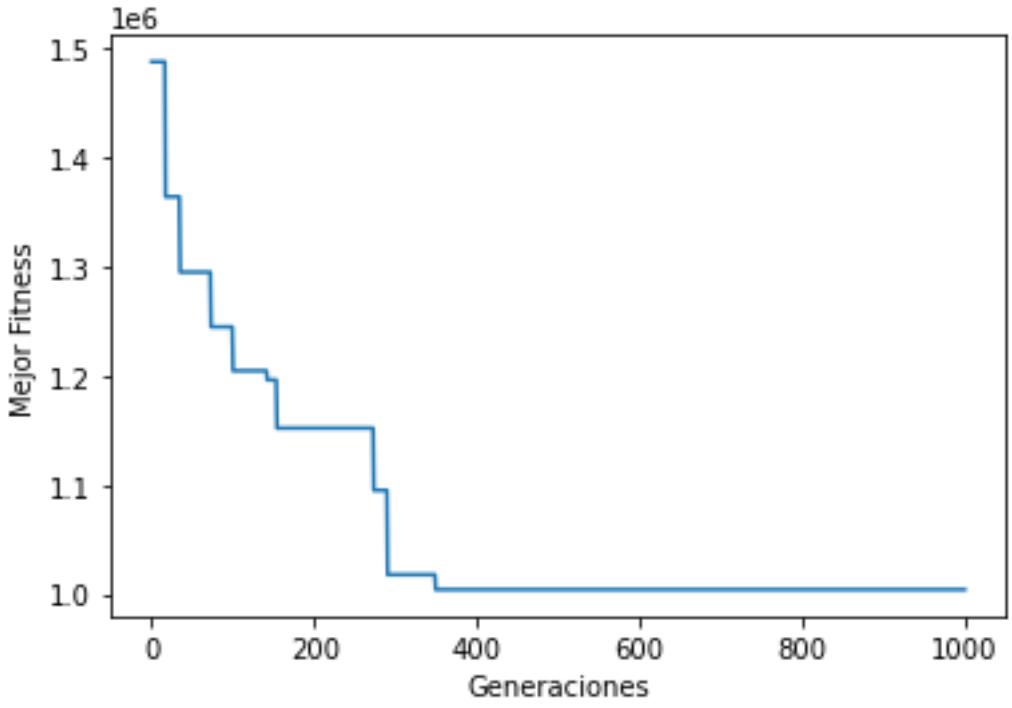
El distrito 7 (darksalmon) tiene: 82191 habitantes.

El distrito 8 (rebeccapurple) tiene: 34753 habitantes.

El distrito 9 (magenta) tiene: 84462 habitantes.

El distrito 10 (silver) tiene: 120737 habitantes.

El distrito 11 (chartreuse) tiene: 86393 habitantes.



Vamos con un experimento con 15 zonas, esta vez necesitaremos más generaciones:

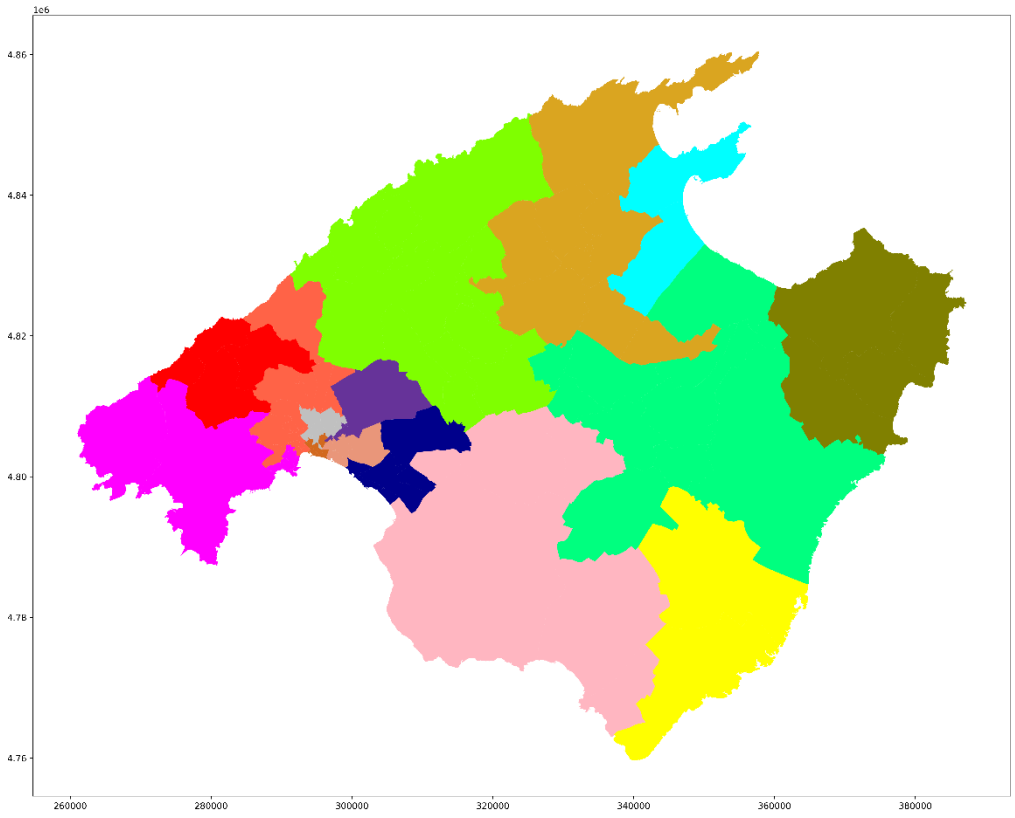
population_size = 5

mutation_rate = 0.1

child_generation = 4

max_generations = 1000

distritos= 15



Se ha obtenido un fitness de 965471 con una compacidad de 818542, y una suma de las diferencias de los habitantes del distrito con la media de habitantes de los distritos de 146929. Además, la distribución de poblaciones ha quedado de la siguiente manera:

Nótese que hay dos colores parecidos, pero son diferentes, uno es rosa y el otro beige. La distribución de la población ha quedado bastante uniforme:

El distrito 0 tiene: 71150 habitantes.

El distrito 1 tiene: 94679 habitantes.

El distrito 2 tiene: 88159 habitantes.

El distrito 3 tiene: 87429 habitantes.

El distrito 4 tiene: 76459 habitantes.

El distrito 5 tiene: 81510 habitantes.

El distrito 6 tiene: 98927 habitantes.

El distrito 7 tiene: 89556 habitantes.

El distrito 8 tiene: 68114 habitantes.

El distrito 9 tiene: 85626 habitantes.

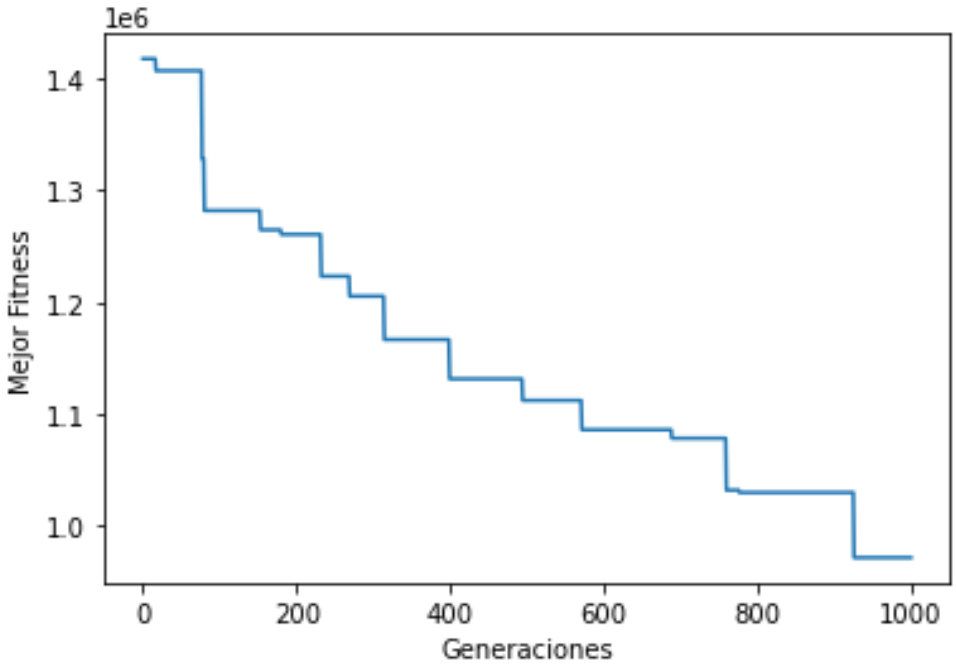
El distrito 10 tiene: 86067 habitantes.

El distrito 11 tiene: 60691 habitantes.

El distrito 12 tiene: 60219 habitantes.

El distrito 13 tiene: 84826 habitantes.

El distrito 14 tiene: 72112 habitantes.


1e6
1.4
1.3
1.2
1.1
1.0
Mejor Fitness
0
200
400
600
800
1000
Generaciones

- Por último un experimento con 20 distritos:

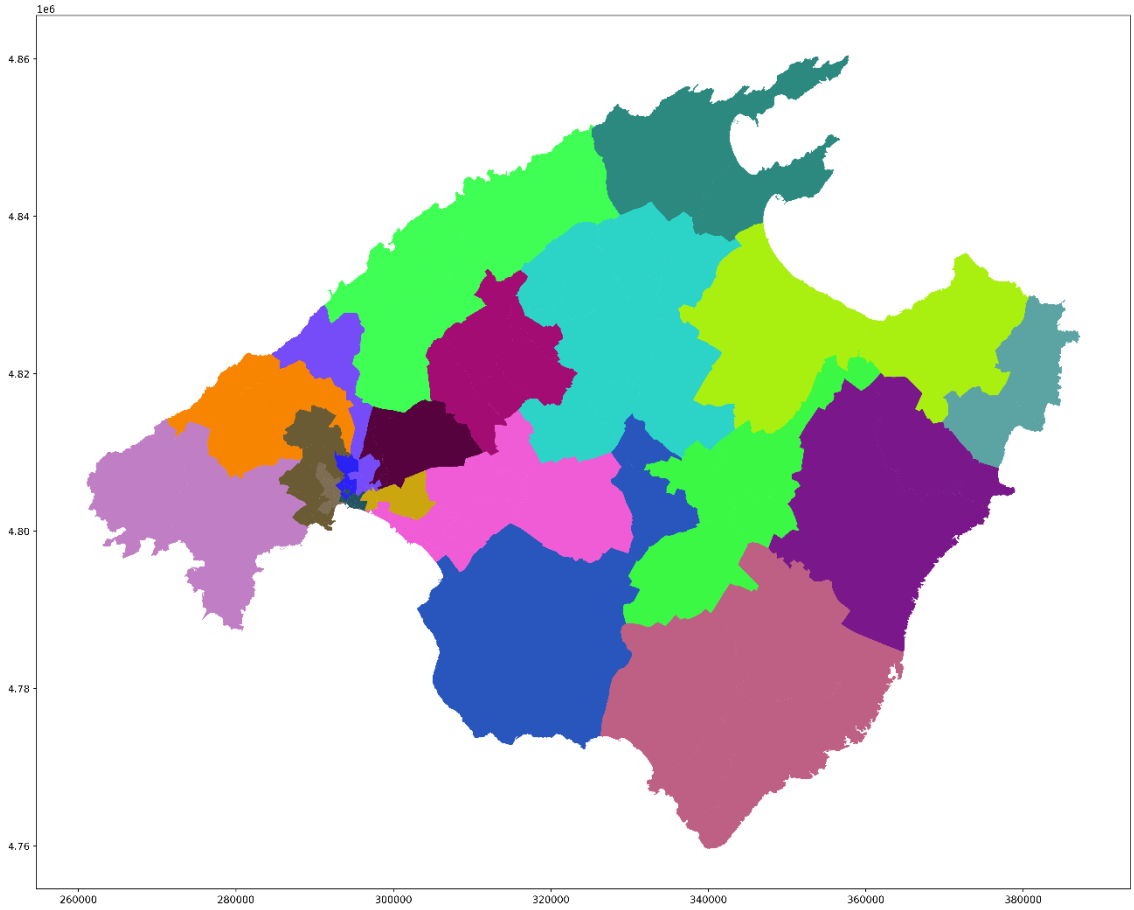
population_size = 5

mutation_rate = 0.1

max_generations = 1000

child_generation = 3

distritos = 20



Se han utilizado 20 colores aleatorios, por este motivo hay algún color casi idéntico. Se ha obtenido un fitness de 951011 con una compacidad de 678835, y una suma de las diferencias de los habitantes del distrito con la media de habitantes de los distritos de 272175. Además, la distribución de poblaciones ha quedado de la siguiente manera:

El distrito 0 tiene: 32106 habitantes.

El distrito 1 tiene: 37477 habitantes.

El distrito 2 tiene: 63143 habitantes.

El distrito 3 tiene: 62847 habitantes.

El distrito 4 tiene: 37730 habitantes.

El distrito 5 tiene: 22046 habitantes.

El distrito 6 tiene: 51838 habitantes.

El distrito 7 tiene: 53186 habitantes.

El distrito 8 tiene: 23975 habitantes.

El distrito 9 tiene: 63821 habitantes.

El distrito 10 tiene: 50389 habitantes.

El distrito 11 tiene: 67813 habitantes.

El distrito 12 tiene: 46954 habitantes.

El distrito 13 tiene: 73298 habitantes.

El distrito 14 tiene: 71819 habitantes.

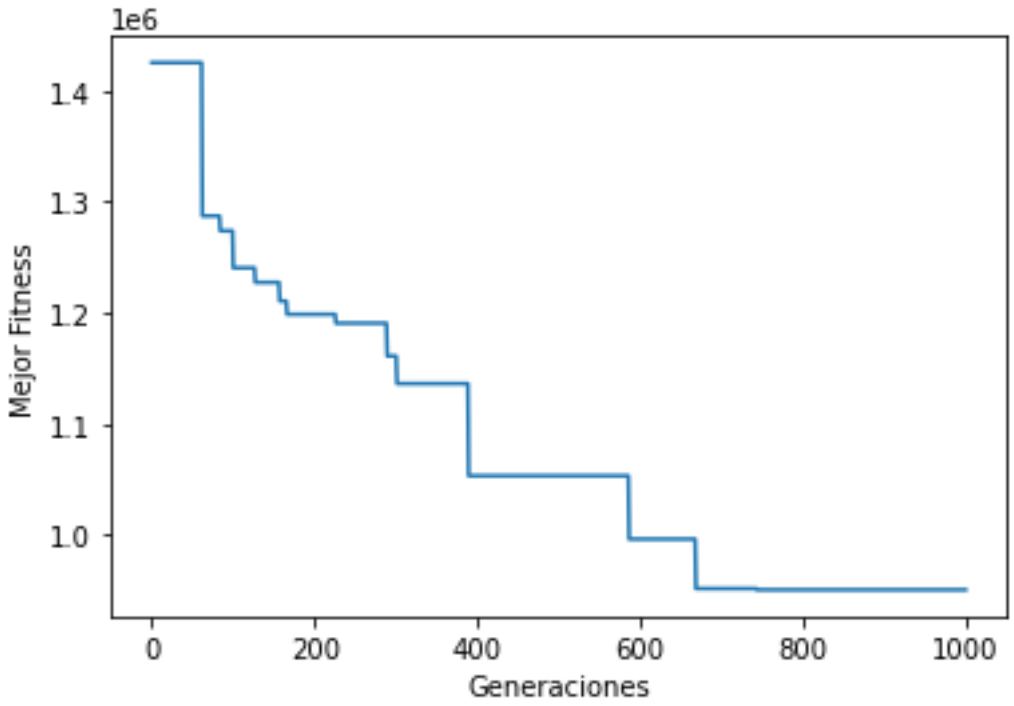
El distrito 15 tiene: 73840 habitantes.

El distrito 16 tiene: 63467 habitantes.

El distrito 17 tiene: 44574 habitantes.

El distrito 18 tiene: 53269 habitantes.

El distrito 19 tiene: 28687 habitantes.



5.Conclusiones

Hemos diseñado un algoritmo que consigue dividir el territorio en los distritos que le pidamos, aunque siempre sea un resultado diferente aun ejecutando con los mismos parámetros. No obstante, el tiempo de cálculo probablemente podría ser optimizado, ya que tarda unos 10 minutos para unas 100-150 generaciones