UNIVERSIDAD
DE GRANADA

METAHEURÍSTICA

Práctica 2.

Problema del Agrupamiento con Restricciones

Lukas Häring García

1. Introducción. **3**

2. Descripción de los algoritmos. **5**
2.1. Algoritmo Greedy (COPKM). 7
2.2. Algoritmo de Búsqueda Local, Primer Mejor (BL-PM). 8
2.3. Algoritmo de Búsqueda Local, Mejor Vecino (BL-MV). 9

3. Análisis de los resultados **11**
3.1 Dataset “Rand”. 11
3.1.1. Análisis Tabla de Resultados Greedy. 11
3.1.2. Análisis Tabla de Resultados Búsqueda Local, Primer Mejor. 12
3.1.3. Análisis Tabla de Resultados Búsqueda Local, Mejor Vecino. 13
3.1.4. Análisis Tabla de Resultados A. Genético Generacional Elitista. 14
3.1.4.1. Algoritmo Genético Generacional Uniforme. 14
3.1.5. Análisis Tabla de Resultados A. Genético Estacionario. 16
3.1.5.1. Algoritmo Genético Estacionario Uniforme. 16
3.1.5.2. Algoritmo Genético Estacionario Segmento Fijo. 17
3.1.6. Análisis Tabla de Resultados A. Meméticos. 18
3.1.6.1. Algoritmo Memético 10 Generaciones 100% Población. 18
3.1.6.2. Algoritmo Memético 10 Generaciones 10% Población. 19
3.1.6.3. Algoritmo Memético 10 Generaciones Mejores 10% Población. 20
3.1.7. Análisis Tabla de Resultados Medios. 21
3.2 Dataset “Iris”. 23
3.2.1. Análisis Tabla de Resultados Greedy. 23
3.2.2. Análisis Tabla de Resultados Búsqueda Local, Primer Mejor. 24
3.2.3. Análisis Tabla de Resultados Búsqueda Local, Mejor Vecino. 25
3.2.4.1. Algoritmo Genético Generacional Uniforme. 26
3.2.5. Análisis Tabla de Resultados A. Genético Estacionario. 28
3.2.5.1. Algoritmo Genético Estacionario Uniforme. 28
3.2.5.2. Algoritmo Genético Estacionario Segmento Fijo. 29
3.2.6. Análisis Tabla de Resultados A. Meméticos. 30
3.2.6.1. Algoritmo Memético 10 Generaciones 100% Población. 30
3.2.6.2. Algoritmo Memético 10 Generaciones 10% Población. 31
3.2.6.3. Algoritmo Memético 10 Generaciones Mejores 10% Población. 32
3.2.7. Análisis Tabla de Resultados Medios. 33
3.3 Dataset “Ecoli”. 35
3.3.1. Análisis Tabla de Resultados Greedy. 35
3.3.2. Análisis Tabla de Resultados Búsqueda Local, Primer Mejor. 36
3.3.3. Análisis Tabla de Resultados Búsqueda Local, Mejor Vecino. 37
3.3.4.1. Algoritmo Genético Generacional Uniforme. 38

3.3.5. Análisis Tabla de Resultados A. Genético Estacionario. 40
3.3.5.1. Algoritmo Genético Estacionario Uniforme. 40
3.3.5.2. Algoritmo Genético Estacionario Segmento Fijo. 41
3.3.6. Análisis Tabla de Resultados A. Meméticos. 42
3.3.6.1. Algoritmo Memético 10 Generaciones 100% Población. 42
3.3.6.2. Algoritmo Memético 10 Generaciones 10% Población. 43
3.3.6.3. Algoritmo Memético 10 Generaciones Mejores 10% Población. 44
3.3.7. Análisis Tabla de Resultados Medios. 45
3.4 Dataset “Newthyroid”. 47
3.4.1. Análisis Tabla de Resultados Greedy. 47
3.4.2. Análisis Tabla de Resultados Búsqueda Local, Primer Mejor. 48
3.4.3. Análisis Tabla de Resultados Búsqueda Local, Mejor Vecino. 49
3.4.4.1. Algoritmo Genético Generacional Uniforme. 50
3.4.5. Análisis Tabla de Resultados A. Genético Estacionario. 52
3.4.5.1. Algoritmo Genético Estacionario Uniforme. 52
3.4.5.2. Algoritmo Genético Estacionario Segmento Fijo. 53
3.4.6. Análisis Tabla de Resultados A. Meméticos. 54
3.4.6.1. Algoritmo Memético 10 Generaciones 100% Población. 54
3.4.6.2. Algoritmo Memético 10 Generaciones 10% Población. 55
3.4.6.3. Algoritmo Memético 10 Generaciones Mejores 10% Población. 56
3.4.7. Análisis Tabla de Resultados Medios. 57

4. Bibliografía **61**

1. Introducción.

Existen problemas que actualmente no se conoce ninguna estrategia para la búsqueda de una solución óptima en tiempo polinómico, es por ello que autores han ideado nuevas estrategias, cada vez más abstracta, para encontrar o aproximar la solución óptima en tiempo polinómico.

El problema propuesto por la asignatura es conocido como “Problema del Agrupamiento”, este problema se puede resolver en un tiempo $O(n^{dk+1} log\ n)$ [1], tal que son fijados “k”, el número de clusters, y “d”, la dimensión. “n” el número de entidades. Este tiempo está categorizado como polinómicos, es por ello que ha sido modificado para éste convertirse en un problema duro.

Ahora, el problema presenta dos tipos de restricciones, deben unirse **(*“Must-Link”*)** y no pueden unirse **(“*Not-Link*”)**. Utilizando los parámetros anteriores sobre unos *datasets*, vamos a proponer diferentes métodos para la aproximación de la solución óptima: La primera, la estrategia *codiciosa* **(Greedy)**, esta será el pilar para poder comparar los demás métodos; La segunda, la búsqueda local, en vez de explorar todo el conjunto de soluciones (que no es posible en tiempo polinómico), vamos a explorar un subconjunto de elementos e intentar converger a una solución aceptable; Otros métodos más ambiciosos inspirados en la naturaleza...

El propósito de esta práctica es la experimentación con algoritmos no rudimentarios para obtener una aproximación más o menos óptima en un tiempo aceptable, además de importar algoritmos bioinspirados para resolver algoritmos complejos.

En la segunda práctica vamos a utilizar algoritmos bio-inspirados para obtener soluciones aceptables, utilizaremos la técnica Darwinista evolutiva y una técnica híbrida, donde vamos a probar esta técnica junto a la búsqueda local.

Para las técnicas evolutivas, vamos a probar diferentes técnicas de cruce y selección, veremos que estas técnicas se ven favorecidas según la naturaleza del problema, siendo conveniente probar diferentes algoritmos.

2. Descripción de los algoritmos.

Para el desarrollo de la siguiente práctica, vamos a suponer que se conoce todo el contenido expuesto en la práctica anterior, los cálculos de la distancia intra-cluster, desviación general, cálculo del factor lambda, etc.

La representación de los datos será igual a la anterior, vamos a utilizar un vector en representación entera, vamos a utilizar el mismo esquema y su valor "f", para tratarlo como un "cromosoma".

La función objetivo está explicada en la práctica anterior, basta con recordar que se calcula de la siguiente forma y que no ha sido modificada para esta práctica:

$$f(A) = \lambda \cdot infeasibility(A) + general\_deviation(A)$$

A un cromosoma pueden ocurrirle tres acciones: selección o competición por la supervivencia; cruce, con otro cromosoma para añadir diversidad y mutación, que ayudará a salir de la convergencia a una solución.

Un conjunto de cromosomas representa una población, cada "pasada" de una población, genera otra utilizando las técnicas expuestas anteriormente.

Para la generación de la población inicial, vamos a generar soluciones aleatorias, que tienen la estructura que ya se ha comentado en la práctica anterior, estas soluciones deben cumplir la restricción de que todos los grupos tienen que tener al menos un elemento.

```
function Poblacion_inicial(num_grupos, tamaño_cromosoma, tamaño_poblacion)
   poblacion_inicial = [] : Cromosoma
   desde i=0 hasta num_grupos - 1, hacer:
     elemento_inicial = vector_aleatorio(num_grupos, tamaño_cromosoma)
     mientras incumple_restriccion(elemento_inicial, num_grupos):
         elemento_inicial = vector_aleatorio(num_grupos, tamaño_cromosoma)
     añadir(poblacion_inicial, elemento_inicial)
    devolver población_inicial
```

Vemos que incumple_restricciones, devolverá cierto o no, si existe un grupo vacío. Este devolverá un grupo de elementos (no tienen por que ser distintos) de posibles soluciones, que será el primer grupo en pasar nuestro algoritmo genético.

En la literatura existen muchas técnicas de selección, ruleta, torneo, estocástico, etc. Nosotros vamos a utilizar la técnica más sencilla, **torneo binario**. Seleccionamos dos elementos aleatorios distintos de la población y aquél que tenga mejor valor “f” (menor), lo metemos en la nueva población, repitiendo este proceso tantos elementos como habían en la población anterior.

Para el cruce, se han propuesto dos tipos diferentes, cruce uniforme y cruce segmento fijo, es obvio que vamos a necesitar dos individuos a cruzar. Todos los cruces para este problema van a hacer uso de la técnica de reparación para evitar que ninguno de los grupos quede vacío después del cruce.

```
function Cruce_Uniforme(individuo1, individuo2):
  tamaño = tamaño(individuo1)
  hijo_resultado = [] : Cromosoma
  indices_permutados = permutador([1,... tamaño])
  para i en indices_permutados[0 … tamaño / 2] hacer
    hijo_resultado[i] = individuo1[i]
  para i en indices_permutados[ tamaño / 2 … tamaño] hacer
    hijo_resultado[i] = individuo2[i]
  return reparar(hijo_resultado)
```

Como vemos, el cruce uniforme coge la primera mitad de la lista desordenada de índices y la utiliza para el primer individuo, la otra mitad para el siguiente individuo. Simula de forma eficiente el coger diferentes posiciones distintas de cada individuo diferente para generar un hijo.

```
function Cruce_Segmento_Fijo(individuo1, individuo2):
  tamaño = tamaño(individuo1)
  hijo_resultado = [] : Cromosoma
  posicion = random(0, tamaño)
  longitud = random(0, tamaño)
  i = posicion
  j = (posicion  + longitud) % tamaño
  segmento1 = [min(i,j)...max(i,j)]
  segmento2 = [0..min(i, j)] + [max(i,j)..tamaño]
  si i > j : intercambiar(segmento1, segmento2)
  para i en segmento1:
    hijo_resultado[i] = individuo1[i]
  //
  // De manera similar al Cruce Uniforme para distribuir el segmento 2 entre los
  // dos individuos, ya que el segmento 1 lo hemos distribuido sobre uno.
  return reparar(hijo_resultado)
```

Este algoritmo es algo más complejo, se crean dos subsegmentos (de ahí su nombre) del cromosoma, este está definido por una posición y un tamaño, de ahí obtenemos dos punteros, posición (a) y posición + tamaño (b), que se pueden cruzar, de los cuales podemos obtener de forma fácil los dos subsegmentos de índices, de 0 a min((a), (b)) unido con (max((a), (b)) hasta tamaño), representa los elementos que no están en el rango, el otro, se obtiene mirando dentro de dicho intervalo min((a), (b)) hasta max((a), (b)), que representa los genes dentro del intervalo. Ahora bien, como (a) y (b) se pueden cruzar, debido a que (b) puede dar la vuelta, entonces los subsegmentos se intercambiarán. Como se especifica en el pdf, tenemos que utilizar el primer subsegmento siempre para el primer padre, el otro, lo repartiremos entre los dos individuos, de manera similar al cruce Uniforme, permutamos dichos individuos, dividimos en dos sub-subsegmentos y repartimos en cada uno de los padres. Finalmente el resultado se repara si el resultado inclumpiese la restricción.

```
function reparar(hijo):
  mientras inclumple_restriccion(hijo):
    para i en [0, … k - 1]:
      si i no está en hijo:
        hijo[aleatorio(n)] = i
  return hijo
```

Como vemos, mientras se incumpla una restricción, vamos a buscar qué grupo está vacío (es el causante de que se incumpla) y vamos a ponerlo en una posición aleatoria, hasta que sea reparado.

Finalmente, vamos a ver la mutación, se trata de una mutación simple, además, esta tiene que ser válida, es decir, tampoco puede dejar un grupo vacío ni puede ser igual a la anterior.

```
function mutacion_uniforme(hijo):
  gen_posicion = random(len(hijo))
  gen_anterior = hijo[gen_posicion]
  hijo[gen_posicion] = random(0, k)
  mientras inclumple_restriccion(hijo) or gen_anterior eq hijo[gen_posicion]:
        hijo[gen_posicion] = random(0, k)
  return hijo
```

Vemos que devuelve un hijo con un único gen modificado, distinto al anterior y válido.

2.1. Algoritmo Genético Generacional.

El algoritmo generacional utiliza el total de la población para generar la siguiente población, para ello, utilizará como hemos comentado antes, los tres operadores expuestos.

La condición de parada será haber llamado 100.000 veces la función “f”. En primer lugar, vamos a generar la población inicial de forma aleatoria como hemos expuesto antes, además, calculando de cada uno su función “f”.

Mientras dicha función tenga menos de 100.000 llamadas, vamos a coger la población actual y vamos a coger por **torneo binario** tantos como elementos haya en la población, esta es la fase de selección.

De dicho conjunto, vamos a generar pares de elementos a cruzar, he optado por elegir pares aleatorios ya que da mayor diversidad, aunque podría haberse cogido 2 a 2 consecutivamente y optimizar, se aplicará sobre estos dos elementos, una probabilidad de cruce, si da positivo, se cruzan dos veces y los hijos entrarán en el conjunto de la siguiente población, si fuera negativo, entrarán ambos padres seleccionados.

Para gestionar la **mutación**, se utiliza un contador de elementos mutados, un contador real, este mantiene información del valor de la actual del número de genes a mutar por esperanza matemática, cuando este valor supere 1, significa que tiene que mutar un elemento.

Una vez esto, calculamos el valor de “f” para cada uno de los elementos de la nueva población, si el elitismo está activado, reemplazará aquellos “n” elementos peores por los mejores de la anterior.

```
function AGG(X, R, k, tam_poblacion, prob_cruce, prob_mutacion):
   genes = tamaño(X)
   poblacion = Poblacion_inicial(k, genes, tam_poblacion)
   esperanza_mutar = prob_mutacion * genes * tam_poblacion
   f_llamadas = tam_poblacion

   mientras f_llamadas < 100.000 hacer:
     seleccionados = []
     // Seleccion
     mientras i en 1..tam_poblacion:
       seleccionados = Seleccion(poblacion, k)
     // Cruce
     sig_poblacion = []
     mientras i en 1..tam_poblacion:
       pares = generar_distinto(0… tam_poblacion-1)
       si aleatorio(0, 1) < probabilidad_cruce:
         añadir(sig_poblacion, cruzar(poblacion [pares [0] ], poblacion [pares [1]]))
       si no:
         añadir(sig_poblacion, poblacion [pares [0] ], poblacion [pares [1]])

      // Mutacion
      si esperanza_mutar >= 1:
        mutaciones = entero(esperanza_mutar)
        esperanza_mutar  -= mutaciones
        mutar = genera_nveces(0..tam_poblacion-1, mutaciones)
        para i en mutar :
          sig_poblacion[i] = mutar(sig_poblacion[i])

      // Evaluamos
      evaluar(sig_poblacion)

      si elitismo > 0:
         para i en 0..elitismo:
            indice_mejor = buscar_indice_mejor(poblacion)
            indice_peor = buscar_indice_peor(sig_poblacion)
            sig_poblacion[indice_peor ] = poblacion[indice_mejor]

      poblacion = sig_poblacion
      f_llamadas += f_llamadas
      esperanza_mutar += prob_mutacion * genes * tam_poblacion

  devolver mejor(poblacion)
```

2.2. Algoritmo Genético Estacionario.

Todos presentan el mismo esquema, pero este mantiene la población anterior, solo trabaja con dos elementos en cada pasada.

Vemos que la condición de parada es la misma y que el resultado devuelto también lo es, el mejor de la población final.

Se generan dos elementos por cada pasada, se cruzan ambos (sin probabilidad), mutamos si es necesario, con la misma condición anterior, aunque recordar que ahora solo se generan 2 elementos.

Para la mutación, se mutará uno de los dos cromosomas que se han obtenido tras el cruce, para ello se generará un número entero uniformemente entre 0, 1 y se mutará uno u otro según dicho número.

Finalmente, si alguno de los cromosomas es mejor que alguno de los peores del población anterior, los sustituimos, en caso contrario, mantenemos la población anterior.

```
function AGE(X, R, k, tam_poblacion, prob_mutacion):
    genes = tamaño(X)
    poblacion = Poblacion_inicial(k, genes, tam_poblacion)
    esperanza_mutar = prob_mutacion * genes * 2
    f_llamadas = tam_poblacion

    mientras f_llamadas < 100.000 hacer:
      // Seleccion
      seleccionado1 = Seleccion(poblacion, k)
      seleccionado2 = Seleccion(poblacion, k)
      // Cruce
      sig_poblacion = [
           cruzar(seleccionado1, seleccionado2),
           cruzar(seleccionado1, seleccionado2),
      ]

       // Mutacion
       si esperanza_mutar >= 1:
          mutaciones = entero(esperanza_mutar)
          esperanza_mutar  -= mutaciones
          si random_entero(0, 1) == 0:
             sig_poblacion[0] = mutar(sig_poblacion[0])
          contrario:
             sig_poblacion[1] = mutar(sig_poblacion[1])

       // Evaluamos
       evaluar(sig_poblacion)

       si mejor_f(sig_poblacion) >= peor_f(poblacion):
          sustituimos_peor_mejor(poblacion, sig_poblacion)

       f_llamadas += 2
       esperanza_mutar += prob_mutacion * genes * 2

  devolver mejor(poblacion)
```

2.3. Algoritmo Memético.

Un algoritmo memético combina algoritmo de búsqueda con uno genético, el algoritmo genético utilizado es AGE, mientras que el algoritmo de búsqueda es conocido como “Búsqueda Local Suave”.

Comentar que esta búsqueda es muy simple, parecida a Greedy, va probando cambios en los genes de manera lineal, si mejora la función de evaluación, lo alteramos, en caso contrario, mantenemos un contador de errores, si superase cierto umbral, devolveremos el cromosoma.

La técnica de búsqueda se aplica cada 10 iteraciones del algoritmo genético, como el algoritmo genético es un AGE, vemos que el que más influye sobre las llamadas de “f” va a ser BLS, si hubiera elegido AGG, podría cambiar la cosa.
Destacar también que la esperanza de mutar se ve constante mientras se aplica BLS.

AM tiene dos cualidades en esta práctica, podemos realizar búsqueda local sobre los mejores cromosomas o aleatoriamente, además de a qué porcentaje de estos representa. Por ejemplo AM10,0.1mej representa una población de 10, una búsqueda 10% de los mejores elementos de la población.

El algoritmo BLS (expuesto en el pdf), va tener también como parámetro cuantas ejecuciones de f lleva hasta el momento, sigue la misma estructura que la del pdf, la única diferencia es que una de las condiciones de parada es no superar el umbral de ejecuciones en ningún momento e ir incrementando cada vez que se ejecute.

```
function AM(X, R, k, tam_poblacion, prob_mutacion, bls, porc_repr, mejores):
    genes = tamaño(X)
    poblacion = Poblacion_inicial(k, genes, tam_poblacion)
    esperanza_mutar = prob_mutacion * genes * 2
    f_llamadas = tam_poblacion
    f_BLS = 0
    mientras f_llamadas < 100.000 hacer:
      si f_BLS == bls:
         f_BLS = 0
         si mejores:
            sub_poblacion = subgrupo(mejores(poblacion), porc_repr)
         contrario:
            sub_poblacion = subgrupo(poblacion, porc_repr)

         mientras i en sub_poblacion:
            bls_poblacion, f_bls_llamadas = BLS(sub_poblacion, f_llamadas)
            reemplazar(poblacion, bls_poblacion)
            f_llamadas = f_bls_llamadas

       contrario: f_BLS += 1
         // Seleccion
         seleccionado1 = Seleccion(poblacion, k)
         seleccionado2 = Seleccion(poblacion, k)
         // Cruce
         sig_poblacion = [
              cruzar(seleccionado1, seleccionado2),
              cruzar(seleccionado1, seleccionado2),
         ]
          // Mutacion
          si esperanza_mutar >= 1:
            mutaciones = entero(esperanza_mutar)
            esperanza_mutar  -= mutaciones
           si random_entero(0, 1) == 0:
            sig_poblacion[0] = mutar(sig_poblacion[0])
         contrario:
           sig_poblacion[1] = mutar(sig_poblacion[1])
          // Evaluamos
          evaluar(sig_poblacion)
          si mejor_f(sig_poblacion) >= peor_f(poblacion):
            sustituimos_peor_mejor(poblacion, sig_poblacion)

        f_llamadas += 2
        esperanza_mutar += prob_mutacion * genes * 2

   devolver mejor(poblacion)
```

3. Análisis de los resultados

3.1 Dataset “Rand”.

El factor obtenido es $\lambda = 0.0072$, está definido el problema para 3 clusters.
El tiempo utilizado para ejecutar todas estas pruebas sobre el conjunto de Rand ha sido aproximadamente de 4h.

3.1.1. Análisis Tabla de Resultados Greedy.

10% COPKM	**RAND (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	1.00	280.00	3.01	1.15s
Ejecución 2	1.00	226.00	2.63	0.77s
Ejecución 3	0.92	158.00	2.05	1.12s
Ejecución 4	1.00	200.00	2.44	1.53s
Ejecución 5	1.21	238.00	2.92	1.91s
Media	1.02	220.40	2.61	1.30s

20% COPKM	**RAND (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	0.76	68.00	1.01	1.16s
Ejecución 2	1.02	408.00	2.49	1.17s
Ejecución 3	0.86	302.00	1.95	1.59s
Ejecución 4	0.77	156.00	1.33	1.52s
Ejecución 5	0.85	256.00	1.77	1.57s
Media	0.85	238.00	1.71	1.40s

Como podemos ver, la tasa de infeasibility es bastante alta, lo que nos dice que greedy nos ofrece una solución bastante mala. Aún así, el agregado es bastante

bajo debido a que el factor tiende a darle poca importancia a la infeasibility, debido a que los datos están poco dispersos.

3.1.2. Análisis Tabla de Resultados Búsqueda Local, Primer Mejor.

10% BL-PM	**RAND (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	0.72	0	0.72	3.02s (1522 its)
Ejecución 2	0.72	0	0.72	3.28s (1671 its)
Ejecución 3	0.72	0	0.72	3.17s (1613 its)
Ejecución 4	0.72	0	0.72	2.63s (1353 its)
Ejecución 5	0.72	0	0.72	2.90s (1484 its)
Media	0.72	0	0.72	3.00s (1529 its)

20% BL-PM	**RAND (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	0.72	0	0.72	3.62s (1818 its)
Ejecución 2	0.72	0	0.72	3.25s (1618 its)
Ejecución 3	0.72	0	0.72	3.25s (1640 its)
Ejecución 4	0.72	0	0.72	3.03s (1575 its)
Ejecución 5	0.72	0	0.72	2.27s (1123 its)
Media	0.72	0	0.72	3.08s (1555 its)

3.1.3. Análisis Tabla de Resultados Búsqueda Local, Mejor Vecino.

10% BL-MV	**RAND (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	0.72	0	0.72	51.17s (30900 its)
Ejecución 2	0.72	0	0.72	57.59s (35100 its)
Ejecución 3	0.72	0	0.72	46.18s (28200 its)
Ejecución 4	0.72	0	0.72	44.76s (27300 its)
Ejecución 5	0.72	0	0.72	42.79s (26100 its)
Media	0.72	0	0.72	48.50s (29520 its)

20% BL-MV	**RAND (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	0.72	0	0.72	51.08s (30600 its)
Ejecución 2	0.72	0	0.72	53.82s (32100 its)
Ejecución 3	0.72	0	0.72	46.61s (27900 its)
Ejecución 4	0.72	0	0.72	44.65s (26700 its)
Ejecución 5	0.72	0	0.72	42.35s (25200 its)
Media	0.72	0	0.72	47.70s (28500 its)

3.1.4. Análisis Tabla de Resultados A. Genético Generacional Elitista.

Se ha utilizado un elitismo de 1 elemento, es decir para cada “pase” de la población, el peor resultante se sustituye por el mejor de la generación anterior, aunque en mi código se ha generalizado para poder realizarse con un valor de elitismo N (genérico). Además, se ha utilizado una probabilidad de cruce de 0.7 (70%).

3.1.4.1. Algoritmo Genético Generacional Uniforme.

Se ha utilizado el cruce uniforme, con selección por torneo de dos individuos.

10% AGG-UN	**RAND (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	0.72	0	0.72	217.21s
Ejecución 2	0.72	0	0.72	291.50s
Ejecución 3	0.72	0	0.72	210.26s
Ejecución 4	0.72	0	0.72	214.34s
Ejecución 5	0.72	0	0.72	279.07s
Media	0.72	0	0.72	242.48s

20% AGG-UN	**RAND (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	0.72	0	0.72	211.24s
Ejecución 2	0.72	0	0.72	263.01s
Ejecución 3	0.72	0	0.72	219.24s
Ejecución 4	0.72	0	0.72	207.45s
Ejecución 5	0.72	0	0.72	217.32s
Media	0.72	0	0.72	223.65s

3.1.4.2. Algoritmo Genético Generacional Uniforme.

Se ha utilizado el cruce por segmento fijo, con selección por torneo de dos individuos.

10% AGG-SF	**RAND (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	0.72	0	0.72	221.34s
Ejecución 2	0.72	0	0.72	231.21s
Ejecución 3	0.72	0	0.72	217.54s
Ejecución 4	0.72	0	0.72	250.12s
Ejecución 5	0.72	0	0.72	232.10s
Media	0.72	0	0.72	230.46s

20% AGG-SF	**RAND (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	0.72	0	0.72	216.25s
Ejecución 2	0.72	0	0.72	210.32s
Ejecución 3	0.72	0	0.72	224.10s
Ejecución 4	0.72	0	0.72	241.21s
Ejecución 5	0.72	0	0.72	260.05s
Media	0.72	0	0.72	230.38s

3.1.5. Análisis Tabla de Resultados A. Genético Estacionario.

La probabilidad de cruce es del 100%, eso significa que siempre se van a cruzar. Estos dos elementos del cruce de los padres se sustituyen por los peores elementos de la generación anterior, si son mejores.

3.1.5.1. Algoritmo Genético Estacionario Uniforme.

Se ha utilizado el cruce uniforme, con selección por torneo de dos individuos.

10% AGE-UN	**RAND (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	0.72	0	0.72	217.21s
Ejecución 2	0.72	0	0.72	223.34s
Ejecución 3	0.72	0	0.72	220.55s
Ejecución 4	0.72	0	0.72	250.21s
Ejecución 5	0.72	0	0.72	245.05s
Media	0.72	0	0.72	231.27s

20% AGE-UN	**RAND (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	0.72	0	0.72	211.24s
Ejecución 2	0.72	0	0.72	223.81s
Ejecución 3	0.72	0	0.72	217.22s
Ejecución 4	0.72	0	0.72	243.10s
Ejecución 5	0.72	0	0.72	235.67s
Media	0.72	0	0.72	226.20s

3.1.5.2. Algoritmo Genético Estacionario Segmento Fijo.

Se ha utilizado el cruce por segmento fijo, con selección por torneo de dos individuos.

10% AGE-SF	**RAND (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	0.72	0	0.72	226.17s
Ejecución 2	0.72	0	0.72	230.21s
Ejecución 3	0.72	0	0.72	231.78s
Ejecución 4	0.72	0	0.72	250.12s
Ejecución 5	0.72	0	0.72	223.46s
Media	0.72	0	0.72	232.35s

20% AGE-SF	**RAND (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	0.72	0	0.72	217.27s
Ejecución 2	0.72	0	0.72	227.10s
Ejecución 3	0.72	0	0.72	223.67s
Ejecución 4	0.72	0	0.72	232.78s
Ejecución 5	0.72	0	0.72	228.23s
Media	0.72	0	0.72	225.81s

3.1.6. Análisis Tabla de Resultados A. Meméticos.

Como hemos comentado anteriormente, vemos que AGE es mejor a AGG, además, utilizamos el algoritmo de cruce por segmento fijo.

3.1.6.1. Algoritmo Memético 10 Generaciones 100% Población.

Cada 10 iteraciones, vamos a aplicar Búsqueda Local Suave al 100% de la población.

10% AM-10,1.0	**RAND (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	0.72	0	0.72	227.39s
Ejecución 2	0.72	0	0.72	220.20s
Ejecución 3	0.72	0	0.72	217.83s
Ejecución 4	0.72	0	0.72	230.63s
Ejecución 5	0.72	0	0.72	232.02s
Media	0.72	0	0.72	225.60s

20% AM-10,1.0	**RAND (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	0.72	0	0.72	232.34s
Ejecución 2	0.72	0	0.72	236.12s
Ejecución 3	0.72	0	0.72	223.56s
Ejecución 4	0.72	0	0.72	230.12s
Ejecución 5	0.72	0	0.72	220.01s
Media	0.72	0	0.72	228.42s

3.1.6.2. Algoritmo Memético 10 Generaciones 10% Población.

Cada 10 iteraciones, vamos a aplicar Búsqueda Local Suave al 10% de la población.

10% AM-10,0.1	**RAND (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	0.72	0	0.72	221.49s
Ejecución 2	0.72	0	0.72	217.31s
Ejecución 3	0.72	0	0.72	224.76s
Ejecución 4	0.72	0	0.72	228.13s
Ejecución 5	0.72	0	0.72	232.12s
Media	0.72	0	0.72	224.76s

20% AM-10,0.1	**RAND (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	0.72	0	0.72	223.59s
Ejecución 2	0.72	0	0.72	231.82s
Ejecución 3	0.72	0	0.72	221.36s
Ejecución 4	0.72	0	0.72	231.14s
Ejecución 5	0.72	0	0.72	224.16s
Media	0.72	0	0.72	226.41s

3.1.6.3. Algoritmo Memético 10 Generaciones Mejores 10% Población.

Cada 10 iteraciones, vamos a aplicar Búsqueda Local Suave al 10% de la mejor población.

10% AM-10,0.1 mej	**RAND (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	0.72	0	0.72	227.67s
Ejecución 2	0.72	0	0.72	218.21s
Ejecución 3	0.72	0	0.72	213.13s
Ejecución 4	0.72	0	0.72	224.71s
Ejecución 5	0.72	0	0.72	224.76s
Media	0.72	0	0.72	223.15s

20% AM-10,0.1 mej	**RAND (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	0.72	0	0.72	223.59s
Ejecución 2	0.72	0	0.72	226.21s
Ejecución 3	0.72	0	0.72	216.16s
Ejecución 4	0.72	0	0.72	221.74s
Ejecución 5	0.72	0	0.72	223.43s
Media	0.72	0	0.72	224.76s

3.1.7. Análisis Tabla de Resultados Medios.

10%	**RAND (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
COPKM	1.02	220.40	2.61	1.30s
BL-PM	0.72	0	0.72	3.28s
BL-MV	0.72	0	0.72	57.59s
AGG-UN	0.72	0	0.72	242.48s
AGG-SF	0.72	0	0.72	230.46s
AGE-UN	0.72	0	0.72	231.27s
AGE-SF	0.72	0	0.72	232.35s
AM-10,1.0	0.72	0	0.72	225.60s
AM-10,0.1	0.72	0	0.72	223.15s
AM-10,0.1mej	0.72	0.0	0.72	227.67s

20%	**RAND (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
COPKM	0.85	238.00	1.71	1.40s
BL-PM	0.72	0	0.72	3.08s (1555 its)
BL-MV	0.72	0	0.72	47.70s (28500 its)
AGG-UN	0.72	0	0.72	223.65s
AGG-SF	0.72	0	0.72	230.38s
AGE-UN	0.72	0	0.72	226.20s
AGE-SF	0.72	0	0.72	225.81s
AM-10,1.0	0.72	0	0.72	228.42s
AM-10,0.1	0.72	0	0.72	226.41s
AM-10,0.1mej	0.72	0	0.72	224.76s

Vemos que greedy es el que peores resultados obtiene, mientras que claramente BL encuentra la solución en tiempos razonables. Vemos además que todos los algoritmos evolutivos y el meméticos son capaces de encontrar la solución óptima, aunque el tiempo de ejecución es el inconveniente ya que al ser una condición de parada pre-fijada.

En este problema, al ser un espacio sencillo, no podemos distinguir si dichos algoritmos son mejores o peores entre sí, pero podemos afirmar sin ninguna duda que todos son buenos ya que todos ofrecen la solución aparentemente óptima.

Finalmente comentar que entre ambos porcentajes de restricciones, tampoco se encuentran diferencias significativas, por lo que no podemos decir mucho sobre los algoritmos.

3.2 Dataset “Iris”.

El factor obtenido es $\lambda = 0.0063$, también definido para 3 clusters.
El tiempo utilizado para ejecutar todas estas pruebas sobre el conjunto de Iris, ha sido similar al anterior, alrededor de 4h.

3.2.1. Análisis Tabla de Resultados Greedy.

10% COPKM	**IRIS (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	1.16	522.00	4.47	2.26s
Ejecución 2	1.10	690.00	5.47	1.87s
Ejecución 3	0.78	454.00	3.66	1.13s
Ejecución 4	0.91	414.00	3.54	1.50s
Ejecución 5	0.94	384.00	3.38	1.89s
Media	0.98	492.8	4.10	1.73s

20% COPKM	**IRIS (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	1.13	1044.00	4.44	1.59s
Ejecución 2	0.93	866.00	3.68	2.35s
Ejecución 3	0.70	632.00	2.70	1.20s
Ejecución 4	0.85	598.00	2.74	2.41s
Ejecución 5	0.88	498.00	2.46	2.14s
Media	0.90	727,6	3.20	1.94s

3.2.2. Análisis Tabla de Resultados Búsqueda Local, Primer Mejor.

10% BL-PM	**IRIS (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	0.67	0.00	0.67	2.69s (1344 its)
Ejecución 2	0.67	0.00	0.67	2.86s (1440 its)
Ejecución 3	0.67	0.00	0.67	2.93s (1450 its)
Ejecución 4	0.67	0.00	0.67	3.29s (1742 its)
Ejecución 5	0.67	0.00	0.67	3.42s (1804 its)
Media	0.67	0.00	0.67	3.04s (1556 its)

20% BL-PM	**IRIS (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	0.67	0.00	0.67	3.63s (1781 its)
Ejecución 2	0.67	0.00	0.67	3.51s (1734 its)
Ejecución 3	0.67	0.00	0.67	3.00s (1486 its)
Ejecución 4	0.67	0.00	0.67	3.46s (1704 its)
Ejecución 5	0.67	0.00	0.67	3.55s (1766 its)
Media	0.67	0.00	0.67	3.63s (1781 its)

3.2.3. Análisis Tabla de Resultados Búsqueda Local, Mejor Vecino.

10% BL-MV	**IRIS (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	0.67	0.00	0.67	62.02s (37200 its)
Ejecución 2	0.67	0.00	0.67	53.33s (32100 its)
Ejecución 3	0.67	0.00	0.67	51.58s (30900 its)
Ejecución 4	0.67	0.00	0.67	45.38s (27300 its)
Ejecución 5	0.67	0.00	0.67	42.88s (25800 its)
Media	0.67	0.00	0.67	51.04s (30660 its)

20% BL-MV	**IRIS (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	0.67	0.00	0.67	52.45s (30000 its)
Ejecución 2	0.67	0.00	0.67	49.78s (28800 its)
Ejecución 3	0.67	0.00	0.67	47.08s (27300 its)
Ejecución 4	0.67	0.00	0.67	45.13s (26100 its)
Ejecución 5	0.67	0.00	0.67	44.68s (25500 its)
Media	0.67	0.00	0.67	47.82s (27540 its)

3.2.4. Análisis Tabla de Resultados A. Genético Generacional Elitista.

Se ha utilizado un elitismo de 1 elemento, es decir para cada "pase" de la población, el peor resultante se sustituye por el mejor de la generación anterior, aunque en mi código se ha generalizado para poder realizarse con un valor de elitismo N (genérico). Además, se ha utilizado una probabilidad de cruce de 0.7 (70%).

3.2.4.1. Algoritmo Genético Generacional Uniforme.

Se ha utilizado el cruce uniforme, con selección por torneo de dos individuos.

10% AGG-UN	**IRIS (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	0.67	0.00	0.67	217.64s
Ejecución 2	0.67	0.00	0.67	218.90s
Ejecución 3	0.67	0.00	0.67	222.89s
Ejecución 4	0.67	0.00	0.67	212.15s
Ejecución 5	0.67	0.00	0.67	217.09s
Media	0.67	0.00	0.67	217.74s

20% AGG-UN	**IRIS (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	0.67	0.00	0.67	210.89s
Ejecución 2	0.67	0.00	0.67	213.62s
Ejecución 3	0.67	0.00	0.67	216.10s
Ejecución 4	0.67	0.00	0.67	209.78s
Ejecución 5	0.67	0.00	0.67	218.43s
Media	0.67	0.00	0.67	213.76s

3.2.4.2. Algoritmo Genético Generacional Uniforme.

Se ha utilizado el cruce por segmento fijo, con selección por torneo de dos individuos.

10% AGG-SF	**IRIS (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	0.67	0.00	0.67	219.06s
Ejecución 2	0.67	0.00	0.67	221.84s
Ejecución 3	0.67	0.00	0.67	216.50s
Ejecución 4	0.67	0.00	0.67	217.92s
Ejecución 5	0.67	0.00	0.67	224.33s
Media	0.67	0.00	0.67	219.93s

20% AGG-SF	**IRIS (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	0.67	0.00	0.67	216.67s
Ejecución 2	0.67	0.00	0.67	221.56s
Ejecución 3	0.67	0.00	0.67	223.27s
Ejecución 4	0.67	0.00	0.67	212.31s
Ejecución 5	0.67	0.00	0.67	217.76s
Media	0.67	0.00	0.67	218.31s

3.2.5. Análisis Tabla de Resultados A. Genético Estacionario.

La probabilidad de cruce es del 100%, eso significa que siempre se van a cruzar. Estos dos elementos del cruce de los padres se sustituyen por los peores elementos de la generación anterior, si son mejores.

3.2.5.1. Algoritmo Genético Estacionario Uniforme.

Se ha utilizado el cruce uniforme, con selección por torneo de dos individuos.

10% AGE-UN	**IRIS (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	0.67	0.00	0.67	217.32s
Ejecución 2	0.67	0.00	0.67	212.08s
Ejecución 3	0.67	0.00	0.67	221.44s
Ejecución 4	0.67	0.00	0.67	220.26s
Ejecución 5	0.67	0.00	0.67	218.92s
Media	0.67	0.00	0.67	218.00s

20% AGE-UN	**IRIS (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	0.67	0.00	0.67	216.82s
Ejecución 2	0.67	0.00	0.67	221.77s
Ejecución 3	0.67	0.00	0.67	211.30s
Ejecución 4	0.67	0.00	0.67	210.21s
Ejecución 5	0.67	0.00	0.67	226.92s
Media	0.67	0.00	0.67	220.87s

3.2.5.2. Algoritmo Genético Estacionario Segmento Fijo.

Se ha utilizado el cruce por segmento fijo, con selección por torneo de dos individuos.

10% AGE-SF	**IRIS (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	0.67	0.00	0.67	219.22s
Ejecución 2	0.67	0.00	0.67	208.09s
Ejecución 3	0.67	0.00	0.67	212.63s
Ejecución 4	0.67	0.00	0.67	222.49s
Ejecución 5	0.67	0.00	0.67	226.36s
Media	0.67	0.00	0.67	217.76s

20% AGE-SF	**IRIS (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	0.67	0.00	0.67	219.23s
Ejecución 2	0.67	0.00	0.67	218.21s
Ejecución 3	0.67	0.00	0.67	222.55s
Ejecución 4	0.67	0.00	0.67	224.96s
Ejecución 5	0.67	0.00	0.67	220.46s
Media	0.67	0.00	0.67	211.27s

3.2.6. Análisis Tabla de Resultados A. Meméticos.

Como hemos comentado anteriormente, vemos que AGE es mejor a AGG, además, utilizamos el algoritmo de cruce por segmento fijo.

3.2.6.1. Algoritmo Memético 10 Generaciones 100% Población.

Cada 10 iteraciones, vamos a aplicar Búsqueda Local Suave al 100% de la población.

10% AM-10,1.0	**IRIS (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	0.67	0.00	0.67	232.34s
Ejecución 2	0.67	0.00	0.67	229.14s
Ejecución 3	0.67	0.00	0.67	227.96s
Ejecución 4	0.67	0.00	0.67	222.71s
Ejecución 5	0.67	0.00	0.67	238.12s
Media	0.67	0.00	0.67	230.05s

20% AM-10,1.0	**IRIS (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	0.67	0.00	0.67	228.45s
Ejecución 2	0.67	0.00	0.67	208.12s
Ejecución 3	0.67	0.00	0.67	210.68s
Ejecución 4	0.67	0.00	0.67	212.50s
Ejecución 5	0.67	0.00	0.67	226.62s
Media	0.67	0.00	0.67	217.27s

3.2.6.2. Algoritmo Memético 10 Generaciones 10% Población.

Cada 10 iteraciones, vamos a aplicar Búsqueda Local Suave al 10% de la población.

10% AM-10,0.1	**IRIS (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	0.67	0.00	0.67	226.21s
Ejecución 2	0.67	0.00	0.67	221.34s
Ejecución 3	0.67	0.00	0.67	222.96s
Ejecución 4	0.67	0.00	0.67	231.45s
Ejecución 5	0.67	0.00	0.67	228.64s
Media	0.67	0.00	0.67	226.12s

20% AM-10,0.1	**IRIS (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	0.67	0.00	0.67	221.71s
Ejecución 2	0.67	0.00	0.67	212.34s
Ejecución 3	0.67	0.00	0.67	223.38s
Ejecución 4	0.67	0.00	0.67	209.67s
Ejecución 5	0.67	0.00	0.67	218.78s
Media	0.67	0.00	0.67	217.18s

3.2.6.3. Algoritmo Memético 10 Generaciones Mejores 10% Población.

Cada 10 iteraciones, vamos a aplicar Búsqueda Local Suave al 10% de la mejor población.

10% AM-10,0.1 mej	**IRIS (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	0.67	0.00	0.67	224.57s
Ejecución 2	0.67	0.00	0.67	222.42s
Ejecución 3	0.67	0.00	0.67	228.53s
Ejecución 4	0.67	0.00	0.67	225.98s
Ejecución 5	0.67	0.00	0.67	223.96s
Media	0.67	0.00	0.67	225.09s

20% AM-10,0.1 mej	**IRIS (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	0.67	0.00	0.67	223.51s
Ejecución 2	0.67	0.00	0.67	232.13s
Ejecución 3	0.67	0.00	0.67	228.33s
Ejecución 4	0.67	0.00	0.67	225.28s
Ejecución 5	0.67	0.00	0.67	243.52s
Media	0.67	0.00	0.67	224.61s

3.2.7. Análisis Tabla de Resultados Medios.

10%	**IRIS (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
COPKM	0.98	492.8	4.10	1.73s
BL-PM	0.67	0.00	0.67	3.04s (1556 its)
BL-MV	0.67	0.00	0.67	62.02s (37200 its)
AGG-UN	0.67	0.00	0.67	217.74s
AGG-SF	0.67	0.00	0.67	213.76s
AGE-UN	0.67	0.00	0.67	219.93s
AGE-SF	0.67	0.00	0.67	218.00s
AM-10,1.0	0.67	0.00	0.67	217.76s
AM-10,0.1	0.67	0.00	0.67	230.05s
AM-10,0.1mej	0.67	0.00	0.67	226.12s

20%	**IRIS (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
COPKM	0.98	727.6	4.10	1.73s
BL-PM	0.67	0.00	0.67	3.63s (1781 its)
BL-MV	0.67	0.00	0.67	47.82s (27540 its)
AGG-UN	0.67	0.00	0.67	213.76s
AGG-SF	0.67	0.00	0.67	218.31s
AGE-UN	0.67	0.00	0.67	220.87s
AGE-SF	0.67	0.00	0.67	211.27s
AM-10,1.0	0.67	0.00	0.67	230.05s
AM-10,0.1	0.67	0.00	0.67	217.27s
AM-10,0.1mej	0.67	0.00	0.67	217.18s

Este problema sigue siendo bastante sencillo, todos los algoritmos ofrecen las soluciones óptimas, al parecer, aún así, los tiempos de los algoritmos nuevos son con creces superiores a Greedy o Búsqueda Local.

3.3 Dataset “Ecoli”.

El factor obtenido es $\lambda = 0.0268$, un problema con 8 clusters.
El tiempo utilizado para ejecutar todas estas pruebas sobre el conjunto de Ecoli ha sido aproximadamente de 19h, aunque el código ha sido optimizado, se puede observar claramente la ineficiencia de Python para este problema, comparando los tiempos de otros alumnos que utilizaron C++, estos tiempos son absurdos.

3.3.1. Análisis Tabla de Resultados Greedy.

10% COPKM	**ECOLI (k = 8)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	23.54	1326.00	59.12	208.91s
Ejecución 2	28.54	1526.00	69.49	201.19s
Ejecución 3	26.73	1622.00	70.25	447.20s
Ejecución 4	32.74	1316.00	68.05	252.68s
Ejecución 5	34.22	1470.00	73.66	145.05s
Media	29.16	1452.00	68.11	251.01s

20% COPKM	**ECOLI (k = 8)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	26.07	2574.00	60.60	139.98s
Ejecución 2	29.19	2760.00	66.22	198.87s
Ejecución 3	32.94	3356.00	77.96	332.54s
Ejecución 4	30.97	2294.00	61.75	258.97s
Ejecución 5	33.18	2704.00	69.45	252.50s
Media	30.47	2737.60	67.20	236.57s

3.3.2. Análisis Tabla de Resultados Búsqueda Local, Primer Mejor.

10% BL-PM	**ECOLI (k = 8)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	22.57	32.00	23.43	248.91s (22842 its)
Ejecución 2	22.72	36.00	23.69	210.88s (19171 its)
Ejecución 3	22.56	42.00	23.68	289.42s (27032 its)
Ejecución 4	22.58	34.00	23.50	292.90s (26797 its)
Ejecución 5	26.29	172.00	30.91	210.31s (19129 its)
Media	23.35	63.20	25.04	250.48s (22994 its)

20% BL-PM	**ECOLI (k = 8)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	22.49	84.00	23.58	256.63s (22880 its)
Ejecución 2	21.98	134.00	23.78	253.50s (23528 its)
Ejecución 3	22.39	90.00	23.60	185.05s (16341 its)
Ejecución 4	22.36	116.00	23.92	222.64s (20570 its)
Ejecución 5	22.27	86.00	23.43	186.59s (16803 its)
Media	22.29	102.0	23.66	220.88s (20024 its)

3.3.3. Análisis Tabla de Resultados Búsqueda Local, Mejor Vecino.

10% BL-MV	**ECOLI (k = 8)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	44.78	3040	126.25	901.13s (MAX its)
Ejecución 2				s (its)
Ejecución 3				s (its)
Ejecución 4				s (its)
Ejecución 5				s (its)
Media				s (its)

20% BL-MV	**ECOLI (k = 8)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1				s (its)
Ejecución 2				s (its)
Ejecución 3				s (its)
Ejecución 4				s (its)
Ejecución 5				s (its)
Media				s (its)

3.3.4. Análisis Tabla de Resultados A. Genético Generacional Elitista.

Se ha utilizado un elitismo de 1 elemento, es decir para cada “pase” de la población, el peor resultante se sustituye por el mejor de la generación anterior, aunque en mi código se ha generalizado para poder realizarse con un valor de elitismo N (genérico). Además, se ha utilizado una probabilidad de cruce de 0.7 (70%).

3.3.4.1. Algoritmo Genético Generacional Uniforme.

Se ha utilizado el cruce uniforme, con selección por torneo de dos individuos.

10% AGG-UN	**ECOLI (k = 8)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	25.33	132.00	28.88	1393.31s
Ejecución 2	26.62	92.00	29.09	1359.95s
Ejecución 3	27.91	176.00	32.63	1443.37s
Ejecución 4	26.68	216.00	32.47	1426.29s
Ejecución 5	23.44	146.00	27.36	1409.08s
Media	26.00	152.40	30.09	1.406.40s

20% AGG-UN	**ECOLI (k = 8)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	22.73	178.00	25.12	1641.93s
Ejecución 2	22.66	160.00	24.80	1637.19s
Ejecución 3	25.22	42.00	25.78	1647.52s
Ejecución 4	22.81	266.00	26.38	1615.09s
Ejecución 5	26.65	412.00	32.18	1560.60s
Media	24.04	211.60	26.85	1620.47s

3.3.4.2. Algoritmo Genético Generacional Uniforme.

Se ha utilizado el cruce por segmento fijo, con selección por torneo de dos individuos.

10% AGG-SF	**ECOLI (k = 8)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	29.14	226.00	36.27	1666.22s
Ejecución 2	26.14	680.00	44.39	1584.12s
Ejecución 3	30.35	406.00	41.24	1472.27s
Ejecución 4	32.11	892.00	56.05	1318.52s
Ejecución 5	33.32	560.00	48.35	1284.23s
Media	30.21	552.8	45.25	1465.08s

20% AGG-SF	**ECOLI (k = 8)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	23.30	98.00	24.62	1594.03s
Ejecución 2	25.40	380.00	30.49	1616.96s
Ejecución 3	32.84	630.00	41.29	1526.46s
Ejecución 4	32.71	956.00	45.53	1532.31s
Ejecución 5	25.42	448.00	31.43	1543.83s
Media	27.93	502.40	34.47	1562.72s

3.3.5. Análisis Tabla de Resultados A. Genético Estacionario.

La probabilidad de cruce es del 100%, eso significa que siempre se van a cruzar. Estos dos elementos del cruce de los padres se sustituyen por los peores elementos de la generación anterior, si son mejores.

3.3.5.1. Algoritmo Genético Estacionario Uniforme.

Se ha utilizado el cruce uniforme, con selección por torneo de dos individuos.

10% AGE-UN	**ECOLI (k = 8)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	22.52	32.00	23.38	1445.74s
Ejecución 2	22.84	30.00	23.64	1429.42s
Ejecución 3	22.60	38.00	23.62	1599.78s
Ejecución 4	22.72	22.00	23.31	1606.80s
Ejecución 5	22.71	36.00	23.68	1611.95s
Media	22.68	31.60	23.53	1538.74s

20% AGE-UN	**ECOLI (k = 8)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	22.74	86.00	23.89	1565.92s
Ejecución 2	22.42	118.00	24.00	1642.44s
Ejecución 3	26.10	218.00	29.03	1580.60s
Ejecución 4	22.52	94.00	23.78	1639.16s
Ejecución 5	22.50	100.00	23.84	1538.55s
Media	23.26	123.20	24.91	1593.33s

3.3.5.2. Algoritmo Genético Estacionario Segmento Fijo.

Se ha utilizado el cruce por segmento fijo, con selección por torneo de dos individuos.

10% AGE-SF	**ECOLI (k = 8)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	22.57	28.00	23.33	1058.78s
Ejecución 2	22.50	42.00	23.62	1561.43s
Ejecución 3	22.40	34.00	23.32	1511.74s
Ejecución 4	24.04	40.00	25.11	1644.01s
Ejecución 5	22.36	36.00	23.32	1530.31s
Media	22.77	36.00	23.74	1461.26s

20% AGE-SF	**ECOLI (k = 8)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	22.29	112.00	23.79	1570.46s
Ejecución 2	22.64	100.00	23.98	1562.18s
Ejecución 3	22.27	86.00	23.42	1557.52s
Ejecución 4	21.80	186.00	24.30	1551.36s
Ejecución 5	21.94	154.00	24.01	1596.61s
Media	22.19	164.80	23.90	1567.63s

3.3.6. Análisis Tabla de Resultados A. Meméticos.

Como hemos comentado anteriormente, vemos que AGE es mejor a AGG, además, utilizamos el algoritmo de cruce por segmento fijo.

3.3.6.1. Algoritmo Memético 10 Generaciones 100% Población.

Cada 10 iteraciones, vamos a aplicar Búsqueda Local Suave al 100% de la población.

10% AM-10,1.0	**ECOLI (k = 8)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	26.07	82.00	28.27	1503.61s
Ejecución 2	22.52	44.00	23.70	1436.76s
Ejecución 3	22.22	32.00	23.08	1449.09s
Ejecución 4	23.47	44.00	24.65	1526.41s
Ejecución 5	22.96	40.00	24.04	1538.25s
Media	23.49	48.40	24.75	1490.83

20% AM-10,1.0	**ECOLI (k = 8)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	22.42	82.00	23.52	1528.28s
Ejecución 2	22.51	92.00	23.75	1501.06s
Ejecución 3	22.03	114.00	23.56	1516.36s
Ejecución 4	22.40	62.00	23.23	1530.62s
Ejecución 5	22.55	100.00	23.89	1561.56s
Media	22.38	90.00	23.59	1527.58s

3.3.6.2. Algoritmo Memético 10 Generaciones 10% Población.

Cada 10 iteraciones, vamos a aplicar Búsqueda Local Suave al 10% de la población.

10% AM-10,0.1	**ECOLI (k = 8)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	22.55	44.00	23.70	1490.67s
Ejecución 2	22.59	32.00	23.44	1411.40s
Ejecución 3	22.43	26.00	23.13	1439.56s
Ejecución 4	22.58	34.00	23.50	1459.71s
Ejecución 5	26.25	86.00	28.56	1554.55s
Media	23.28	44.40	24.47	1471.19s

20% AM-10,0.1	**ECOLI (k = 8)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	22.12	138.00	23.97	1513.69s
Ejecución 2	22.30	90.00	23.50	1522.52s
Ejecución 3	22.12	104.00	23.51	1529.94s
Ejecución 4	22.71	78.00	23.76	1520.33s
Ejecución 5	22.64	100.00	23.98	1523.39s
Media	22.38	102.00	23.74	1521.97s

3.3.6.3. Algoritmo Memético 10 Generaciones Mejores 10% Población.

Cada 10 iteraciones, vamos a aplicar Búsqueda Local Suave al 10% de la mejor población.

10% AM-10,0.1 mej	**ECOLI (k = 8)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	22.38	42.00	23.51	1447.03s
Ejecución 2	26.54	86.00	28.85	1536.20s
Ejecución 3	22.62	42.00	23.75	1561.82s
Ejecución 4	22.55	34.00	23.65	1517.68s
Ejecución 5	22.90	28.00	23.56	1516.65s
Media	23.40	46.40	24.66	1515.88s

20% AM-10,0.1 mej	**ECOLI (k = 8)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	22.26	106.00	23.68	1528.29s
Ejecución 2	22.56	78.00	23.61	1528.01s
Ejecución 3	22.26	108.00	23.71	1529.17s
Ejecución 4	22.30	102.00	23.67	1519.17s
Ejecución 5	22.45	106.00	23.88	1525.64s
Media	22.37	100.00	23.71	1526.06s

3.3.7. Análisis Tabla de Resultados Medios.

10%	**ECOLI (k = 8)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
COPKM	29.16	1452.00	68.11	251.01s
BL-PM	23.35	63.20	25.04	250.48s (22994 its)
BL-MV				
AGG-UN	26.00	152.40	30.09	1.406.40s
AGG-SF	30.21	552.8	45.25	1465.08s
AGE-UN	22.68	31.60	23.53	1538.74s
AGE-SF	27.93	502.40	34.47	1562.72s
AM-10,1.0	23.49	48.40	24.75	1490.83
AM-10,0.1	23.28	44.40	24.47	1471.19s
AM-10,0.1mej	23.40	46.40	24.66	1515.88s

20%	**ECOLI (k = 8)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
COPKM	30.47	2737.60	67.20	236.57s
BL-PM	22.29	102.0	23.66	220.88s (20024 its)
BL-MV				
AGG-UN	24.04	211.60	26.85	1620.47s
AGG-SF	27.93	502.40	34.47	1562.72s
AGE-UN	23.26	123.20	24.91	1593.33s
AGE-SF	22.19	164.80	23.90	1567.63s
AM-10,1.0	22.38	90.00	23.59	1527.58s
AM-10,0.1	22.38	102.00	23.74	1521.97s
AM-10,0.1mej	22.37	100.00	23.71	1526.06s

En este problema, un problema ya complejo como habíamos estudiado anteriormente, vemos que los resultados ya son distintos entre sí.

Para las restricciones del 10% , vemos que AGE de media es superior a AGG, razón por la que se optó por utilizar AGE para AM. Entre los dos algoritmos de AGE, vemos un claro ganador utilizando la técnica de cruce uniforme, esto puede ser ya que el cruce uniforme da más variabilidad entre los cromosomas, ya que elige genes de forma uniforme sin importar la vecindad de estos, mientras que por segmento fijo, si que lo tiene en cuenta, haciendo que sea más probable que dos genes vecinos sean heredados por un hijo y si estos sean malos, también arrastre el error.

Comentar además que no se ve una mejora de AM sobre AGE, esto puede ser debido a que las semillas (que son las mismas que para la práctica anterior), puedan haber añadido este error, pudiendo rechazar la hipótesis de que AM sea peor que AGE.

De la misma forma, no podemos decir nada definitivo sobre AM con diferentes parámetros ya las diferencias son décimas, lo que puede haber causado la semilla.

Uno de los motivos por el cual probablemente AM no sea mejor que AGE es debido a que AM ha sido ejecutado con AGE, como cada 10 iteraciones se aplica BLS sobre una población que varía muy poco, ya que AGE solo varía 2 cromosomas por pasada, probablemente BLS tenga mucha más fuerza sobre AGE y AGE solo se utilice como regularizador para BLS.

Este análisis puede realizarse de la misma forma sobre el 20% ya que los resultados son muy similares, quizás habiendo utilizado AGG en vez de AGE para AM, todo esto hubiera cambiado y se hubieran apreciado grandes cambios.

Los tiempos de ejecución son muy altos, los motivos son principalmente la generación de números aleatorios y que Python es un lenguaje interpretado, requiriendo más pasos para la ejecución.

Realizando un análisis global sobre los resultados, estos parecen buenos en general utilizando las diferentes técnicas bio-inspiradas. Comparando los tiempos con otros alumnos, explotariamos mucho este algoritmo si lo hubiéramos escrito en C++.

3.4 Dataset “Newthyroid”.

El factor obtenido es $\lambda = 0.0365$, Con una agrupación de 3 clusters.
Como este dataset también tiene una alta complejidad, se han ejecutado todas las pruebas en aproximadamente 14 horas. Como se ha comentado anteriormente, Python no es una buena referencia para realizar este problema.

3.4.1. Análisis Tabla de Resultados Greedy.

10% COPKM	**NEWTHYROID (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	13.21	1034.00	51.03	8.67s
Ejecución 2	12.84	750.00	40.27	5.48s
Ejecución 3	13.37	782.00	41.97	5.43s
Ejecución 4	12.91	720.00	39.24	4.33s
Ejecución 5	10.32	1398.00	61.45	4.51s
Media	12.53	936.80	46.79	5.68s

20% COPKM	**NEWTHYROID (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	12.95	1802.00	45.89	9.33s
Ejecución 2	12.99	1366.00	37.96	4.58s
Ejecución 3	13.07	1448.00	39.55	5.61s
Ejecución 4	12.84	1510.00	40.45	4.60s
Ejecución 5	13.36	2506.00	59.17	4.73s
Media	13.04	1726.40	44.60	5.77s

3.4.2. Análisis Tabla de Resultados Búsqueda Local, Primer Mejor.

10% BL-PM	**NEWTHYROID (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	14.09	4.00	14.23	9.42s (2740 its)
Ejecución 2	14.09	4.00	14.23	9.87s (2883 its)
Ejecución 3	10.09	194.00	17.99	11.14s (3466 its)
Ejecución 4	14.09	4.00	14.23	8.84s (2703 its)
Ejecución 5	10.87	192.00	17.90	15.37s (4929 its)
Media	12.81	79.6	15.72	10.93s (3344 its)

20% BL-PM	**NEWTHYROID (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	14.29	0.00	14.29	8.43s (2499 its)
Ejecución 2	10.81	462	19.26	7.95s (2413 its)
Ejecución 3	10.86	462	19.30	11.16s (3374 its)
Ejecución 4	14.29	0.00	14.29	8.13s (2481 its)
Ejecución 5	14.29	0.00	14.29	7.30s (2190 its)
Media	12.91	184.8	16.28	8.59s (2591 its)

3.4.3. Análisis Tabla de Resultados Búsqueda Local, Mejor Vecino.

10% BL-MV	**NEWTHYROID (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	14.09	4.00	14.23	180.81s (62780 its)
Ejecución 2	14.09	4.00	14.23	204.91s (69660 its)
Ejecución 3	14.09	4.00	14.23	215.48s (70520 its)
Ejecución 4	14.09	4.00	14.23	181.88s (60630 its)
Ejecución 5	14.09	4.00	14.23	196.49s (66220 its)
Media	14.09	4.00	14.23	195.91s (65962 its)

20% BL-MV	**NEWTHYROID (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	14.29	0.0	14.29	183.64s (62350 its)
Ejecución 2	14.29	0.0	14.29	201.02s (66650 its)
Ejecución 3	14.29	0.0	14.29	187.91s (61490 its)
Ejecución 4	14.29	0.0	14.29	178.54s (58480 its)
Ejecución 5	14.29	0.0	14.29	184.76s (60200 its)
Media	14.29	0.0	14.29	187.17s (61834 its)

3.4.4. Análisis Tabla de Resultados A. Genético Generacional Elitista.

Se ha utilizado un elitismo de 1 elemento, es decir para cada “pase” de la población, el peor resultante se sustituye por el mejor de la generación anterior, aunque en mi código se ha generalizado para poder realizarse con un valor de elitismo N (genérico). Además, se ha utilizado una probabilidad de cruce de 0.7 (70%).

3.4.4.1. Algoritmo Genético Generacional Uniforme.

Se ha utilizado el cruce uniforme, con selección por torneo de dos individuos.

10% AGG-UN	**NEWTHYROID (k = 8)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	10.87	192.00	17.90	852.54s
Ejecución 2	10.62	192.00	17.65	886.86s
Ejecución 3	14.09	4.00	14.23	893.45s
Ejecución 4	14.09	4.00	14.23	878.39s
Ejecución 5	14.09	4.00	14.23	874.12s
Media	12.75	79.20	15.65	877.07s

20% AGG-UN	**NEWTHYROID (k = 8)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	14.29	0.0	14.29	869.90s
Ejecución 2	14.29	0.0	14.29	939.57s
Ejecución 3	14.29	0.0	14.29	864.11s
Ejecución 4	14.29	0.0	14.29	872.16s
Ejecución 5	14.29	0.0	14.29	873.99s
Media	14.29	0.0	14.29	883.74s

3.4.4.2. Algoritmo Genético Generacional Uniforme.

Se ha utilizado el cruce por segmento fijo, con selección por torneo de dos individuos.

10% AGG-SF	**NEWTHYROID (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	14.09	4.00	14.23	856.71s
Ejecución 2	14.09	4.00	14.23	874.82s
Ejecución 3	14.09	4.00	14.23	871.43s
Ejecución 4	14.09	4.00	14.23	873.13s
Ejecución 5	10.71	190.00	17.66	877.08s
Media	13.41	41.20	14.92	870.87s

20% AGG-SF	**NEWTHYROID (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	14.29	0.0	14.29	921.02s
Ejecución 2	14.29	0.0	14.29	904.63s
Ejecución 3	14.29	0.0	14.29	909.41s
Ejecución 4	14.29	0.0	14.29	921.99s
Ejecución 5	14.29	0.0	14.29	967.41s
Media	14.29	0.0	14.29	924.89s

3.4.5. Análisis Tabla de Resultados A. Genético Estacionario.

La probabilidad de cruce es del 100%, eso significa que siempre se van a cruzar. Estos dos elementos del cruce de los padres se sustituyen por los peores elementos de la generación anterior, si son mejores.

3.4.5.1. Algoritmo Genético Estacionario Uniforme.

Se ha utilizado el cruce uniforme, con selección por torneo de dos individuos.

10% AGE-UN	**NEWTHYROID (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	14.09	4.00	14.23	855.90s
Ejecución 2	14.09	4.00	14.23	903.07s
Ejecución 3	14.09	4.00	14.23	906.87s
Ejecución 4	14.09	4.00	14.23	903.60s
Ejecución 5	10.87	196.00	18.04	893.14s
Media	13.45	42.40	14.99	892.52s

20% AGE-UN	**NEWTHYROID (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	10.82	458.00	19.20	940.31s
Ejecución 2	14.29	0.00	14.29	940.70s
Ejecución 3	14.29	0.00	14.29	931.66s
Ejecución 4	14.29	0.00	14.29	943.66s
Ejecución 5	14.29	0.00	14.29	934.87s
Media	12.59	91.60	15.27	938.24

3.4.5.2. Algoritmo Genético Estacionario Segmento Fijo.

Se ha utilizado el cruce por segmento fijo, con selección por torneo de dos individuos.

10% AGE-SF	**NEWTHYROID (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	14.09	4.00	14.23	876.85s
Ejecución 2	14.09	4.00	14.23	900.93s
Ejecución 3	14.09	4.00	14.23	907.52s
Ejecución 4	14.09	4.00	14.23	904.56s
Ejecución 5	10.73	196.00	10.73	895.95s
Media	13.42	42.40	13.53	897.16s

20% AGE-SF	**NEWTHYROID (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	14.29	0.00	14.29	964.09s
Ejecución 2	14.29	0.00	14.29	948.64s
Ejecución 3	14.29	0.00	14.29	954.55s
Ejecución 4	14.29	0.00	14.29	954.64s
Ejecución 5	14.29	0.00	14.29	951.35s
Media	14.29	0.00	14.29	954.65s

3.4.6. Análisis Tabla de Resultados A. Meméticos.

Como hemos comentado anteriormente, vemos que AGE es mejor a AGG, además, utilizamos el algoritmo de cruce por segmento fijo.

3.4.6.1. Algoritmo Memético 10 Generaciones 100% Población.

Cada 10 iteraciones, vamos a aplicar Búsqueda Local Suave al 100% de la población.

10% AM-10,1.0	**NEWTHYROID (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	14.09	4.00	14.23	891.87s
Ejecución 2	14.09	4.00	14.23	911.03s
Ejecución 3	14.09	4.00	14.23	863.49s
Ejecución 4	14.09	4.00	14.23	881.45s
Ejecución 5	10.87	192.00	17.90	904.45s
Media	13.45	41.60	14.96	890.46s

20% AM-10,1.0	**NEWTHYROID (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	14.29	0.00	14.29	869.90s
Ejecución 2	14.29	0.00	14.29	873.99s
Ejecución 3	14.29	0.00	14.29	864.11s
Ejecución 4	14.29	0.00	14.29	872.16s
Ejecución 5	14.29	0.00	14.29	913.21s
Media	14.29	0.00	14.29	878.67s

3.4.6.2. Algoritmo Memético 10 Generaciones 10% Población.

Cada 10 iteraciones, vamos a aplicar Búsqueda Local Suave al 10% de la población.

10% AM-10,0.1	**NEWTHYROID (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	14.09	4.00	14.23	883.19s
Ejecución 2	14.09	4.00	14.23	842.23s
Ejecución 3	14.09	4.00	14.23	853.17s
Ejecución 4	10.90	194.00	17.99	826.72s
Ejecución 5	10.89	190.00	17.84	842.37s
Media	12.81	79.20	15.70	849.54s

20% AM-10,0.1	**NEWTHYROID (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	10.78	466.00	19.30	899.66s
Ejecución 2	14.29	0.00	14.29	840.75s
Ejecución 3	14.29	0.00	14.29	907.08s
Ejecución 4	14.29	0.00	14.29	863.53s
Ejecución 5	14.29	0.00	14.29	858.47s
Media	13.59	93.20	15.29	873.90s

3.4.6.3. Algoritmo Memético 10 Generaciones Mejores 10% Población.

Cada 10 iteraciones, vamos a aplicar Búsqueda Local Suave al 10% de la mejor población.

10% AM-10,0.1 mej	**NEWTHYROID (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	10.87	192.00	17.90	894.75s
Ejecución 2	14.09	4.00	14.23	885.97s
Ejecución 3	14.09	4.00	14.23	875.02s
Ejecución 4	14.09	4.00	14.23	854.12s
Ejecución 5	10.88	196.00	18.04	900.40s
Media	12.80	80.00	15.72	882.05s

20% AM-10,0.1 mej	**NEWTHYROID (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	14.29	0.00	14.29	931.42s
Ejecución 2	14.29	0.00	14.29	937.77s
Ejecución 3	14.29	0.00	14.29	928.37s
Ejecución 4	14.29	0.00	14.29	931.41s
Ejecución 5	14.29	0.00	14.29	929.83s
Media	14.29	0.00	14.29	931.76s

3.4.7. Análisis Tabla de Resultados Medios.

10%	**NEWTHYROID (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
COPKM	12.53	936.80	46.79	5.68s
BL-PM	12.81	79.6	15.72	10.93s (3344 its)
BL-MV	14.09	4.00	14.23	195.91s (65962 its)
AGG-UN	12.75	79.20	15.65	877.07s
AGG-SF	13.41	41.20	14.92	870.87s
AGE-UN	13.45	42.40	14.99	892.52s
AGE-SF	13.42	42.40	13.53	897.16s
AM-10,1.0	13.45	41.60	14.96	890.46s
AM-10,0.1	12.81	79.20	15.70	849.54s
AM-10,0.1mej	12.80	80.00	15.72	882.05s

20%	**NEWTHYROID (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
COPKM	13.04	1726.40	44.60	5.77s
BL-PM	12.91	184.8	16.28	8.59s (2591 its)
BL-MV	14.29	0.0	14.29	187.17s (61834 its)
AGG-UN	14.29	0.0	14.29	883.74s
AGG-SF	14.29	0.0	14.29	924.89s
AGE-UN	12.59	91.60	15.27	938.24s
AGE-SF	14.29	0.00	14.29	954.65s
AM-10,1.0	14.29	0.00	14.29	878.67s
AM-10,0.1	13.59	93.20	15.29	873.90s
AM-10,0.1mej	14.29	0.00	14.29	931.76s

Para este problema podemos observar que los tiempos de ejecución se han reducido, pudiendo afirmar que parece un problema más sencillo. Además conocemos que este dataset contiene dos posibles soluciones óptimas, por lo que las medias de las tasas podrían ser muy dispares, teniendo que observar únicamente el agregado.

En primer lugar sobre el 10% de restricciones vemos que todos los algoritmos bioinspirados son similares, aunque AGE un poco mejor, aunque no ocurre lo mismo con 20%, parece que AGG es algo mejor aunque puede ser causa de la semilla.

Vemos que con AM obtenemos peores resultados, esto puede ser debido a lo que hemos comentado anteriormente, se explota aún más la búsqueda suave local que AGE, donde el regulador puede tener menor peso.

Podemos finalizar diciendo todos los algoritmos son bastante buenos, los tiempos similares, para este problema podríamos tener alguna duda si utilizar AGE o AGG, aunque por la categoría de los problemas, se premia la localidad, por lo que elegiría AGE.

4. Bibliografía

1. *K. Alsabti, S. Ranka, V. Singh*. (**1998**). **An Efficient K-Means Clustering Algorithm**. "*Inthispaper, we present a novel algorithm for performing k-means clustering. It organizes all the patterns in a k-d tree structure such that one can find all the patterns which are closest to a given prototype efficiently. The main intuition behind our approach is as follows. All the prototypes are potential candidates for the closest prototype at the root level.However, for the children of the root node, we may be able to prune the candidate set by using si...*". Obtenido de https://www.cs.utexas.edu/~kuipers/readings/Alsabti-hpdm-98.pdf
2. *P. Krömer, H. Zhang, Y. Liang, J-S. P. S*. (**2018**) .**Proceedings of the Fifth Euro-China Conference on Intelligent Data Analysis**. *"This volume of Advances in Intelligent Systems and Computing highlights papers presented at the Fifth Euro-China Conference on Intelligent Data Analysis and Applications (ECC2018), held in Xi'an, China from October 12 to 14 2018. The conference was co-sponsored "* Obtenido de https://books.google.es/books?id=GzKBDwAAQBAJ&pg=PA513&lpg=PA513&dq=newthyroid+dataset+kmean&source=bl&ots=04b14UkELW&sig=ACfU3U1C4JwYT8QMCkFrC79eNn-ShPEDVA&hl=es&sa=X&ved=2ahUKEwiQjomnm67oAhWcA2MBHVgVCAwQ6AEwCnoECAkQAQ#v=onepage&q=newthyroid%20dataset%20kmean&f=false
3. *C. Blum, A. Roli* (**2003**). **Metaheuristics in Combinatorial Optimization: Overviewand Conceptual Comparison**. "*The field of metaheuristics for the application to combinatorial optimization problems isa rapidly growing field of research. This is due to the importance of combinatorialoptimization problems for the scientific as well as the industrial world...*" Obtenido de https://www.iiia.csic.es/~christian.blum/downloads/blum_roli_2003.pdf