METAHEURÍSTICAS

PRÁCTICA 3: Búsqueda por Trayectorias

UNIVERSIDAD
DE GRANADA

Problema: Agrupamiento con Restricciones

Alba Casillas Rodríguez
76738108B
3ºA – Grupo de Prácticas 1 (Miércoles 17:30-19:30)
albacaro@correo.ugr.es

Índice

1 - Descripción del problema

PROBLEMA DEL AGRUPAMIENTO CON RESTRICCIONES

El agrupamiento, también llamado ***clustering***, es una tarea de aprendizaje no supervisado que consiste en agrupar un conjunto de objetos de tal manera que los objetos que estén en el mismo ***grupo (cluster)*** sean más ***similares*** entre sí que con los de otros grupos (clusters).

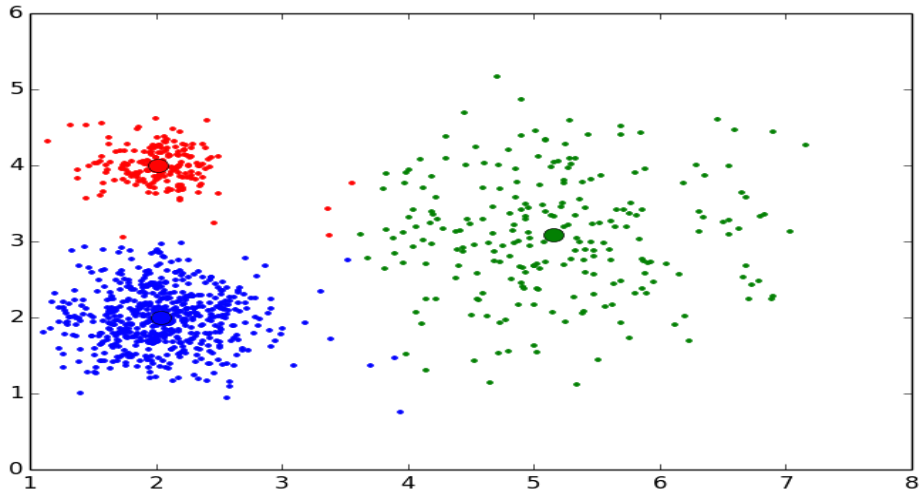
Esta similitud entre los objetos está definido mediante una ***distancia***, normalmente la Euclídea; por lo que haremos uso del término ***centroide***, siendo este el centro geométrico del cluster.

En nuestro problema, al clustering clásico le añadiremos restricciones; convirtiendo este problema en uno de aprendizaje semi-supervisado.

La partición de nuestro grupo de datos, deberá tener en cuenta ***restricciones fuertes***, que obligan a que todos los clusters deben contener al menos una instancia, cada instancia debe pertenecer a un único cluster, y la unión de los clusters debe ser el conjunto de datos.

Además, también deberán cumplirse unas ***restricciones débiles***, donde la partición del conjunto de datos debe minimizar el número de restricciones incumplidas (pudiendo incumplir algunas). Estas restricciones consisten en que dada una pareja de instancias, algunas deberán pertenecer al mismo cluster (***must-link***); y otras, por el contrario, no podrán estar en un mismo cluster (***cannot-link***).

El clustering es una tarea muy importante en exploración de data mining, y una técnica común en el análisis estadístico de datos, aprendizaje automático, reconocimiento de imágenes, etc.



2 - Consideraciones comunes de los algoritmos

- ***2.1- Representación de la solución :*** Representamos nuestra solución como un vector de tamaño "n", cuyos valores van desde 1 hasta "k", siendo este el número de clusters definido.

$$\text{Solución} = \{S_1 , S_2 , \dots , S_n\} \setminus S_i \in K$$

Inicialmente, esta solución es creada mediante un generador de números aleatorios.

Deberemos asegurarnos de que nuestra solución sea **factible**, es decir, no dejaremos ningún cluster vacío.

- ***2.2 – Función objetivo :*** Será nuestra función a minimizar. Está se calculará mediante la fórmula:

$$f = C + (\text{infeasibility} * \lambda)$$

- C será la desviación general.
- Infeasibility será el número de restricciones incumplidas.
- λ(lambda) será el parámetro de escalado para dar relevancia al infeasibility.

- ***2.3 – Cálculo del Infeasibility*** *[Modificada con respecto a la práctica1]:* Llamaremos infeasibility al número de restricciones incumplidas dada una pareja de instancias del conjunto de datos.

$$\text{Infeasibility} = \Sigma^{n}_{i=0} \Sigma^{n}_{j=i+1} V(x_i, x_j)$$

Aunque sea una de las funciones utilizadas en la práctica 1, para esta práctica he modificado la estructura de datos usada para guardar las restricciones, con motivo de agilizar las ejecuciones de conjuntos de datos.

He sustituido la matriz de restricciones por una lista que almacena datos de la manera:
[dato1, dato2, valor] , donde valor toma los valores: 1 (ML) o -1 (CL)

De esta manera, el valor de infeasibility pasa a ser calculado de la siguiente manera:

```
def calcular_infeasibility(datos, v_solucion, restricciones):
    para i ∈ {0...len(restricciones)}:
        si  restricciones[i][2] == -1 && v_solucion[i][0] == v_solucion[i][1]:
            # INCUMPLE CANNOT-LINK
            infeasibility++
        si  restricciones[i][2] == 1 && v_solucion[i][0] != v_solucion[i][1]:
            # INCUMPLE CANNOT-LINK
            infeasibility++
```

- *2.4 – Generación de la solución aleatoria:* La solución inicial estará formada por un vector de tamaño “n” cuyos valores serán aleatorios desde 1 hasta “k”, siendo este el número de clusters.

```
def generar_solucion_aleatoria(k, n):
    solucion = []

    para i ∈ {0...n-1}:
        solucion[i] = random{1..k}

    reparar(solucion,n)

    return solucion
```

donde reparar es una función auxiliar, la cual se encargará de revisar si hay algún cluster vacío. En el caso de haberlo, realizará una reparación en la que se le asignará a dicho cluster una posición aleatoria de nuestra solución. Cada vez que hagamos una reparación, volveremos a comprobar si hay más clusters vacíos o si alguno se vació al reparar otro; para que la solución sea factible.

3 – Descripción de los algoritmos: Esquema general

ENFRIAMIENTO SIMULADO

Es un algoritmo ***basado en trayectorias simples*** inspirada en la termodinámica y donde se realiza una exploración del entorno en busca de una solución mejor, permitiendo aún así seleccionar soluciones peores a la actual siguiendo el criterio: a mayor temperatura, mayor será la posibilidad de aceptar una solución peor. Con esto se consigue un equilibrio exploración-explotación, puesto que al principio favorecemos la exploración eligiendo soluciones peores pero, a medida que la temperatura disminuye, se explota más, eligiendo primordialmente soluciones mejores.

```
def Enfriamiento_Simulado:
    # Nuestra solución aleatoria inicial será ahora mismo nuestra mejor solución
    # hasta el momento
    solucion ← generar_solucion_aleatoria(n,k)
    centroides ← recalcular_centroides(datos, solucion, centroides)
    coste_sol ← f_objetivo(datos, restricciones, solucion, centroides,n,d,lambda)

    mejor_solucion ← solucion #Actualizamos también su coste y centroides

    # Calculamos la temperatura inicial
    temperatura_inicial ← calcular_temperatura(mu,phi, temperatura, coste_sol)
    temperatura ← temperatura_inicial

    mientras que num_exitos != 0 && num_evaluaciones <= MAX_EVAL:
        num_exitos = 0 num_vecinos = 0

        mientras que num_vecinos < max_vecinos && num_extiso < max_exitos:
            # En cada iteración del bucle se genera una única solución vecina
            # aleatoria que compararemos con la actual.
            # Nuestro operador de vecino será el cambio de clúster;
            # le asignaremos a una posición aleatoria un cluster de valor
            # diferente al suyo actual
            vecino ← solucion
            posicion ← random(0,(n-1))

            valor ← vecino[posicion]
            nuevo_valor ← random(1,k)

            mientras que nuevo_valor == valor:
                nuevo_valor ← random(1,k)

            vecino[posicion] ← nuevo_valor
```

```
                reparar(vecino, n) # Comprobamos que la solución es FACTIBLE
                num_vecinos++

                centroides_vecino ← recalcular_centroides(datos,vecino...)
                coste_vecino ← f_objetivo(datos,restricciones,vecino....)
                num_evaluaciones++

                diferencia ← coste_vecino – coste_solucion
                aleatorio ← random

                si diferencia < 0 || aleatorio <= exp(-diferencia/temperatura):
                        solucion ← vecino # Actualizamos centroides y coste
                        num_exitos++

                        si coste_solucion < mejor_coste:
                                # Actualizamos centroides y coste
                                mejor_solucion ← solucion


        temperatura ← esquema_enfriamiento(temperatura)

    return mejor_solucion
```

BÚSQUEDA MULTIARRANQUE

Se trata de una técnica basada en trayectorias múltiples que generará un determinado número de soluciones aleatorias iniciales (en nuestro caso, 10) que serán sometidas a una búsqueda local, quedándonos con la mejor de las soluciones obtenidas.

```
def Busqueda_Multiarranque:
    mejor_solucion ← [] , mejor_coste ← INF
    para i ϵ {0...MAX_ITERACIONES}:
        solucion ← generar_solucion_aleatoria(n,k)
        centroides ← recalcular_centroides(datos,solucion,centroides)

        # A cada solución aleatoria le aplicaremos una Búsqueda Local
        solucion, coste_sol, centroides = BusquedaLocal(datos,restricciones...)

        si coste_sol < mejor_coste:
            #Actualizaremos los centroides y el coste
            mejor_solucion ← solucion

    return mejor_solucion
```

BÚSQUEDA LOCAL ITERATIVA

Es un algoritmo de búsqueda basada en trayectorias múltiples donde se permite evitar óptimos locales aplicando una mutación sobre la mejor solución en el momento. Nosotros hemos implementado dos versiones del mismo: el ILS que utilizará la búsqueda local para explorar el entorno, y el ILS-ES, donde cambiamos la búsqueda local por nuestro algoritmo de enfriamiento simulado.

```
def Busqueda_Local_Iteartiva:
    solucion_inicial ← generar_solucion_aleatoria(n,k)
    centroides_ini ← recalcular_centroides(datos, solucion_ini, centroides)
    coste_ini ← f_objetivo(datos, restricciones, solucion_inicial, centroides_ini...)

    # Si nos encontramos en el ILS ejecutaremos la búsuqeda local, pero si
    # estamos ejecutando ILS_ES ejecutaremos el algoritmo ES
    si ES == false:
        solucion, coste_sol, centroides = BusquedaLocal(solucion_inicial, datos...)
    si no:
        solucion, coste_sol, centroides = Enf_Simulado(solucion_inicial, datos...)

    mejor_solucion ← solucion # La de menor coste entre solucion y solucion_ini

    para i ϵ {0...MAX_ITERACIONES}:
        # Aplicamos la mutación
        solucion_mutada ← mutacion(mejor_solucion)
        centroides_mutados ← recalcular centroides
        coste_mutado ← f_objetivo(solucion_mutada, centroides_mutados...)

    si ES == false:
        solucion, coste_sol, centroides = BusquedaLocal(solucion_mutada...)
    si no:
        solucion, coste_sol, centroides = Enf_Simulado(solucion_mutada, datos...)

    if coste_sol < mejor_coste:
        # Actualizamos también los centroides y coste
        mejor_solucion ← coste_sol
```

3 – Descripción de los algoritmos: Particularidades

3.2.1 – ES , temperatura inicial y esquema de enfriamiento :

Para la temperatura inicial, utilizaremos la siguiente fórmula:

$$T_0 = \frac{\mu \cdot C(S_0)}{-\ln(\phi)}$$

donde mu y phi valdrán 0.3. Por tanto:

temperatura_inicial = (mu * coste_solucion) / -(np.log(phi))

El esquema de enfriamiento empleará el esquema de Cauchy modificado:

$$T_{k+1} = \frac{T_k}{1 + \beta \cdot T_k} \quad ; \quad \beta = \frac{T_0 - T_f}{M \cdot T_0 \cdot T_f}$$

de manera que (temp_final = 10^{-3}):

valor_M = NUM_MAX_EVALUACIONES / max_vecinos
beta = temp_inicial – temp_final / valor_M*temp_inicial*temp_final

temperatura = temperatura / (1 + beta * temperatura)

3.2.2 – BL, exploración del entorno y operador generador de vecinos :

La búsqueda local será la misma planteada en la práctica 1, cuyo pseudocódigo será el mismo (habiendo arreglado sus fallos):

```
def BusquedaLocal(datos, restricciones, n, d, solucion, lambda):
    num_evaluaciones ← 0
    hay_cambio ← True

    mientras que (hay_cambio) && (num_iteraciones < MAX_ITERACIONES):
        hay_cambio ← False
        # EXPLORACIÓN DEL VECINDARIO Y OPERADOR GENERADOR DE VECINO
        # Realizaremos la técnica del cambio de cluster para generar todos los
        # posibles vecinos, cambiando de cluster cada elemento de la solución
```

```
for i ϵ {0...n}:
        for j i ϵ {0...k}:
                Si j ! = v_solucion[i]: #Si no es el cluster en el que ya está
                        vecinos  ← v_solucion

                        # Generamos el vecindario asignando a cada elemento
                        # un cluster nuevo
                        vecinos.back()[i] = j

                        #A no ser que el cluster se quede vacío, donde
                        # generaríamos una solución infactible y por tanto, no
                        # tendremos en cuenta este vecino borrándolo
                        si cluster_vacio():
                                borrar vecino

        #Para no coger siempre los primeros vecinos:
        shuffle(vecinos)


        #Comparamos el valor de la función objetivo para la solución actual
        # con el de cada vecino hasta encontrar uno que sea mejor.
        #Como nos basamos en una búsqueda local de EL PRIMERO EL
        # MEJOR, en cuanto encontremos una solución que mejore la
        # actual, nos quedaremos con ella
        for v ϵ vecinos:
                centroide_vecino ← recalcular_centroides(vecino[v]...)
                sol1  ←  funcion_objetivo(vecino)
                sol2  ←  funcion_objetivo(v_solucion)

                si sol1 < sol2
                        v_solucion ←  v
                        hay_cambio ← TRUE
                        break

return v_solucion
```

3.2.3 – ILS, operador de mutación :

Para salir de óptimos locales realizaremos un cambio brusco en la solución aplicando una mutación basada en una mutación por segmento. Seleccionaremos un segmento de longitud fija de la solución a mutar, y se le aplicará desde una posición aleatoria.

A este segmento le reasignaremos de forma aleatoria las etiquetas de las instancias asociadas.

```
def mutacion(solucion,n):
    sol_aux ← solucion
    inicio_seg ← random(0,(n-1))
    tamanio ← int(0.1*n)
    fin_seg ← ((inicio_seg + tamanio)mod n)

    contador ← inicio_seg

    # Reasignaremos las etiquetas de todo el segmento elegido
    mientras que contador != fin_seg:
        valor ← random(1,k)
        sol_aux[contador] ← valor
        contador ← (contador+1)mod n

    # Comprobaremos que la solución sea factible
    reparar(sol_aux,n)

    return sol_aux
```

4 – Algoritmos comparativos

ALGORITMO GREEDY

La solución greedy para este problema estará basado en el algoritmo ***k-medias,*** donde además se tendrá en cuenta el cumplimiento de las ***restricciones*** débiles.

K-medias es un método de agrupamiento en el cual se divide un conjunto de n objetos en k grupos, donde cada objeto pertenece al gurpo cuyo valor medio es más cercano.

Por ello, inicialmente barajaremos los índices para recorrer el conjunto de datos de forma aleatoria sin repetición y asignaremos cada instancia al clúster más cercano de entre los que produzcan un menor aumento en el valor de infeasibility.

Guardaremos la ***infeasibility*** de asignar a cada instancia los clústeres, para que de haber varios con infeasibility minima, podamos elegir aquel que tiene menor distancia dato-centroide.

Una vez hemos obtenido la nueva partición, se deberá ***re-calcular los centroides.***

Iteraremos el algoritmo mientras no haya cambio en la partición.

```
def GREEDY(datos, restricciones, centroides, n, d):
        random.shuffle(rsi) # rsi será un vector cuyos valores van de 0 a n

        mientras que ( hay_cambio ) && ( num_iteraciones <
MAX_ITERACIONES=10000):
                num_iteraciones++
                anterior_solucion ← v_solucion

                for i ϵ {0...rsi}:
                        for j i ϵ {0...k}:
                                v_solucion[i] ←  j #Asignamos a cada instancia un cluster "j"
                                infeasibility ← caculamos infeasibility(v_solucion)

                                #Guardo en un vector los valores de infeasibility de asignar a
una                                     #instancia X_i al cluster j
                                v_infeasibility[j] ← infeasibility
```

```
                #De los valores de v_infeasibility, guardaremos en un vector auxiliar
                        #(v_minimos) las posiciones de los valores mínimos.
                for c ∈ {0...k}:
                        si v_infeasibility[c] < minimo:
                                minimo ← valor_infeasibility
                                v_minimos ← posicion


                #Tras obtener las posiciones cuyo valor del infeasibility es mínimo.
                # Nos quedaremos con el centroide que tenga la menor distancia
                for c ∈ {0...len(v_minimos)}:
                        distancia ← distancia(datos,centroides)
                        v_distancias[c] ← distancia

                for c ∈ {0...len(v_distancias)}:
                        si v_distancias[c] < minimo:
                                minimo ← v_distancias[c]
                                centroide_cercano ← v_minimos[c]

                #Una vez obtenemos el centroide más cercano con menor valor de
                        #infeasibility, lo asignamos al vector solución

                v_solucion[i] ← centroide_cercano

        #Recalculamos centroides
        recalcular_centrodies(datos, v_solucion, centroides, d)

        #Comprobamos si ha habido un cambio en la partición
        cambio ← cambio(v_solucion, anterior_solucion)

return v_solucion
```

5 – Manual de usuario

Para ejecutar la práctica solo debe indicarse qué conjunto de datos se desea ejecutar.

Tras indicar el conjunto de datos, se ejecutarán ambos conjuntos de restricciones 5 veces.

NO HACE FALTA INDICAR LAS SEMILLAS.

Las semillas utilizadas: (1,3,2,7, 5) han sido guardadas en un vector, de manera que por cada ejecución se seleccionará una semilla diferente.

Esta práctica se ha implementado en python3.

Para ejecutar escribir en terminal: python3 practica$_3$.py

** Cabe destacar que en el programa ejecutable se ha comentado el código con el que se realizan de manera automática las tablas y gráficas proporcionadas para el análisis de resultados.

6 – Tablas de resultados

6.1 – Enfriamiento Simulado (ES)

Resultados obtenidos en el ES con 10% de restricciones.

	Iris				Ecoli				Rand				Newthyroid			
	Tasa_C	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T
Ejecución 1	0,66	0,00	0,66	21,25	21,63	40,00	22,69	315,47	0,71	0,00	0,71	18,19	13,83	6,00	14,05	161,95
Ejecución 2	0,66	0,00	0,66	24,98	21,51	55,00	22,98	440,61	0,71	0,00	0,71	14,29	13,83	6,00	14,05	138,59
Ejecución 3	0,66	0,00	0,66	27,19	21,17	50,00	22,50	474,22	0,71	0,00	0,71	17,72	13,83	6,00	14,05	119,92
Ejecución 4	0,66	0,00	0,66	23,65	21,58	45,00	22,78	608,27	0,71	0,00	0,71	16,67	13,83	6,00	14,05	143,64
Ejecución 5	0,66	0,00	0,66	26,07	21,51	47,00	22,75	423,18	0,71	0,00	0,71	21,20	13,83	6,00	14,05	152,30
Media	0,66	0,00	0,66	24,63	21,48	47,40	22,74	452,35	0,71	0,00	0,71	17,61	13,83	6,00	14,05	143,28

Resultados obtenidos en el ES con 20% de restricciones.

	Iris				Ecoli				Rand				Newthyroid			
	Tasa_C	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T
Ejecución 1	0,66	0,00	0,66	33,77	21,62	40,00	22,69	313,55	0,71	0,00	0,71	14,61	14,28	0,00	14,28	132,06
Ejecución 2	0,66	0,00	0,66	33,40	21,51	55,00	22,98	434,15	0,71	0,00	0,71	20,84	14,28	0,00	14,28	180,69
Ejecución 3	0,66	0,00	0,66	31,44	21,17	50,00	22,51	472,94	0,71	0,00	0,71	20,40	14,28	0,00	14,28	242,67
Ejecución 4	0,66	0,00	0,66	33,16	21,58	45,00	22,78	596,70	0,71	0,00	0,71	16,37	14,28	0,00	14,28	199,26
Ejecución 5	0,66	0,00	0,66	45,31	21,50	47,00	22,76	425,61	0,71	0,00	0,71	16,64	14,28	0,00	14,28	191,72
Media	0,66	0,00	0,66	35,42	21,48	47,40	22,74	448,59	0,71	0,00	0,71	17,77	14,28	0,00	14,28	189,28

6.2 – Búsqueda Multiarranque (BMB)

Resultados obtenidos en el BMB con 10% de restricciones.

	Iris				Ecoli				Rand				Newthyroid			
	Tasa_C	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T
Ejecución 1	0,67	0,00	0,66	43,24	22,20	119,00	25,38	946,30	0,71	0,00	0,71	38,00	13,83	6,00	14,05	127,20
Ejecución 2	0,67	0,00	0,66	47,03	21,69	171,00	23,59	955,34	0,71	0,00	0,71	38,44	13,83	6,00	14,05	106,74
Ejecución 3	0,67	0,00	0,66	44,59	21,78	141,00	25,53	954,86	0,71	0,00	0,71	42,67	13,83	6,00	14,05	118,66
Ejecución 4	0,67	0,00	0,66	42,96	21,67	148,00	25,62	968,83	0,71	0,00	0,71	40,31	13,83	6,00	14,05	128,61
Ejecución 5	0,67	0,00	0,66	45,55	22,52	105,00	25,32	966,96	0,71	0,00	0,71	39,56	13,83	6,00	14,05	113,16
Media	0,67	0,00	0,66	44,67	21,97	136,80	25,09	958,46	0,71	0,00	0,71	39,80	13,83	6,00	14,05	118,87

Resultados obtenidos en el BMB con 20% de restricciones.

	Iris				Ecoli				Rand				Newthyroid			
	Tasa_C	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T
Ejecución 1	0,67	0,00	0,66	51,93	21,74	156,00	25,90	950,72	0,71	0,00	0,71	42,36	14,28	0,00	14,28	130,88
Ejecución 2	0,67	0,00	0,66	51,03	22,33	164,00	26,70	947,21	0,71	0,00	0,71	45,48	14,28	0,00	14,28	105,84
Ejecución 3	0,67	0,00	0,66	49,34	22,58	139,00	26,29	947,10	0,71	0,00	0,71	43,81	14,28	0,00	14,28	143,51
Ejecución 4	0,67	0,00	0,66	48,42	22,42	122,00	25,67	944,25	0,71	0,00	0,71	44,77	14,28	0,00	14,28	157,86
Ejecuión 5	0,67	0,00	0,66	50,95	21,37	145,00	25,23	956,47	0,71	0,00	0,71	43,37	14,28	0,00	14,28	144,01
Media	0,67	0,00	0,66	50,33	22,09	145,20	25,96	949,15	0,71	0,00	0,71	43,96	14,28	0,00	14,28	136,42

6.3 – Búsqueda Local Iterativa

Resultados obtenidos en el ILS con 10% de restricciones.

	Iris				Ecoli				Rand				Newthyroid			
	Tasa_C	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T
Ejecución 1	0,66	0,00	0,66	27,77	33,15	102,00	23,87	921,77	0,78	0,00	0,71	23,96	13,83	6,00	14,05	37,96
Ejecución 2	0,66	0,00	0,66	23,25	33,19	75,00	23,59	957,65	0,75	0,00	0,71	22,81	13,83	6,00	14,05	66,60
Ejecución 3	0,66	0,00	0,66	24,13	35,92	57,00	23,80	938,68	0,76	0,00	0,71	23,30	13,83	6,00	14,05	59,26
Ejecución 4	0,66	0,00	0,66	25,02	36,17	97,00	23,83	987,94	0,75	0,00	0.71	22,38	13,83	6,00	14,05	62,60
Ejecución 5	0,66	0,00	0,66	23,78	37,97	58,00	23,25	931,23	0,80	0,00	0,71	23,61	13,83	6,00	14,05	55,99
Media	0,66	0,00	0,66	24,79	35,28	77,80	23,67	947,45	0,77	0,00	0,71	23,21	13,83	6,00	14,05	56,48

Resultados obtenidos en el ILS con 20% de restricciones.

	Iris				Ecoli				Rand				Newthyroid			
	Tasa_C	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T
Ejecución 1	0,66	0,00	0,66	30,03	32,74	87,00	23,91	950,72	0,71	0,00	0,71	25,81	14,28	0,00	14,28	93,56
Ejecución 2	0,66	0,00	0,66	31,77	35,89	66,00	23,28	976,48	0,71	0,00	0,71	25,70	14,28	0,00	14,28	95,57
Ejecución 3	0,66	0,00	0,66	28,56	28,95	106,00	24,91	993,49	0,71	0,00	0,71	26,29	14,28	0,00	14,28	79,29
Ejecución 4	0,66	0,00	0,66	29,20	35,38	71,00	23,55	971,68	0,71	0,00	0,71	27,13	14,28	0,00	14,28	108,82
Ejecución 5	0,66	0,00	0,66	29,73	22,16	62,00	23,82	934,99	0,71	0,00	0,71	27,06	14,28	0,00	14,28	88,48
Media	0,66	0,00	0,66	29,86	31,02	78,40	23,89	965,47	0,71	0,00	0,71	26,40	14,28	0,00	14,28	93,14

Resultados obtenidos en el ILS-ES con 10% de restricciones.

	Iris				Ecoli				Rand				Newthyroid			
	Tasa_C	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T
Ejecución 1	0,68	0,00	0,66	93,96	21,70	176,00	26,39	989,09	0,79	0,00	0,71	78,22	13,83	6,00	14,05	251,29
Ejecución 2	0,68	0,00	0,66	76,91	41,38	237,00	26,04	1001,61	0,79	0,00	0,71	91,27	13,83	6,00	14,05	214,91
Ejecución 3	0,68	0,00	0,66	87,31	34,78	116,00	24,91	993,06	0,79	0,00	0,71	85,97	13,83	6,00	14,05	225,91
Ejecución 4	0,68	0,00	0,66	88,03	42,36	91,00	25,37	996,10	0,79	0,00	0,71	79,92	13,83	6,00	14,05	204,44
Ejecución 5	0,68	0,00	0,66	77,25	43,88	199,00	27,58	1010,77	0,79	0,00	0,71	80,88	13,83	6,00	14,05	227,18
Media	0,68	0,00	0,66	84,69	36,82	163,80	26,06	998,13	0,79	0,00	0,71	83,25	13,83	6,00	14,05	224,75

Resultados obtenidos en el ILS-ES con 20% de restricciones.

	Iris				Ecoli				Rand				Newthyroid			
	Tasa_C	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T
Ejecución 1	0,68	0,00	0,66	94,93	21,70	176,00	26,39	982,40	0,71	0,00	0,71	85,22	14,28	0,00	14,28	332,84
Ejecución 2	0,68	0,00	0,66	100,50	41,38	237,00	26,04	992,02	0,71	0,00	0,71	91,47	14,28	0,00	14,28	306,46
Ejecución 3	0,68	0,00	0,66	100,87	34,78	116,00	24,91	993,49	0,71	0,00	0,71	86,96	14,28	0,00	14,28	361,85
Ejecución 4	0,68	0,00	0,66	100,96	42,36	91,00	25,37	985,88	0,71	0,00	0,71	91,65	14,28	0,00	14,28	339,60
Ejecución 5	0,68	0,00	0,66	93,75	43,88	199,00	27,58	990,29	0,71	0,00	0,71	87,53	14,28	0,00	14,28	343,21
Media	0,68	0,00	0,66	98,20	36,82	163,80	26,06	988,82	0,71	0,00	0,71	88,57	14,28	0,00	14,28	336,79

6.3 – Tabla global comparativa

NOTA: Para realizar las tablas de resultados globales, he arreglado los errores que tuve en la práctica 1, para obtener unos resultados más fiables. Aún así, añado en las tablas los resultados obtenidos la primera vez y los nuevos.

Resultados globales con 10% de restricciones

	Iris				Ecoli				Rand				Newthyroid			
	Tasa_C	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T
COPKM	0.14	55,40	7,55	4,39	8,13	448,40	645,61	1652,94	0.16	45,80	8,54	3,64	x	x	x	x
BL	0,11	0,00	0,11	36,13	7,69	709,20	1015,93	2388,08	0,13	0,00	0,13	34,72	x	x	x	x
COPKM-Arreg	1,57	215,00	2,86	0,26	37,24	537,40	51,56	13,93	1,47	164,40	2,64	0,28	19,7	161,00	25,58	0,752
BL-Arreglado	0,67	0,00	0,67	4,14	22,39	55,40	23,86	156,64	0,71	0,00	0,71	3,48	11,6	118,00	15,91	11,73
ES	0,66	0,00	0,66	24,63	21,48	47,40	22,74	452,35	0,71	0,00	0,71	17,61	13,83	6,00	14,05	143,28
BMB	0,67	0,00	0,66	44,67	21,97	136,80	25,09	958,46	0,71	0,00	0,71	39,8	13,83	6,00	14,05	118,87
ILS	0,66	0,00	0,66	24,79	35,28	77,80	23,67	947,45	0,71	0,00	0,71	23,21	13,83	6,00	14,05	56,48
ILS-ES	0,68	0,00	0,66	84,69	36,82	163,80	26,06	998,13	0,79	0,00	0,71	83,25	13,83	6,00	14,05	224,75

Resultados globales con 20% de restricciones

	Iris				Ecoli				Rand				Newthyroid			
	Tasa_C	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T
COPKM	0.12	113,40	7,95	3,29	9,97	386,40	288,68	1652,89	0.17	106,00	10,17	3,39	x	x	x	x
BL	0,11	0,00	0,11	35,88	7,73	1383,60	1005,71	2250,57	0,13	0,00	0,13	29,42	x	x	x	x
COPKM-Arreg	0,76	62,40	0,95	0,53	43,13	644,40	51,71	14,07	1,38	217,40	2,15	0,49	18,61	250,40	23,17	1,45
BL-Arreglado	0,67	0,00	0,67	4,91	21,83	162,20	23,99	290,89	0,72	0,00	0,72	4,37	14,28	0,00	14,28	15,6
ES	0,66	0,00	0,66	35,42	21,48	47,40	22,74	448,59	0,71	0,00	0,71	17,77	14,28	0,00	14,28	189,28
BMB	0,67	0,00	0,66	50,33	22,09	145,20	25,96	949,15	0,71	0,00	0,71	43,96	14,28	0,00	14,28	136,42
ILS	0,66	0,00	0,66	29,86	31,02	78,40	23,89	965,47	0,71	0,00	0,71	26,4	14,28	0,00	14,28	93,14
ILS-ES	0,68	0,00	0,66	98,2	36,82	163,80	26,06	998,82	0,71	0,00	0,71	88,57	14,28	0,00	14,28	336,79

7– Análisis global de resultados

Para analizar el rendimiento sobre los algoritmos, se han ejecutado sobre cuatro conjuntos de datos, uno más que en la práctica anterior:

- ***Iris***: Contiene información sobre características de tres tipos de flor de Iris. Tiene tres clases (k = 3).

- ***Ecoli***: Contiene medidas sobre ciertas características de diferentes tipos de células. Tiene ocho clases (k=8).

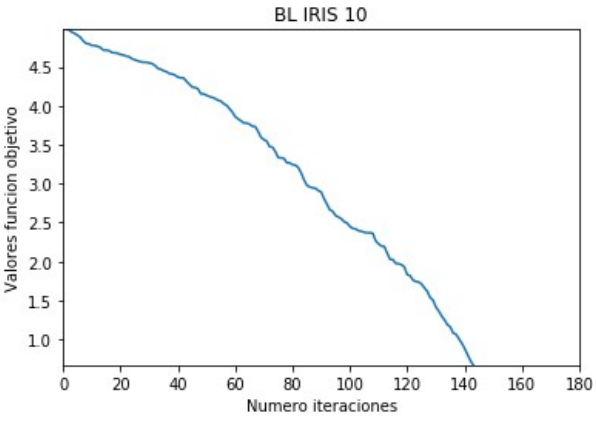
- ***Rand***: Conjunto de datos artificial. Contiene tres clases (k = 3).

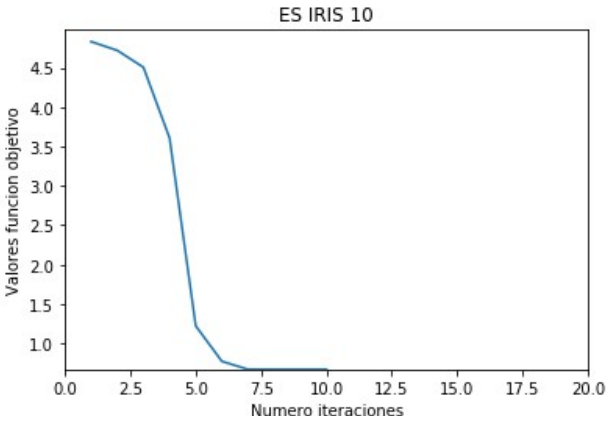
- ***Newthyroid***: Contiene medidas cuantitativas tomadas sobre la glándula tiroides de 215 pacientes. Tiene tres clases (k = 3).

Compararemos ambos algoritmos basándonos en sus capacidades para obtener soluciones de calidad, el número de restricciones no satisfechas, y rapidez de estos, comenzando por comparar los algoritmos genéticos.

Comenzamos comparando las técnicas basadas en trayectorias simples, es decir al algoritmo de ***Búsqueda Local*** y ***Enfriamiento Simulado***. Tanto en los resultados para 10% como 20% de restricciones, no obtenemos diferencias muy significativas en los valores, sobre todo en los conjuntos más sencillos como *"Iris"* y *"Rand"*, ambos son capaces de llegar a los mismos resultados; sin embargo, observamos que en *"Newthyroid"*, a pesar de que la Búsuqeda Local haya conseguido un agregado menor, el Enfriamiento Simulado ha sido capaz de disminuir en gran medida el valor de *"infeasibility"*, haciendo que el valor objetivo resultante sea menor en este que en la BL. Aún así, el mayor cambio lo podemos observar en el conjunto de datos *"Ecoli"* , donde claramente los resultados son mejores en el Enfriamiento Simulado ya que consigue disminuir todos los parámetros. Aunque el ES requiera más tiempo de ejecución, nos decidimos por el debido a las mejoras en sus resultados.

A pesar de ello resulta interesante, sobre todo para conjuntos como *"Iris"* donde se obtienen los mismos resultados, comparar el comportamiento de ambos algoritmos, por lo que he decidido realizar gráficas de convergencias para el caso de Iris con un 10% de restricciones:

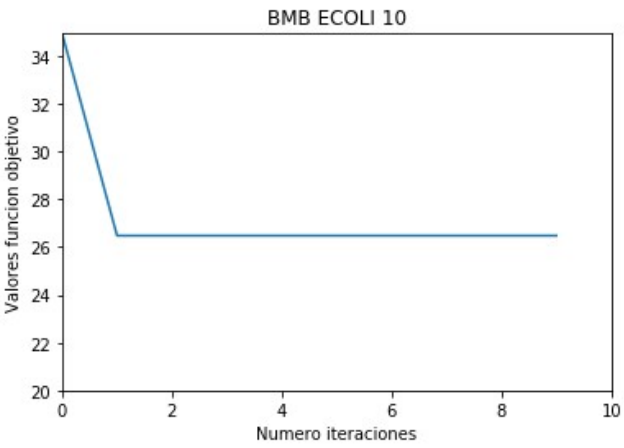


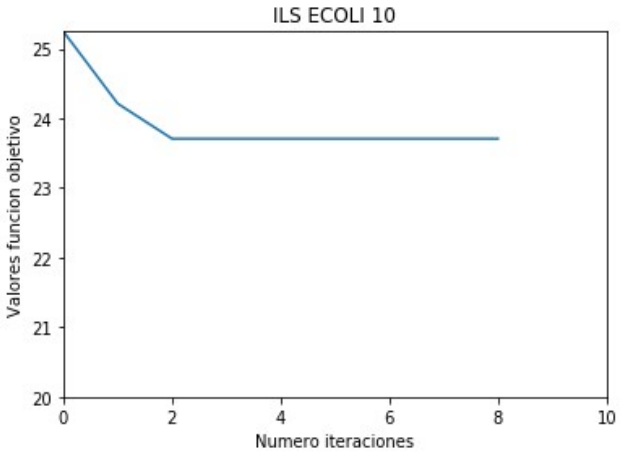


Como podemos observar, aunque ambos algoritmos lleguen al óptimo, es descomunal la diferencia de iteraciones que necesita cada uno para conseguirlo, siendo la Búsqueda Local unas 140 aproximadamente y el Enfriamiento Simulado, menos de 15. Esto se debe a que enfría demasiado rápido y de manera que no le supone ninguna dificultad encontrar el óptimo.

Otra cosa de la que nos podemos percatar sobre el algoritmo ES es el hecho de que en las primeras iteraciones, este descenso es más suave, debido a que al principio la temperatura es más alta, lo que da paso a que se elijan soluciones peores más a menudo; por ende, a medida que esta temperatura dismiuye las soluciones pasan a ser mayoritariamente mejores y a descender con más velocidad.

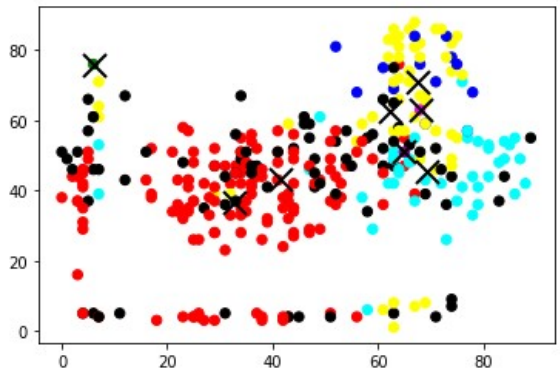
Compararemos también los algoritmos de trayectorías múltiples: ***Búsqueda Multiarranque*** y ***Búsqueda Local Iterativa***. Ambos aplican una Búsqueda Local en cada iteración, por lo que no es extraño que obtengamos resultados muy similares entre ellos. Aún asi, el ILS mejora un poco con respecto al BMB gracias a que disminuye el valor de *"infeasibility"*, como se ve en los resultados del conjunto *"Ecoli"*; por lo que podemos concluir que el hecho de utilizar una mutación en el conjunto de datos, aunque empeore el resultado, puede ser de gran utilidad para salir de óptimos locales en los que estamos atascados y mejorar el resultado final.



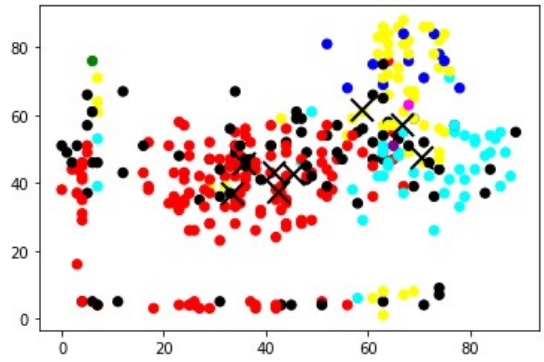


Generamos dos gráficas para el conjunto *"Ecoli"* con 10% de restricciones, donde podemos ver el comportamiento del BMB e ILS. En el caso de la Búsqueda Multiarranque, es evidente que desde la primera Búsqueda Local encontramos un óptimo local del que no somos capaces de salir, ya que para todas las soluciones aleatorias generadas llegamos a un mismo resultado en el que nos quedamos estancados; sin embargo, podemos destacar la diferencia entre los valores inicial (34 y algo) y final (26 y algo), la cual es medianamente grande. En el caso de la Búsqueda Local Iterativa, el descenso es algo más suave para las primeras iteraciones, pero no tardamos en quedarnos atascados en un óptimo local, aunque en este caso la diferencia de los valores inicial y final sea menor.

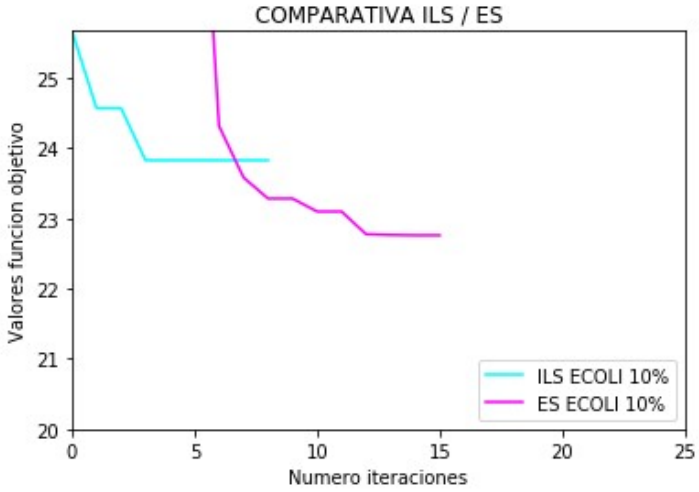
Hasta ahora, contemplamos al ***Enfriamiento Simulado*** como mejor algoritmo de búsuqeda de trayectorias simples y a la ***Búsqueda Local Iterativa*** como mejor de entre los de búsqueda de trayectorias múltiples. Entre ambos, es evidente que el Enfriamiento Simulado obtiene mejores resultados ya que consigue obtener un valor menor de la función objetivo en, además, menos tiempo; esto es debido a que el ILS al implementarse usando BL, es susceptible a que quede atascado en un óptimo local, ya que la Búsqueda Local mejora la solución cuando obtiene una mejora inmediata.



ES – ECOLI 10%



ILS -ECOLI 10%



Las dos primeras imágenes son la distribución de los centroides sobre unos mismos datos, no es difícil ver que para el algoritmo Enfriamiento Simulado, los centroides están algo mejor distribuidos que en la Búsqueda Local Iterativa; resultados que a su vez se ven reflejados en las tablas de las ejecuciones, donde el valor del parámetro *"ifeasibility"* es menor en el ES.

A continuación, también generamos una gráfica donde comparamos los valores de la función objetivo para ambos algoritmos, en ella vemos que aunque al Enfriamiento Simulado le tome un mayor número de iteraciones, consigue unos mejores resultados gracias a una disminución menos brusca del coste.

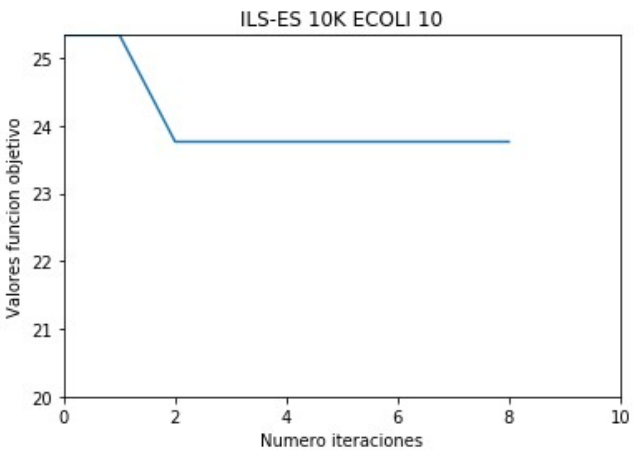
Podemos asegurar que estos algoritmos explotan más que diversifican, lo que genera buenos resultados. Además, el hecho de que el ES favorezca al principio la exploración, hace que se genere un equilibro exploración-explotación que funciona bastante bien con conjuntos de datos más complejos como el *"Ecoli".*

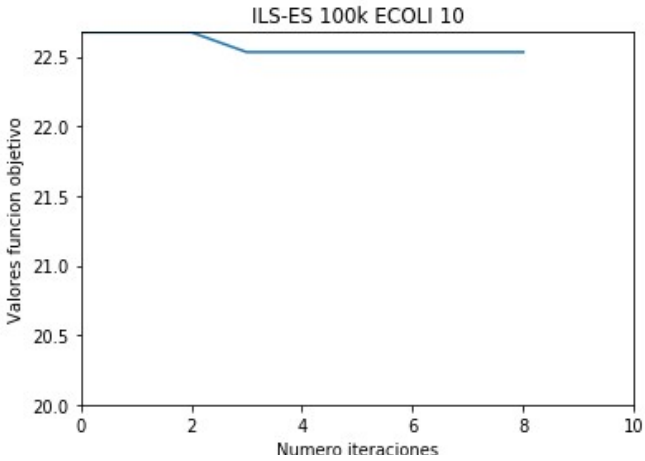
Sorprendentemente, el ***algoritmo híbrido ILS-ES*** obtiene peores resultados que el resto de los algoritmos, a pesar de combinar los dos mejores algoritmos elegidos. Una explicación para esto podría ser que a la hora de realizar la práctica, usar solmaente 10000 evaluaciones de la función objetivo no son suficientes para un buen enfriamiento de la temperatura del ES; por lo que sus resultados se ven emperoados.

Debido a esto, veo una buena oportunidad para experimentar qué ocurriría si ejecutásemos el ILS-ES con 100000 evaluaciones de la función objetivo:

La ejecución la haré con la semilla 7 y para el conjunto de datos Ecoli con 10% de restricciones.

	Desviación	Infeasibility	F.Objetivo	Tiempo
ILS-ES 10K	39,53	92	23,76	689,56
ILS-ES 100K	43,82	37	22,53	2625,88





Aunque solo se haya probado con una única ejecución, no solo el ILS-ES con 100K evaluaciones es mejor que el de 10K , sino que el resultado de la función objetivo es mejor que el resultado medio de todos los algoritmos implementados, a pesar de que no se diferencia mucho de los valores obtenidos

con el Enfriamiento Simulado. Con esto, queda demostrado que 10K no son suficientes para generar un enfriamiento que supere al resto de los algoritmos y no obtener las peores soluciones.

El ILS-ES de 10k nos proporciona un valor menor de la desviación general, sin embargo, es mayor la diferencia de los valores de *"infeasibility"* , causa de que el valor de la función objetivo sea el más bajo. Aún así, este algoritmo presenta una gran desventaja: el tiempo de ejecución , el cual es inmensamente mayor en comparación al resto de resultados.

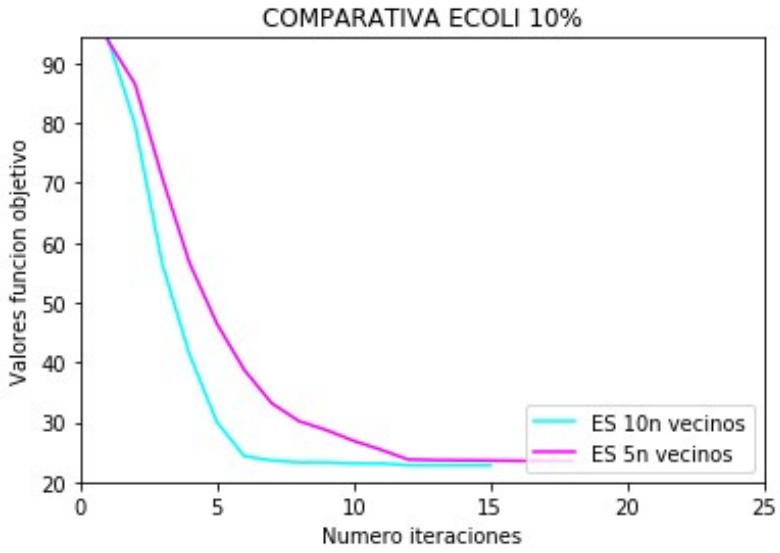
Experimentación:

Debido a los buenos resultados del Enfriamiento Simulado, resultaría interesante ver qué resultados nos proporciona en dos casos diferentes. Todos los resultados será obbtenidos con el conjunto de datos Ecoli debido a que es el qué mas juego ha dado durante la práctica, para un porcentaje de 10% de restricciones.

El primero de ellos, ha sido suponer que un max_vecinos = 10*n eran demasiados, por lo que hemos cambiado este valor por un max_vecinos = 5*n

Los resultados obtenidos son los siguientes:

	Ecoli, vecinos=10*n				Ecoli, vecinos=5*n			
	Tasa_C	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T
Ejecución 1	21,63	40,00	22,69	315,47	21,18	93,00	23,65	182,43
Ejecución 2	21,51	55,00	22,98	440,61	21,92	41,00	23,02	243,60
Ejecución 3	21,17	50,00	22,50	474,22	21,42	61,00	23,04	340,01
Ejecución 4	21,58	45,00	22,78	608,27	21,59	46,00	22,81	356,21
Ejecución 5	21,51	47,00	22,75	423,18	21,50	47,00	22,75	425,61
Media	21,48	47,40	22,74	452,35	21,52	57,60	23,05	309,57



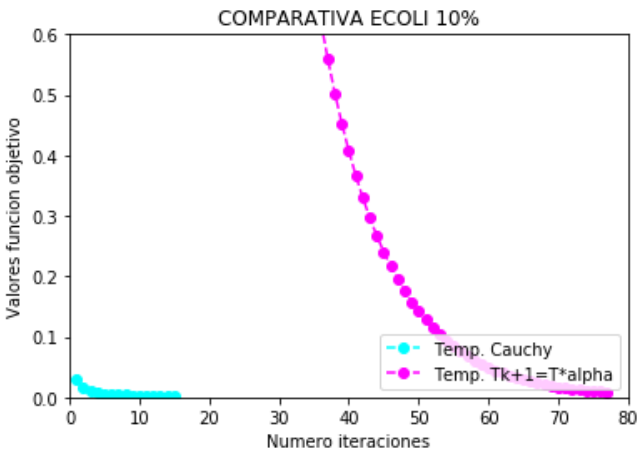
Si disminuimos el número de vecinos en el número de enfriamientos, vemos como la disminución del valor del coste toma una curvatura más suave, donde le toma más iteraciones en encontrar el óptimo final, como era de esperar. Además, fijándonos en los valores de la tabla, aunque no difieren mucho , son algo peores que cuando usamos un mayor número de vecinos, ya que estamos generando menos 'oportunidades' de encontrar soluciones mejores.

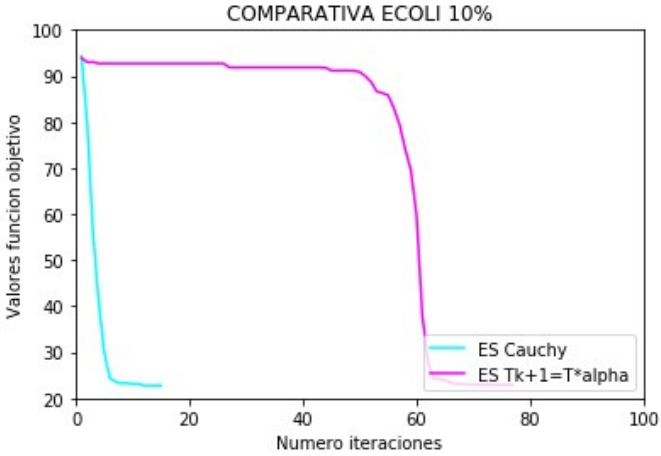
También, para comprobar si el Esquema de Cauchy enfría demasiado rápido, lo he sustituido por un esquema proporcional:

$$T_{k+1} = \alpha \cdot T_k \text{ con } \alpha \in [0.9, 0.99].$$

Los resultados obtenidos son los siguientes:

	Ecoli , Cauchy				*Ecoli, Tk+1 = T*alpha*			
	Tasa_C	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T
Ejecución 1	21,63	40,00	22,69	315,47	21,17	58,00	22,72	797,86
Ejecución 2	21,51	55,00	22,98	440,61	21,03	67,00	22,81	679,85
Ejecución 3	21,17	50,00	22,50	474,22	21,26	63,00	22,94	616,89
Ejecución 4	21,58	45,00	22,78	608,27	21,32	72,00	23,24	637,55
Ejecución 5	21,51	47,00	22,75	423,18	21,14	64,00	22,85	610,59
Media	21,48	47,40	22,74	452,35	21,18	64,80	22,91	668,55





Observándo las gráficas, vemos como el esquema de Cauchy enfría muy rapido, ya que partiendo de una misma temperatura inicial, Cauchy consigue disminuirla en gran medida mientras que el otro esquema implementado disminuye mucho más lentamente la temperatura. Esto, a su vez, se ve relfejado en los valores de la función objetivo, donde en el ES con Cauchy disminuye rápidamente su valor, mientras que en el otro los valores disminuyen muy lentamente, quedandose en gran parte de las iteraciones un poco “atascado”, ya que se eligen muchas soluciones peores hasta que la temperatura disminuye lo suficiente como para empezar a comportarse igual que en el esquema de Cauchy.

Este comportamiento tampoco mejora ninguno de los parámetros de estudio, aunque no haya una gran variación entre ellos; exceptuando el tiempo que, no hablando en su favor, empeora.

Para concluir este análisis, aseguramos que técnicas como la BMB y sobre todo, el ILS, son capaces de ofrecernos muy buenos resultados y no deben ser despreciados. Sin embargo, reforzamos el hecho de que el Enfriamiento Simulado es una metaheurística muy potente, la cual es capaz de proporcionarnos muy buenos resultados en poco tiempo. A su vez, el ILS-ES puede llegar a ser una técnica muy eficaz siempre que se estime un número de evaluaciones y un esuqmea de enfriamiento adecuados.

Viéndose no solo estos resultados, sino también los de las otras prácticas realizadas a lo largo del curso, he de recalcar que no exsite una metaheurística que sea ideal pata todos los problemas. Por eso, es beneficioso probar y adaparar varías técnicas a un mismo problema en base a saber cuál será más adecuada para este.

7 – Bibliografía

- Entendimiento del problema

- Tema 5: Métodos basados en trayectorias
- Seminario 3: Problemas de optimización con técnicas basadas en trayectorias

-Programación del problema

- https://stackoverflow.com/
- https://docs.python.org/3/library/

-Análisis de resultados y muestra de resultados

- https://www.aprendemachinelearning.com/k-means-en-python-paso-a-paso/
- https://matplotlib.org/mpl_toolkits/mplot3d/tutorial.html
-https://datatofish.com/export-dataframe-to-excel/