Práctica final

Cat Swarm Optimization

Problema del Agrupamiento con Restricciones (PAR)

Metaheurística GII 19-20
Grupo MH1 - Miércoles 17:30h

Cumbreras Torrente, Paula
49087324-B
paulacumbreras@correo.ugr.es

ugr | Universidad de Granada

“Un algoritmo debe ser visto para ser creído”
-Donald Ervin Knuth

Índice de contenidos

Descripción del problema

El problema del agrupamiento con restricciones (PAR) consiste en un problema de agrupamiento o clustering al que se le ha añadido un conjunto de restricciones. El clustering es una técnica de aprendizaje no supervisada cuyo objetivo es agrupar los elementos que tengan alguna similitud entre ellos de un conjunto de datos en varios clusters, simplificando así la información de dichos datos.

Este problema suele ser aplicado en aplicaciones de minería de datos, aplicaciones Web, marketing, diagnóstico médico, análisis de ADN, biología computacional, análisis de plantas y animales, farmacología…

El problema elegido consiste en una generalización del ya mencionado. PAR es una técnica de aprendizaje semi-supervisada. Las restricciones expresan un requerimiento del usuario y describen propiedades que han de contener los grupos de elementos. El conjunto de restricciones que se aplicarán a nuestro problema pueden ser clasificadas en:

- Restricciones fuertes (hard): todas las restricciones de esta clase han de ser cumplidas para que una solución sea factible. Las restricciones fuertes tratadas en nuestro problema serán:
  - Todos los clusters deben contener al menos una instancia:
  - Cada instancia debe pertenecer a un solo cluster:
  - La unión de los clusters debe ser el conjunto de datos X:
- Restricciones débiles (soft): se busca que el número de restricciones violadas de esta clase sea minimizado. Las restricciones débiles a aplicar serán restricciones de instancia:
  - Must-Link: dos instancias dadas han de ser asignadas al mismo cluster.
  - Can't-Link: dos instancias dadas no podrán agruparse en el mismo cluster.

Se nos han proporcionado 4 conjuntos de datos para la realización de del problema:

- Iris: 150 datos con 4 propiedades a agrupar en 3 clases.
- Ecoli: 336 datos con 7 propiedades a agrupar en 8 clases.
- Rand: 150 datos con 2 propiedades a agrupar en 3 clases.
- Newthyroid: 215 datos con 5 propiedades a agrupar en 3 clases

Para cada conjunto de datos se estudiarán 2 instancias distintas de PAR a partir de los conjuntos de restricciones, también proporcionados, los cuales corresponderán al 10% y 20% del total de restricciones posibles.

Para esta práctica que estudiará la metaheurística CSO (Cat Swarm Optimization) junto a un algoritmo híbrido memético con Búsqueda Local y una mejora de la misma. Se compararán los resultados obtenidos con dichos algoritmos, evaluados en función de la desviación general, de la infactibilidad y del tiempo obtenido con diferentes semillas de aleatoriedad, con los resultados obtenidos con los algoritmos de la primera práctica (Búsqueda Local y Greedy).

Descripción de la aplicación de los algoritmos empleados

El esquema de estructuras de datos empleado para la primera práctica (en la que se estudiaron los algoritmos Greedy y Búsqueda Local) es el siguiente:

- Los elementos se almacenan en Nodos, estructuras formadas por un entero sin signo cluster donde se registrará el cluster al que pertenecen (por defecto -1), y un vector de double datos donde se almacenan las distintas propiedades del elemento.
- El conjunto de todos los Nodos se almacena en un vector de Nodos X en el orden de lectura.
- Las restricciones se almacenan en una matriz (vector de vector) de enteros M.
- Los Cluster quedan representados como una estructura compuesta por un vector de double donde se almacenará su centroide y por un vector de enteros sin signos donde se almacenará el índice en X de los distintos Nodos que lo compongan.
- El conjunto de todos los Clusters se almacena en un vector de Cluster C.

Para esta práctica, se ha definido un atributo posición para las clases generadas que funciona de forma similar a los cromosomas en los algoritmos evolutivos de las prácticas anteriores, sustituyendo al vector de Clusters C:

- Un vector de tantos enteros sin signo como elementos tenga el conjunto seleccionado. En él se almacenan los números de los clusters a los que han sido asignados cada uno de los elementos de X.

Esta forma de representación permite trabajar con los clusters con más facilidad, pero no contiene la suficiente información para evaluar la partición obtenida; por ello, se añade un método de conversión que adaptará una partición del atributo posición a C

Conversión

Para todos los elementos i de la partición
  asignar a la variable cluster del elemento X[i] el valor de partición[i]
  añadir al vector datos del cluster C[solución[i]] la posición i
Fin para
Actualizar centroides
Devolver C (X ha sido pasado por referencia)

Una vez se haya ejecutado el algoritmo seleccionado, aparecerán por pantalla una serie de evaluaciones de la partición obtenida:

- Tasa_C: Desviación general de la partición.

Desviación general

Para todos los clusters
  calcular distancia media intra-cluster
  añadir el valor obtenido al valor de una variable desviación
Devolver desviación entre el número de clusters

Distancia media intra-cluster

```
Para cada elemento i del cluster
  para cada propiedad j del elemento
    añadir en variable[j] la distancia entre el dato j del elemento i y el dato j del
centroide del cluster
Para cada propiedad del vector variable
  añadir su valor a una variable distancia
Devolver raíz cuadrada de distancia entre el número de elementos del cluster
```

- Tasa_Inf: Infactibilidad o número de violaciones de restricciones

Infactibilidad

```
Para cada fila de M
  para cada columna de M
    si M[i][j] es 1 y los elementos i y j de la solución son distintos
      incrementar infactibilidad en 1
    si M[i][j] es -1 y los elementos i y j de la solución son el mismo
      incrementar infactibilidad en 1
Devolver infactibilidad
```

- Agregado: Función objetivo o función a minimizar

Función objetivo

```
Calcular infactibilidad
Calcular desviación
Lambda es la diferencia entre la distancia máxima entre los elementos del
conjunto y el número total de restricciones.
Devolver desviación más infactibilidad por lambda
```

- Tiempo: Calculado desde el inicio del algoritmo hasta el final del mismo, excluyendo la lectura de ficheros e inicialización de datos. Devuelto en nanosegundos.

Como parámetro de entrada habrá que indicar si se desea que se muestre por pantalla el resultado de la partición C. En caso afirmativo se mostrarán los índices en X de los elementos almacenados en cada cluster; un ejemplo de salida sería:

```
Semilla = 8

Tasa_C  |  Tasa_i | Agreg   |   Tiempo
0.106822  0         0.106822  457676653

Cluster 0:
{ 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124
125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148
149 }
Cluster 1:
{ 1 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37
38 39 40 41 42 43 44 45 46 47 48 49 0 2 }
Cluster 2:
{ 51 52 53 54 55 56 57 58 59 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82
83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 50 60 }
```

Aparte de los ya mencionados, otros operadores que han sido necesarios para la implementación de los algoritmos son:

- Cálculo de centroides: Asigna a cada cluster el vector promedio de las instancias de X que los componen

Calculo centroides

Para cada cluster de la partición
  para cada elemento i del cluster
  para cada propiedad j del elemento
  añadir a variable [i] la propiedad j del elemento i
  para cada propiedad i del vector centroide
  asignar a variable [i] el valor que tenía dividido entre el número de elementos del cluster
  asignar el vector variable al centroide del cluster

- Operador para evitar que una solución tenga algún cluster vacío: Realiza modificaciones aleatorias hasta que esta restricción sea cumplida

No vacío

Generar vector check de tamaño num_clus
Para cada elemento i de la solución
  Aumentar check[i] en una unidad
Fin para
Mientras no se cumpla la restricción
  Para cada i de num_cluster
  si check[i] vale 0
  seleccionar aleatoriamente un cluster rand
  almacenar valor de solución[rand] en aux
  modificar el valor de solución[rand] a i
  aumentar check[i] en una unidad
  reducir check[aux] en una unidad
  salir del para
  si ningún elemento de check vale 0
  se cumple la restricción
  Fin para
Fin mientras
Devolver solución

Pseudocódigo del método de búsqueda

Cat Swarm Optimization (CSO)

La metaheurística elegida fue diseñada por Shu-Chuan Chu (Cheng Shiu University), Pei-wei Tsai y Jeng-Shyang Pan (National Kaohsiung University of Applied Sciences) y está basada en el comportamiento de los felinos, en especial de los gatos doméstico, del que distingue dos comportamientos: en reposo pero observando el entorno, decidiendo cuál será su próximo movimiento (seeking) y el estado de avanzar hacia un objetivo (tracing).

Para la realización del mismo se ha diseñado una clase Gato que tiene los siguientes parámetros:

- El ya mencionado vector de posiciones, de n dimensiones.
- Un vector de velocidades, de n dimensiones, que indicará la velocidad con la que un gato avanza en cada posición (corresponde con la "velocidad" con la que un elemento puede cambiar de cluster).
- Un límite de velocidad para el vector de velocidades.
- Un comportamiento, que podrá ser SEEKING o TRACING.

También se han añadido a la clase Gato las funciones para llevar a cabo las acciones correspondientes a ambos comportamientos:

Seeking con los siguientes parámetros:

- SMP (seeking memory pool): número de puntos a la vista del gato, es decir, el número de posiciones que se evaluarán.
- SRD (seeking range of selected dimension): distancia máxima a la que pueden encontrarse las posiciones a evaluar.
- CDC (counts of dimension to change): número de dimensiones que variarán.
- SPC (self-position considering): booleano que será true si el gato considera también como posible posición final la posición inicial.

```
Si SPC generar vector candidatos de indices aleatorios con tamaño SMP-1
Si no generar vector candidatos de indices aleatorios con tamaño SMP
Generar vector posiciones de vector de enteros sin signos
Generar vector valoraciones de valores reales
Para cada elemento i de candidatos
  añadir posición del gato a posiciones
  generar vector de índices aleatorios de tamaño CDC
  para cada elemento j de índices
    generar valor r aleatorio entre -1 y 1
    asignar a posiciones[i][j] = (posiciones[i][j]·r·SRD)%numero_clus *
  fin para
  almacenar en valoraciones la función objetivo de posiciones[i]
  si la valoración es máximo o mínimo respecto de las anteriores
    almacenar posición y función objetivo
Fin para
Si SPC
  añadir posición del gato a posiciones
  comprobar si es máximo o mínimo
Generar vector probabilidad de valores reales
```

Para cada i del número de dimensiones
  si el valor máximo es distinto del mínimo
  añadir a probabilidad (valoración[actual]-valoración[mayor]) / (valoración[mayor]-valoración[menor])
  si no añadir a probabilidad 1
Fin para
Actualizar la posición del gato con la posición con mayor probabilidad (si hay empate se selecciona aleatoriamente)
Comprobar que ningún cluster se queda vacío

Tracing con los siguientes parámetros: ● C: coeficiente. ● Mejor posición: posición obtenida por otro gato con mejor o igual valoración que la posición actual.

Para cada i del número de dimensiones
  generar un valor r aleatorio entre 0 y 1
  si velocidad[i]+r·C·(mejor posición[i] - posición[i]) es menor que el límite
  asignar a velocidad[i] dicho valor
  si no asignar el límite
  actualizar el valor de posición[i] a (posición[i]+velocidad[i])% número de clusters*
Fin para
Comprobar que ningún cluster se queda vacío

* Este valor ha de ser redondeado pues se trata de un valor real.

El algoritmo CSO genera un "enjambre" de gatos, a los que se le asigna un comportamiento. La cantidad de gatos que estará en un modo u otro viene determinado por un factor MR (mixture ratio). Para el problema estudiado, el factor que mejores resultados ha generado ha sido 0.6 (el 60% de los gatos estará en modo Seeking y el 40% en modo Tracing). Una vez se hayan realizado las acciones, se volverán a repartir los modos, y se repetirá hasta que se llegue al máximo de iteraciones o a la condición de parada, que en nuestro caso es que el algoritmo falle en encontrar una mejor solución un cierto número de veces.

```
Ajustar parámetros para los métodos
Generar un vector Swarm de Gatos
Redimensionar Swarm con cantidad de gatos a generar
Para cada i de cantidad de gatos
  añadir a Swarm[i] un nuevo gato con posición y velocidades aleatoria
  estudiar la función objetivo de la posición generada
  si es la mejor función objetivo almacenar i como mejor gato
Fin para
Mientras iteraciones < máximo de iteraciones y fallos < máximo de fallos
  generar un vector de índices aleatorios de tamaño MR·número de gatos
  para cada i de número de gatos
    si i está en índices, asignar comportamiento Seeking a Swarm[i]
    si no asignar comportamiento Tracing
  fin para
  para cada i de número de gatos
    si el comportamiento de Swarm[i] es Seeking, aplicar función seeking
    si no, aplicar función tracing
    si la función objetivo de Swarm[i] es mejor que la del mejor gato
      ha habido mejora
      asignar a mejor gato la posición i
  fin para
```

```
  si no ha habido cambios aumentar fallos en una unidad
  aumentar iteraciones en una unidad
Fin mientras
Devolver la posición del mejor gato
```

Cat Swarm Optimization Memético con BL (CSO-M)

El algoritmo memético diseñado utiliza la misma clase Gato diseñada para el CSO. El algoritmo es muy similar, con la excepción de que cada cierto número de iteraciones se aplica una búsqueda local suave. Puesto que la variación es poca, se añade a continuación un pseudocódigo resumido

```
Inicialización
Generación de gatos
Mientras iteraciones < máximo de iteraciones y fallos < máximo de fallos
  asignar comportamiento a gatos
  ejecutar funciones relativas a los comportamientos
  si no ha habido mejora aumentar fallos en una unidad
  si iteraciones módulo nº iteraciones para BL vale 0
    aplicar búsqueda local sobre la mejor posición
    actualizar la posición del mejor gato
    almacenar su función objetivo
  aumentar iteraciones en una unidad
Fin mientras
Devolver la posición del mejor gato
```

La búsqueda local empleada recorre el vector posición una única vez para evitar aumentar mucho el coste computacional

Búsqueda local suave
Almacenar función objetivo de la posición Mientras fallos < máximo de fallos y no se haya recorrido el cromosoma entero obtener un vecino con mutación si la función objetivo del vecino es menor que la de la posición la posición pasa a ser el vecino si no, aumentar fallos en una unidad aumentar contador en una unidad Fin mientras Comprobar que ningún cluster se queda vacío Devolver posición
Operador de mutación
Generar un elemento aleatorio entre 0 y dimensiones -1 Generar un nuevo cluster aleatorio distinto a posición[elemento] entre 0 y número de clusters -1 Asignar a posición[elemento] el nuevo cluster Devolver posición

Propuesta de mejora de Cat Swarm Optimization (CSO-P)

La propuesta de mejora diseñada se centra en las acciones de los gatos. Se ha creado una clase nueva denominada Pantera para la que se ha definido un comportamiento nuevo:

Cleaning. Los felinos son conocidos por ser muy limpios; la función asociada a este comportamiento consistirá en una pequeña búsqueda local sobre un conjunto de elementos, pudiendo entender la BL como una "limpieza" de sí mismo (los elementos mal situados serían suciedad). Las funciones de los comportamientos anteriores, ahora basados en un felino salvaje, han sido modificadas también.

Seeking con los siguientes parámetros:

- SRD (seeking range of selected dimension): distancia máxima a la que pueden encontrarse las posiciones a evaluar.
- CDC (counts of dimension to change): número de dimensiones que variarán.

La Pantera solo considerará movimientos unidimensionales; si puede desplazarse en una dimensión, esta variación se añadirá a la posición actual. Para entenderlo mejor, supongamos un ejemplo en un mundo 2D con una pantera y otros animales salvajes. A diferencia de un gato doméstico, la pantera no puede moverse a sus anchas, debe tener en cuenta el peligro (la nueva valoración es peor que la anterior). Si puede desplazarse a la derecha y hacia arriba, entonces irá en diagonal, pero estudiará ambas direcciones de forma individual.

```
Generar vector posiciones de vector de enteros sin signos
Generar vector valoraciones de valores reales
Si SPC almacenar en actual un valor aleatorio entre 0 y dimensiones - 1
Generar un vector de índices aleatoriamente de tamaño dimensiones
Para cada elemento i de dimensiones
  añadir posición del gato a posiciones
  si que i sea distinto que actual
    asignar a posiciones[i][índices[i]]=(posiciones[i][índices[i]]·r·SRD)%numero_clus *
Fin para
Para cada i de dimensiones
  si la función objetivo de posiciones[i] es mejor que la actual
     asignar al elemento índices[i] de posición el valor de posiciones[i][índices[i]]
Fin para
Comprobar que ningún cluster se queda vacío
```

Tracing con los siguientes parámetros:

- C: coeficiente.
- Mejor posición: posición obtenida por otro gato con mejor o igual valoración que la posición actual.

Las panteras son animales muy veloces, por lo que se ha eliminado el límite de velocidad. Además, antes de avanzar, estudiará si hay peligro en la posición a la que se dirige (entendiendo por peligro una peor valoración)

```
Generar un vector auxiliar de enteros sin signo con el tamaño de dimensiones
Para cada i del número de dimensiones
  generar un valor r aleatorio entre 0 y 1
  asignar a velocidad[i] el valor de velocidad[i]+r·C·(mejor posición[i] - posición[i])
  almacenar en auxiliar[i] a (posición[i]+velocidad[i])% número de clusters*
Fin para
Si la función objetivo de auxiliar es mejor que la de la posición actual
  posición = auxiliar
Comprobar que ningún cluster se queda vacío
```

Cleaning con el siguiente parámetro:

- CDC (counts of dimension to change): número de dimensiones que variarán.

```
Generar un vector de índices aleatorio de tamaño número de dimensiones
```

Reducir el vector de índices a CDC elementos
Mientras no se realicen CDC iteraciones
  generar un vecino por cambio de cluster del elemento índices[contador]
  si la función objetivo del vecino es mejor que la actual
  asignar a la posición actual la del vecino
Fin mientras
Comprobar que ningún cluster se queda vacío

Operador cambio de clúster
Generar un nuevo clúster distinto de posición[elem] aleatoriamente entre 0 y número de clusters - 1 Generar un vector vecino de enteros sin signo igual a posición Asignar a vecino[elem] el nuevo cluster Devolver vecino
* Este valor ha de ser redondeado pues se trata de un valor real.

El algoritmo diseñado es muy similar al CSO original: genera un "enjambre" de panteras y les asigna un comportamiento. La distribución de los comportamientos viene dado por los factores MR_L (porcentaje de panteras que estará en modo Seeking) y CL (porcentaje de panteras que estarán en modo Cleaning). Para el problema estudiado, la división que mejores resultados generaba ha sido asignar un 47% a Seeking, un 23% a Cleaning y el 30% restante a Tracing.

```
Ajustar parámetros para los métodos
Generar un vector Swarm de Panteras
Redimensionar Swarm con cantidad de panteras a generar
Para cada i de cantidad de panteras
  añadir a Swarm[i] una nueva pantera con posición y velocidades aleatoria
  estudiar la función objetivo de la posición generada
  si es la mejor función objetivo almacenar i como mejor pantera
Fin para
Mientras iteraciones < máximo de iteraciones y fallos < máximo de fallos
  generar un vector de índices aleatorios de tamaño (MR_P+CL)·número de panteras
  separar los índices en índices seeking (tamaño MR_P·panteras) y cleaning (tamaño
CL·panteras)
  para cada i de número de panteras
    si i está en índices seeking, asignar comportamiento Seeking a Swarm[i]
    si i está en índices cleaning, asignar comportamiento Cleaning a Swarm[i]
    si no asignar comportamiento Tracing
  fin para
  para cada i de número de panteras
    si el comportamiento de Swarm[i] es Seeking, aplicar función seeking
    si el comportamiento de Swarm[i] es Tracing, aplicar función tracing
    si no, aplicar función cleaning
    si la función objetivo de Swarm[i] es mejor que la de la mejor pantera
      ha habido mejora
      asignar a mejor pantera la posición i
  fin para
  si no ha habido cambios aumentar fallos en una unidad
  aumentar iteraciones en una unidad
Fin mientras
Devolver la posición de la mejor pantera
```

Pseudocódigo del algoritmo de comparación

Los algoritmos de comparación empleados han sido los algoritmos de Búsqueda Local y Greedy COPKM empleados para la práctica anterior.

- La Búsqueda Local es un proceso iterativo que parte de una solución aleatoria y la mejora realizando modificaciones locales respecto a la función objetivo.

Búsqueda Local

```
Para cada elemento i de X
  asignar un cluster aleatorio a cluster
  asignar a dicho cluster la posición i
  modificar hasta que ningún cluster quede vacío
Calcular centroides
Almacenar función objetivo
Mientras haya cambios y contador < 100000
  cambios = false
  para cada elemento i de X
    para cada cluster j mientras no se encuentre una mejora
      si cluster no está vacío
        calcular nuevo cluster
      si probar vecino
        eliminar el elemento i del cluster al que pertenecía
        asignar el elemento i al nuevo cluster
        modificar el valor de cluster del elemento i
        mejora encontrada y cambios=true
    fin para
  fin para
  aumentar contador en una unidad
Fin mientras
```

- Los algoritmos greedy o voraces son aquellos que, para resolver el problema, eligen la opción óptima local en cada paso tratando así de llegar a una solución general óptima.

Greedy

```
Almacenar infactibilidad
Calcular centroides aleatorios
Mientras haya cambios
  Para cada elemento de X
    Para cada cluster j de C
      si probar cluster es menor que máximo auxiliar
        vaciar lista de empates
        modificar máximo auxiliar
        añadir j a la lista de empates
      si son iguales añadir j a la lista de empates
    Fin para
    si la lista de empates tiene más de 1 cluster
      si algún cluster k de la lista está vacío, nuevo cluster = k
      si no nuevo cluster = desempatar por distancia
    si no nuevo cluster = el cluster de la lista
    si no es la primera iteración eliminar el elemento del cluster al que pertenecía
      asignar el elemento a nuevo cluster
```

```
        modificar cluster del elemento
    Fin para
    Si infactibilidad es igual a infactibilidad antes, diferencia = false
    Aumentar contador
Fin mientras
```

Procedimiento y manual de uso

Para el desarrollo de esta práctica no se ha partido de ningún framework de metaheurísticas ni de un código proporcionado, más allá de la librería random dada para la generación de números pseudoaleatorios. La estructura empleada en la práctica consiste en unas librerías llamadas "cso.h", "gato.h" y "pantera.h" donde se han incluido todas las funcionalidades necesarias para los algoritmos de esta práctica y en un main, alojado en "cluster.cpp" desde donde se llaman a las funciones necesarias para la ejecución según los parámetros que se indiquen. También se hace uso de la librería "util.h" creada para la primera práctica. Para las estructuras de datos se ha usado la librería std.

Se ha incluido un archivo makefile en la carpeta de software para la compilación del programa. Se ha creado un único ejecutable para todos los algoritmos, llamado cluster; éste se encuentra en el directorio bin. Para ejecutar el programa, es necesario pasar como parámetros, en el orden indicado, los siguientes datos:

- Modo: hace referencia al algoritmo que se ejecutará. En esta práctica, los parámetros aceptados serán:
    - "COPKM" para la ejecución del algoritmo Greedy
    - "BL" para la ejecución del algoritmo de Búsqueda Local
    - "CSO" para el algoritmo Cat Swarm Optimization
    - "CSO-M" para el algoritmo Cat Swarm Optimization Memético
    - "CSO-P" para el algoritmo Cat Swarm Optimization Mejorado
- Conjunto: hace referencia al conjunto de datos sobre el que se ejecutará el problema. Para evitar tener que introducir la ruta completa de todos los ficheros en cada ejecución, éstos han de encontrarse dentro de un directorio llamado "datos", en el mismo directorio desde donde se llama a la función. Los parámetros aceptados serán:
    - "I" para el conjunto Iris
    - "E" para el conjunto Ecoli
    - "R" para el conjunto Rand
    - "NT" para el conjunto Newthyroid
- Porcentaje restricciones: hace referencia el porcentaje del total de restricciones posibles que se aplicarán al problema. Los parámetros aceptados serán "10" y "20"
- Nº repeticiones: cantidad de veces que deseemos que se ejecute el algoritmo
- Resultado: si deseamos que se muestre, además de la salida estándar, el reparto de elementos en los distintos clusters, este parámetro será "S"; en caso contrario será "N"
- [Semillas]: hace referencia a las semillas de generación de números aleatorios. Se pueden introducir tantas como se deseen. Si no se introduce ninguna, se considerarán las semillas 1, 5, 10, 20 y 25.

Experimentos y análisis de resultados

Descripción de los casos del problema empleados y de los valores de los parámetros considerados en las ejecuciones de cada algoritmo

Para cada algoritmos se han realizado 50 ejecuciones por conjunto de datos, es decir, 25 por cada conjunto de restricciones. Se han elegido 5 semillas, en nuestro caso {1, 5, 10, 20 y 25}, y para todas ellas se ha ejecutado el programa 5 veces.

Resultados obtenidos según el formato especificado

Se han incluido los resultados de todas las ejecuciones de los algoritmos en formato ods en archivos individuales con el mismo nombre que el parámetro especificado en la sección anterior para cada algoritmo en el directorio "ejecuciones". En el mismo directorio se han añadido 3 archivos de comparación:

- "comparacion_BL_COPKM.ods" muestra una comparación entre los resultados medios obtenidos para cada semilla con los algoritmos de la primera práctica.
- "comparacion_CSO.ods" muestra una comparación entre los resultados medios obtenidos para cada semilla con los tres algoritmos de esta práctica.
- "comparacion.ods" muestra la comparación entre las medias de los resultados obtenidos para todas las semillas con todos los algoritmos evaluados (práctica 1 y práctica final). Aunque los tiempos se han obtenido en nanosegundos con una mayor precisión, en esta comparación se muestran en segundos para facilitar la comprensión.

Comparación de todos los algoritmos con 10% de restricciones

10,00 %	Iris				Ecoli			
Algoritmo	Tasa_C	Tasa_inf	Agregado	Tiempo	Tasa_C	Tasa_inf	Agregado	Tiempo
COPKM	0,19309116	585	0,377306	0,045448948	2,3718714	2638	6,73392	43,16101505
BL	0,1013662	181,6	0,1585518	0,63099293	2,509918	2684,6	6,10053	44,87238801
CSO	0,2895662	457,4	0,4336004	1,170799806	7,518898	1823,6	9,957936	5,887912341
CSO-M	0,230873	306	0,3272316	1,304533892	7,061924	1647,8	9,265834	8,108351114
CSO-P	0,1658902	207,8	0,2313258	3,687212463	6,306184	1511,2	8,32739	29,20215697

10,00 %	Rand				Newthyroid			
Algoritmo	Tasa_C	Tasa_inf	Agregado	Tiempo	Tasa_C	Tasa_inf	Agregado	Tiempo
COPKM	0,3833204	582,2	0,5914688	0,066989823	2,1200092	1120,6	4,159124	0,095534881
BL	0,1171912	9	0,1204086	0,431677721	0,7248988	1185,2	2,881566	2,425872726
CSO	0,3978052	448,8	0,5582602	1,187072698	1,982858	1116,2	4,013962	2,032802795
CSO-M	0,3357014	327	0,4526108	1,277679012	2,038534	907,8	3,690424	2,461776165
CSO-P	0,2688784	237,6	0,3538248	3,049554226	1,847698	753,2	3,218268	7,790460901

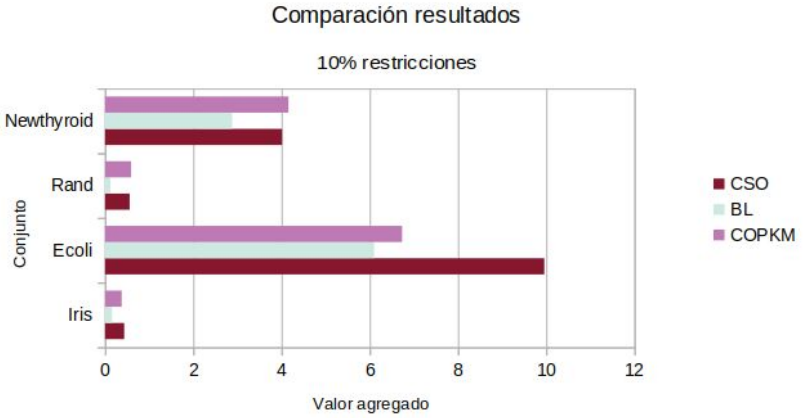
Comparación de todos los algoritmos con 20% de restricciones

20,00 %	Iris				Ecoli			
Algoritmo	Tasa_C	Tasa_inf	Agregado	Tiempo	Tasa_C	Tasa_inf	Agregado	Tiempo
COPKM	0,19309116	1178,4	0,5641664	0,052322799	3,0555152	5008,8	11,936774	42,26533541
BL	0,1016958	302,6	0,1969836	0,572985412	2,708116	5335,8	9,844656	57,67299581
CSO	0,2924846	899,2	0,5756408	1,245044598	7,489274	3595,6	12,29832	6,735830773
CSO-M	0,2554034	729,6	0,4851526	1,390590663	7,143498	3175,6	11,3908	8,530917647
CSO-P	0,1681822	434,2	0,3049106	4,027001548	6,539426	2916,4	10,44006	34,02295315

20,00 %	Rand				Newthyroid			
Algoritmo	Tasa_C	Tasa_inf	Agregado	Tiempo	Tasa_C	Tasa_inf	Agregado	Tiempo
COPKM	0,2990654	1212,8	0,732666	0,063900305	2,841118	2347	7,111866	0,170259734
BL	0,1171538	19,4	0,1240894	0,472646106	0,7357046	2418	5,135648	2,533093233
CSO	0,394227	913	0,7206434	1,255172645	2,059214	2221,4	6,101412	2,238401668
CSO-M	0,3293758	605,2	0,5457472	1,394793807	2,212782	1683,6	5,276366	2,686771321
CSO-P	0,253414	405,8	0,3984956	3,906678343	2,139424	1105,2	4,15051	7,564522329

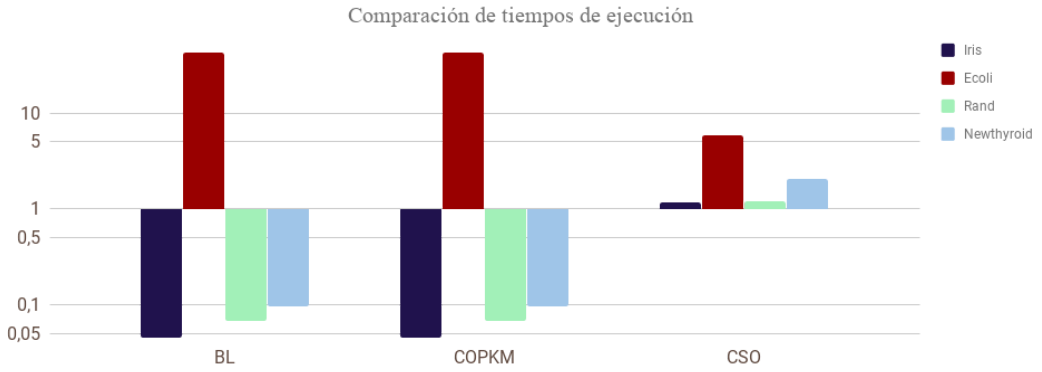
Análisis de resultados

El algoritmo CSO no obtiene soluciones especialmente buenas, aunque probablemente sea debido a la naturaleza de nuestro problema. La adaptación de los valores reales mediante redondeo también perjudica el buen funcionamiento que este algoritmo podría llegar a tener si trabajáramos con ellos.

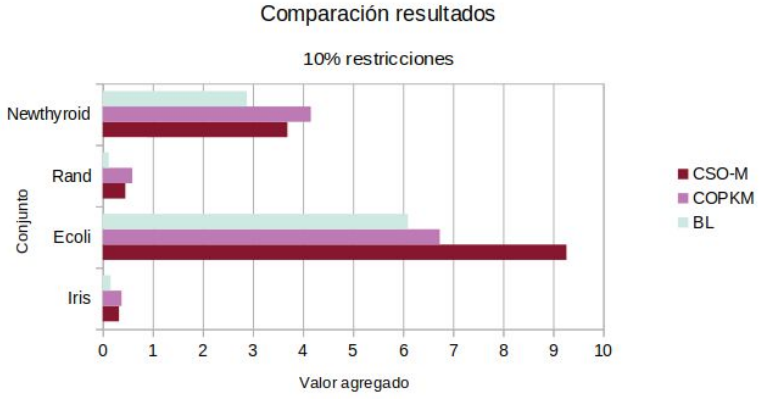


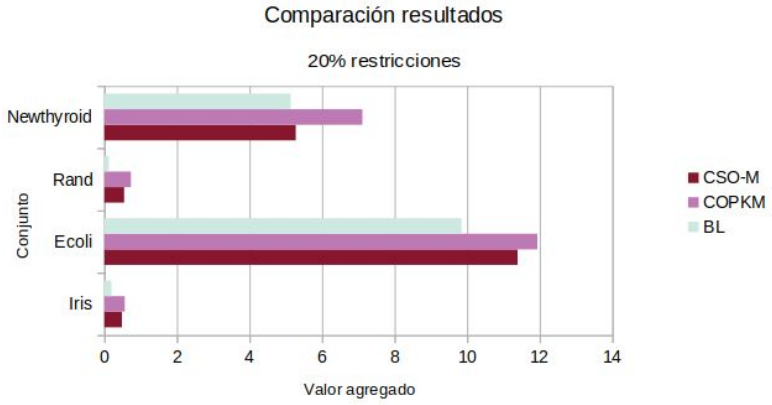
Como se puede observar en las gráficas, la función objetivo obtenida con CSO solo es ligeramente menor que la obtenida con COPKM para los conjuntos Rand y Newthyroid, y es bastante superior a la BL. En cuanto a la relación entre desviación general e infactibilidad, CSO obtiene mejores valores para la Tasa_inf que para la Tasa_C. A continuación se muestra una gráfica en la que se comparan los tiempos de ejecución con el conjunto del 10% de restricciones; en ella se puede observar que CSO obtiene tiempos ligeramente peores para Iris, Rand y Newthyroid, pero es bastante rápido en la ejecución de Ecoli, que es el conjunto que más problemas ha generado en relación al tiempo a lo largo del desarrollo de las prácticas. Solamente se ha añadido la gráfica de las comparaciones de tiempo al ejecutar con

el 10% de restricciones porque éstos y los tiempos para el conjunto del 20% son proporcionales.

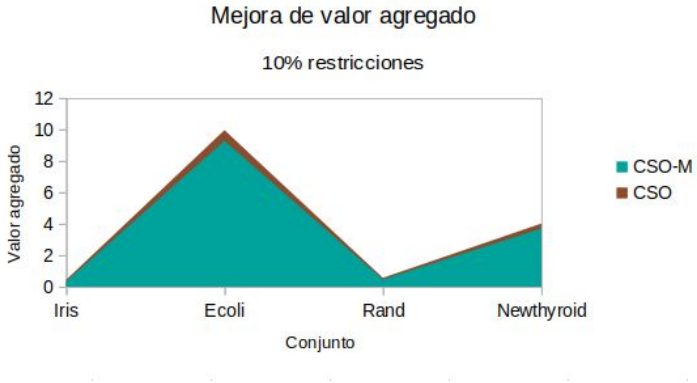


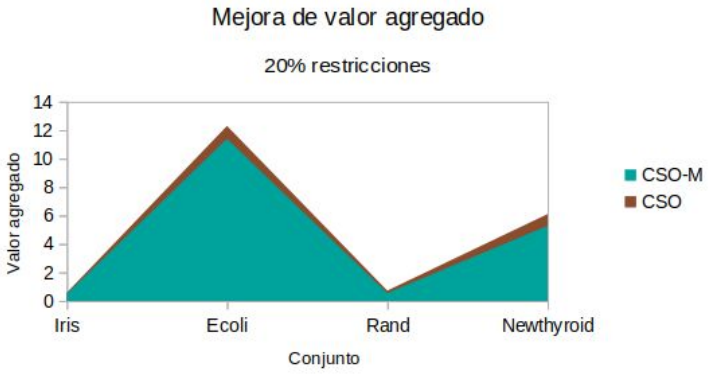
El algoritmo memético diseñado para CSO, con uso de BL, se ha obtenido una leve mejora, aunque sin llegar a ser bueno en comparación con la búsqueda local. Este nuevo algoritmo presenta mejores resultado que CSO en unos tiempos aceptables; aunque aumentando la frecuencia en la que se aplica la búsqueda local sobre la posición del mejor gato (actualmente establecida a 10 iteraciones) se obtienen mejores resultados, a costa del tiempo de ejecución. Ya que aun ejecutando la búsqueda local en cada iteración no se obtienen resultados mejores que la BL original (ni que COPKM para Ecoli con el conjunto del 10%), así como el tiempo de ejecución resulta perjudicado para tan escasa mejora del valor agregado, se ha mantenido la frecuencia ya mencionada.



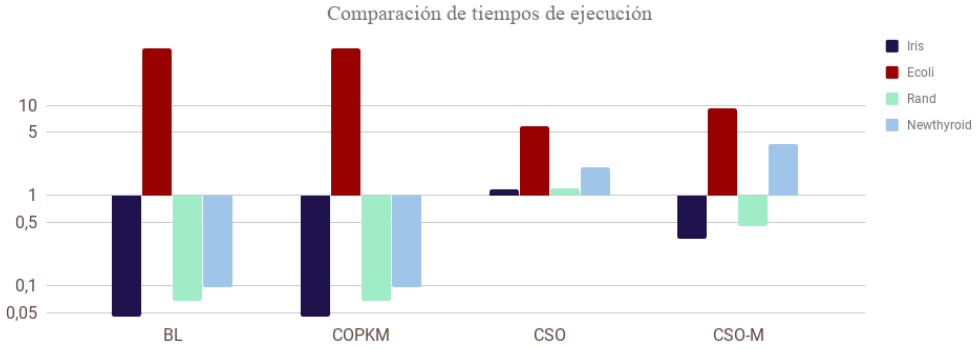


Considero que la mejora que proporciona el hibridar nuestro algoritmo no es demasiado notoria debido al gran peso que CSO le da a la exploración: aunque la BL nos proporcione una solución parcialmente óptima, si el mejor gato entra en modo Tracing, aunque ya se encuentre en la posición óptima es posible que aun así se desplace, pudiendo en algunos casos empeorar los resultados. Esto se arregla en la propuesta de mejora, poniendo una condición para que el gato se deplace (que la nueva posición sea mejor). A continuación se muestra la mejora en el valor agregado obtenida en cuanto a CSO.

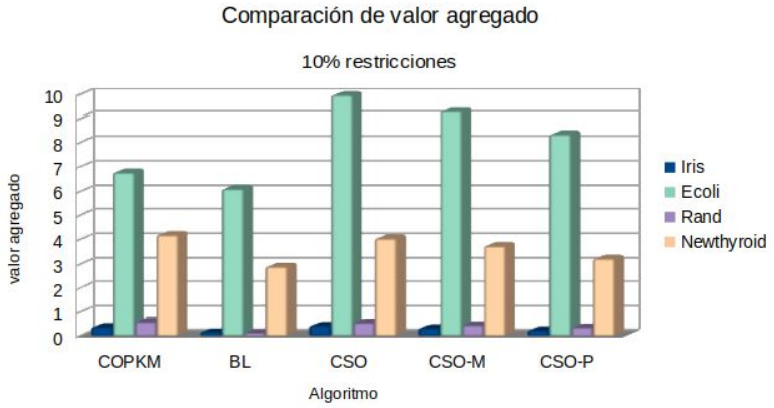


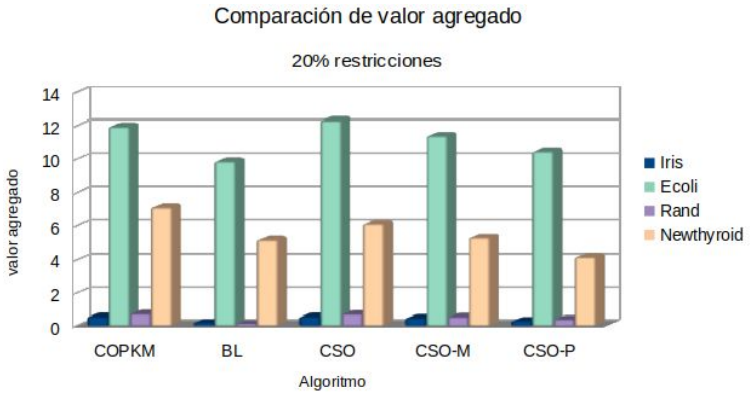


Lo que se ha tratado de representar en esta gráfica es la mejora del algoritmo híbrido frente al clásico, mostrando la diferencia entre los valores agregados alcanzados para cada conjunto. Como se puede observar, no existe gran diferencia. La relación entre las tasas de desviación general y de infactibilidad es la misma que con CSO. A continuación se muestra la misma gráfica de tiempos mostrada anteriormente, pero con los tiempos de CSO-M añadidos, donde se puede observar que éste último algoritmo resulta un poco más lento que CSO para los conjuntos grandes (Ecoli y Newthyroid) pero es más rápido para los pequeños. En relación a los algoritmos de comparación, solamente presenta una mejora en tiempo para el conjunto Ecoli, justo como ocurría con CSO. Como ya se mencionó anteriormente, al aumentar la frecuencia con la que se ejecuta la BL, el tiempo aumenta; estos son los tiempos obtenidos para una frecuencia de 10 iteraciones.

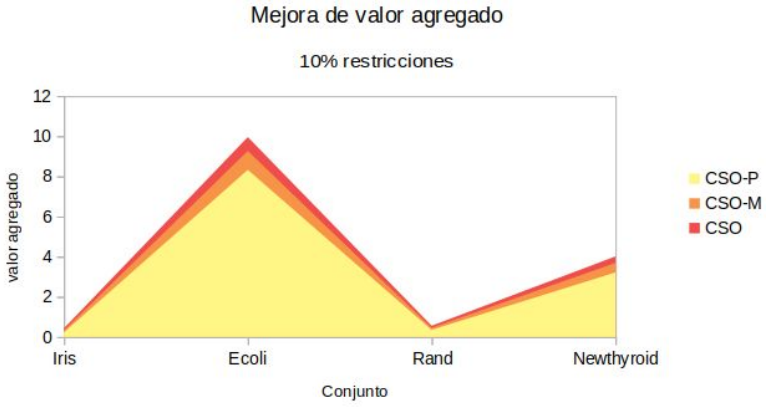


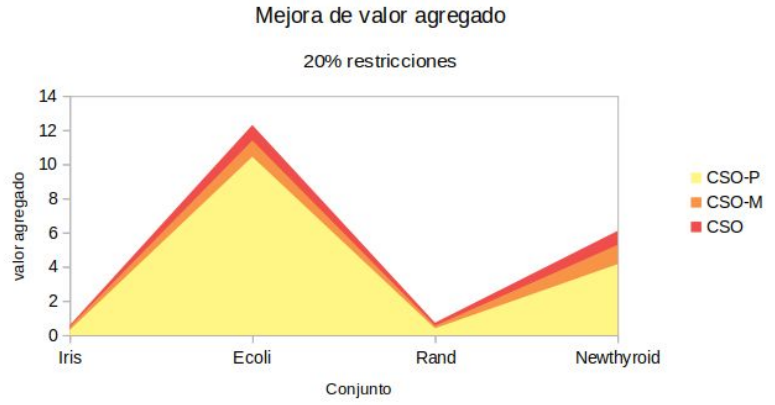
Por último analizamos la propuesta de mejora realizada, junto a una comparación final de todos los algoritmos. En este algoritmo también se incluye, como ya se comentó, una búsqueda local pero más reducida, en la que sólo se generan vecinos para un cierto número de elementos para no aumentar demasiado el coste computacional. Además se trata el posible empeoramiento de los resultados surgido por el modo Tracing mediante una condición. A continuación se muestra la comparación de los valores agregados obtenidos con los 5 algoritmos estudiados.



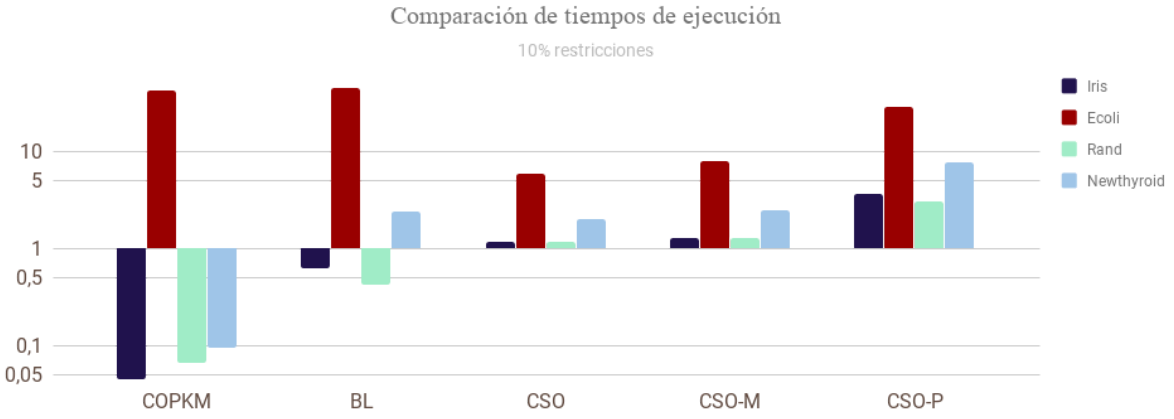


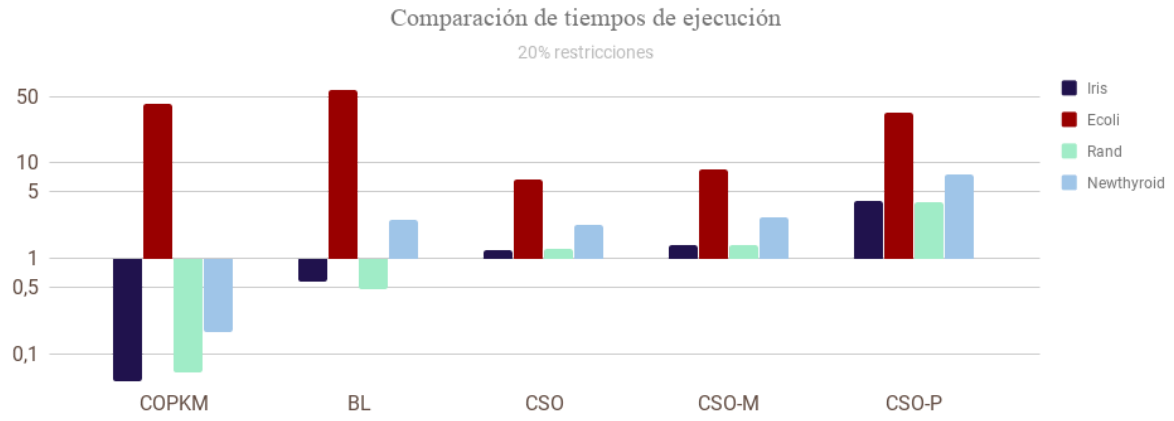
Se puede observar que CSO-P, aun mejorando con respecto a CSO y CSO-M, sigue sin ser suficientemente bueno para alcanzar a BL. Sí obtiene unos resultados más similares a los de COPKM. La relación del algoritmo CSO entre la desviación general y la tasa de infactibilidad se mantiene. A continuación se muestra la mejora obtenida en cuanto al valor agregado con las modificaciones realizadas.





Vemos que aunque el valor agregado no difiere demasiado del obtenido con CSO-M, sumada a la diferencia entre CSO-M y CSO, sí puede tratarse de una mejora notable con respecto a la metaheurística original, que como ya comenté en un principio, no parece ser idónea para nuestro problema. Hablando de los tiempos, no podemos decir lo mismo: CSO-P resulta bastante lenta, empeorando en todos los conjuntos con respecto a las otras 2 versiones ya comentadas. Aunque el tiempo empleado para el conjunto Ecoli es menor que el empleado por greedy y búsqueda local, sigue siendo demasiado alto. A continuación se muestran las gráficas con los tiempos obtenidos.





La conclusión que obtengo es la misma con la que empecé el análisis: CSO no es apropiado, la conversión de los valores reales hace que el algoritmo pierda la buena funcionalidad que podría tener el modo Tracing: su forma de acercarse a una mejor solución en nuestro caso se convierte en una modificación de clusters que en ningún momento asegura que nos ofertará una mejor solución. El algoritmo propuesto por mí es demasiado costoso para los resultados que ofrece. El modo Seeking diseñado para la clase Pantera está inspirado en el original, por lo que su funcionamiento sería más lógico en problemas de valores reales, otra vez, no en el nuestro.

Bibliografía

- https://www.researchgate.net - *"(PDF) Cat Swarm Optimization"*
- https://github.com
    - *"GitHub - thieunguyen5991/metaheuristic"*
    - *"GitHub - DACUS1995/Swarm-intelligence-experiments"*
    - *"GitHub - orouskhani/BinaryCSO"*
    - *"GitHub - dhbwstudienarbeit2019/CSO"*

"Mi gato nunca se ríe o se lamenta, siempre está razonando."
– Miguel de Unamuno