UNIVERSIDAD DE GRANADA

ETSIIT
Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

Práctica 3: Búsquedas por Trayectorias para el Problema del Agrupamiento con Restricciones

Metaheurísticas

Grupo A3 — Lunes de 17:30 a 19:30

Autor:
Lugli, Valentino Glauco · YB0819879 · valentinolugli@correo.ugr.es

Índice

1. Descripción del Problema

Esta práctica aborda el Problema del Agrupamiento con Restricciones, siendo este una generalización del problema del Agrupamiento Clásico. Este problema tiene como objetivo clasificar un conjunto de datos u observaciones en un grupo definido de clusters o agrupamientos de acuerdo a posibles similitudes entre los datos. Es un problema de aprendizaje no supervisado que permite descubrir grupos inicialmente desconocidos en un conjunto de datos, también permite el agrupamiento de objetos similares entre sí.

La variante que se trabajará, llamada con Restricciones o Contrained Clustering, generaliza el anterior problema añadiendo restricciones, de esta manera el problema se vuelve de aprendizaje semi-supervisado en donde se tiene un conjunto de datos X con n instancias, donde se desea encontrar una partición C del mismo que minimice la desviación general y cumpla con las restricciones del conjunto llamadas R.

Se utilizarán restricciones de instancias para R, es decir, dada una pareja de instancias se establecerán dos tipos de restricciones: Must-Link, en la que se indica que ese par de instancias deben pertenecer a un mismo cluster y Cannot-Link, que indica que ese par no debe pertenecer al mismo cluster. Se interpretarán las restricciones de manera débil, es decir que se desea minimizar el número de restricciones incumplidas pero se pueden incumplir algunas.

2. Descripción de los algoritmos empleados

2.1. Consideraciones generales

En esta práctica se realizaron 4 algoritmos que Búsquedas por Trayectorias: Enfriamiento Simulado, Búsqueda Multiarranque Básica, Búsqueda Local Reiterada y un híbrido de la Búsqueda Local Reiterada con Enfriamiento Simulado (ILS-ES).

Los algoritmos representan la solución de manera similar, hacen uso de un vector de enteros de una longitud n en donde se indica que para la i-ésima instancia en ese vector, esta está asociada a un cluster j, con $j \in \{0, .., k-1\}$.

Los datos referentes a las instancias están almacenados para ambos problemas en una matriz X de tamaño $n \times d$ y hacen uso de, o bien una matriz de restricciones R denominada MR de tamaño $n \times n$ o bien de un vector de restricciones que internamente utiliza un tipo de dato abstracto sencillo denominado `triplet` el cual contiene un par de índices de la matriz R y que clase de restricción poseen.

2.2. Función Objetivo y Operadores Comunes

2.2.1. Registro triplet

Registro utilizado en todas los algoritmos para almacenar las restricciones incumplidas.

```
registro triplet
    entero: x_0
    entero: x_1
    entero: restriccion
```

```
fregistro
```

2.2.2. Evaluación de la Función Objetivo

La función para evaluar los algoritmos es exactamente la misma que fue utilizada en el algoritmo de Búsqueda Local, se trata de una función principal llamada solamente `fitness`, la cual se utiliza para unir las dos partes de la misma, la distancia instracluster promedio sumado a la infeasibilidad por el lambda.

```
funcion fitness(vector de entero: S, vector de vector de flotante: X, vector de
   triplet: ML, vector de triplet: CL, flotante: lambda, entero: k): flotante
inicio:
    retornar intraClusterDistance(S, X, k) + (lambda * infeasibility(S, ML, CL))
ffunc
```

Para obtener la distancia intracluster promedio, se necesita saber que instancias están en qué clusters, para ello lo primero que se realiza es calcularlas, para el cluster `k`, se llama a la función `calcularInstancia()`, internamente esta función va almacenando que instancias del vector `S` se encuentran asociadas al cluster `i`, estas se retornan y se almacenan en `instanciasDeCluster`, luego, con estas instancias se promedian sus coordenadas en todas las dimensiones para obtener la posición del cluster `i`, esto se almacena en el vector `coords` y finalmente, se realiza el cálculo de la distancia intracluster de ese cluster con sus elementos asociados, y cuando se han realizado todos los clusters, se devuelve el promedio de todos ellos.

```
funcion intraClusterDistance(vector de entero: S, vector de vector de flotante: X,
   entero: k): flotante
variables:
    vector de entero: instanciasDeCluster
    vector de flotante: coords, actDist, total <- 0
inicio:
    para i <- 0 hasta k hacer
        // Obtener las instancias asociadas a un cluster.
        instanciasDeCluster <- calcularInstancia(X, i, S)
        // Obtener las coordenadas del centroide de un cluster.
        coords <- calcularCoordenadasCentroide(instancia, X)
        //Obtener la distancia intracluster de un cluster.
        actDist <- calcIntraDist(coords, instancia, X)
        total += actDist
    fpara

    retornar totalDist / k
ffunc
```

Como se ha comentado, se realiza aquí la media de las diferencias para un cluster en particular, en la función principal se realiza es el promedio de las distacias de todos los clusters.

```
funcion calcIntraDist(vector de flotante: coords, vector de entero: instancia,
   vector de vector de flotante X): flotante
variables:
    entero: numInstancias <- instancia.tamaño(), numDimensiones <- X[0].tamaño()
    flotante: suma <- 0, distanciaActual
inicio:
```

```
    para i <- 0 hasta numInstancias hacer
        para j <- 0 hasta numDimensiones hacer
            distanciaActual += ( X[ instancia[i] ][j] - coords[j] )**2
        fpara
        distanciaActual <- sqrt(distanciaActual)
        suma += distanciaActual
    fpara

    retornar suma / numInstancias
ffunc
```

Para la infesasibilidad, se recorre una vez los vectores de las restricciones, sumando si se incumplen las restricciones en el vector `S`.

```
funcion infeasibility(vector de entero: S, vector de triplet: ML, vector de triplet
    : CL): entero
variables:
    entero: contador <- 0, numML <- ML.tamaño(), numCL <- CL.tamaño()
inicio:
    para i=0 hasta numML hacer
        // Si x_0 y x_1 están en clusters diferentes, no se cumple ML
        si(S[ML[i].x_0] != S[ML[i].x_1]) entonces
            contador++
        fsi
    fpara

    para i=0 hasta numCL hacer
        // Si x_0 y x_1 están en clusters iguales, no se cumple CL
        si(S[CL[i].x_0] == S[CL[i].x_1]) entonces
            contador++
        fsi
    fpara

    retornar contador
ffunc
```

2.2.3. Generación de solución aleatoria

Se reutiliza la misma función para la generación de soluciones iniciales aleatorias de Búsqueda Local que aparece en la primera práctica debido a que el esquema de representación es exactamente el mismo y no se piden componentes extra para la generación inicial de soluciones; el cambio ahora es dónde se llama esta función dentro de los algoritmos.

```
procedimiento solucionInicial(vector de entero: &S, entero: k, vector de entero:
   listaCluster)
inicio:
    // Para verificar que al menos hay un cluster se rellena con los clusters
       existentes
    para i<-0 hasta k hacer
        S[i] <- i
        listaCluster[i]++
    fpara
    // Luego se rellena con un valor aleatorio que se encuentre de entre los
       valores posibles de un cluster
```

```
    para i<-k hasta n hacer
        S[i] <- randomInt(0, k)
        listaCluster[S[i]]++
    fpara

    // Se baraja la solución
    randomShuffle(S)
fproc
```

2.3. Enfriamiento Simulado

2.3.1. Función principal

La función de Enfriamiento Simulado, antes de comenzar el bucle principal realiza los cálculos necesarios: Se calcula el valor de β dependiendo de los valores de temperaturas inicial y final; ya que se utilizó Cauchy Modificado como esquema de enfriamiento, aunque también se realizaron pruebas utilizando el esquema proporcional. De igual manera de calcula la temperatura inicial, el máximo número de vecinos, de éxitos y de iteraciones.

Una vez en el bucle principal, una vez se reinicializan los contadores, se entra en el bucle interno; por cada iteración se cuenta la generación de un vecino en el contador apropiado, luego se genera un vecino el cual puede repetirse pero sin dejar ningún clúster sin instancias; luego se obtiene el fitness del mismo y si este fitness es mejor que el actual o si es peor, entra por el criterio de metrópolis si el valor aleatorio es menor que el cálculo del criterio en esa evaluación.

En ambos casos, se reemplaza la solución actual `S` por el vecino y su fitness, además del contador de clústers asociado. Este contador se mantiene para evitar dejar clústers vacíos.

Si, en todo caso, el fitness es mejor que el mejor hasta ahora, se actualiza el registro con el nuevo valor del fitness y las instancias que lo generan.

Una vez ocurre que se generan la cantidad máxima de vecinos o de éxitos, se sale de este bucle y se enfría; se repite hasta que el número de iteraciones se lleguen o el algoritmo entre en un óptimo y no ocurran más éxitos en el bucle interno.

```
funcion ES(vector de vector de flotante: X, vector de triplet: ML, vector de
   triplet: CL, entero: clusters, doble: lambda): vector de entero
variables:
    entero: cuentaVecinos, cuentaExitos, maxVecinos, maxExitos, iteraciones,
       maxIteraciones

    doble: mu, phi, beta, fitnessActual, fitnessVecino, diffFitness, tInicial,
       tFinal, tAct

    vector de entero: S(X.tamaño()), vecino(X.tamaño()), numK(k, 0), vecinoK(k,0)

    mejorDato: mejorSolucion
inicio:
    maxVecinos <- 6 * X.tamaño() // Originalmente 10
    maxExitos <- 0.1 * maxVecinos
    maxIteraciones <- 100000 / maxVecinos
    mu <- 0.3
```

```
phi <- 0.3
tFinal <- 0.00001 // Originalmente 0.001

solucionInicial(S, k, numK)
fitnessActual <- fitness(S, X, ML, CL, lambda, clusters)

mejorSolucion.instancias <- S
mejorSolucion.fitness <- fitnessActual

/*Cálculo Temperatura Inicial*/
tInicial <- calcularTempInicial(fitnessActual, mu, phi)

// Si en un grupo de datos resulta que tInicial salió más grande, se reduce por
    ese valor.
si (tInicial > tFinal) entonces
    tFinal <- tFinal * tInicial
fsi
/*Cálculo de Beta*/
beta <- (tInicial - tFinal) / (maxIteraciones * tInicial * tFinal)

tActual <- tInicial

hacer
    iteraciones++
    cuentaVecinos <- 0
    cuentaExitos <- 0

    hacer
        cuentaVecinos++
        generarVecino(S, vecino, numK, vecinoK)

        fitnessVecino <- fitness(vecino, X, ML, CL, lambda, clusters)

        diffFitness <- fitnessVecino - fitnessActual

        /*
         * Se utiliza el criterio de metrópolis, si diffFitness es mayor o
         * igual a cero, entonces, ese vecino entrará dependiendo de lo que
         * se obtenga por el RNG y el valor del criterio.
         */
        si (diffFitness < 0 ∨ Rand() < exp(-diffFitness/ actTemp) ) entonces

            cuentaExitos++
            S <- vecino
            numK <- vecinoK
            fitnessActual <- fitnessVecino

            si (fitnessActual < mejorSolucion.fitness) entonces
                mejorSolucion.instancias <- S
                mejorSolucion.fitness <- fitnessActual
            fsi
        fsi

    mientras (cuentaVecinos < maxVecinos ∧ cuentaExitos < maxExitos)
```

```
        /*Enfriamiento*/
        tAct <- enfriarCauchy(tAct, beta)
        // tAct <- enfriar(tAct)

    mientras(iteraciones < maxIteraciones ∧ cuentaExitos ≠ 0)

    retornar mejorSolucion.instancias
ffunc
```

2.3.2. Registro mejorDato

Registro utilizado para almacenar la mejor solución.

```
registro mejorDato
    vector de entero: instancias
    flotante: fitness
fregistro
```

2.3.3. Cálculo de Temperatura Inicial

El cálculo de la temperatura inicial fue implementado directamente de la descripción dada de la misma en el guión de prácticas; también con los valores correspondientes a μ y ϕ.

```
funcion calcularTempInicial(doble: fitness, doble: mu, doble: phi): doble
inicio
    retornar (mu * fitness) / -log(phi)
ffunc
```

2.3.4. Esquema de Enfriamiento

El esquema enfriamiento que se utiliza en la implementación final es el enfriamiento Cauchy Modificado, realizando cambios en los parámetros se encontró una configuración que hace uso de este. Se incluye también el enfriamiento proporcional ya que se utilizó en otras pruebas.

```
funcion enfriar(doble: temperatura): doble
inicio
    retornar tAct * 0.95
ffunc

funcion enfriarCauchy(doble: temperatura, doble: beta): doble
inicio
    retornar tAct / 1 + (beta * tAct)
ffunc
```

2.4. Búsqueda Multiarranque Básica

2.4.1. Función Principal

El algoritmo genera 10 soluciones aleatorias a las cuales se le aplica una Búsqueda Local, de estas se guarda la mejor. Cabe notar que la generación de soluciones aleatorias sucede internamente dentro

de la Búsqueda Local, aprovechando que el código de la Búsqueda Local es el mismo de la primera práctica y ya que son soluciones aleatorias sin ninguna relación entre ellas, se prefirió mantenerlas allí para facilitar el desarrollo del código.

```
funcion BMB(vector de vector de flotante: X, vector de triplet: ML, vector de
   triplet: CL, entero: k, doble: lambda) : vector de entero
variables:
    doble: mejorFitness <- MAX_DOBLE, fitnessActual
    vector de entero: S, mejorS
inicio

    para i<-1 hasta 10 hacer

        S <- busquedaLocal(X, ML, CL, k, lambda, fitnessActual)

        si (fitnessActual < mejorFitness) entonces
            mejorFitness <- fitnessActual
            mejorSol <- S
        fsi

    fpara
    retornar mejorSol
ffunc
```

2.4.2. Búsqueda Local

Se ha reutilizado la Búsqueda Local de la primera práctica con ligeras modificaciones que no afectan su funcionalidad, en particular, para BMB se modificó para que se acepte el parámetro fitness por referencia, de esta manera además de devolver el vector de enteros de las instancias, devuelve el fitness asociado.

Para ILS, se modificó para que se por parámetro además de tener el fitness, se tiene también el vector de enteros de la solución mutada, en este caso no se realiza internamente el cálculo de la solución inicial, pero el resto del algoritmo es idéntico.

Función Principal

```
funcion BL(vector de vector de entero: X, vector de triplet: ML, vector de triplet:
    CL, entero: k, doble: lambda): vector de entero
variables:
    entero: evaluaciones
    vector de entero: S(X.tamaño), viejaS, vecindarioVirtual, listaCluster(0, k)
    par[entero "posicion", entero "clusterNuevo"]: vecinoVirtual
inicio:
    // Esto está comentado en ILS ya que S viene como un parámetro más ya
       inicializado fuera.
    solucionInicial(S, k, listaCluster)
    mejorFitness <- evaluarFitness(S, X, ML, CL, lambda, k)
    evaluaciones <- 0
    hacer
        viejaS <- S
        vecindarioVirtual <- generarVecinos(S, k, listaCluster)
```

```
        para i<-0 hasta VecindarioVirtual.tamaño() hacer
            vecinoActual <- vecindarioVirtual[i]
            actualS <- S
            actualS[vecinoActual.posicion] = vecinoActual.clusterNuevo
            actualFitness <-  evaluarFitness(actualS, X, ML, CL, lambda, k)
            evaluaciones++

            si(actualFitness < mejorFitness) entonces
                actualizarSolucion(S, vecinoActual, listaCluster)
                mejorFitness <- actualFitness
                break
            fsi

            si(evaluaciones >= 100000) entonces
                break
            fsi
        fpara

    mientras(evaluaciones < 100000 ∧ S != viejaS)
    // En BMB e ILS también se tiene
    // fitness <- mejorFitness
    // que se tiene como un parámetro más de la función.
    retornar S
ffunc
```

Operador de Generación de Vecinos

```
func generarVecinos(vector de entero: S, entero: k, vector de entero: listaCluster)
    :
vector de par[entero, entero]: vecino

inicio:
    // La generación en general consiste que para dada posición i en S, generar
       todos los clusters menos el que ya se encuentra asignado, esto solamente si
       el cluster actual posee más de un miembro.
    para i<-0 hasta S.tamaño() hacer
        para j<-0 hasta k hacer
            // Para evitar repetir el cluster actual, si S[i] == j directamente se
               salta y para asegurarse de que no se sobreescriba un cluster que
               posee 1 solo integrante, si el cluster S[i] posee 1 solo miembro,
               también se salta la generación de vecinos.
            si S[i] != j ∧ listaCluster[S[i]] > 1 entonces
                //Par {posición, cluster}
                vecino.push({i, j})
            fsi
        fpara
    fpara

    // Para recorrer aleatoriamente el vecindario se baraja el vector
    randomShuffle(vecino)

    retornar vecino
ffunc
```

2.5. Búsqueda Local Reiterada

2.5.1. Función Principal

```
funcion ILS(vector de vector de flotante: X, vector de triplet: ML, vector de
   triplet: CL, entero: k, doble: lambda) : vector de entero
variables:
    doble: mejorFitness, fitnessActual
    // S inicializado al tamaño de X y numero de clústers a la cantidad de clusters
        todos a cero.
    vector de entero: S(X.tamaño()), sPrima, sDoblePrima, mejorS, numK(0 , k)
inicio
    solucionInicial(S, k, numK)
    S <- busquedaLocal(S, X, ML, CL, k, lambda, fitnessActual)
    /* Si es ILS-ES esta llamada se cambia por una llamada a la función de
     * Enfriamiento Simulado, el resto se mantiene idéntico.
     */

    mejorS <- S
    mejorFitness <- fitnessActual

    para i<-1 hasta 9 hacer
        sPrima <- mutarSolucion(S, k)
        sDoblePrima <- busquedaLocal(sPrima, X, ML, CL, k, lambda, fitnessActual)

        si (fitnessActual < mejorFitness) entonces
            mejorFitness <- fitnessActual
            mejorS <- S
        fsi

        S <- mejorS

    fpara

    retornar mejorS

ffunc
```

2.5.2. Operador de Mutación

Se realiza la mutación como se especifica, inicialmente se copia todo `S` en `nuevoS` y luego, dado un valor aleatorio de dónde comenzar la mutación, se empieza a mutar cada instancia en el rango de `seg` y `seg+tamañoSegmento mod m`, siempre tomando en cuenta que no se deje ningún clúster sin instancias.

```
funcion mutarSolucion(vector de entero: S, entero: k) : vector de entero
variables:

    entero: seg, tamañoSegmento, m, nuevoCluster
    vector de entero: cuentaCluster(k, 0), nuevoS

inicio

    seg <- RandInt(0, S.tamaño())
```

```
    tamañoSegmento <- 0.1 * S.tamaño()
    m <- S.tamaño()

    nuevoS <- S
    // Rellena el contador de clusters como está en la solución actual.
    para i<-1 hasta S.tamaño() hacer
        cuentaCluster[ nuevoS[i] ]++
    fpara

    para i<-1 hasta tamañoSegmento hacer

        nuevoCluster = RandInt(0, k)

        // Si en el cluster elegido tiene más de un miembro, aceptar el cambio.
        si( cuentaCluster[ nuevoS[(i + seg)  % m ] ] > 1)
            // Reducir el número de miembros del clúster original.
            cuentaCluster[ nuevoS[(i + seg)  % m ] ]--
            // Aumentarlo en el nuevo.
            cuentaCluster[nuevoCluster]++
            // Realizar el cambio.
            nuevoS[nuevoS[(i + seg)  % m] = nuevoCluster
        fsi

    fpara

    retornar nuevoS

ffunc
```

2.6. Manual de Usuario

2.6.1. Compilación

Para la compilación, se ha provisto de un fichero `Makefile` el cual posee las siguientes recetas:

- `Make all`: Compila todos los algoritmos, así como la librería `random.h` proporcionada.
- `Make ES_exe`: Compila el ejecutable del algoritmo de Enfriamiento Simulado.
- `Make BMB_exe`: Compila el ejecutable del algoritmo de Búsqueda Multiarranque Básica
- `Make ILS_exe`: Compila el ejecutable que contiene los algoritmos ILS e ILS-ES.
- `Make clean_obj`: Elimina los ficheros objeto.
- `Make clean`: Elimina los ejecutables y los ficheros objeto.

Existen otras recetas pero estas son utilizadas de manera interna por lo que se recomienda utilizar solo las recetas presentadas.

2.6.2. Ejecución

Para realizar una única ejecución, los algoritmos utilizan los siguientes parámetros de entrada:

```
$./algoritmo_exe n d k setPath constPath randomSeed
```

Donde

- `n`: Número de instancias del conjunto de datos.
- `d`: Número de dimensiones del conjunto de datos.
- `k`: Número de clusters.
- `setPath`: Ruta hacia el fichero `*.dat`.
- `constPath`: Ruta hacia el fichero `*.const`
- `randomSeed`: Número que se utiliza para inicializar el generador de números aleatorios.

Se incluye también un script de Bash el cual se utilizó para obtener los datos requeridos.

```
$bash gatherData_P3.sh
```

El script espera que en la ubicación dónde sea ejecutado exista, además de los programas ejecutables, las carpetas "input", donde estarían los datos de entrada y "output", donde se almacenarán las salidas de los programas siguiendo el formato `algoritmo_restricciones.txt`.

3. Resultados y Análisis

3.1. Casos de Problema y Parámetros Empleados

Para la resolución de esta práctica, se utilizaron los tres grupos de datos provistos en la práctica, estos son:

- Zoo: 101 instancias, 16 atributos y 7 clusters. Se trata de clasificar animales para agruparlos en grupos similares.
- Glass: 214 instancias, 9 atributos y 7 clusters. Se trata de agrupar diferentes tipos de vidrios.
- Bupa: 345 instancias, 5 atributos y 16 clusters. Se trata de agrupar personas dependiendo de sus hábitos de consumo de alcohol.

Además, se incluye también otro pequeño grupo de datos que fue utilizado para poder depurar los algoritmos, denominado simplemente "Test", fue inspirado en el ejemplo que se puede observar en las diapositivas del Seminario 2 de la asignatura. Posee 8 instancias, 2 atributos y 2 clusters.

Referente a los parámetros del algoritmo, se utilizaron los grupos de datos las instancias, atributos y clusters originales. Se hizo uso de la página web `https://www.random.org/integers/` para obtener semillas de números enteros.

Las semillas que se utilizaron son las siguientes:

```
860681 980472 206894 954919 426969
```

3.2. Resultados Obtenidos

Haciendo uso del script de Bash anteriormente mencionado, se obtuvieron los siguientes resultados de 5 ejecuciones de cada variante de los algoritmos, tanto para 10 % como 20 % de restricciones.

3.2.1. Enfriamiento Simulado

	Zoo				Glass				Bupa			
	Infeasable	**Distancia**	**Fitness**	**T**	**Infeasable**	**Distancia**	**Fitness**	**T**	**Infeasable**	**Distancia**	**Fitness**	**T**
Ejecución 1	20	0,758478	0,923088	0,503541	48	0,230801	0,297252	1,26029	127	0,119063	0,164163	10,1518
Ejecución 2	11	0,641073	0,731608	0,55802	47	0,213129	0,278196	1,31555	107	0,121842	0,159839	14,5885
Ejecución 3	18	0,702461	0,850609	0,387912	73	0,189308	0,29037	1,30855	136	0,123479	0,171775	9,81104
Ejecución 4	11	0,610507	0,701042	0,561161	52	0,199242	0,271231	1,58135	87	0,121858	0,152753	12,7044
Ejecución 5	17	0,709947	0,849864	0,732547	39	0,261455	0,315447	1,14495	78	0,122007	0,149706	12,158
Media	**15,40**	**0,6845**	**0,8112**	**0,5486**	**51,80**	**0,2188**	**0,2905**	**1,32**	**107,00**	**0,1216**	**0,1596**	**11,88**

10 % Restricciones

	Zoo				Glass				Bupa			
	Infeasable	**Distancia**	**Fitness**	**T**	**Infeasable**	**Distancia**	**Fitness**	**T**	**Infeasable**	**Distancia**	**Fitness**	**T**
Ejecución 1	72	0,606308	0,922097	0,458984	71	0,252232	0,304234	1,30951	185	0,129808	0,163992	13,9399
Ejecución 2	65	0,855446	1,14053	0,336535	114	0,25006	0,333556	1,6202	157	0,127076	0,156086	13,0711
Ejecución 3	38	0,771787	0,938454	0,284728	78	0,253165	0,310294	1,15381	248	0,120396	0,16622	10,946
Ejecución 4	31	0,729943	0,865908	0,290132	64	0,253911	0,300786	0,821113	115	0,131178	0,152427	16,0636
Ejecución 5	30	0,761582	0,893161	0,295662	62	0,251017	0,296427	1,33949	174	0,120452	0,152603	11,5102
Media	**47,20**	**0,7450**	**0,9520**	**0,3332**	**77,80**	**0,2521**	**0,3091**	**1,25**	**175,80**	**0,1258**	**0,1583**	**13,11**

20 % Restricciones

3.2.2. Búsqueda Multiarranque Básica

	Zoo				Glass				Bupa			
	Infeasable	**Distancia**	**Fitness**	**T**	**Infeasable**	**Distancia**	**Fitness**	**T**	**Infeasable**	**Distancia**	**Fitness**	**T**
Ejecución 1	9	0,617829	0,691903	5,03253	31	0,21148	0,254396	16,9453	420	0,170465	0,319613	60,2351
Ejecución 2	10	0,617217	0,699522	4,5429	51	0,185209	0,255813	17,023	448	0,151212	0,310303	62,2699
Ejecución 3	11	0,629907	0,720442	5,02625	28	0,227799	0,266562	17,8539	424	0,164329	0,314897	59,889
Ejecución 4	9	0,61324	0,687314	4,22401	52	0,187794	0,259783	17,4482	340	0,165354	0,286093	62,1714
Ejecución 5	10	0,623538	0,705842	4,4618	32	0,211974	0,256275	17,985	375	0,153536	0,286704	60,3838
Media	**9,80**	**0,6203**	**0,7010**	**4,6575**	**38,80**	**0,2049**	**0,2586**	**17,45**	**401,40**	**0,1610**	**0,3035**	**60,99**

10 % Restricciones

	Zoo				Glass				Bupa			
	Infeasable	**Distancia**	**Fitness**	**T**	**Infeasable**	**Distancia**	**Fitness**	**T**	**Infeasable**	**Distancia**	**Fitness**	**T**
Ejecución 1	14	0,743321	0,804724	3,72964	31	0,248821	0,271526	16,8962	744	0,157718	0,295191	61,3832
Ejecución 2	11	0,738038	0,786284	3,58743	36	0,243035	0,269402	16,0165	650	0,153988	0,274091	62,1045
Ejecución 3	9	0,723794	0,763268	3,71644	31	0,241654	0,26436	15,4468	648	0,161448	0,281182	60,7491
Ejecución 4	39	0,598241	0,769293	3,98729	30	0,241691	0,263663	15,3636	663	0,166623	0,289128	61,534
Ejecución 5	18	0,714041	0,792988	4,33118	29	0,25059	0,27183	16,1487	720	0,163204	0,296241	60,8268
Media	**18,20**	**0,7035**	**0,7833**	**3,8704**	**31,40**	**0,2452**	**0,2682**	**15,97**	**685,00**	**0,1606**	**0,2872**	**61,32**

20 % Restricciones

3.2.3. Búsqueda Local Reiterada

	Zoo				Glass				Bupa			
	Infeasable	**Distancia**	**Fitness**	**T**	**Infeasable**	**Distancia**	**Fitness**	**T**	**Infeasable**	**Distancia**	**Fitness**	**T**
Ejecución 1	7	0,606577	0,664191	2,87112	45	0,210193	0,272491	12,1946	112	0,121628	0,161401	62,3838
Ejecución 2	9	0,609193	0,683267	2,57923	18	0,248887	0,273806	10,9192	85	0,131273	0,161458	62,5701
Ejecución 3	8	0,607618	0,673461	2,61367	16	0,248128	0,270278	10,1514	134	0,130996	0,178581	62,445
Ejecución 4	8	0,60932	0,675164	2,49137	11	0,248801	0,264029	10,9287	92	0,126731	0,159401	61,7017
Ejecución 5	9	0,619512	0,693586	2,57803	51	0,195181	0,265786	11,8548	114	0,123476	0,163959	61,9379
Media	**8,20**	**0,6104**	**0,6779**	**2,6267**	**28,20**	**0,2302**	**0,2693**	**11,21**	**107,40**	**0,1268**	**0,1650**	**62,21**

10 % Restricciones

	Zoo				Glass				Bupa			
	Infeasable	**Distancia**	**Fitness**	**T**	**Infeasable**	**Distancia**	**Fitness**	**T**	**Infeasable**	**Distancia**	**Fitness**	**T**
Ejecución 1	18	0,718634	0,797581	2,12846	38	0,249065	0,276898	9,40998	227	0,131312	0,173255	63,2559
Ejecución 2	27	0,600638	0,719059	2,31546	39	0,250347	0,278912	9,50483	213	0,12656	0,165917	62,0875
Ejecución 3	24	0,715431	0,820694	2,29219	22	0,248096	0,264209	9,65629	206	0,120883	0,158946	63,3817
Ejecución 4	12	0,710826	0,763458	2,80737	39	0,250713	0,279277	10,0528	223	0,130892	0,172097	62,2885
Ejecución 5	16	0,733622	0,803798	2,38372	27	0,250981	0,270756	9,52821	199	0,127776	0,164547	63,9498
Media	**19,40**	**0,6958**	**0,7809**	**2,3854**	**33,00**	**0,2498**	**0,2740**	**9,63**	**213,60**	**0,1275**	**0,1670**	**62,99**

20 % Restricciones

3.2.4. Algoritmo Híbrido ILS-ES

	Zoo				Glass				Bupa			
	Infeasable	**Distancia**	**Fitness**	**T**	**Infeasable**	**Distancia**	**Fitness**	**T**	**Infeasable**	**Distancia**	**Fitness**	**T**
Ejecución 1	13	0,628291	0,735287	5,5299	29	0,252638	0,292786	14,5871	322	0,156454	0,270801	31,9345
Ejecución 2	10	0,639288	0,721593	4,68855	20	0,247841	0,275529	13,1197	321	0,159886	0,273877	31,2614
Ejecución 3	15	0,663913	0,78737	5,6236	43	0,21107	0,2706	12,6861	319	0,164259	0,27754	31,2516
Ejecución 4	4	0,717516	0,750438	4,27864	33	0,225838	0,271524	12,6584	342	0,164701	0,286149	31,2922
Ejecución 5	16	0,617676	0,749363	4,72939	59	0,203641	0,285321	12,5819	285	0,151495	0,252702	31,4452
Media	**11,60**	**0,6533**	**0,7488**	**4,9700**	**36,80**	**0,2282**	**0,2792**	**13,13**	**317,80**	**0,1594**	**0,2722**	**31,44**

10 % Restricciones

	Zoo				Glass				Bupa			
	Infeasable	**Distancia**	**Fitness**	**T**	**Infeasable**	**Distancia**	**Fitness**	**T**	**Infeasable**	**Distancia**	**Fitness**	**T**
Ejecución 1	28	0,718036	0,840843	3,98961	60	0,250192	0,294138	12,7128	867	0,160763	0,320963	32,4033
Ejecución 2	25	0,641449	0,751098	4,01553	66	0,25066	0,299	12,5637	575	0,158431	0,264677	32,2683
Ejecución 3	17	0,732119	0,806681	3,97482	60	0,248143	0,292088	13,0117	574	0,160471	0,266531	31,7343
Ejecución 4	9	0,733168	0,772642	3,8578	45	0,250371	0,28333	13,8573	602	0,161257	0,272491	31,4522
Ejecución 5	47	0,602376	0,808517	3,92525	40	0,250345	0,279642	13,2244	710	0,151731	0,282921	31,6463
Media	**25,20**	**0,6854**	**0,7960**	**3,9526**	**54,20**	**0,2499**	**0,2896**	**13,07**	**665,60**	**0,1585**	**0,2815**	**31,90**

20 % Restricciones

3.3. Resultados Globales

	Zoo				Glass				Bupa			
	Infeasable	**Distancia**	**Fitness**	**T**	**Infeasable**	**Distancia**	**Fitness**	**T**	**Infeasable**	**Distancia**	**Fitness**	**T**
COPKM	31,60	1,0096	1,2697	0,3691	435,60	0,4077	1,0108	8,3806	798,60	0,2026	0,4862	94,9271
BL	16,00	0,6916	0,8233	0,5457	26,80	0,2331	0,2702	1,9462	109,20	0,1236	0,1624	57,5137
ES	15,40	0,68	0,81	0,55	51,80	0,22	0,29	1,32	107,00	0,12	0,16	11,88
BMB	9,80	0,6203	0,7010	4,6575	38,80	0,2049	0,2586	17,4511	401,40	0,1610	0,3035	60,9898
ILS	8,20	0,6104	0,6779	2,6267	28,20	0,2302	0,2693	11,2097	107,40	0,1268	0,1650	62,2077
ILS-ES	11,60	0,65	0,75	4,97	36,80	0,23	0,28	13,13	317,80	0,16	0,27	31,44

10 % Restricciones

	Zoo				Glass				Bupa			
	Infeasable	**Distancia**	**Fitness**	**T**	**Infeasable**	**Distancia**	**Fitness**	**T**	**Infeasable**	**Distancia**	**Fitness**	**T**
COPKM	30,60	1,1434	1,2776	0,2994	323,40	0,4031	0,6400	10,2299	1.471,80	0,2424	0,5143	77,2708
BL	25,40	0,7356	0,8470	0,3061	42,80	0,2489	0,2803	1,8000	171,60	0,1248	0,1565	39,4667
ES	47,20	0,75	0,95	0,33	77,80	0,25	0,31	1,25	175,80	0,13	0,16	13,11
BMB	18,20	0,7035	0,7833	3,8704	31,40	0,2452	0,2682	15,97	685,00	0,1606	0,2872	61,3195
ILS	19,40	0,6958	0,7809	2,3854	33,00	0,2498	0,2740	9,63	213,60	0,1275	0,1670	62,9927
ILS-ES	25,20	0,69	0,80	3,95	54,20	0,25	0,29	13,07	665,60	0,16	0,28	31,90

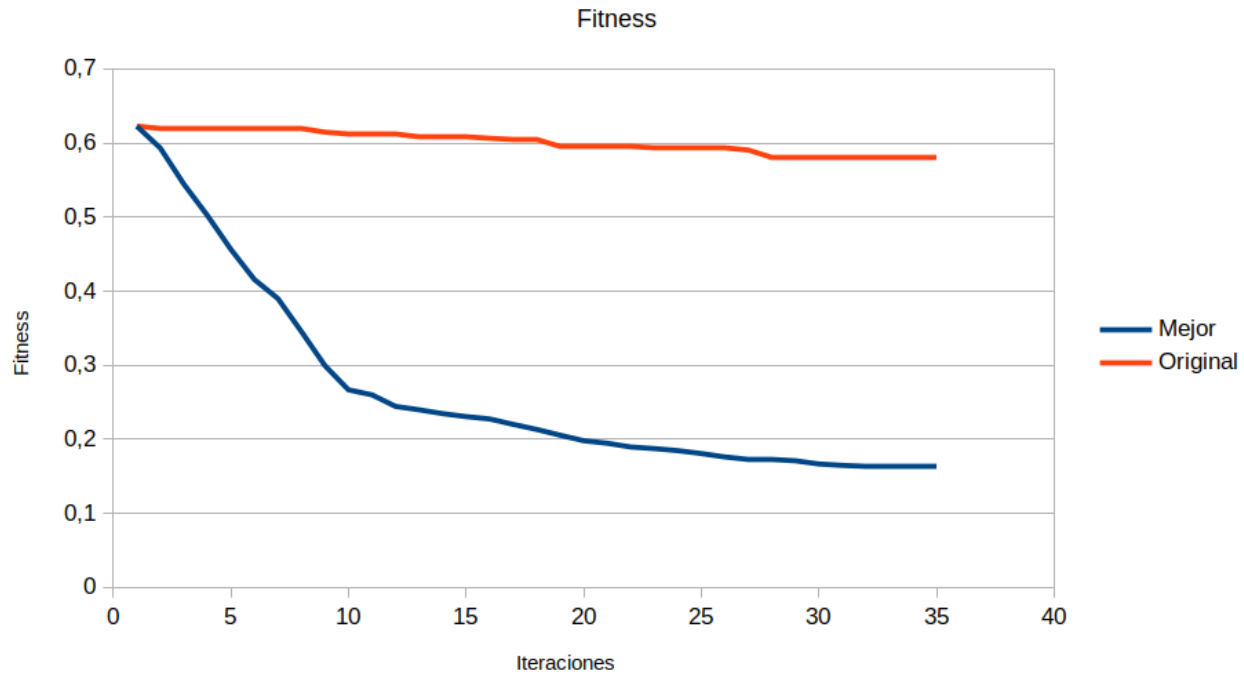
20 % Restricciones

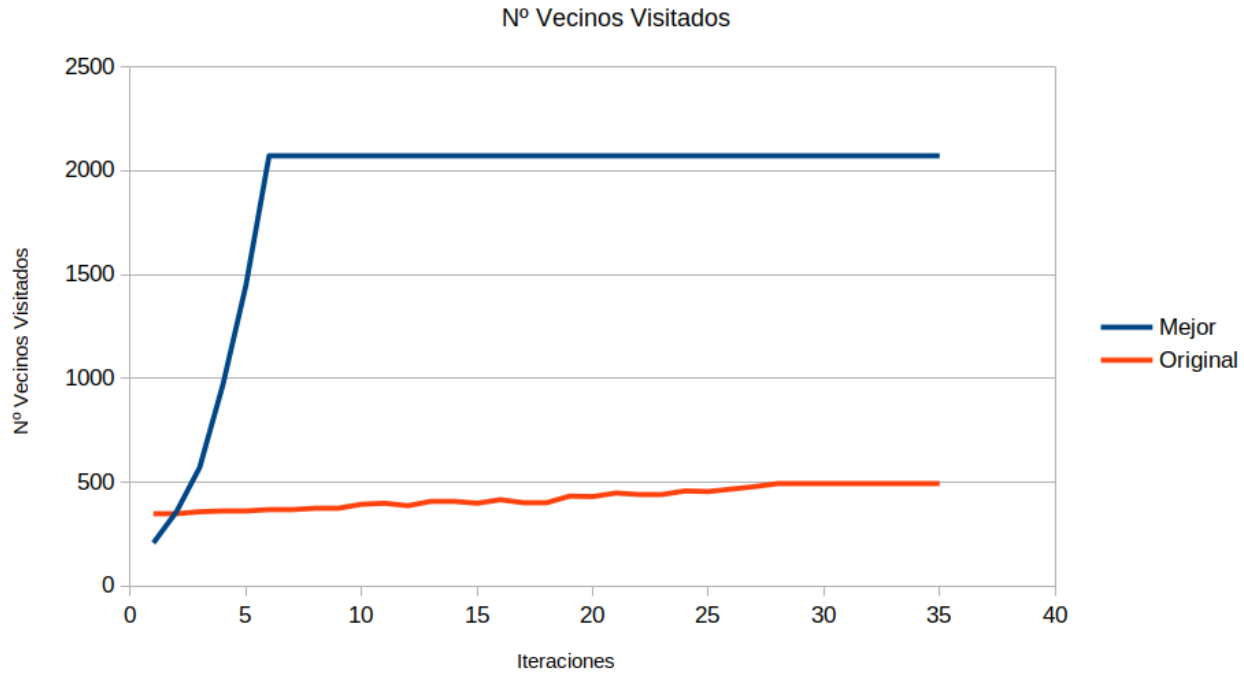
3.4. Análisis de Resultados

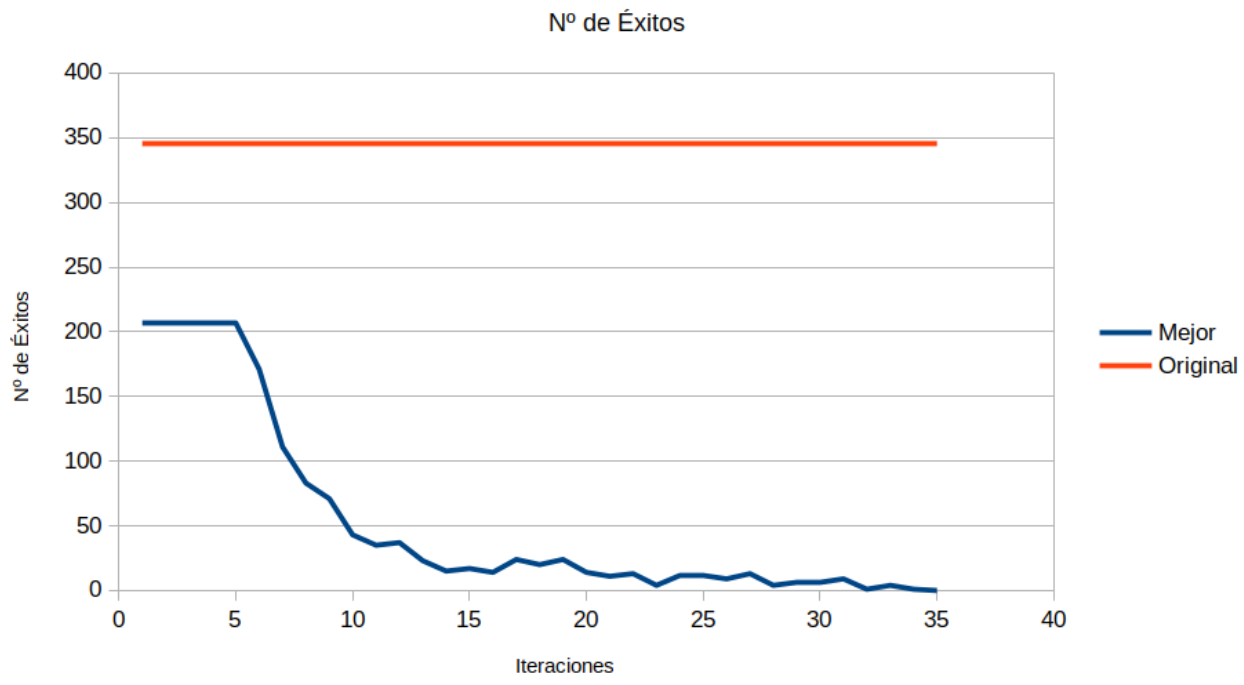
Los resultados globales a primera vista, se puede notar que todos los algoritmos tienen un desempeño similar en cuanto a fitness respecto a la Búsqueda Local original, sin embargo, hay que hacer notar ciertas cosas para obtener estos resultados.

En primer lugar con el algoritmo de Enfriamiento Simulado. Este algoritmo es muy sensible a sus parámetros, esto quiere decir que la decisión de las temperaturas, el esquema de enfriamiento, la cantidad de vecinos o de éxitos tienen un peso muy grande a la hora de obtener buenas soluciones.

Originalmente, los parámetros del algoritmo indicaban utilizar `maxVecinos <- 10 * X.tamaño` y `tFinal <- 0.001`; con estos dos parámetros los resultados obtenidos eran relativamente buenos en los grupos de datos más pequeños, es decir, Zoo y Glass. Pero los resultados con Bupa daban más que esperar; indagando en las ejecuciones se podía observar que para este grupo de datos, el algoritmo no terminaba de explorar cuando ya se acababan las iteraciones, es decir, ocurría que se llegaba al máximo de éxitos y no al de vecinos explorados. Según los datos que se obtuvieron, aún no le daba tiempo al algoritmo de asentarse por el la cantidad de datos y de restricciones que tiene. Se puede observar en la gráfica como se está enfriando muy rápido debido a que el fitness baja muy poco, no se llega a la meseta del máximo de vecinos visitados cuando ya se terminan las iteraciones.


Fitness
0,7
0,6
0,5
0,4
0,3
0,2
0,1
0
Fitness
0
5
10
15
20
25
30
35
40
Iteraciones
Mejor
Original


Nº Vecinos Visitados
2500
2000
1500
1000
500
0
Nº Vecinos Visitados
0
5
10
15
20
25
30
35
40
Iteraciones
Mejor
Original



BUPA 10 % - Evolución de Diferentes Parámetros por Iteración

A partir de distintas pruebas, incluyendo cambiar el esquema de enfriamiento a uno proporcional, reducir o ampliar la cantidad de vecinos y de temperaturas, se llegó al balance descrito anteriormente, con `maxVecinos <- 6 * X.tamaño` y `tFinal <- 0.00001`.

La razón es que de esta manera se dan más iteraciones a los algoritmos y además que al bajar la temperatura final, dado que ahora los valores de temperatura serian más bajos esto cierra la oportunidad de que entren soluciones menos factibles por el criterio de metrópolis, ya que a menor cantidad de temperatura más reducido sale el número y es más difícil que se acepte una solución peor.

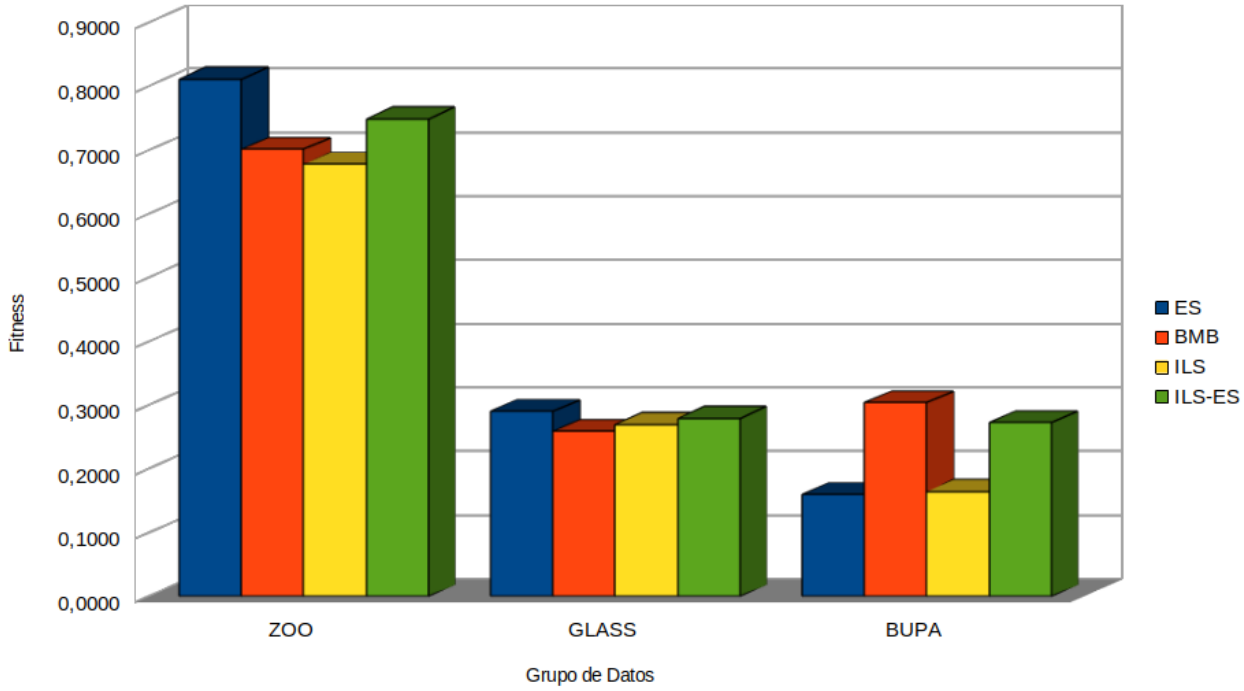
Estos dos parámetros permitieron mejorar significativamente la calidad del algoritmo para todos los conjuntos de datos. Cabe notar que en la configuración original se obtenían para Glass y Zoo mejores resultados, pero se prefirió que aunque se diesen resultados ligeramente peores en estos dos grupos de datos, mejorase la calidad de Bupa.

Para la Búsqueda Multiarranque Básica, lo que se puede concluir es que es una técnica sumamente útil porque aprovecha ya la utilizad de explotar de la Búsqueda Local y sencillamente, realizando múltiples arranques aleatorios permite que el algoritmo BL se mueva por el espacio de búsquedas, es otra manera de ver como explorar el mismo; a diferencia de otras técnicas que mutan la solución, aquí se prefiere simplemente ir probando aleatoriamente por todo el espacio de búsquedas e ir almacenando la mejor solución encontrada. Se puede notar que los resultados obtenidos por BMB son muy buenos para Zoo y Glass. En Bupa es ligeramente peor pero aún así es una técnica factible ya que además de dar buenos resultados es muy sencilla de implementar y los tiempos de ejecución son aceptables.

En la Búsqueda Local Reiterada o ILS, a diferencia de BMB en cuanto a explorar el espacio,

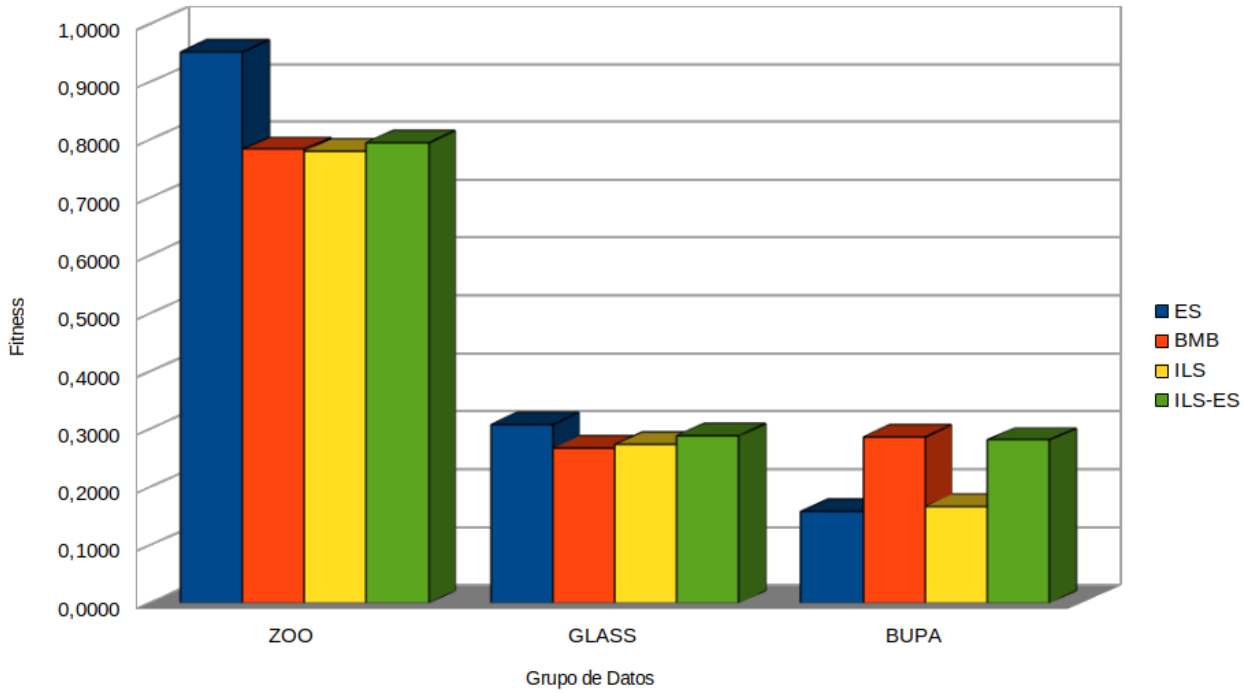
aquí para el elemento de exploración se realiza una mutación fuerte de la solución actual. Es otra manera sumamente válida de ir variando la solución, ya que al realizarse con la mejor solución del momento, se puede tener una cierta seguridad que se está explorando una zona del espacio de soluciones cercano a un óptimo al que se desea llegar, y efectivamente, ILS es la técnica que mejora todos los resultados de todos los grupos de datos superando a BL tanto en 10 % como 20 % de restricciones, es admisible concluir que esta técnica es muy poderosa porque combina de manera excepcional la exploración y explotación de las soluciones. La variante ILS-ES da resultados más pobres, aunque igualmente aceptables; se encuentran aún mejores que los del Greedy COPKM. La diferencia entre esta y la ILS original es el cambiar a Enfriamiento Simulado como función de explotación.

Se utilizaron los mismos parámetros que daban la mejor solución para el problema original, pero como ahora se reducen las iteraciones a 10000 en vez de 100000, esto afecta el funcionamiento en general, el problema recae en que se debería realizar un estudio aparte para optimizar los parámetros de esta versión del ES, nuevamente demostrando la sensibilidad de esta técnica a sus parámetros iniciales a diferencia de la BL.



Comparación de Fitness para 10 % Restricciones

Para este grupo de datos se puede observar que los resultados que se obtienen son consistentes tanto para 10 % como para 20 % de restricciones, con algún algoritmo teniendo una ligera ventaja en ciertos datos.



Comparación de Fitness para 20 % Restricciones

En general se puede concluir que las técnicas de Búsquedas por Trayectorias dan muy buenos resultados, aunque en ciertos casos no logren superar la Búsqueda Local, se encuentran en cerca de la calidad de la misma y además, poseen la ventaja de que su ejecución es rápida y son algoritmos de fácil implementación a diferencia de los algoritmos genéticos, que si bien son una herramienta poderosa, son también algoritmos lentos y que requieren un estudio exhaustivo del problema para que con ellos se obtengan resultados favorables, de igual manera, la calidad es comparable con los genéticos pero son indudablemente mucho más rápidos.