UNIVERSIDAD DE GRANADA

ETSIIT
Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

Práctica 1: Técnicas de Búsqueda Local y Algoritmos Greedy 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 dos algoritmos para resolver el PAR, un algoritmo de comparación Greedy basado en *k-means* y un algoritmo de Búsqueda Local.

Ambos 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 $j \in \{0, .., k-1\}$.

De igual forma, 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.

El resto de la implementación varía de este punto en adelante debido a las necesidades particulares de cada algoritmo.

2.2. Función Objetivo y Operadores

La función objetivo a minimizar en los algoritmos será la llamada "Fitness" la cual se describe de la siguiente manera:

```
funcion fitness(matriz X, vector S, vector de triplet ML, CL, lambda, k):
   doble
```

```
inicio:
    retornar intraClusterDistance(S, X, k) + (lambda * infeasibility(S, ML,
       CL))
ffunc
```

De los cuales estos operadores se definen como:

```
funcion intraClusterDistance(vector S, matriz X, k): doble
inicio:
    para i=0 hasta k hacer
        // Obtener las instancias asociadas a un cluster.
        instancia <- calcularInstancia(X, i, S)
        // Obtener las coordenadas del centroide de un cluster.
        coords <- calcularCoordenadasCentroide(instancia, X)
        //Obtener la distancia intracluster de un cluster.
        distanciaAct = calcIntraDiff(coords, instancia, X)
        totalDist += actDiff
    fpara

    retornar totalDist / k
ffunc
```

```
funcion calcIntraDiff(vector coords, vector instancia, matriz X): doble
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 / instancias
ffunc
```

```
funcion infeasibility(vector S, vector de triplet ML, vector de triplet CL):
    entero
inicio:
    contador <- 0
    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
```

2.3. Algoritmo de Búsqueda Local

```
funcion BL(matriz X, vector de triplet ML, CL, entero k, doble lambda):
   vector de entero
inicio:
    solucionInicial(S, k, listaCluster)
    mejorFitness <- evaluarFitness(S, X, ML, CL, lambda, k)
    iteraciones <- 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)
            iteraciones++

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

            si(iteraciones >= 100000) entonces
                break
            fsi
        fpara

    mientras(iteraciones < 100000 && S != viejaS)

    retornar S
ffunc
```

```
procedimiento solucionInicial(vector &S, entero k, vector 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
```

```
func generarVecinos(vector S, entero k, vector listaCluster): vector de
   pares
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 n 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
```

2.4. Algoritmo Greedy COPKM

```
funcion greedy(matriz X, matriz MR, entero k, vector centroides): vector de
   enteros
inicio
    randomShuffle(RSI)
    hacer
        viejoC <- C

        para i=0 hasta n hacer
            // Chequear la infeasibilidad de asignar la instancia i al
               cluster j
            para j=0 hasta k hacer
                // Esta función es diferente a la infeasibility
                   anteriormente declarada
```

```
                inferiorC[j] = infeasibilityGreedy(RSI[i], j, C, MR)
            fpara

            // De todos los clusters, elegir aquel de menor infeasibilidad
               ...
            maxInfeas <- INT_MAX
            para j=0 hasta k hacer
                si(inferiorC[j] <= maxInfeas) entonces
                    si(inferiorC[j] != maxInfeas) entonces
                        masBajoC.clear()
                    fsi
                    masBajoC.push(inferiorC[j])
                    maxInfeas <- inferiorC[j]
                fsi
            fpara

            // ... si hay mas de uno, aquel que tenga la distancia mínima a
               i.
            si inferiorC.tamaño() > 1 entonces
                C[RSI[i]] <- distanciaMinima(X, centroides, masBajoC, RSI[i
                   ])
            sino
                C[RSI[i]] <- inferiorC.frente()
            fsi

        fpara

        // Actualizar centroides y repetir hasta que no hayan más cambios.
        para i=0 hasta k hacer
            actualizarDistancia(i, centroides, X, C)
        fpara

    mientras (C != viejoC)
    retornar C
ffunc
```

```
funcion infeasibilityGreedy(entero instancia, entero cluster, vector C,
   matriz MR): entero
inicio:
    contador <- 0
    para i=0 hasta C.tamaño() hacer
        si (C[i] == cluster) entonces
            // La función V posee la funcionalidad descrita en el guión.
            contador += V(MR, C, instancia, i, cluster)
        fsi
    fpara

    retornar contador
ffunc
```

3. Procedimiento considerado al desarrollar la práctica

Para esta práctica se consideró el uso del lenguaje de programación C++ debido a su versatilidad, la familiaridad del autor con el mismo y también por ser un lenguaje rápido en su ejecución, a diferencia de lenguajes interpretados, esta consideración se realiza debido a la advertencia de que los tiempos de ejecución pueden ser elevados en los algoritmos.

Los códigos fueron implementados por el autor basándose en las diapositivas del Seminario 1, junto con el guión de prácticas y en particular para el algoritmo de Búsqueda Local, de las diapositivas del tema visto en teoría.

Se ha hecho uso de la librería proporcionada para la generación de números aleatorios en los algoritmos además de librerías de C++ Estándar ya sea para barajar un vector o tomar el tiempo de ejecución.

3.1. Manual de Usuario

3.1.1. Compilación

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

- `Make all`: Compila ambos algoritmos, así como la librería `random.h` proporcionada.
- `Make COPKM_exe`: Compila el ejecutable del algoritmo Greedy.
- `Make BL_exe`: Compila el ejecutable del algoritmo de Búsqueda Local.
- `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.

3.1.2. 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, es decir, para realizar las 30 ejecuciones requeridas. Para utilizarlo solamente se ha de llamar desde la consola:

```
$bash gatherData.sh
```

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

4. Resultados y Análisis

4.1. 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 set de datos que fue utilizado para poder depupar 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, aunque ciertas semillas fueron posteriormente reemplazadas debido a los resultados obtenidos en el algoritmo COPKM.

Las semillas que se utilizaron son las siguientes:

```
860681 980472 206894 161195 121263
```

4.2. Resultados Obtenidos

Haciendo uso del script de Bash anteiormente mencionado, se obtenieron los siguientes datos de realizar un total de 30 ejecuciones, cinco ejecuciones por cada grupo de datos, con 10 % de restricciones y otros cinco con 20 % de restricciones; todo esto con el algoritmo Greedy COPKM y otra vez con el algoritmo de Búsqueda Local.

4.2.1. Algoritmo Greedy COPKM

	Zoo				Glass				Bupa			
	Infeasable	**Error_Dist**	**Agr.**	**T**	**Infeasable**	**Error_Dist**	**Agr.**	**T**	**Infeasable**	**Error_Dist**	**Agr.**	**T**
Ejecución 1	51	0,1471	1,4716	0,2941	432	0,0601	1,02248	6,50765	778	0,0171	0,479558	93,756
Ejecución 2	30	0,1778	1,3295	0,3146	397	0,0233	0,937155	6,81128	815	0,0204	0,489479	92,2859
Ejecución 3	33	0,1155	1,2919	0,4338	445	0,0617	1,04204	7,9188	790	0,0173	0,483622	101,911
Ejecución 4	34	0,0132	1,1978	0,6411	445	0,0395	1,01986	9,08787	805	0,0142	0,492072	93,0941
Ejecución 5	10	0,0703	1,0575	0,1617	459	0,0325	1,03226	11,5772	805	0,0199	0,486365	93,5884
Media	**31,60**	**0,1048**	**1,2697**	**0,3691**	**435,60**	**0,04**	**1,01**	**8,38**	**798,60**	**0,02**	**0,49**	**94,93**

10 % Restricciones

	Zoo				Glass				Bupa			
	Infeasable	**Error_Dist**	**Agr.**	**T**	**Infeasable**	**Error_Dist**	**Agr.**	**T**	**Infeasable**	**Error_Dist**	**Agr.**	**T**
Ejecución 1	31	0,3695	1,17674	0,347422	508	0,0492	0,78558	10,0454	1364	0,0226	0,495019	78,4901
Ejecución 2	31	0,3027	1,20823	0,271955	333	0,0219	0,630105	6,11537	1539	0,0207	0,525512	68,7008
Ejecución 3	32	0,2174	1,41465	0,147763	367	0,0464	0,679444	7,5101	1413	0,0222	0,503663	81,7358
Ejecución 4	27	0,3027	1,32595	0,361867	254	0,0510	0,601366	9,173	1513	0,0231	0,523062	79,8078
Ejecución 5	32	0,2174	1,2625	0,367815	155	0,0257	0,503554	18,3055	1530	0,0212	0,524294	77,6195
Media	**30,60**	**0,2819**	**1,2776**	**0,2994**	**323,40**	**0,04**	**0,64**	**10,23**	**1.471,80**	**0,02**	**0,51**	**77,27**

20 % Restricciones

4.2.2. Algoritmo de Búsqueda Local

	Zoo				Glass				Bupa			
	Infeasable	**Error_Dist**	**Agr.**	**T**	**Infeasable**	**Error_Dist**	**Agr.**	**T**	**Infeasable**	**Error_Dist**	**Agr.**	**T**
Ejecución 1	21	0,2040	0,964562	0,558438	26	0,1273	0,273015	1,76895	85	0,0971	0,153543	52,1666
Ejecución 2	11	0,1561	0,699385	0,461931	41	0,1524	0,268643	2,10445	135	0,0904	0,177974	62,9562
Ejecución 3	12	0,2970	0,799529	0,471867	12	0,1154	0,265539	2,21285	87	0,0922	0,159114	51,1391
Ejecución 4	25	0,1561	0,954445	0,524284	30	0,1387	0,267151	1,9166	102	0,1025	0,154186	64,7627
Ejecución 5	11	0,2970	0,698333	0,711969	25	0,1222	0,276746	1,7283	137	0,1020	0,167112	56,5437
Media	**16,00**	**0,22**	**0,82**	**0,55**	**26,80**	**0,13**	**0,27**	**1,95**	**109,20**	**0,10**	**0,16**	**57,51**

10 % Restricciones

	Zoo				Glass				Bupa			
	Infeasable	**Error_Dist**	**Agr.**	**T**	**Infeasable**	**Error_Dist**	**Agr.**	**T**	**Infeasable**	**Error_Dist**	**Agr.**	**T**
Ejecución 1	28	0,1560	0,882756	0,258688	37	0,1142	0,277238	1,91436	125	0,0885	0,155008	36,4591
Ejecución 2	35	0,1897	0,871719	0,380453	33	0,1138	0,274694	1,70347	165	0,0978	0,153139	39,2873
Ejecución 3	22	0,1689	0,845309	0,333361	60	0,1146	0,293669	1,48629	186	0,0990	0,155839	43,1435
Ejecución 4	25	0,1897	0,824717	0,227914	40	0,1136	0,279964	1,85157	169	0,0976	0,154056	37,9047
Ejecuión 5	17	0,1689	0,810448	0,33029	44	0,1206	0,275902	2,04448	213	0,0951	0,164645	40,5389
Media	**25,40**	**0,17**	**0,85**	**0,31**	**42,80**	**0,12**	**0,28**	**1,80**	**171,60**	**0,10**	**0,16**	**39,47**

20 % Restricciones

4.2.3. Resultados Globales

	Zoo				Glass				Bupa			
	Infeasable	**Error_Dist**	**Agr.**	**T**	**Infeasable**	**Error_Dist**	**Agr.**	**T**	**Infeasable**	**Error_Dist**	**Agr.**	**T**
COPKM	31,60	0,10	1,27	0,37	435,60	0,04	1,01	8,38	798,60	0,02	0,49	94,93
BL	16,00	0,22	0,82	0,55	26,80	0,13	0,27	1,95	109,20	0,10	0,16	57,51

10 % Restricciones

	Zoo				Glass				Bupa			
	Infeasable	**Error_Dist**	**Agr.**	**T**	**Infeasable**	**Error_Dist**	**Agr.**	**T**	**Infeasable**	**Error_Dist**	**Agr.**	**T**
COPKM	30,60	0,28	1,28	0,30	323,40	0,04	0,64	10,23	1.471,80	0,02	0,51	77,27
BL	25,40	0,17	0,85	0,31	42,80	0,12	0,28	1,80	171,60	0,10	0,16	39,47

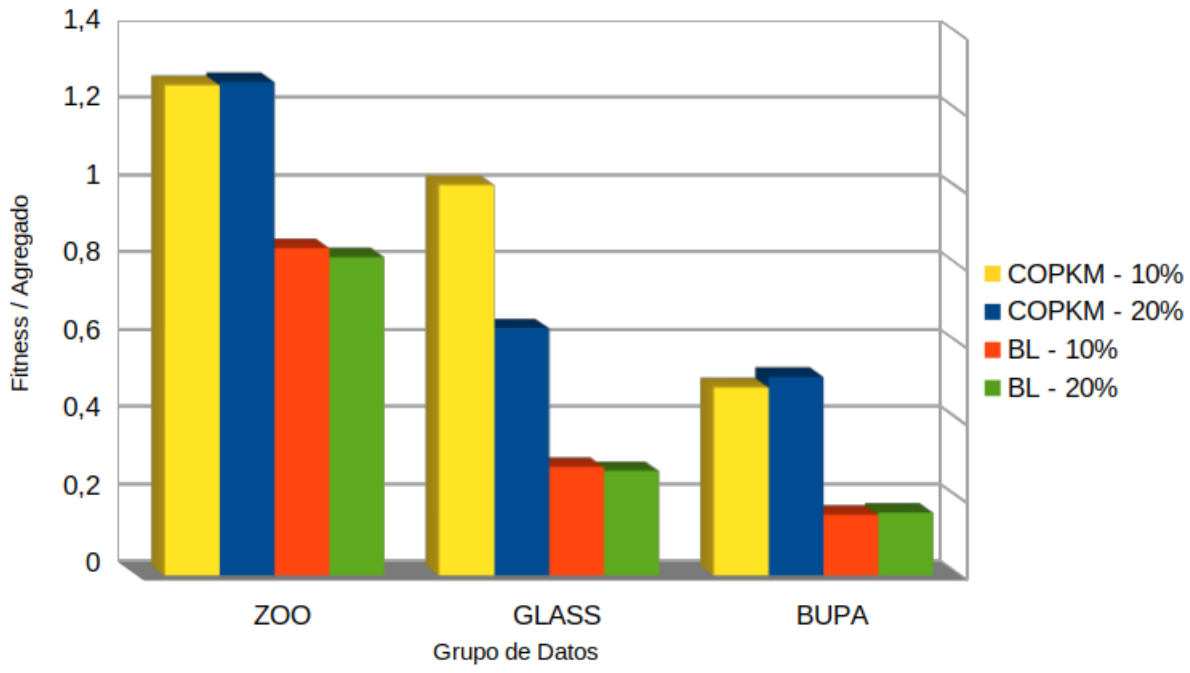
20 % Restricciones

4.3. Análisis de Resultados

Dadas las tablas anteriores de resultados, es inmediatamente obvio la mejora que implica utilizar una metaheurística que un simple Greedy para este problema, en general se puede observar una reducción significativa en el valor Agregado o "Fitness" de la solución.

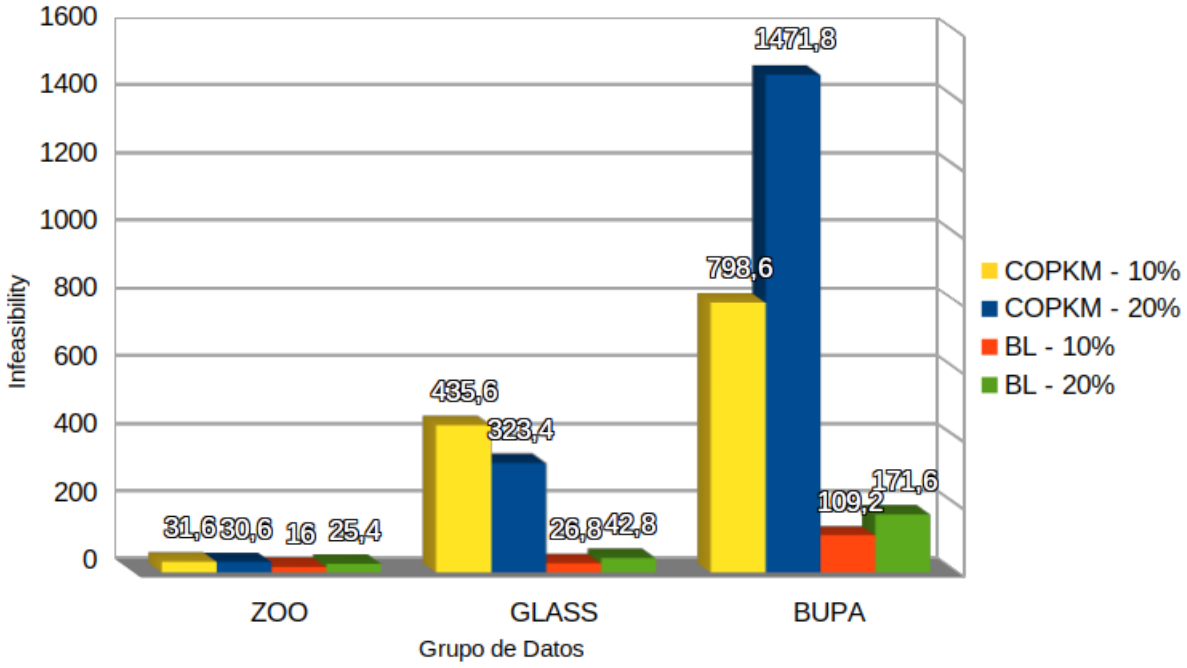
Que el valor Agregado sea menor indica que, por una parte se está obteniendo una Infeasibility reducida y una distancia entre las instancias y el centroide también reducida por parte de la Búsqueda Local, ¿por qué es mejor que el Greedy?

En principio, se puede pensar que el Greedy va probando de una a una instancia, viendo cual cluster le conviene más, aquél con menos restricciones incumplidas y más cercano, esta estrategia tiene el fallo de que al principio de la ejecución cuando la mayoría de los clusters no poseen ningún miembro, una opción que inicialmente parecía contener menos restricciones a la larga resulta que da una infeasibilidad mayor, aunque por otro lado logra una distancia de los clusters más cercana en la mayoría de los casos la cantidad de restricciones incumplidas superan alguna ganancia que se pudiera obtener de tener la distancia cluster reducida, por otra parte, la Búsqueda Local comienza ya con una solución generada, donde al menos cada cluster tiene una instancia, de esta forma este algoritmo ve al problema de una manera más "global", por lo que al generar un vecino y chequear si este vecino mejora la solución, esta mejora ya toma en cuenta todas las instancias, sus restricciones y distancias, obteniendo entonces una solución de mayor calidad por este hecho a pesar de que la distancia intracluster sea mayor.

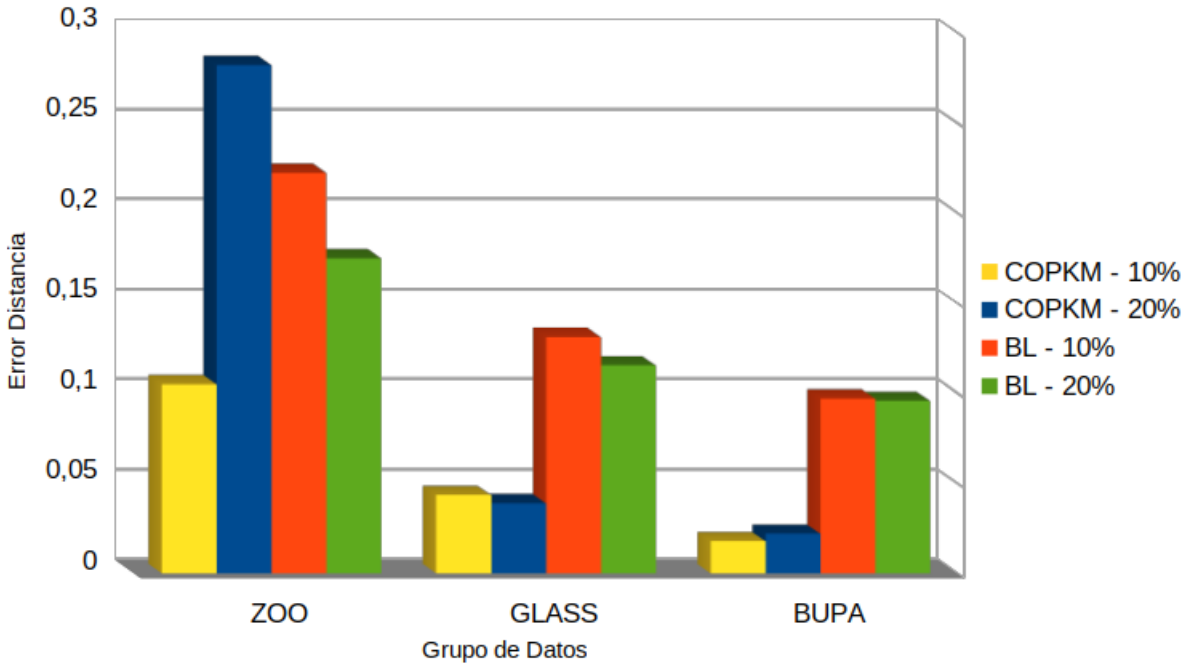


COPKM vs BL – Fitness

De hecho, si se promedia el valor del Fitness tanto del 10 % como del 20 % de Restricciones, se obtiene que la Búsqueda Local realiza una mejora de dos veces sobre el algoritmo Greedy, es decir, redude el valor del Fitness más o menos por la mitad, lo cual se puede corroborar con el gráfico.



COPKM vs BL – Infeasibility

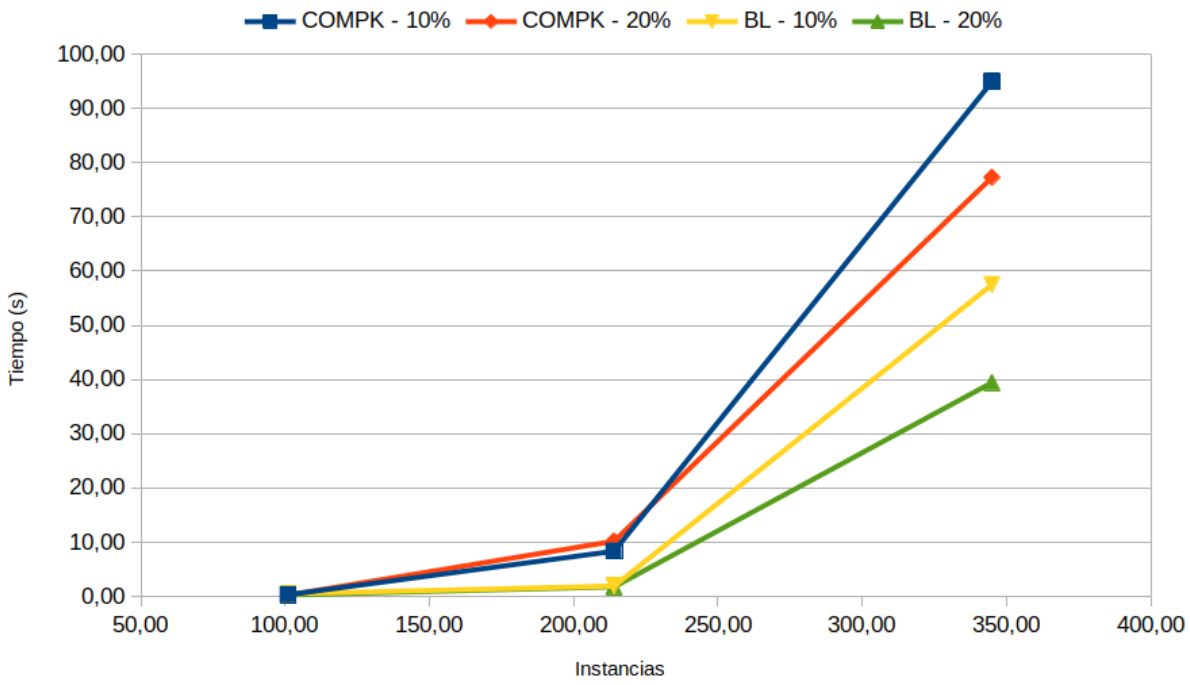


COPKM vs BL – Error de Distancia

Nuevamente, como se ha comentado, si bien el Greedy da un error de distancia más bajo, esto ocurre teniendo muchas más restricciones incumplidas por la manera en que funciona, de hecho, la diferencia de restricciones incumplidas resulta abismal, como puede observarse.

Esta tendencia se mantiene de igual manera para los tiempos de ejecución, puede observarse que sistemáticamente el algoritmo de Búsqueda Local obtiene tiempos mucho menores que los que obtiene el algoritmo Greedy, esto indica que, en primer lugar, el orden de complejidad temporal del

Greedy es mucho mayor que el que posee la Búsqueda Local lo cual resulta lógico pues su manera de resolver el problema requiere de dar muchas más vueltas a los datos, por ejemplo en la obtención de la infeasibility, en el cual es un chequeo muy intensivo ya que debe comparar para cada instancia y para cada cluster, mientras que en la Búsqueda Local esto se realiza con una búsqueda de orden lineal del tamaño de restricciones del problema dado, además que el Greedy realiza una función adicional de asignar el cluster con el centroide más cercano a una instancia, este proceso consume también tiempo porque es necesario chequear cuales de los centroides con menor infeasibility se encuentra más cerca, cosa que no sucede con la Búsqueda Local.



COPKM vs BL – Tiempo de Ejecución

Ambos algoritmos finalizan antes con las restricciones al 20 %, esto podría verse como que se reduce el espacio de soluciones al haber más restricciones y por lo tanto ambos entran en un óptimo local antes.

Se puede concluir entonces que, tanto como en la calidad de la solución como en el tiempo de ejecución, el algoritmo de Búsqueda Local triunfa sobre el Algoritmo Greedy COPKM para resolver el problema PAR.