Práctica 2.b:
Técnicas de Búsqueda basadas en Poblaciones para el Problema del Agrupamiento con Restricciones.

3º CSI
Problema del Agrupamiento con Restricciones (PAR)

AGs y AMs

David Erena Casado 26055001A davidерena@correo.ugr.es
Grupo 1 (Miercoles 17:30 – 19:30)

Indice:

- Problema del Agrupamiento por Restricciones - PAR
- Aplicación de los algoritmos empleados al problema
- Esquema de los algoritmos implementados
- Experimentos y análisis de resultados

Problema del Agrupamiento por Restricciones – PAR:

El agrupamiento o análisis de clusters o clustering en inglés, persigue la clasificación de objetos de acuerdo a posibles similitudes entre ellos.

El problema del PAR consiste en una generalización del agrupamiento clásico. Permite incorporar al proceso de agrupamiento un nuevo tipo de información: las restricciones. Dado un conjunto de datos X con n instancias, el problema consiste en encontrar una partición $C=\{c_1,\dots,c_{1k}\}$ del mismo que minimice la desviación general y cumpla con las restricciones de instancia existentes en el conjunto R. Dada una pareja de instancias, se establece una restricción de tipo Must-Link (ML) si deben pertenecer al mismo grupo y de tipo Cannot-Link (CL) si no pueden pertenecer al mismo grupo.

A partir de ahora, consideraremos n como el número de instancias/datos y k como el número de clústers que consideramos para el problema.

Aplicación de los algoritmos empleados al problema:

En cuanto a la forma de representar una solución, he decidido hacerlo de dos formas:

S1.- Un vector donde la posición i indica el cluster al que se asigna el dato i.

S2.- Un vector de vectores donde el vector i almacena los índices de los datos que pertenecen al cluster i.

Aunque a la hora de mostrar la solución uso S1.

La razón es que acceder a los datos de un cluster es una tarea que se va a realizar muchas veces a lo largo de la ejecución, y si solo trabajásemos con S1 tendríamos que recorrer el vector entero para poder guardar los datos pertenecientes al cluster, mientras que usando S2 no necesitamos esto ya que lo tenemos actualizado en el momento.
A su vez, S1 es necesaria ya que, aparte de para mostrar la solución, para operaciones lógicas (como ver si dos elementos pertenecen al mismo cluster) se complica con S2 (recorrer S2 cada vez que comprobemos algo), y con S1 es inmediato.

Se pierde un poco en espacio y un poco en velocidad al tener que tener ambos actualizados, pero con lo que ahorramos en las operaciones es una mejora.

_____ Funciones comunes: _____

calcularLambda

Se usa una vez al principio del programa para calcular cuánto penalizan las infracciones cometidas.
Se calcula dividiendo la mayor distancia entre dos datos del conjunto (siendo la distancia entre dos datos el módulo del vector resta de estos) entre el número de restricciones existentes.

calcularCentroide

Calcula el centroide asociado a un cluster, realizando la media aritmética de las características de los datos que pertenezcan a este.

vectorCentroides

Calcula los centroides de todos los clusters, usando la función calcularCentroide.

Ci :

Calcula la distancia intracluster, es decir, la media aritmética de distancias entre cada elemento de un cluster y su centroide (módulo de la resta de estos dos vectores).

C :

Calcula la media aritmética de los Ci's de cada cluster.

violaRestriccion :

Nos dice si una solución viola una restricción de la matriz de restricciones, es decir, si dos datos dados tenían una relación ML no están en el mismo cluster, o tenían una relación CL y comparten clúster.

infeasibility :

Devuelve el número de restricciones infringidas en una solución. Recorre el triángulo superior de la matriz (sin contar la diagonal principal) ya que es simétrica, y aplica violaRestriccion a cada posición i,j (que indicaría la restricción asociada a los datos i y j).

funcionObjetivo :

Calcula la bondad de una solución, dada por la expresión:

funcionObjetivo = C + (infeasibility * LAMBDA)

restaVector :

Resta dos vectores:

$$v-u=(v_1-u_1, v_2-u_2, \ldots, v_n-u_n)$$

moduloVector :

Calcula el módulo de un vector:

$$|v|=\sqrt{(v_1{}^2+v_2{}^2+\ldots+v_n{}^2)}$$

borrarPorValor :

Borra un elemento de un vector, buscando por valor hasta que alguno coincida con el dado. Para ello, recorre el vector desde el inicio y desde el final hasta llegar a la mitad de este, y en caso de encontrar el elemento lo borra y termina.

Con los algoritmos genéticos y meméticos, añadimos las siguientes funciones comunes:

- **create_genome (Generar solución aleatoria) :**

Generamos un vector aleatorio con (longitud de cromosoma) elementos, si es válido como solución, lo aceptamos. Si no, generamos otro y repetimos.
1.- Hacer:
2.- repetir = *false*
3.- Generamos un vector de números aleatorios entre 0 y numClusters-1
4.- Contamos las apariciones de cada clúster en el vector.
5.- Si alguno es 0, repetir = *true*
6.- Mientras repetir = *true*

- **Selección :**

Elegimos dos cromosomas diferentes, y elegimos el ganador del torneo binario entre ellos.
Si es en el esquema generacional (o memético, que usa el mismo esquema), realizamos tamaño de población torneos, y si es AGE, solo 2.

- **Cruce uniforme:**

Generamos *n/2* posiciones aleatorias de un padre, y asignamos al hijo los valores encontrados en dichas posiciones. Luego recorremos el vector hijo, y donde no se haya asignado valor, asignamos el del otro padre.

- **Cruce por segmento fijo :**

Generamos inicio (*ini*) y longitud (*long*) del segmento como n.º aleatorios entre 0 y n-1. Luego, copiamos el segmento de longitud *long* empezado en *ini* de un padre en el hijo (Mientras no se hayan copiado *long* elementos, copiamos el elemento *i* del padre en el hijo (comenzando *i* en *ini*, y *i=i+1 mod* vector.size)

- **Mutación :**

Calculamos las mutaciones esperadas (*numMut*) para ahorrar generar muchos números aleatorios. Realizamos *numMut* iteraciones, y en cada una generamos un n.º aleatorio, a partir del cual calculamos un cromosoma y un gen para mutar (con el módulo de número de cromosomas y modulo de longitud de cromosoma, respectivamente) y mutamos dicho cromosoma en dicho gen (si no se puede, generamos otros números hasta que se pueda, dentro de la misma iteración). A la hora de mutar, generamos otro número de cluster diferente al actual y lo cambiamos.

Esquema de los algoritmos implementados:

Aquí se presentan los esquemas de 1 generación de cada algoritmo. Se ejecuta este proceso mientras se hayan producido menos de 100000 evaluaciones de la función objetivo.
En todos comenzamos generando una población de soluciones aleatorias, mediante el método anteriormente mostrado.

_____ AGG: _____

1.- Ordenamos de menor a mayor f la población.
2.- Guardamos el mejor elemento.
3.- SELECCIÓN → Generamos 50 padres por el método de torneo binario con dos padres diferentes seleccionados aleatoriamente.
4.- CRUCE → Generamos (0.7 * tamaño de población) hijos, emparejando los padres de forma fija (padre n.º 0 con padre n.º 1, padre n.º 2 con padre n.º 3, y así sucesivamente) y los guardamos en descendencia.
5.- Generamos nueva_generation como descendencia + padres no usados en el cruce.
6.- MUTACIÓN → Mutamos nueva_generation (n.º de mutaciones esperadas) veces
7.- ELITISMO → Insertamos el mejor elemento de la población si no está en nueva_generation, buscando y eliminando el peor elemento actual para poder incluirlo.
8.- REEMPLAZAMIENTO → población = nueva_generation

_____ AGE: _____

1.- SELECCIÓN → Elegimos 2 padres mediante torneo binario.
2.- CRUCE → Generamos descendencia (2 hijos).
3.- MUTACIÓN → Mutamos la descendencia (n.º de mutaciones esperadas) veces
4.- REEMPLAZAMIENTO → Sustituimos los hijos generados por los peores individuos de la población. Para ello, miramos el peor individuo, y comparamos con los hijos, sustituyendo por el primero mejor (de haberlo). Después, volvemos a coger el peor elemento de la población en este momento, y repetimos (con los dos hijos si no se ha realizado cambio o solo con uno de haberlo habido).

En cuanto a los AMs, usan la estructura del AGG (usando mutación por segmento fijo), con el añadido de que cada 10 iteraciones, aplicamos el método de búsqueda local suave (BLS) a una parte de la población; en nuestro caso, a toda la población, al 10% de esta o al 10% de la mejor población (dependiendo de la versión del algoritmo que tratemos).

_____ BLS: _____

(Acepta cromosoma S, numeroClusters y numeroFallos)
1.- Trabajamos con copia de S
2.- Generamos un vector con indices de S aleatorios.
2.- Mientras ((mejora o fallos < numeroFallos) y i<longitudCromosoma)
3.- mejora = false
4.- Para cada indice rsi
5.- Si se puede cambiar de cluster
Para cada cluster
6.- Si al cambiar el dato rsi a ese cluster mejoro la solución, actualizo mejor solución, y mejora=true.
7.- Si no se ha mejorado, incrementamos fallos
8.- ++i

9.- S = mejor solución

Material considerado y manual de uso:

En cuanto al material usado en la implementación, se han usado las funciones de generación de números aleatorios de PRADO (random.h y random.cpp) y las siguientes librerías de C++ : iostream, fstream, string, vector, set, cmath, numeric, algorithm, limits.h, chrono, functional .
Además, se usó la página https://www.geeksforgeeks.org/genetic-algorithms/ como referencia para la implementación de los algoritmos.

En cuanto al uso del código, este viene dividido en 2 carpetas: *BIN* y *FUENTES*. Las acompaña un *makefile*, con lo cual, para compilar el código, solo necesita usar el comando *make* posicionándose en la terminal sobre la carpeta donde se encuentre. Dentro de *FUENTES*, están las carpetas *include* (archivos .h), *obj* (archivos .o) y *src* (archivos .cpp), y en *BIN* se encontrará el ejecutable y una carpeta *data* con los datos para ejecutar el programa.

Una vez compilado, el programa, que se encontrará en la carpeta BIN, se ejecuta con los siguientes argumentos:

./bin/practica2 archivoDeDatos archivoDeRestricciones NumeroDeClusters Semilla

Experimentos y análisis de resultados:

He realizado 5 ejecuciones para cada cada porcentaje de restricciones (10% y 20%) y conjunto de datos (iris, ecoli, rand y newthyroid).
Además, he actualizado los valores para búsqueda local y COPKM con los nuevos conjuntos de datos.

Para conjuntos de datos con 10% de restricciones:

	IRIS				ECOLI				RAND				NEWTHYROID			
	Tasa_C	Tasa_inf	Agr,	T	Tasa_C	Tasa_inf	Agr,	T	Tasa_C	Tasa_inf	Agr,	T	Tasa_C	Tasa_inf	Agr,	T
COPKM	0,669305	0	0,669305	0,0011068982	32,46344	22,6	33,08032	0,013064754	0,715599	0	0,715599	0,0018561216	14,2871	0	14,2871	0,001665902
BL	0,669305	0	0,669305	0,0412102	21,59652	105,8	24,43508	2,503564	0,715599	0	0,757315	0,03876064	11,47994	79	14,36906	0,171333

	IRIS				ECOLI				RAND				NEWTHYROID			
	Tasa_C	Tasa_inf	Agr,	T	Tasa_C	Tasa_inf	Agr,	T	Tasa_C	Tasa_inf	Agr,	T	Tasa_C	Tasa_inf	Agr,	T
AGG-UN	0,669305	0	0,669305	4,713256	23,67786	143,6	27,53056	22,65584	0,715599	0	0,715599	4,778364	13,19272	24	14,0704	9,265934
AGG_SF	0,669305	0	0,669305	4,701788	26,17966	210	31,81386	22,90906	0,715599	0	0,715599	4,724774	12,6411	46,6	14,3453	9,219118
AGE_UN	0,669305	0	0,669305	5,210806	21,86638	75,4	23,88936	23,99094	0,715599	0	0,715599	5,219308	13,24394	24,2	14,12894	9,94116
AGE_SF	0,669305	0	0,669305	5,205424	21,71882	94,2	24,24614	23,71036	0,715599	0	0,715599	5,245924	12,0229	69,4	14,56092	9,812176
AM-1.0,10	0,669305	0	0,669305	4,596106	21,99724	79,2	24,12216	22,18634	0,715599	0	0,715599	4,617126	12,60006	43,4	14,18724	9,006006
AM-0.1,10	0,669305	0	0,669305	4,593218	21,9211	81,6	24,1104	4572,80556	0,715599	0	0,715599	4,612686	13,21228	24,2	14,09728	9,11248
AM-0.1,10mej	0,669305	0	0,669305	4,598688	21,42468	111	24,40278	22,62968	0,715599	0	0,715599	4,579682	12,65428	43,6	14,24878	9,148902

Para los conjuntos de datos con un 20% de restricciones:

	IRIS				ECOLI				RAND				NEWTHYROID			
	Tasa_C	Tasa_inf	Agr,	T	Tasa_C	Tasa_inf	Agr,	T	Tasa_C	Tasa_inf	Agr,	T	Tasa_C	Tasa_inf	Agr,	T
COPKM	0,669305	0	0,669305	0,001630774	30,31352	1,2	30,32962	0,01400534	0,715599	0	0,715599	0,0015716732	14,2871	0	14,2871	0,00196247
BL	0,669305	0	0,669305	0,04904058	21,77668	179,8	24,18868	3,555722	0,715599	0	0,715599	0,0496854	12,21254	151,6	14,98406	0,1938444

	IRIS				ECOLI				RAND				NEWTHYROID			
	Tasa_C	Tasa_inf	Agr,	T	Tasa_C	Tasa_inf	Agr,	T	Tasa_C	Tasa_inf	Agr,	T	Tasa_C	Tasa_inf	Agr,	T
AGG-UN	0,669305	0	0,669305	6,120092	22,96608	308	27,09782	29,51196	0,715599	0	0,715599	5,914376	13,56186	45,4	14,39184	12,1408
AGG_SF	0,669305	0	0,669305	5,942132	24,6265	312,8	28,82262	29,39096	0,715599	0	0,715599	5,895242	12,16352	137,8	14,68276	12,13474
AGE_UN	0,669305	0	0,669305	6,490444	21,86736	152,4	23,91178	30,79072	0,715599	0	0,715599	6,484298	12,91094	101,2	14,76104	12,9228
AGE_SF	0,669305	0	0,669305	6,48152	21,95504	163,4	24,147	30,49766	0,715599	0	0,715599	6,454126	13,59384	50,8	14,52254	12,7004
AM-1,0,10	0,669305	0	0,669305	5,840796	21,96348	180,8	24,38888	29,137	0,715599	0	0,715599	5,762206	14,2871	0	14,2871	12,33014
AM-0,1,10	0,669305	0	0,669305	5,839542	21,69284	191,2	24,25778	29,15254	0,715599	0	0,715599	5,777938	12,1657	142,2	14,76534	12,27874
AM-0,1,10mej	0,669305	0	0,669305	5,838466	21,84446	175,2	24,19472	29,26158	0,715599	0	0,715599	5,750422	13,58752	46,2	14,43214	12,3162

Entre algoritmos genéticos y meméticos, vemos que no hay una gran diferencia, ni en calidad de resultados ni en tiempo de ejecución. Tal vez, destacar que AGG nos da resultados peores que AGE y AMs, especialmente en infeasibility.

Los datos resultantes no nos ayudan a concluir nada sobre la calidad de cada algoritmo sobre otro. Esto se debe al fuerte uso de probabilidades de cada algoritmo.

En comparación con BL y COPKM, lo primero a destacar es el tiempo, ya que los algoritmos genéticos y meméticos tardan mucho más que estos otros dos. Esto se debe, obviamente, a que tanto genéticos como meméticos trabajan sobre poblaciones de soluciones, que aporta una gran carga de trabajo, mientras que BL y COPKM solo trabajan sobre una.
Si nos fijamos en la calidad de las soluciones obtenidas, observamos que tampoco nos supone una gran diferencia un algoritmo y otro. Es más, COPKM y BL nos proporcionan soluciones de calidad similar o mejor a los genéticos y meméticos en un tiempo muchísimo menor.
