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

3º CSI
Problema del Agrupamiento con Restricciones (PAR)

ES BMB ILS ILS-ES

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

Indice:

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 mediante una clase con los siguientes elementos para cada objeto:

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.
Centroides.- Centroides usados en el cálculo de la función objetivo – Necesario para factorizar el cálculo de f en vecinos.
f.- Valor de la función objetivo.

Aunque a la hora de mostrar la solución uso S1, ya que el resto de elementos no aporta información útil en este aspecto.

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 referentes a la función objetivo: _____

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).

1 Total = 0
2 Para cada elemento del cluster
  2.1 distancia = Modulo(centroide – elemento)
  2.2 Total += distancia
3 Return Total/NumElementosCluster

C :

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

1 Total = 0
2 Desde i=0…NumClusters
  2.1 Total += Ci(Cluster n.º i)
3 Devolver Total/NumClusters

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.

1 resultado = 0
2 Desde i=0… NumFilas
  2.1 Desde j=i+1… NumColumnas
    2.1.1 resultado += violaRestriccion(i, j, matrizRestricciones, solucion)
3 Devolver resultado

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.

GenerarSoluciónAleatoria (BMB - ILS) :

Asignamos números aleatorios entre 0 y numClusters-1 a cada posición de S1.
Repetimos esto hasta que se consiga una solución válida (ningún clúster vacío).
Tras esto, generamos S2 y evaluamos la solución.

Enfriamiento Simulado (ES):

ES se inspira en el comportamiento de los cuerpos físicos al enfriarse, ya que pueden ir a estados “peores” con cierta probabilidad.
Partimos de una solución dada o aleatoria (generada como ha sido explicado anteriormente). Calculamos los siguientes parámetros:

- MaxVecinos: Número máximo de vecinos a evaluar.
- MaxExitos: Considerando como éxito un cambio de solución.
- MaxEvaluaciones: Número máximo de evaluaciones de la función objetivo
- Temperatura inicial
- Temperatura final
- NumEnfriamientos: Marcará cúantas iteraciones realizará el algoritmo.
- Beta(β): Usada en el esquema de enfriamiento.

En cuanto a la temperatura, se iniciará al valor dado por:

$$TemperaturaInicial = \frac{0.3}{-\log(0.3)} \cdot f(solucionInicial)$$

Y en cada iteración se enfriará según la siguiente fórmula:

$$Temperatura = \frac{Temperatura}{1 + \beta \cdot Temperatura}$$

Para generar un vecino, generaremos dos n.º aleatorios que representen un dato a cambiar de clúster y el clúster al que cambia hasta que sea un cambio válido. Después, creamos la solución asociada al vecino, y calculamos el valor de f para ella. Para agilizarlo, factorizamos el cálculo de los centroides de la solución; es decir, los centroides del vecino serán los de la solución actual quitando un elemento de uno de ellos y añadiéndolo a otro. Como fórmula, se expresa así:

$$CCA = \frac{CCA \cdot CAsize - datoACambiar}{CAsize - 1}$$

Siendo CCA el centroide del cluster antiguo y CA el cluster antiguo.

$$CCN = \frac{CCN \cdot CNsize + datoACambiar}{CNsize + 1}$$

Siendo CCN el centroide del cluster nuevo y CN el cluster nuevo.

Comenzamos el algoritmo:

1 MaxVecinos = n.º de datos * 10
2 MaxExitos = MaxVecinos * 0.1
3 MaxEvaluaciones = 100000
4 NumEnfriamientos = MaxEvaluaciones / MaxVecinos
5 Temperatura Final = 1/1000
6 Hacer
  6.1 solucionActual = solucion aleatoria
  6.2 Temperatura Inicial = 0.3 * f(solucionActual) / -log(0.3)
  Mientras Temperatura Inicial < Temperatura final

7 Desde j=0...NumEnfriamientos y evaluaciones < MaxEvaluaciones *//Bucle exterior*
  7.1 numExitos = 0
  7.2 Desde i=0...maxVecinos *//Bucle interior*
    7.2.1 Generamos un vecino aleatorio.
    7.2.2 Si el vecino es mejor o con un 10% de probabilidades, se acepta.
    7.2.3 Si se acepta:
      7.2.3.1 numExitos +=1.
      7.2.3.2 Se sustituye la solucion actual por el vecino.
      7.2.3.3 Comparamos con la mejor solución obtenida y sustituimos si es mejor que la considerada mejor hasta ahora.
    7.2.4 Si hemos llegado al máximo número de éxitos, salimos del bucle interior.
  7.3 Si no hemos conseguido ningún éxito, salimos del bucle exterior.
  7.4 Enfriamos la temperatura mediante el esquema de enfriamiento.
8 Devolvemos la mejor solución encontrada.

Búsqueda Multiarranque Básica (BMB) :

BMB basa su funcionamiento en realizar varias búsquedas locales (empezando en puntos aleatorios) con menos iteraciones que una BL normal para cada una y quedándonos con la mejor solución encontrada, de esta forma podemos evitar caer en óptimos locales.
Para implementarla, necesitamos hablar sobre cómo generamos el entorno de una solución, como exploramos dicho entorno y cómo generamos un vecino de una solución.

Para generar la lista de candidatos, generamos pares que almacenen los el indice del valor a cambiar y el clúster al que se cambia. No generamos pares con un valor y el clúster en el que ya está o si el clúster en el que está actualmente solo contiene a ese dato.

Para explorar el entorno, barajamos la lista de candidatos y accedemos a la última posición de esta, eliminando ese par de la lista, y generamos el vecino asociado a ese par.

Para generar el vecino, seguimos el mismo procedimiento que en ES, pero en este caso sin tener que generar los números aleatorios que definen el vecino ya que vienen incluido en el par.

En cuanto al algoritmo, este sería su pseudocódigo:

1. Generamos una solución inicial con BL
2. Mejor solución = solución actual
3. Desde i=0...9
    - 3.1 Generamos una solución con BL
    - 3.2 Si es mejor que la mejor solución, la sustituimos.
4. Devolvemos la mejor solución encontrada.

Búsqueda Local Iterada (ILS) :

Para ILS, realizamos una mutación en la mejor solución encontrada antes de aplicarle BL y comparar con la mejor solución encontrada hasta el momento.

Pseudocódigo del esquema de búsqueda :

1 solucionActual = solucionMejor = solucion Aleatoria
2 solucionActual = BL a partir de solucionActual
3 Actualizamos solucionMejor si solucionActual es mejor
4 Desde i=0...9
  4.1 mutada = solucionMejor mutada
  4.2 solucionActual = BL a partir de mutada
  4.3 Actualizamos solucionMejor si mutada es mejor
5 Devolver solucionMejor

Para la mutación, seleccionamos un segmento de la solución, cuyos elementos serán cambiados en la mutada, mientras que el resto se mantendrá igual.
Pseudocódigo del esquema de mutación :

1 inicio = dato aleatorio
2 mas = aleatorio(0,1)
3 Hacer
  3.1 Si mas == 0
    3.1.1 fin = inicio – 0.1*NumDatos
    3.1.2 Si fin < 0 mas = 1
  3.2 Si mas == 1
    3.2.1 fin = inicio + 0.1*NumDatos
    3.2.2 Si fin >= NumDatos mas = 0
  Mientras fin < 0 o fin >= NumDatos
4 Desde i = fin+1 % NumDatos...inicio *//Copiamos lo que haya fuera del segmento*
  4.1 mutado[i] = solucion[i]
  4.2 i = (i+1) % NumDatos
5 Hacer
  5.1 Desde i=inicio... fin % NumDatos *//Para el segmento, clusters aleatorios*
    5.1.1 mutado[i] = cluster aleatorio
    5.1.2 i = (i+1) % NumDatos
  Mientras mutado no sea una solución válida
6 Devolver solución asociada a mutado

El paso 3 del algoritmo se realiza ya que podemos escoger, desde una posición, un 10% en un sentido u otro de la solución. Sin embargo, podría darse el caso de que escoger el 10% en una determinada dirección no sea posible. Con el paso 3 nos encargamos de solucionar dicha situación, ya que comprueba que se puede hacer en la dirección escogida, y si no, lo hace en la otra.

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 .

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/practica3  archivoDeDatos  archivoDeRestricciones  NumeroDeClusters  Semilla   AlgoritmoAUsar
```

Con AlgoritmoAUsar a 0, se usará ES; con 1, BMB; con 2, ILS y con 3, ILS-ES.

Experimentos y análisis de resultados:

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
ES	0,669305	0	0,669305	0,2085976	21,3707	94,6	23,91216	5,911276	0,715599	0	0,715599	0,1843594	12,05262	60,6	14,27164	0,6136322
BMB	0,669305	0	0,669305	0,8263618	22,06322	173,8	26,72806	23,66016	0,715599	0	0,715599	0,8149312	13,835	6	14,0544	3,033164
ILS	0,669305	0	0,669305	0,6133684	21,7848	78,8	23,8991	20,35656	0,715599	0	0,715599	0,5724548	13,2427	24	14,12046	2,393042
ILS-ES	0,669305	0	0,669305	0,7887618	21,58608	78,6	23,69502	18,65768	0,715599	0	0,715599	0,6931782	13,835	6	14,0544	2,543378

Vemos un mejor rendimiento por parte de BMB e ILS-ES en Newthyroid, siendo este último algoritmo el mejor también en Ecoli.
También vemos que BMB rinde peor que BL para Ecoli, pero mejor en Newthyroid. Esto muy probablemente se deba a que BMB usa menos iteraciones por búsqueda que BL (10000 contra 100000), por lo que puede que, aunque el punto de partida sea muy bueno en algunas de las iteraciones de BMB, no lleguemos a explotarlo del todo. Sin embargo, para Newthyroid, tenemos que es un problema más sencillo que Ecoli, por lo que podemos realizar menos iteraciones sin que ello afecte al resultado.

Con 20% de restricciones:

	IRIS				ECOLI				RAND				NEWTHYROID			
	Tasa_C	Tasa_inf	Agr,	T	Tasa_C	Tasa_inf	Agr,	T	Tasa_C	Tasa_inf	Agr,	T	Tasa_C	Tasa_inf	Agr,	T
COPKM	0,669305	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
ES	0,669305	0	0,669305	0,2204846	21,7876	172	24,09796	8,394868	0,715599	0	0,715599	0,2239868	14,2871	0	14,2871	0,799012
BMB	0,669305	0	0,669305	0,9533632	22,08474	262,6	25,60944	31,15988	0,715599	0	0,715599	0,9234188	14,2871	0	14,2871	3,530898
ILS	0,669305	0	0,669305	0,7332274	22,03726	145,2	23,98518	30,83932	0,715599	0	0,715599	0,6833928	14,2871	0	14,2871	2,867504
ILS-ES	0,669305	0	0,669305	0,9142246	21,92734	174,2	24,2643	28,3773	0,715599	0	0,715599	0,838788	14,2871	0	14,2871	3,321556

Aquí observamos que en Newthyroid todos menos BL consiguen el mismo rendimiento. En este caso, como BL depende del punto de partida, puede suceder que en dos o tres ejecuciones el resultado obtenido sea peor (empeorando la media), mientras que en BMB, al realizar muchos más intentos (aunque más cortos, pero a juzgar por el rendimiento con 10% de restricciones, no parece un problema), es mucho más probable que consiga un punto de partida mejor.
En cuanto a Ecoli, tenemos que ILS y ES consiguen los mejores resultados, mientras que ILS-ES es peor que ambos. El peor rendimiento en ILS-ES puede justificarse por los parámetros que tenemos que ajustar y, además, en ILS-ES mutamos (con lo que visitaremos estados peores) y usamos ES, que también puede aceptar soluciones peores, lo cual ha favorecido mucho la exploración del entorno pero en este caso puede que no

haya sido lo más conveniente, ya que por cuestión de velocidad reducimos las iteraciones de ES a 10000. ILS y ES exploran menos pero explotan más la solución, ya que ILS funciona con BL, que no visita estados peores como ES, y ES funciona con 100000 iteraciones.
