METAHEURÍSTICA
GRADO EN INGENIERÍA INFORMÁTICA
UNIVERSIDAD DE GRANADA

Práctica 1.a Técnicas de Búsqueda Local y Algoritmos Greedy para el Problema de la Mínima Dispersión Diferencial

---

José Luis Molina Aguilar

9 de abril de 2022

Curso 2021-2022
DNI : 77556436E
Correo : joselu201@correo.ugr.es
Grupo : A3, MARTES 17:30 - 19:30

Índice

Índice de figuras

1. Descripción Problema de Mínima Dispersión Diferencial

El problema de Mínima Dispersión Diferencial es un problema de optimización combinatoria que entra en la clase de problemas **NP-Completo**
Este es un problema en el que las heurísticas obtienen buenas soluciones en menos tiempo.

1.1. Descripción

Dado un conjunto de n elementos todos ellos conectados entre sí, representado por una matriz de distancias de tamaño nxn obtener un subconjunto m tal que la diferencia entre la máxima distancia acumulada y la mínima distancia acumulada de los elementos de m se minimiza.
El conjunto $m < n$ y por lo tanto lo que estamos buscando es $m \subset n \mid MinimizeDD(S_m)$
donde $DD(S_m)$ es la Dispersión Diferencial del conjunto de Soluciones de tamaño m

1.2. Consideraciones

En mi representación de este problema la matriz de distancias descrita anteriormente será una matriz de flotantes llamada **datos**

Además implementaré un vector **distan** la cual almacena la distancia desde un punto al resto, será útil para factorizar en BL.

2. Greedy

El algoritmo Greedy se basa en la heurística de ir añadiendo a la solución el elemento más óptimo de los disponibles, el cual es el que minimice la dispersión.
Elegiremos el primer elemento de m de forma aleatoria para ganar variedad en los resultados.
Después, el resto de elementos a elegir hasta completar la solución será, sobre todos los posibles candidatos, calculamos la dispersión cuando añadimos ese elemento a la solución m y el elemento que la minimice será escogido y añadido a la solución.
Esta aproximación cae fácilmente en óptimos locales ya que es muy dependiente de los del punto de inicio y en cada paso aunque escojamos el elemento que minimiza la Dispersión no significa que, como conjunto solución, sea el correcto.
La ventaja principal del greedy es que obtiene una solución relativamente buena en mucho menos tiempo que el algoritmo perfecto que resuelve este problema.
Para ayudarnos en el desarrollo del Greedy usaremos 3 funciones:

- **distPuntoRestoElemenetos**, que calcula la distancia acumulada de un punto al resto del vector.

- **diff**, el cual dado un vector de soluciones calcule las distancias acumuladas (distPunto-RestoElemenetos) y devuelva la dispersión para ese conjunto.

- **fit_adding**, esta función simplemente calcula la dispersión (mediante diff) si añadimos un nuevo elemento al vector de soluciones.

La representación de la solución la realizaremos con un vector de enteros que almacena los índices de los elementos escogidos.
Por lo que el algoritmo Greedy quedaría:

Algorithm 1 Greedy

```
1:  function GREEDY
2:      Solucion ← ∅
3:      Candidatos ← V                                  ▷ V son todos los indices, n
4:      v_0 ← SelectRandomFrom(Candidatos)
5:      Solucion ← Solucion ∪ {v_0}
6:      Candidatos ← Candidatos \ {v_0}
7:      while |Solucion| < m do
8:          for ele in Candidatos do
9:              min ← FLOATMAX
10:             new_fitness ← fit_adding(Solucion, ele)
11:             if new_fitness < min then
12:                 ele_pos ← ele                       ▷ Guardo el mejor elemento
13:                 min ← new_fitness                   ▷ Actualizo el minimo actual
14:             end if
15:         end for
16:         Solucion ← Solucion ∪ {ele_pos}             ▷ Añado a la solucion
17:         Candidatos ← Candidatos \ {ele_pos}
18:     end while
19:     return Solucion
20: end function
```

Pseudocodigo de **distPuntoRestoElemenetos**

Algorithm 2 distPuntoRestoElemenetos

```
1: function DISTPUNTORESTOELEMENETOS(FILA,VECTOR)
2:     dist ← 0
3:     for i ← 0 to length(vector) do
4:         dist ← dist + datos[fila][vector[i]]
5:     end for
6:     return dist
7: end function
```

Pseudocodigo de **diff**

Algorithm 3 diff

```
1: function DIFF(POSIBLES)
2:     distancias ← ∅
3:     for i ← 0 to length(posibles) do
4:         distancias ← distancias ∪ distPuntoRestoElemenetos(posibles[i], posib))
5:     end for
6:     sort(distancias)
7:     return distancias[length(posibles)] − distancias[0]
8: end function
```

Pseudocodigo de **fit_adding**

Algorithm 4 fit_adding

```
1: function fit_adding(posibles, new_i)
2:     posibles ← posibles ∪ new_i
3:     new_diff ← diff(posibles)
4:     posibles ← posibles \ new_i
5:     return new_diff
6: end function
```

3. Busqueda por Trayectorias Simples (BL)

La búsqueda local se basa en generar una solución aleatoria, la cual como solución válida tiene que satisfacer las restricciones de

- No puede tener elementos repetidos
- Tiene que tener exactamente m elementos
- El orden no es relevante

Para obtener una solución BL aplica un Operador de intercambio, este es:
Dada una solución, intercambiar un elemento de esa solución por otro elemento del conjunto Candidatos, (el cual esto formado por todos los índices menos los que están en solución, S-Solucion = Candidatos), el cual minimice el valor del fitness.
Esto provoca que el espacio de posibilidades de cambio sea de $m \cdot (m-n)$, por lo que a la hora de aplicar esto, una vez que encontremos un elemento que minimice la dispersión se añadirá a la solución, (con añadir me refiero a intercambiar los valores) y seguidamente buscaremos otra vez para el siguiente elemento de la solución, puede llegar un punto en el que una vez recorrido todo el conjunto de soluciones e intentar intercambiarlo por algún elemento del conjunto de Candidatos ninguno minimice el valor actual, en ese caso terminaremos y devolveremos la solución actual.
También utilizaremos un número limitado de iteraciones.

3.1. Factorización del Movimiento de Intercambio

Ejemplo.
Dado el conjunto solución (0,4,6), cambio el elemento 0 por 1, quedaria (1,4,6) por lo que el vector distan quedaría:
D0 = D04 + D06 //Esta ya no lo necesito
D4 = D40 + D46 //Si cambio el 0 por un 1, D4 = D4 -D04 + D14
D6 = D60 + D64

D1 = D14 + D16 //Este tengo que recalcularlo entero
D4 = D4 -D04 + D14
D6 = D6 -D06 + D16

De esta forma no tengo que volver a calcular de nuevo todas las distancia de un punto al resto, sino que simplemente tendré que actualizar el valor de la forma anteriormente descrita.
Lo cual me hace pasar de una complejidad $\mathcal{O}(n^2)$ a $\mathcal{O}(n)$

Algorithm 5 BL

```
1: function BL
2:     Solucion ← SelectRandomSolution
3:     Candidatos ← V                                   ▷ V son todos los indices de n
4:     Shuffle(Candidatos)
5:     index ← 0                                                  ▷ Indice de solucion
6:     MaxIters ← 1000000
7:     cambia ← true
8:     iter ← 0
9:     while iter < MaxIters and cambia do
10:        for i ← 0 to length(candidatos) do
11:            actual_disp ← diff(Solucion)
12:            intercambio ← (index, cand[i])   ▷ Cambio el elemento index por un candidato
13:            new_disp ← distFactorizada(Solucion, intercambio)
14:            if new_disp < actual_disp then
15:                Solucion ← Solucion ∪ {cand[i]}                  ▷ Añado a la solucion
16:                index ← index + 1                ▷ Avanzo a otro elemento de solucion
17:                cambio ← true                          ▷ Anoto que ha habido cambio
18:                i ← 0                          ▷ Vuelvo a mirar con el siguiente elemento
19:            end if
20:            if !cambio and solucion[index] != solucion[solucion.size()] then
21:                        ▷ Si no ha habido cambio, pero no he he comprobado todo Solucion
22:                index ← index + 1                          ▷ Avanzo al siguiente elemento
23:                i ← 0                          ▷ Vuelvo a mirar si algun candidato mejora
24:            else if solucion[index] == solucion[solucion.size()] then
25:                                                          ▷ Si he comprobado todas
26:                cambio ← false
27:            end if
28:            iter ← iter + 1
29:        end for
30:        index ← index + 1
31:    end while
32:    return Solucion
33: end function
```

Pseudocodigo de **distFactorizada**

```
1: function distFactorizada(solucion, cambio)
2:     distan[cambio.first] = distPuntoRestoElemenetos(cambio.second, solucion); ▷ Recalculo el punto que he cambiado
3:     for i ← 0 to length(solucion) do
4:         distan[i] ← distan[i] − datos[solucion[cambio.first]][solucion[i]] + datos[cambio.second][solucion[i]];
5:     end for
6:     Sort(distan)
7:     return (distan[distan.size() − 1] − distan[0]);
8: end function
```

4. Analisis

- La información de uso se encuentra en el README.md.
- Las semillas con las que se han obtenido los resultados entan el main.cpp y son {0,1,2,3,4}.

Finalmente para representar los resultados obtenidos por estos dos algoritmos respecto del perfecto tenemos la siguinte grafica

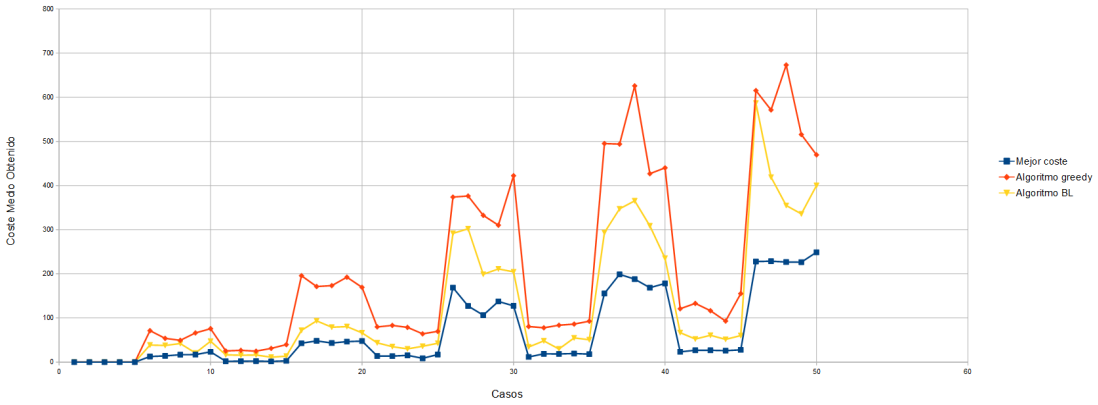

Figura 4.1: Resultados distintos Algoritmos

En la Gráfica 4.1 podemos ver como se comporta cada algoritmo, como vemos la BL es mucho mejor que el Greedy en este caso ya que esta mas cerca del resultado que nos ofrece el algoritmo perfecto.

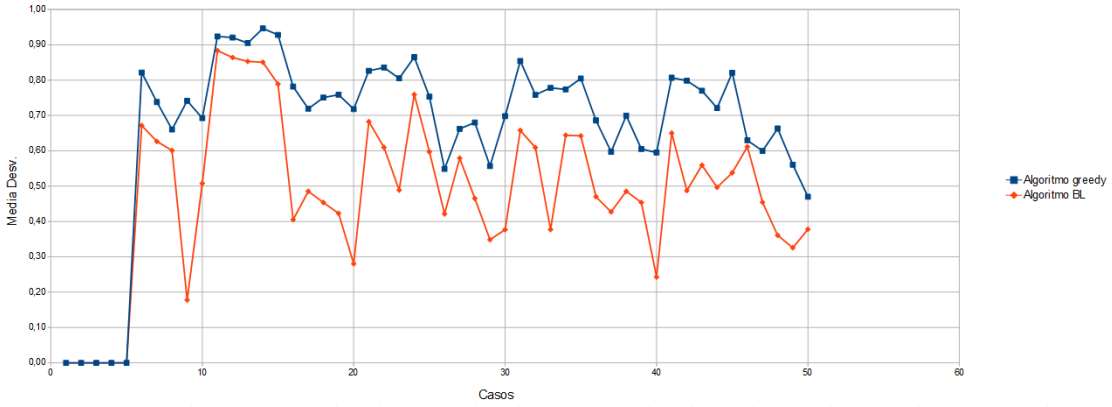

Figura 4.2: Desviación distintos Algoritmos

Pero esta mejora en la precisión conlleva como podemos ver el la Gráfica 4.3 un aumento sustancial en el tiempo de computo aunque si pudiesemos comparar el tiempo que le tomo al algoritmo perfecto, la BL en comparación sería muy rapida.
Además podemos ver que que el tiempo que tarda en dar una solucion es directamente proporcional al numero de elementos de m ya que, como podemos ver podemos diferenciar intervalos que tienen en comun una cosa n, por ejemplo [31,40], $n = 125$, dentro de ese intervalo tenemos que [31-35] $m = 12$ y [35 - 40] $m = 37$, en la Gráfica podemos ver que tarda mucho menos en [31-35] ya que el valor de m es menor que en el intervalo [35 - 40]

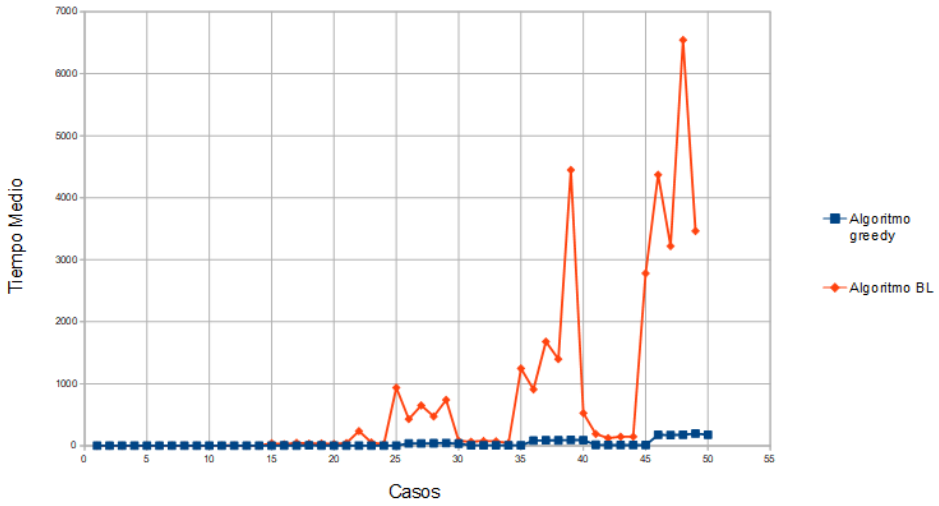

Figura 4.3: Tiempos(ms) para diferentes Algoritmos