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

Luis Miguel Guirado Bautista

10 de Abril de 2022   Curso 2021/2022

Correo: luismgb@correo.ugr.es

DNI: 75942712R

Subgrupo: Martes, 3

Problema: A (MDD)

Índice

1 Motivación

En esta práctica abordaremos el problema de la Mínima Dispersión Diferencial (MDD), que consiste en seleccionar un conjunto de m elementos M t.q $M \subset N$ y $m < n$, cuya dispersión entre sus elementos sea mínima, de modo que puede tratarse como un problema de optimización.

La dispersión de un elemento v de la selección se interpreta como la suma de las distancias de v al resto de puntos de M.

$$\Delta(v) = \sum_{v \in M} d_{vu}$$

La dispersión de una solución M se define como la diferencia entre los valores extremos de las dispersiones de los puntos de M.

$$\Delta(M) = max_{v \in M}(u) - min_{u \in M}(v)$$

Siendo $\mathbb{M}$ el conjunto de posibles soluciones, podemos denotar la solución óptima M^* como:

$$M^* = min_{M \in \mathbb{M}}(\Delta(M))$$

Para solucionar de forma práctica este tipo de problemas usaremos 50 ficheros de datos generados por Glover, Kuo y Dhir en 1998 (GKD). Cada uno de estos ficheros son casos de problemas, con unos valores de n y m predefinidos, y la matriz diagonal superior de distancias euclídeas de cada uno de los puntos de N. Podemos recoger los datos de estos ficheros y representarlos de la siguiente manera.

$$I_{nm} = \begin{pmatrix} 0 & d_{01} & d_{02} & \dots & d_{0n} \\ 0 & 0 & d_{12} & \dots & d_{1_n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & d_{nn} \end{pmatrix}$$

Nota: Por simplicidad, el programa convertirá la matriz diagonal superior a una matriz diagonal superior e inferior.

También haremos uso de dos algoritmos implementados por el estudiante:

- Greedy
- Búsqueda Local

Finalmente haremos un análisis en base a los algoritmos empleados, el tamaño de los casos y el coste y tiempo medios.

2 Algoritmos

2.1 Greedy

Consiste en seleccionar el siguiente elemento de la solución que minimice la dispersión con respecto a la solución actual. El primer elemento es aleatorio ya que $M = \emptyset$, y a partir de ahí se van escogiendo los elementos siguiendo el siguiente criterio.

Para cada $u \notin M$:

$$\partial(u) = \sum_{v \in M} d_{uv}$$

Y para cada $v \in M$, siendo $anterior(v)$ la distancia entre los puntos de M:

$$\partial(v) = anterior(v) + d_{uv}$$

Entonces, siendo:

$$\partial_{max}(u) = max(\partial(u), max_{v \in M}\partial(v))$$

$$\partial_{min}(u) = min(\partial(u), min_{v \in M}\partial(v))$$

La dispersión del siguiente punto u será:

$$g(u) = \partial_{max}(u) - \partial_{min}(u)$$

```
function Greedy-MDD(distancias, N, m)
    actual ← Aleatorio(N)
    s ← [actual]                                  ▷ Lista de elementos pertenecientes a N
    disp ← 0
    while s.size < m do
        (actual, disp) ← EscogerElemento(distancias, s)     ▷ Heurística empleada
        s ← s ∪ actual                                      ▷ Se añade actual a s
    end while
    return (s, disp)
end function
```

function $\textsc{EscogerElemento}(distancias,\, s)$
  $mejor \leftarrow -$
  $min_g \leftarrow \infty$
  for $u \notin s$ **do** ▷ Para cada elemento no seleccionado
    $\partial(u) \leftarrow \sum_{v \in s} distancias[u, v]$
    for $v \in s$ **do** ▷ Para cada elemento seleccionado
      $\partial(v) \leftarrow distancia(s) + distancias[u, v]$
    end for
    $\partial_{max} \leftarrow max(\partial(u), max_{v \in s}(\partial(v)))$
    $\partial_{min} \leftarrow min(\partial(u), min_{v \in s}(\partial(v)))$
    if $\partial_{max} - \partial_{min} < min_g$ **then** ▷ Nos quedamos con el de menor dispersión
      $mejor \leftarrow u$
      $min_g \leftarrow \partial_{max} - \partial_{min}$
    end if
  end for
  return $mejor$, min_g
end function

2.2 Búsqueda Local

Dada una solución completamente aleatoria M, consiste en ir generando soluciones vecinas que mejoren el resultado de la solución original M dentro de un entorno $E(M)$, hasta que, generado todo el entorno de una solución, no se haya encontrado una mejora o se haya alcanzado un número máximo de iteraciones definida por el programador (en este caso, el máximo de iteraciones será de 10^5).

Este algoritmo requiere definir el $E(M)$ y un operador que genere una solución vecina a M dentro de su entorno. Para ello, definiremos el método `Int(M,i,j)`, que se encargará de intercambiar un elemento $i \in M$ por otro $j \notin M$ para que quede una solución vecina M' t.q. $i \notin M'$ y $j \in M'$.

Entonces es fácil deducir que el entorno de M son todas las soluciones vecinas M' que puede generar el operador `Int(M,i,j)`, que son $m \cdot (n - m)$ vecinos. No obstante, el entorno puede llegar a ser muy grande, así que realizaremos una factorización del coste para cada vecino que consideremos para aumentar la eficiencia.

El coste de realizar un intercambio es:

$$Z_{mm}(M, i, j) = (\partial_{max} - \partial_{min}) - Z_{mm}(M) = Z_{mm}(M') - Z_{mm}(M)$$

$$\partial_{max} = max(\partial(v), max_{w \in M}(\partial(w)))$$

$$\partial_{min} = min(\partial(v), min_{w \in M}(\partial(w)))$$

Para todo elemento v:

$$\partial(v) = \sum_{w \in M} d_{vw}$$

Luego para todo elemento $w \in M$, siendo $anterior(w)$ el coste de M:

$$\partial(w) = anterior(w) - d_{wu} + d_{wv}$$

El algoritmo de búsqueda local escogerá al vecino como buena solución si $Z_{mm}(M, i, j) < 0$, es decir, si al realizar el intercambio, se ha producido una mejora en cuanto al coste con respecto a la solución original.

```
function BúsquedaLocal(distancias, N, m, max_iters)
    actual ← GenerarSolucion(N,m)
    iters ← 0
    do
        iters ← iters + 1
        prima ← EscogerVecino(distancias, actual, N)
        if Distancia(prima) < Distancia(actual) then
            actual ← prima
        end if
    while iters ≥ max_iters or Distancia(prima) ≥ Distancia(actual)
    return actual, Distancia(actual)
end function

function EscogerVecino(distancias, actual, N)
    for u ∈ actual do
        for v ∉ actual do
            M' ← Int(actual, u, v)                        ▷ Operador de intercambio
            ∂(v) ← ∑_{w∈M'} d_{vw}
            for w ∈ M' do
                ∂(w) ← Distancia(actual) − d_{wu} + d_{wv}
            end for
            ∂_{max} ← max(∂(v), max_{w∈M'}(∂(w)))
            ∂_{min} ← min(∂(v), min_{w∈M'}(∂(w)))
            Z_{mm}(M') ← ∂_{max} − ∂_{min}
            Z_{mm}(M, u, v) ← Z_{mm}(M') − Z_{mm}(M)
            if Z_{mm}(M, u, v) < 0 then
                return M', Z_{mm}(M')
            end if
        end for
    end for
    return M'
end function
```

3 Desarrollo

La práctica se ha realizado en Python 3.10.2, haciendo uso de los paquetes `time`, `NumPy`, `os`, `Pandas` y `random`. No se ha utilizado ningún framework.

Para la generación de la semilla se ha usado la distribución uniforme discreta de `NumPy` a partir del método `randint`.

Para la lectura de los ficheros de datos GKD se ha usado el método `listdir` de `os`. y para su organización se han usado las estructuras de datos `array` de `NumPy`.

Para la medición de tiempos se ha usado el paquete `time`.

Para la gestión de datos en los algoritmos, además de las estructuras de datos primitivas de Python como `list`, `tuple`, o `dict`, se ha usado la estructura de datos `array`. de `NumPy` y varios de sus métodos como `sum, zeros, shape (atributo)`,...

Para la generación de las tablas de las que se hablará posteriormente se ha usado el paquete `Pandas`.

4 Análisis

Una instancia de ejecución de todo el experimento (5 ejecuciones de cada caso con cada uno de los algoritmos), se encuentra en el archivo `Tablas_MDD_2021-22.ods`. No obstante, al ejecutar el programa, siempre genera una tabla del mismo formato (salvo tablas finales, no son válidas). Basta con ejecutar el archivo mediante cualquier intérprete, aunque es necesario tener los archivos GKD en el directorio `/datos/p1a/` y tener instalados los paquetes mencionados en la sección de desarrollo.