UNIVERSIDAD
DE GRANADA

METAHEURÍSTICAS

CURSO 20-21

Práctica 1. Técnicas de Búsqueda Local y Algoritmos Greedy

PROBLEMA DE LA MÁXIMA DIVERSIDAD (MDP)

Tomás Ruiz Fernández

DNI: 77385078-Z

tomasruiz@correo.ugr.es

Grupo 3, Viernes

Índice

Introducción al problema

Como remarca el guión de la práctica el Problema de Máxima Diversidad (MDP) es un problema de optimización combinatoria el cual consiste en seleccionar un subgrupo de elementos *M* con un tamaño determinado *m* a partir de un conjunto *N* con *n* elementos con el objetivo de maximizar la diversidad entre los elementos seleccionados. Es decir, maximizar la distancia entre los elementos seleccionados.

Utilizaremos la distancia euclídea:

$$d_{ij} = \sqrt{\sum_{k=1}^{K} (s_{ik} - s_{jk})^2}$$

Imagen 1. Distancia euclídea entre dos puntos con dimensión K.Ref, [1]

El problema se podría resumir como:

$$\text{Maximize} \quad \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} d_{ij} x_i x_j$$

$$\text{Subject to} \quad \sum_{i=1}^{n} x_i = m,$$

$$x_i = \{0, 1\}, \quad 1 \leq i \leq n$$

Imagen 2. Fórmula MDP. Ref. [1]

Siendo *x* un vector binario marcando los elementos que pertenecen al subconjunto seleccionado.

Existen varios enfoques de este problema dependiendo de cómo calculemos la diversidad. En este caso se hará con MaxSum, maximizando la suma de las distancias entre los elementos. Como se ve en las fórmulas de la *Imagen 2.*

Algoritmos a aplicar

Algoritmo de Búsqueda Local

Introducción al algoritmo

“Una búsqueda local es un proceso que, dada la solución actual en la que se encuentra el recorrido, selecciona iterativamente una solución de su entorno para continuar la búsqueda.“ (p. 7) Ref. [2]

Como se menciona en las transparencias del tema 2, el algoritmo de búsqueda local empieza en una solución (en este problema aleatoria) e itera buscando en su entorno hasta que llegue a unas condiciones de parada específicas.

El entorno de la solución actual es lo que define qué otras soluciones son vecinas y por ende son candidatas del siguiente paso.

Sabiendo las ideas básicas el pseudocódigo básico de este algoritmo es:

```
Inicio
        GENERA(Solución Inicial)
        Solución Actual ← Solución Inicial;
        Mejor Solución ← Solución Actual;

        Repetir
                Solución Vecina ← GENERA_VECINO(Solución Actual);
                Si Acepta(Solución Vecina)
                        entonces Solución Actual ← Solución Vecina;
                Si Objetivo(Solución Actual) es mejor que Objetivo(Mejor Solución)
                        entonces Mejor Solución ← Solución Actual;
        Hasta (Criterio de parada);

        DEVOLVER (Mejor Solución);
Fin
```

Código 1. Pseudocódigo del algoritmo genérico de búsqueda local. Ref. [2]

Se observan funciones clave del algoritmo como generar la solución inicial, definir la estructura del entorno y la exploración del mismo, estas características afectan al rendimiento del algoritmo. Posteriormente se explica con detalle cómo se aplican estas funciones al problema concreto.

Hay que destacar que la gran desventaja que tiene este algoritmo es que se queda atrapado fácilmente en máximos o mínimos locales y depende mucho de la solución inicial.

Aplicación del algoritmo al problema

Siguiendo los pasos del algoritmo genérico, el primer paso es generar una solución inicial, para una experimentación real del algoritmo sin ninguna "decisión humana" que lo afecte, lo ideal es generar un primer subconjunto M pseudoaleatorio. Este conjunto tendrá *m* elementos y ninguno puede estar repetido.

El entorno de la solución inicial será el conjunto de soluciones a las que se puede llegar intercambiando un elemento del conjunto *M* con uno del conjunto *N/M* (los elementos no seleccionados). El entorno será de tamaño *m*(n-m),* es importante tener en cuenta el tamaño del entorno a explorar para seleccionar la técnica adecuada de generación de vecino y si merece la pena explorar el entorno entero.

En este caso no se utilizará la técnica de búsqueda local del mejor, que exploraría todo el entorno, si no la técnica de búsqueda local del primer mejor. Esta consiste en ir explorando el entorno hasta que se encuentre una solución mejor que la actual, en ese momento se da un paso y la solución actual pasa a ser la encontrada. El algoritmo sería:

```
Procedimiento Búsqueda Local del Primer Mejor
Inicio

        GENERA(Sact);

        Repetir
                Repetir
                        S'  GENERA_VECINO(Sact);
                Hasta (Objetivo(S') mejor que Objetivo(Sact))
                        O (se ha generado E(Sact) al completo)
                Si Objetivo(S') mejor que Objetivo(Sact)
                        entonces Sact  S';
        Hasta (Objetivo(S') peor o igual que Objetivo(Sact));
 DEVOLVER(Sact);

Fin
```

Código 2. Pseudocódigo Búsqueda Local del Primer Mejor. (p. 33) Ref [2]

De esta manera ganamos eficiencia. Otra forma de ganar eficiencia es factorizando el coste de las distancias, esto es calcular la distancia de un elemento *a* de *M* con los demás, así al realizar un intercambio solo tendremos que calcular la distancia del nuevo elemento *b* con *M/a.* Por ende en vez de realizar *n*(n-1)/2* operaciones, realizamos *n* operaciones.

Distancia factorizada:

$$di = \sum_{sj \in Sel} dij = d(si, Sel)$$

Cálculo de la nueva distancia:
Nueva distancia = distancia actual - d(a, M/a) + d(b, M/a)

Este cálculo se puede aprovechar para realizar una exploración inteligente del entorno. Ordenando a los elementos por contribución. Esta contribución sería el cálculo de la distancia factorizada. El procedimiento consistiría en explorar primero el elemento del conjunto *M* que menos contribución tiene y por ende es más probable de encontrar antes una solución mejor.

Habiendo visto estas características, mi construcción del algoritmo sería la siguiente:

```
Inicio
        Sel = {}            // Conjunto de seleccionados
        No_sel = {}     // Conjunto de no seleccionados

        Sel = random(m)        // m = número de elementos a seleccionar
        No_sel = N - Sel        // N = conjunto inicial

        distancia_actual = Suma de las distancias de Sel    // Calculamos el resultado actual

        contribucion = diccionario
        contribucion = d(i, Sel) para todo i (Cálculo de la distancia fact para todos los
        elementos)
        sort(contribucion)

        evaluacion = 0
        Mientras Haya mejora de la solución y evaluacion < 100000
                mejora = False
                evaluacion++

                Por x perteneciente a contribucion
                        Por y perteneciente a No_sel
                                contribucion_nuevo = d(y, Sel/x)
                                nueva_distancia = distancia_actual - contribucion[x] +
                                contribucion_nuevo

                                Si nueva_distancia > distancia_actual
                                        mejora = True
                                        Intercambiamos x por y en Sel y No_sel
                                        Actualizar la contribución

                                        distancia = nueva_distancia
                                        Salimos del bucle

                        Si hay mejora: Salimos del bucle

        Devolvemos distancia
Fin
```

Código 3. Pseudocódigo de Búsqueda Local del Primero Mejor para el problema MDP.

Este sería el pseudocódigo teniendo en cuenta las técnicas explicadas anteriormente. Para generar el conjunto inicial uso el generador de pseudoaleatorios de numpy.

El cálculo de las contribuciones se realiza usando la matriz diagonal superior, por ejemplo si calculamos la de x sería:

```
Por i perteneciente a la fila D[x, x+1:] (Empezamos a valorar la fila después de los 0s)
        Si el elemento i pertenece a Sel
                contribucion += D[x, i]
```

Código 4. Pseudocódigo de cálculo factorial del coste.

Hacemos lo mismo con la columna de x pero iterando desde el principio hasta x (D[:x, x]). Y posteriormente se utilizan esas contribuciones para calcular de manera factorial la distancia al intercambiar los elementos.

Algoritmo Greedy

La heurística del algoritmo Greedy consiste en escoger en cada paso la opción óptima para intentar construir una solución óptima. Al contrario que el algoritmo de búsqueda local, el greedy no empieza un con una solución inicial, sino que empieza desde cero construyendo la solución.

Glover desarrolló un algoritmo greedy para el problema MDP en 1998, consistiendo en calcular el centroide del conjunto que será el primer elemento, e iterar insertando los elementos más lejanos al centroide. En cada iteración calcula el centroide de Sel y repite.

Como en la práctica no disponemos de los puntos sino que de las distancias entre los puntos, lo que se hará es calcular el elemento que más distancia acumulada tenga con los demás y seguidamente ir insertando los elementos que más alejados estén del conjunto actual.

```
Inicio
        sel = {}
        no_sel = S

        centro = elemento x con más d(x, S)

        sel = sel + centro
        no_sel = no_sel - centro

        Mientras que |sel| < m:
                mejor = Max( min (d(x, y)) para todo x ∈ no_sel, para todo y ∈ sel
                sel = sel + mejor
                no_sel = no_sel - mejor
                distancia = distancia + d(mejor, sel)
Fin
```

Código 4. Pseudocódigo Algoritmo Greedy para el problema MDP

Desarrollo de la práctica

Para desarrollar la práctica me he guiado principalmente de lo que se proporciona en Prado, es decir, el Tema 2, el seminario 2 y el guión de prácticas.

Lo he desarrollado en el subsistema Linux de Windows, en Ubuntu 18, utilizando Python3. No he utilizado ningún framework especial, solo numpy y las librerías necesarias para leer los ficheros y calcular el tiempo de ejecución.

El código de la práctica se encuentra en software/FUENTES y no tiene binario ya que está en Python. En la carpeta FUENTES están los ficheros BL.py, Greedy.py, main.py y LEEME. Los ficheros BL y Greedy tienen funciones con cada algoritmo, al que se llamará desde main, pasando como argumento la matriz de distancias D y en el caso de BL la semilla.

Para ejecutar el script main se tiene que usar la siguiente instrucción:
python3 main.py SEMILLA CASO
El parámetro SEMILLA almacenará la semilla para iniciar los números pseudoaleatorios de numpy y el parámetro CASO sirve para pasarle una ruta absoluta o relativa de qué caso se desea ejecutar.
El programa devolverá la ruta pasada como argumento y los datos de la distancia calculada por los algoritmos màs los tiempos de ejecución de cada uno.

En el fichero LEEME.txt se da una explicación detallada de qué realiza cada script y de cómo ejecutar main.py.

Experimentos y análisis

Todas las ejecuciones de BL son con semilla = 22.

Algoritmo de Búsqueda Local			
Caso	**Coste obtenido**	**Desv**	**Tiempo(s)**
MDG-a_1_n500_m50	7365,5800	5,98	0,90
MDG-a_2_n500_m50	7287,9600	6,22	0,86
MDG-a_3_n500_m50	7313,9300	5,74	0,70
MDG-a_4_n500_m50	7293,1500	6,14	0,82
MDG-a_5_n500_m50	7274,3500	6,20	0,78
MDG-a_6_n500_m50	7272,2400	6,45	0,86
MDG-a_7_n500_m50	7294,2500	6,14	0,73
MDG-a_8_n500_m50	7318,4800	5,58	0,74
MDG-a_9_n500_m50	7328,2200	5,69	0,84
MDG-a_10_n500_m50	7340,4300	5,65	0,86
MDG-b_21_n2000_m200	10355176,6100	8,36	65,00
MDG-b_22_n2000_m200	10314270,3700	8,62	59,70
MDG-b_23_n2000_m200	10350204,8100	8,40	59,80
MDG-b_24_n2000_m200	10342541,4300	8,40	64,40
MDG-b_25_n2000_m200	10344195,0000	8,43	61,20
MDG-b_26_n2000_m200	10314665,8600	8,66	59,50
MDG-b_27_n2000_m200	10310573,0200	8,80	56,00
MDG-b_28_n2000_m200	10320275,4500	8,51	61,40

MDG-b_29_n2000_m200	10308631,6200	8,75	61,50
MDG-b_30_n2000_m200	10340831,2200	8,46	63,10
MDG-c_1_n3000_m300	22717631	8,71	221,83
MDG-c_2_n3000_m300	22712780	8,80	228.26
MDG-c_8_n3000_m400	38521342	11,32	386,20
MDG-c_9_n3000_m400	38547353	11,26	408,32
MDG-c_10_n3000_m400	38489039	11,47	372,40
MDG-c_13_n3000_m500	56525658	15,65	581,80

Tabla 1. Resultados Algoritmo de Búsqueda Local.

Algoritmo Greedy			
Caso	***Coste obtenido***	***Desv***	***Tiempo(s)***
MDG-a_1_n500_m50	*6865,9400*	*12,36*	*0,15*
MDG-a_2_n500_m50	*6754,0200*	*13,09*	*0,15*
MDG-a_3_n500_m50	*6741,6000*	*13,12*	*0,15*
MDG-a_4_n500_m50	*6841,5900*	*11,95*	*0,15*
MDG-a_5_n500_m50	*6740,3400*	*13,09*	*0,15*
MDG-a_6_n500_m50	*7013,9400*	*9,77*	*0,15*
MDG-a_7_n500_m50	*6637,4600*	*14,59*	*0,15*

MDG-a_8_n500_m50	*6946,2800*	*10,38*	*0,15*
MDG-a_9_n500_m50	*6898,0100*	*11,22*	*0,15*
MDG-a_10_n500_m50	*6853,6800*	*11,91*	*0,15*
MDG-b_21_n2000_m200	*10314568,3500*	*8,72*	*7,85*
MDG-b_22_n2000_m200	*10283328,5000*	*8,89*	*7,80*
MDG-b_23_n2000_m200	*10224214,1600*	*9,52*	*7,80*
MDG-b_24_n2000_m200	*10263575,4700*	*9,10*	*7,77*
MDG-b_25_n2000_m200	*10250090,8000*	*9,26*	*7,77*
MDG-b_26_n2000_m200	*10196189,8800*	*9,71*	*8,00*
MDG-b_27_n2000_m200	*10358195,6000*	*8,38*	*7,77*
MDG-b_28_n2000_m200	*10277383,1600*	*8,89*	*7,80*
MDG-b_29_n2000_m200	*10291258,6700*	*8,90*	*7,80*
MDG-b_30_n2000_m200	*10263859,3300*	*9,14*	*7,83*
MDG-c_1_n3000_m300	*22943111*	*7,80*	*25,20*
MDG-c_2_n3000_m300	*22982398*	*7,72*	*25,12*
MDG-c_8_n3000_m400	*40434465*	*6,91*	*42,40*

MDG-c_9_n3000_m400	*40488295*	*6,79*	*42,60*
MDG-c_10_n3000_m400	*40455410*	*6,95*	*42,15*
MDG-c_13_n3000_m500	*63170811*	*5,73*	*62,90*

Tabla 2. Resultados Algoritmo Greedy.

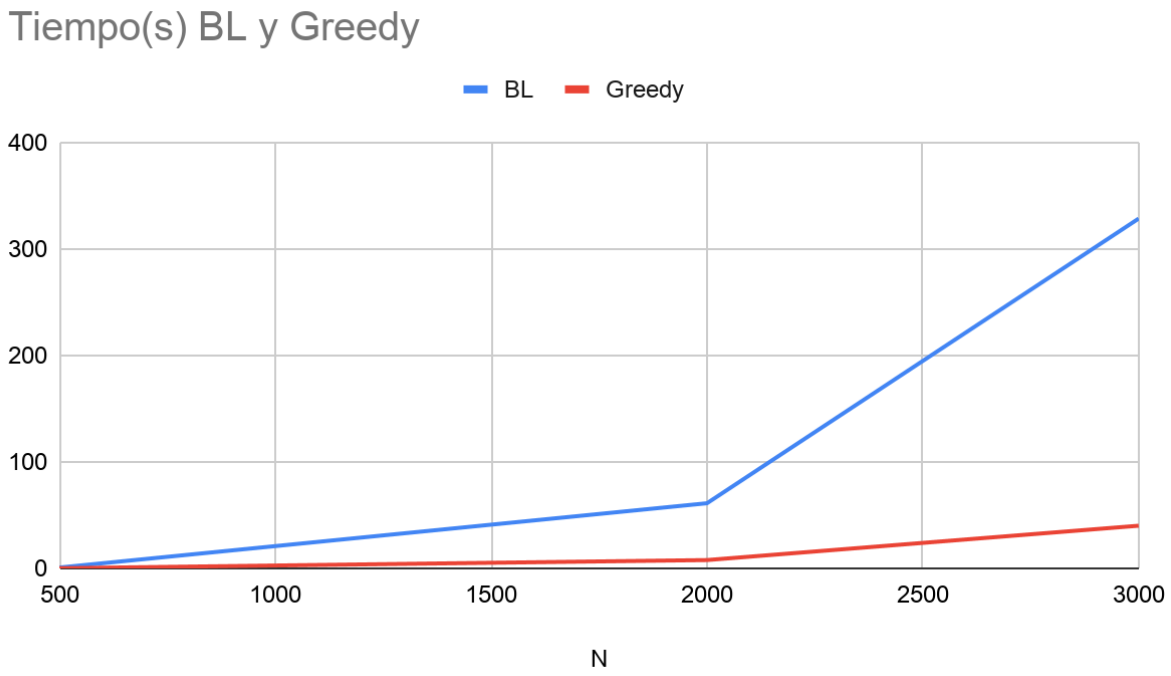
	Algoritmo BL	***Greedy***
MediaDev:	***7,08***	***8,46***
MediaTiempo:	***86,34***	***10,67***

Tabla 3. Resultados globales.

Hay varias cosas que me han llamado la atención, lo primero es el excesivo tiempo que le toma al algoritmo BL realizar las últimas ejecuciones. También como decae en calidad cuanto más grande es *n*, y a su vez cómo mejora el algoritmo Greedy.

En cuanto al algoritmo de Búsqueda Local es lógico que decaiga mucho en tiempo y en calidad cuanto más grande sea n. En ambas depende mucho de la suerte, si a la hora de explorar el entorno encuentra rápido un solución que mejora a la actual el algoritmo tendrá una velocidad decente, si nos ponemos en el caso de que encuentra una solución mejor a la primera y que tiene que hacer m*2 cambios para que no encuentre un vecino que mejora la solución, el algoritmo tendría una eficiencia de *O(M*2*M)*.
En cambio, si nos ponemos en el peor de los casos, el algoritmo podría llegar a tener una eficiencia de *O(100000 * (N-M) * (M))*.
Con la calidad pasa algo parecido. Si nos situamos de primeras en una zona donde hay un máximo local, el algoritmo se quedará en este máximo, esta cualidad no depende de *n*, sino de la forma del conjunto de soluciones.

En cuanto al algoritmo greedy, hablando de calidad es un algoritmo limitado ya que solo mira un camino hacia la solución e ignora los demás. En cuanto a eficiencia es mejor que el algoritmo de Búsqueda Local debido a su simpleza, se puede observar claramente en los resultados.



Podemos observar como el algoritmo de BL escala mucho más que el algoritmo Greedy.

Y en cuanto a calidad lo contrario, el algoritmo de BL desciende mientras que el algoritmo Greedy asciende.

En conclusión el algoritmo de Búsqueda Local depende mucho del conjunto inicial con el que empieza, ya que puede quedarse atrapado en un máximo local, y en eficiencia también depende de la suerte al explorar el entorno. En su contraparte el algoritmo Greedy solo observa un camino hacia la solución pero es más sólido a la hora de analizarlo.

Ambos algoritmos son útiles dependiendo de la situación y aportan soluciones diferentes, en este caso con el problema MDP, BL depende de qué puntos escoja al principio y de la colocación de los puntos, y el algoritmo greedy buscará solo un camino para completar una solución por lo que es muy difícil que encuentre una solución óptima, pero encontrará una solución cercana y en un tiempo asequible.

Referencias

[1] Optsicom Project, Maximum Diversity Problem, http://grafo.etsii.urjc.es/optsicom/mdp/
[2] Transparencias Tema 2, Metaheurísticas 20-21