UNIVERSIDAD
DE GRANADA

METAHEURÍSTICAS
CURSO 20/21

PRÁCTICA 3. BÚSQUEDAS POR TRAYECTORIAS

PROBLEMA DE LA MÁXIMA DIVERSIDAD

Tomás Ruiz Fernández
77385078Z
tomasruiz@correo.ugr.es
Grupo 3, Viernes

Índice

Introducción

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.*

Partes comunes entre los algoritmos

Me apoyaré un poco en el lenguaje de programación Python para explicar cómo represento y cómo hago (con pseudocódigo) los puntos importantes del problema.

Para representar las soluciones al problema usaré *listas* que contienen los puntos que están seleccionados, y al contrario, otra lista con los puntos no seleccionados, para ir explorando el espacio de soluciones vecinos con el operador de intercambio.

A la hora de generar soluciones aleatorias, genero *m* números pseudoaleatorios entre 0 y *n-1*, quedaría algo así (siendo *m* el nº de puntos a seleccionador y *n* el nº de puntos total):

```
aleatorios = generar_pseudoaleatorios(0, n, m)

for i = 0 hasta i = n:
        if i está en aleatorios:
                insertar(i, seleccionados)
        else:
                insertar(i, no seleccionados)
```

Código 1. Generación de solución aleatoria.

Para calcular la función objetivo es importante mencionar que me apoyo en la biblioteca de Python, numpy, por lo que los cálculos son algo diferentes. La función objetivo es la distancia entre los puntos seleccionados (siendo *D* la matriz de distancias):

```
funcion_objetivo
Inicio
        resultado = suma(D[seleccionados, seleccionados])
        # Cogemos las filas y columnas de cada elemento de sel, y realizamos su sumatoria
Fin
```

Código 2. Pseudocódigo función objetivo

Aunque la mayor parte del tiempo se usará el cálculo factorizado de la función objetivo, es decir, al cambiar un elemento por otro, aprovechamos los cálculos ya hechos y simplemente calcularemos la diferencia de cada punto:

```
funcion_objetivo_factorizada:
Inicio
        distancia_punto_viejo = sum(D[seleccionados, viejo]) + sum(D[viejo, seleccionados])
        distancia_punto_nuevo = sum(D[seleccionados, nuevo]) + sum(D[nuevo, sel])

        nueva_distancia = distancia - distancia_punto_viejo + distancia_punto_nuevo
Fin
# Para calcular la distancia del punto nuevo hay que quitar el punto viejo de los
seleccionados o restar sus distancias después
```

Código 3. Pseudocódigo factorización función objetivo.

Para los dos puntos sumamos todas sus distancias con los puntos seleccionados y posteriormente realizamos la diferencia.

Esquemas de los algoritmos

Algoritmo de Búsqueda Local

Ya se explicó en la práctica 1. En este caso haré la misma búsqueda local que en la práctica anterior para el algoritmo BMB e ILS. Utilizaré un diccionario para ordenar qué puntos aportan más al conjunto e iré empezando sustituyendo los que aportan menos. La búsqueda por el entorno la haré siguiendo el orden que tenga el conjunto de no seleccionados, al igual que en la primera práctica.

Algoritmo de Enfriamiento Simulado

La solución inicial de este algoritmo se rige según se ha explicado antes. La inicialización de la temperatura es la que sigue:

*t_inicial = mu * coste(S0) / -ln(fi)*

Siendo fi la probabilidad de aceptar una solución que sea un “mu” peor que la inicial. Como se menciona en el guión se usará mu = fi = 0,3. S0 representa la solución inicial.

El pseudocódigo del algoritmo sería el siguiente:

```
Inicio
        S0 <- Generar solución inicial
        t_inicial <- Generar temperatura inicial
        t = t_inicial

        Mientras t > t_final:
                vecinos = 0
                exitos = 0
```

```
            Mientras vecinos < max_vecinos y exitos < max_exitos:
                S' <- Generar nuevo vecino
                dif_coste = coste(S) - coste(S')
                vecinos++

                if dif_coste < 0 o U(0,1) < exp(-dif_coste/t):
                    Intercambiamos los puntos
                    S <- S'
                    exitos++
            if exitos == 0
                break
            Enfriar(t)
        return coste(S)
Fin
```

Código 4. Pseudocódigo ES.

En este pseudocódigo se ven partes explicadas en el guión de la práctica como las condiciones de parada.

El esquema de enfriamiento es el de Cauchy modificado, es decir:

```
Esquema_Cauchy_modificado:
Inicio
    beta = (t_inicial - t_final) / (M * t_inicial * t_final)
    t_siguiente = t / (1 + beta * t)
Fin
```

Código 5. Pseudocódigo enfriamiento.

Algoritmo de Búsqueda Multiarranque Básica

El algoritmo simplemente lanzará la búsqueda local implementada en la práctica 1, varias veces con soluciones iniciales distintas.

Algoritmo de Búsqueda Local Reiterada

Este algoritmo genera una solución inicial, la mejora con una búsqueda local (o enfriamiento simulado) y hace un bucle repitiendo la secuencia:

- Mutación de la solución
- Realizamos búsqueda local con la mutación
- En el caso de esta práctica nos quedamos con la mejor solución

En este caso solo hay que aclarar el procedimiento de mutación.

Para la mutación cambiaremos 0.1 * *m* puntos aleatorios de los seleccionados (siendo *m* el nº de puntos a seleccionar):

```
mutacion
Inicio
        elegidos_sel = elegir_aleatoriamente(sel, 0,1 * m)
        elegidos_no_sel = elegir_aleatoriamente(sel, 0,1 * m)

        nuevo_sel = sel
        nuevo_no_sel = no_sel

        for x in elegidos_sel:
                Añadir a nuevo no sel
                Quitar de nuevo sel

        Repetir el procedimiento con elegidos_no_sel

        Devoler nuevo_sel y nuevo_no_sel
Fin
```

Código 6. Pseudocódigo función de mutación.

Desarrollo

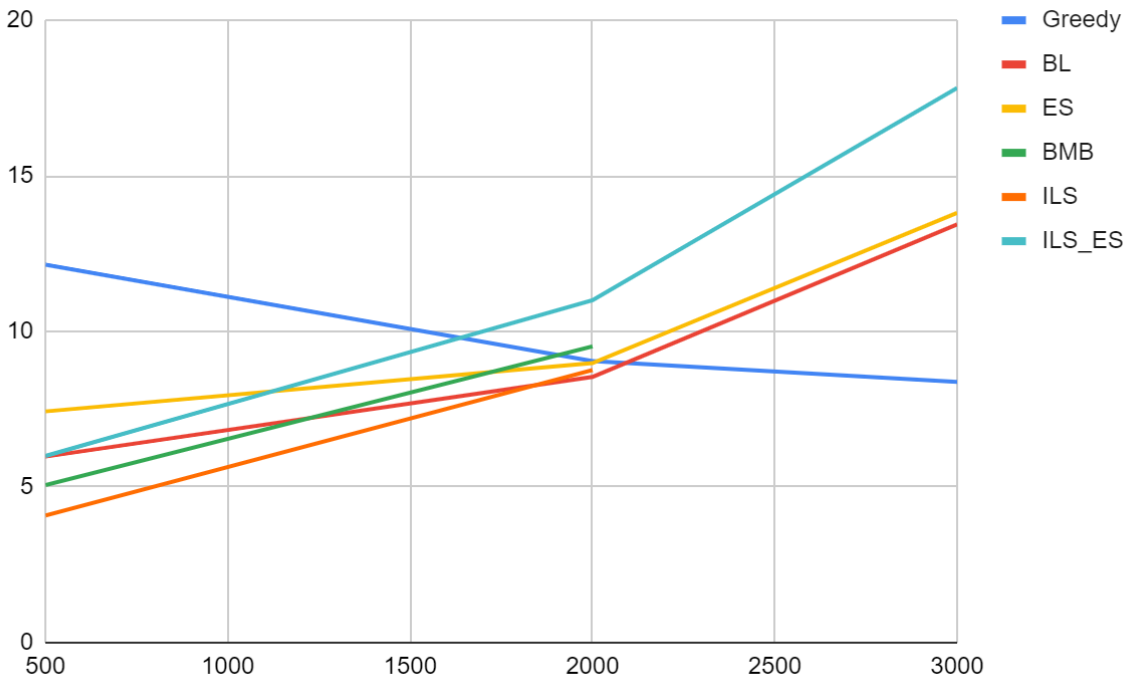
He desarrollado la práctica usando Python (Python 3.6), he utilizado el subsistema de Linux en Windows, con Ubuntu 18.

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.

Experimentos y Análisis

Voy a empezar a hacer el análisis haciendo una comparativa general de todos los algoritmos para ver cómo se comparan los unos a los otros.

Comparativa de desviación con respecto la solución óptima:

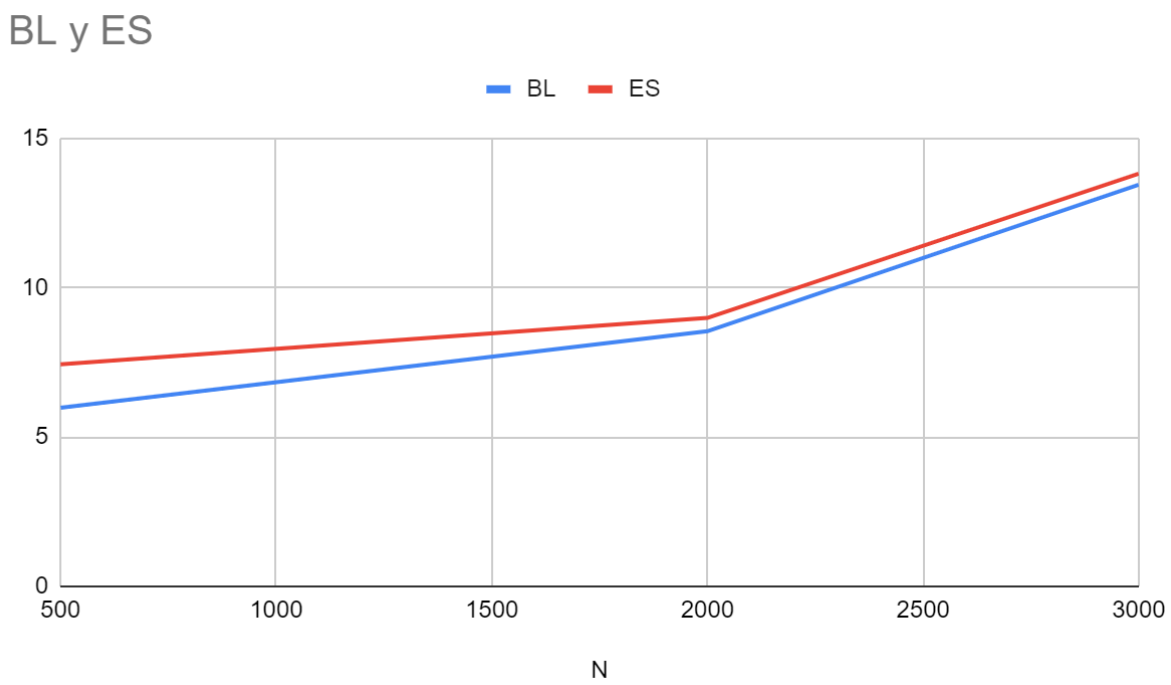


Hay dos algoritmos que no he podido ejecutar en los 3000 casos que son BMB e ILS, ya que tardaban demasiado tiempo en completarse.

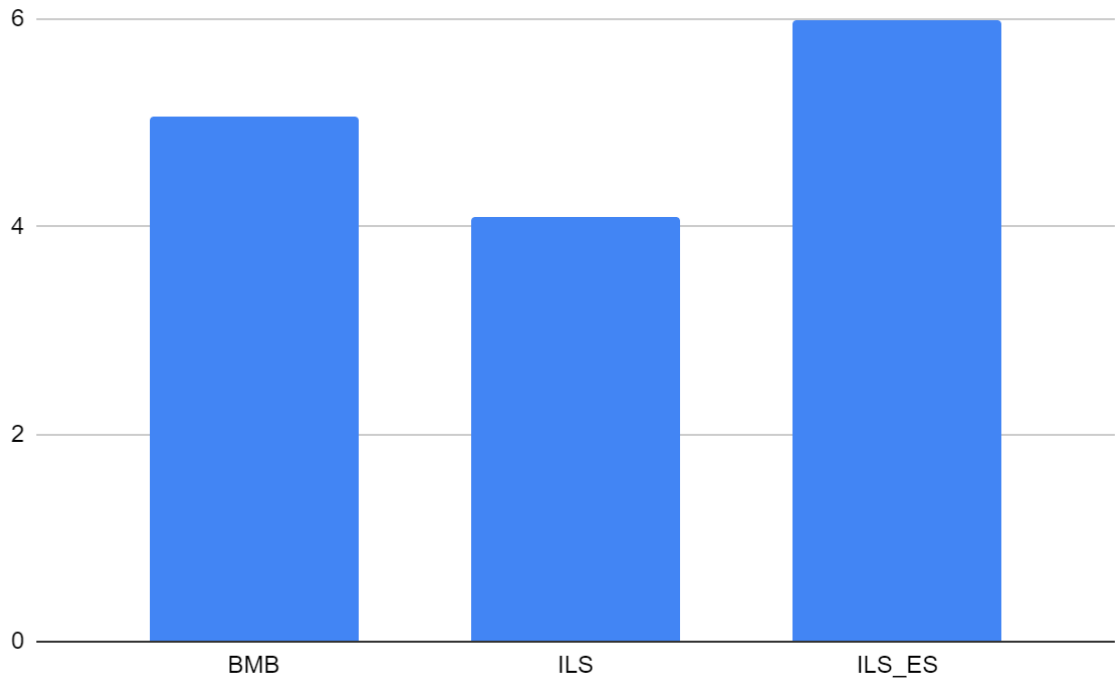
Analizar el comportamiento de estos algoritmos es un poco complejo ya que a veces dependen de la suerte. Pero se ve que en líneas generales siguen unos patrones, cuántos más puntos tengamos, dan una peor solución. Puede ser porque cuando más grande sea el espacio de soluciones es más probable toparse con un máximo local, o simplemente es más difícil encontrar el óptimo, teniendo en cuenta que se empieza con una solución aleatoria.

Entonces tiene sentido decir que BMB, ILS e ILS-ES son los que en momentos puntuales me dan las mejores soluciones cuando el número de puntos no es muy grande (tampoco se si con una N más grande se comportan igual, entiendo que empeoran pero rinden mejor que el resto de algoritmos), ya que abordan más espacio de soluciones.

El algoritmo ES y BL me dan unos resultados similares, lógico también, ya que con el esquema que he implementado el ES, tiende mucho a centrarse en explotar la solución más que explorar. Tiene bastantes opciones de explorar en las primeras iteraciones pero después esto decae mucho.



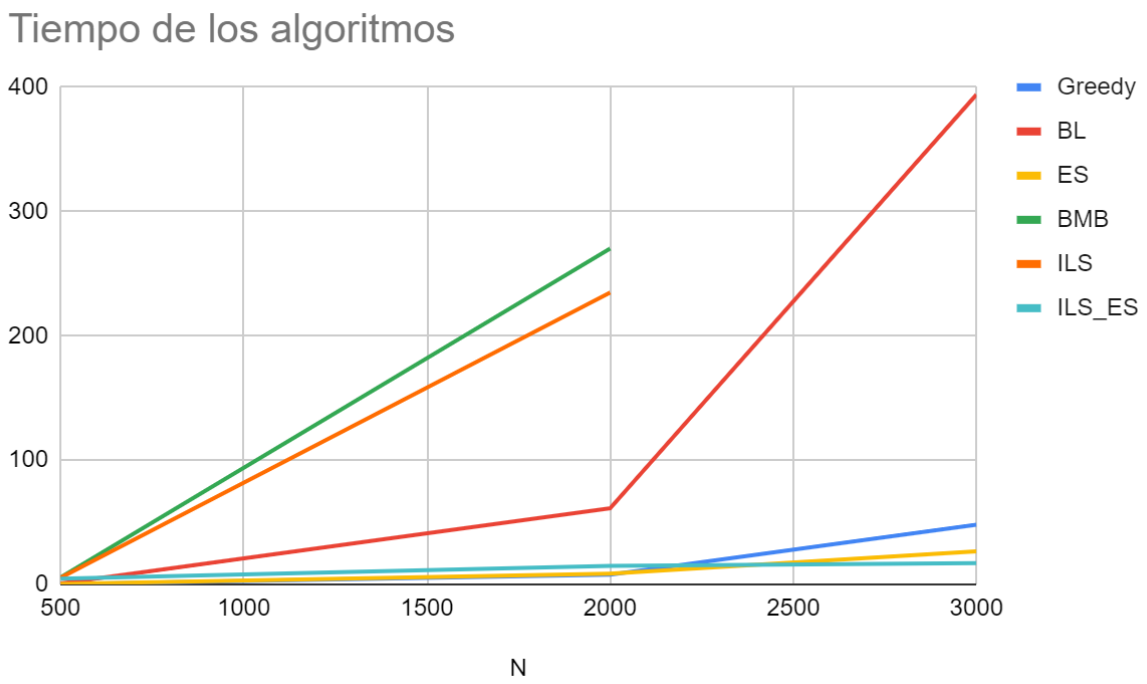
Siguen un patrón casi idéntico.



Esta comparación es con la desviación media de las 10 primera ejecuciones. Son los algoritmos que menor desviación me dan, sobretodo el ILS que parece que esa mutación le beneficia a la hora de explorar.

No menciono al algoritmo Greedy ya que su comportamiento es distinto al del resto de algoritmos.

Comparativa de tiempo de ejecución:



Hay tres grupos en esta gráfica, están el BMB y ILS por una parte como los algoritmo más lentos, tienen la carga del BL más la carga de tener que inicializar cada vez que itera como en el caso del BMB, o la carga de la mutación en el caso del ILS. Además que al empezar con soluciones distintas es más probable que se produzcan cambios cuando llamen a la BL y por ende se ejecute más veces la actualización de la contribución de los puntos, algo que es ineficiente.

El caso de los dos algoritmos que utilizan enfriamiento simulado tienen un detalle y es que cuando no hay éxitos en una iteración, el algoritmo para. Esto le da mucho ventaja en cuestión a la eficiencia.

Como conclusión, no me sorprende lo que he visto en las gráficas y en los resultados, me parece que los algoritmos tienen un equilibrio entre eficiencia y resultados, creo que cada algoritmo se tiene que adaptar al problema utilizando los parámetros adecuados y al final muchas veces depende de la suerte que tenga si inicializa con una solución aleatoria.

Todas las ejecuciones se han hecho con la SEMILLA = 22

DESVIACIÓN	N = 500	N = 2000	N = 3000
Greedy	**12,148**	**9,051**	**8,38**
BL	**5,979**	**8,539**	**13,442**
ES	**7,430666589**	**8,982975575**	**13,81220742**

BMB	**5,062185477**	**9,52**	
ILS	**4,084547597**	**8,76**	
ILS-ES	**6**	**11**	**17,82559308**

TIEMPO(s)	N = 500	N = 2000	N = 3000
Greedy	**0,15**	**7,819**	**48,074**
BL	**0,809**	**61,16**	**394,11**
ES	**0,3946159601**	**8,611440134**	**26,55163302**
BMB	**5,56094048**	**270,21**	
ILS	**5,274715018**	**234,8**	
ILS-ES	**4,645156793**	**14,86399**	**17,10166**

Referencias

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