UNIVERSIDAD DE GRANADA

Metaheurísticas

Práctica 3.a

Búsquedas por Trayectorias para el Problema de la Máxima Diversidad

Nombre: Francisco Javier Bolívar Expósito

DNI: ---

Email: ---

Grupo de prácticas: ---

Curso 2019-2020

Índice

1. Descripción del Problema

El problema de la máxima diversidad (**MDP**) consiste en **seleccionar** un **subconjunto** de *m* elementos de un conjunto más grande de *n* de forma que la **diversidad** entre los elementos escogidos se **maximice**.

Para calcular la diversidad consideraremos la variante del problema ***MaxSum***, en la que la diversidad se calcula como la suma de las distancias entre cada par de elementos seleccionados.

Por lo tanto el problema consistirá en elegir *m* elementos del conjunto inicial de *n* elementos de forma que la suma de las distancias entre ellos sea máxima, siendo la definición matemática:

$$\text{Maximizar} \quad z_{MS}(x) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} d_{ij} x_i x_j$$

$$\text{Sujeto a} \quad \sum_{i=1}^{n} x_i = m$$

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

Xi contiene un 1 si el elemento "i" se ha seleccionado y un 0 si no

MDP es de la clase de problemas **NP-Duros** de optimización combinatoria, por lo que **no es factible** encontrar la solución óptima en un tiempo razonable. Por ello en esta práctica se desarrollan y analizan algoritmos eficientes para encontrar soluciones buenas aunque no sean las óptimas.

2. Aplicación de los algoritmos empleados al problema

La práctica consiste en adaptar varias técnicas metaheurísticas al problema del MDP, siendo estas el Enfriamento Simulado, la Búsqueda Multiarranque Básica, la Búsqueda Local Reiterada y la Hibridación de ILS y ES.

Primero vamos a ver varios aspectos comunes a la implementación de los distintos algoritmos al problema.

2.1 Esquema de representación de soluciones

Se ha usado el mismo esquema de representación de soluciones que en la P1, que usa números enteros por ser más cómodo y eficiente que la representación binaria.

Siendo n = número de elementos en el conjunto inicial, m = número de elementos a seleccionar, la solución se representa como un vector de números enteros que:

- Contiene exactamente m elementos
- No contiene elementos repetidos,
- Cada elemento es un entero perteneciente al intervalo [0, n).

2.2. Función objetivo

$$z_{MS}(x) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} d_{ij} x_i x_j$$

La función objetivo evalúa una solución para MDP siguiendo el modelo MaxSum, teniendo como entrada de la solución un vector con la representación decimal usada en la P1.

El objetivo de los algoritmos es maximizar esta función objetivo, que consiste en calcular la suma de distancias entre pares de elementos seleccionados.

Todas las sumas se realizan de manera vectorizada, en vez de ir recorriendo una a una en un bucle. Si nos damos cuenta la fórmula descrita para la función objetivo con indices es equivalente a la suma de la matriz triangular superior, tomando como matriz una submatriz de la matriz de distancias que solo contenga las filas y columnas de los elementos seleccionados. Preocupándose por no realizar copias temporales de los datos y acceder directamente a estos esta función queda implementada de manera eficiente.

Finalmente el pseudocódigo de la función queda de la siguiente manera:

FuncionObjetivo(Entradas: distancias entre elementos ‘dist’ y vector solución “Sol”
Salida: Coste de la solución):

1. matrizDistanciasSeleccionadas := dist con las filas y columnas de los elementos en ‘Sol’
2. Coste := sumatoriaTriangularSuperior(matrizDistanciasSeleccionadas)

devolver Coste

2.3. Generación de la Solución Inicial

inicializarSolucion():
1. Sol := elegir m números aleatorios diferentes de [0, n)
2. Devolver Sol

Siendo n el número de elementos seleccionables, m el número de elementos a seleccionar, generamos de forma aleatoria como se indica en el siguiente pseudocódigo:

2.4. Operador de vecino de intercambio en ES y BL

A continuación se describe en pseudocódigo la generación de una solución vecina, el calculo factorizado del nuevo valor de la función objetivo solo se lleva a cabo si esta nueva solución se acepta, por lo que se muestra este proceso en la descripción concreta de cada algoritmo.

Como parámetros tenemos ‘Sol’, la solución actual con los elementos seleccionados, ‘s’, el conjunto de elementos no seleccionados, ‘i’ índice o índices no repetidos aleatorios de Sol, ‘j’, índice o índices no repetidos aleatorios de ‘s’.

generarVecino(Sol, s, i, j):
1. Sol_veci := Sol
2. Sol_veci[i] := s[j]
3. Devolver Sol_veci

2.5 Algoritmo de búsqueda por trayectorias

La estructura de la búsqueda por trayectorias es igual para los dos variantes del ILS. La función ‘refine’ será sustituida por el algoritmo de intensificación usado, ya sea la búsqueda local o el enfriamiento simulado. El operador de mutación se aplica sobre 0.1 de los elementos de la solución para generar un cambio brusco.

A continuación se define el pseudocódigo del operador de mutación y el esquema de búsqueda:

operadorMutacion(sol, s):
```
i = elegir 0.1 * m números aleatorios diferentes de [0, longitud de sol)
j = elegir 0.1 * m números aleatorios diferentes de [0, longitud de s)
sol_veci = generarVecino(sol, s, i, j)
Devolver sol_veci
```

```
ils(dist, n, m, n_sol, max_evals_ref):
  1. sol = refine(dist, inicializarSolucion(), max_evals_ref)
  2. mejor_sol = sol
  Desde 0 hasta n_sol – 1
        3. seleccionables = [0, n) diferencia con sol
        sol = operadorMutacion(sol, seleccionables)
        6. sol = refine(dist, sol, max_evals_ref)

        Si ‘sol’ es mejor que ‘mejor_sol’
          7. mejor_sol := sol
        En caso contrario
          8. sol := mejor_sol
 9. Devolver mejor_sol
```

3. Descripción de la implementación de los Algoritmos

3.1.1. Temperatura inicial y esquema de enfriamiento para ES

La temperatura inicial se calcula con la siguiente fórmula:

$$T_0 = \frac{\mu \cdot C(S_0)}{-\ln(\Phi)}$$

Siendo ‘sel’ la solución inicial y con Φ = μ = 0.3 en las ejecuciones la implementación queda así:

T_0 := (u * fObjetivo(dist, sel)) / - ln(u)

Para el esquema de enfriamiento se emplea el esquema de Cauchy modificado, quedando la actualización de la temperatura como:

T_{k+1} := T_k / (1 + beta * T_k)

Siendo ‘beta’:

beta = (T_0 - T_f) / ((max_evals / max_veci) * T_0 * T_f)

3.1.2. Esquema de búsqueda ES

En el problema del MDP buscamos maximizar la función objetivo, por lo que la condición de cambio a una solución vecina cambia ligeramente respecto a la proporcionada en el seminario. Se presenta a continuación el pseudocódigo del esquema de búsqueda de ES:

```
sa():
  T := T0; sol := inicializarSolucion(); mejor_sol := sol; s := [0, n) diferencia con sol
  Z := matriz con suma de distancias de cada elemento en 'sol' con el resto de elementos en 'sol'
  Mientras T > Tf y se suceda al menos 1 éxito en la iteración:
      rand_i := elegir 'max_veci' números aleatorios diferentes de [0, m)
      rand_j := elegir 'max_veci' números aleatorios diferentes de [0, longitud de s)

        Mientras vecinos_generados < max_veci y exitos < max_exitos y evals < max_evals:
            i := rand_i[vecinos_generados]
            j := rand_j[vecinos_generados]
            vecino = generarVecino(sol, s, i, j)

            Δf := Z[i] – contribución vecino (suma de distancias de j a cada elemento de vecino)
            Si (Δf < 0) ó (random(0, 1) <= exp(-Δf / k * T):
              Z := Z – suma de distancias de cada elemento de vecino a i
              Z := Z + suma de distancias de cada elemento de vecino a j
              Z[i] := contribución vecino

              sol := vecino
              Si Z de sol es mejor que Z de mejor_sol:
                  mejor_sol := sol

      T := T / (1 + beta * T)

  Devolver mejor_sol
```

3.2 Operadores BL

```
listaDeCandidatos(Sol):
   Elementos = conjunto [0, filas de la matriz de distancias)
   Candidatos = Eliminar de 'Elementos' aquellos valores que estén en 'Sol'
   Devolver Candidatos
```

```
generarVecino(i, elemento_no_seleccionado):
   sol_veci = sol
   sol_vecina[i] = elemento_no_seleccionado
   Devolver sol_vecina
```

```
ExploracionDelEntorno con fObjtivo fact():
   Z = contribución de cada elemento a la solución (suma de distancias de cada elemento de 'Sol' al
         resto de elementos seleccionados)
   Para cada i en índices ordenados de Z de menor a mayor mientras se encuentre mejora
      Para cada elem en listaDeCandidatos(Sol)
         sol_veci = generarVecino(i, elem)
         Si contribución vecino (suma de distancias de elem a cada elemento de sol_veci) > Z[i]
            Z = Z  – suma de distancias de cada elemento de sol_veci a sol[i]
            Z = Z + suma de distancias de cada elemento de sol_veci a elem
            Z[i] = contribución vecino
            sol = sol_veci
```

3.3. Esquema de Búsqueda BMB

La búsqueda local multiarranque consiste simplemente en ejecutar la BL 'n_sol' veces desde diferentes puntos iniciales aleatorios y quedarnos con la mejor solución. El pseudocódigo queda así:

```
bmb():
  mejor_sol = vector de 0s de tamaño m

  Repetir n_sol veces:
         sol_inicial = inicializarSolucion()
         sol = búsquedaLocal(dist, sol_inicial, max_evals)
         Si fObjetivo(sol) > fObjetivo(mejor_sol):
            mejor_sol = sol

   Devolver mejor_sol
```

3.4. Esquema de Búsqueda variantes ILS

La estructura es similar al pseudocódigo definido como ‘ils’ pero sustituyendo la función de intensificación ‘refine’ por la Búsqueda Local en la primera variante y por el Enfriamiento Simulado en la segunda variante.

4. Implementación de la práctica y manual de usuario

La implementación de la práctica se ha realizado en **Python**, usando como interprete para las pruebas Python 3.8.1.

Se han utilizado 4 módulos, **time** para medir los tiempos de ejecución, **os** para recorrer el directorio de datos, **numpy** para realizar la mayoría de operaciones y **pandas** para guardar los resultados y generar las hojas de cálculo siendo estos dos últimos los no incluidos en Python.

Se ha utilizado un ‘**profiler**’ para detectar las partes del código que más tiempo de ejecución ocupan y optimizarlas. La conclusión obtenida es que las partes que más tiempo de computación requieren son la función objetivo por su enorme cantidad de llamadas y el operador de reparación, por necesitar calcular las contribuciones de los elementos.

Se ha intentado conseguir un buen tiempo de ejecución, siendo los principales cuellos de botella del programa el cálculo de la función objetivo y la generación de números aleatorios. Por lo tanto, se ha intentando realizar siempre las **operaciones de forma vectorizada** con los arrays de Numpy, **evitando copias innecesarias** de estos vectores optimizando el acceso a memoria y se han generado todos los números aleatorios que se vayan a necesitar en un algoritmo de golpe.

Para usar el programa es necesario instalar NumPy y Pandas. Se incluye un archivo **requirements.txt** que se puede usar para instalar la versión usada, utilizando “pip” con ‘pip install -r requirements.txt’.

Todo está contenido en un único script **MDP.py** que cuelga del directorio **FUENTES.** Simplemente será necesario ejecutar el script y se guardaran los resultados para todos los casos de ejecución contenidos en el directorio **BIN** en el directorio raíz de la práctica como ‘resultados.csv’

5. Análisis de los resultados

Para todos los algoritmos se ha usado la semilla 510.

Para el ES se realizan 100.000 evaluaciones de la función objetivo como máximo, y max_veci se calcula como se indica en el guión pero dividiendo el resultado por max_evals / 100000, como max_exitos depende de max_veci de esta forma se escala el número de enfriamientos según la cantidad de evaluaciones de la función objetivo que se establezca, lo que es útil en el caso de ejecución considerado para el ILS.

Se usan 10 iteraciones con 10.000 evaluaciones máximo de la función objetivo en cada ejecución de la BL en la BMB.

Se usan 10 iteraciones con 10.000 evaluaciones máximo de la función objetivo en cada ejecución de la BL en ILS.

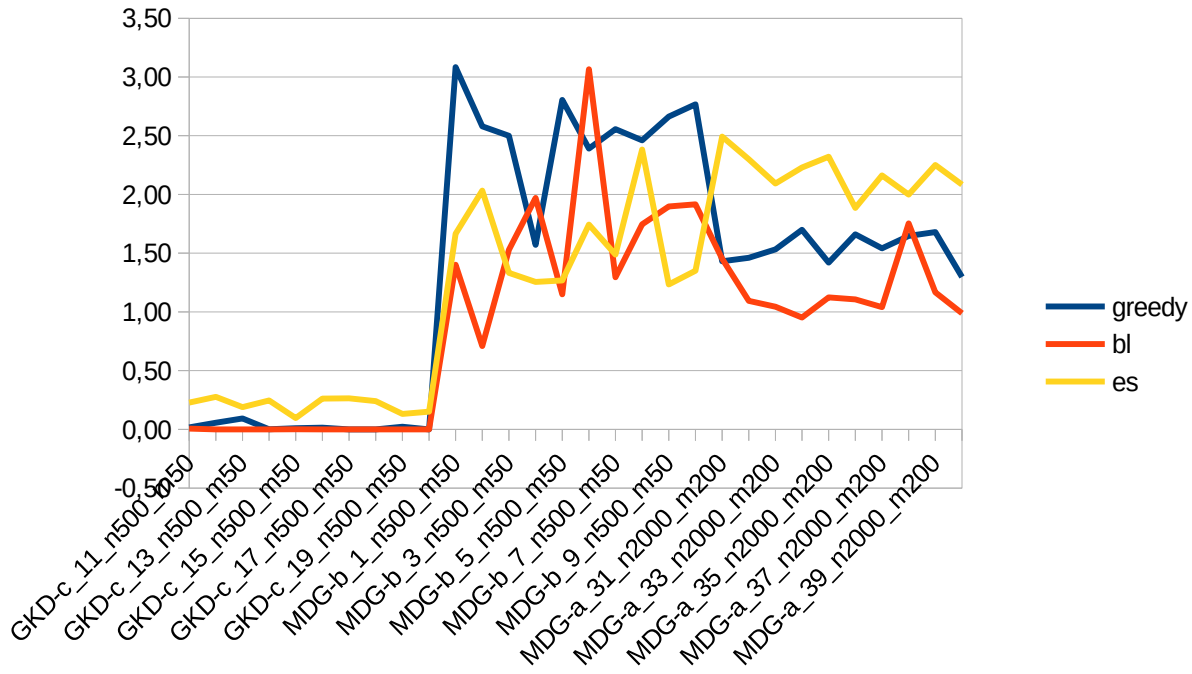
Se usan 10 iteraciones con 10.000 evaluaciones máximo de la función objetivo en cada ejecución de ES en ILS-ES.

Todos los tiempos están medidos en segundos. Se han vuelto a ejecutar todos los algoritmos de la P1 ya que en la P2 se mejoró el tiempo de ejecución de la Búsqueda Local y no sería una comparación justa sin actualizar los tiempos.

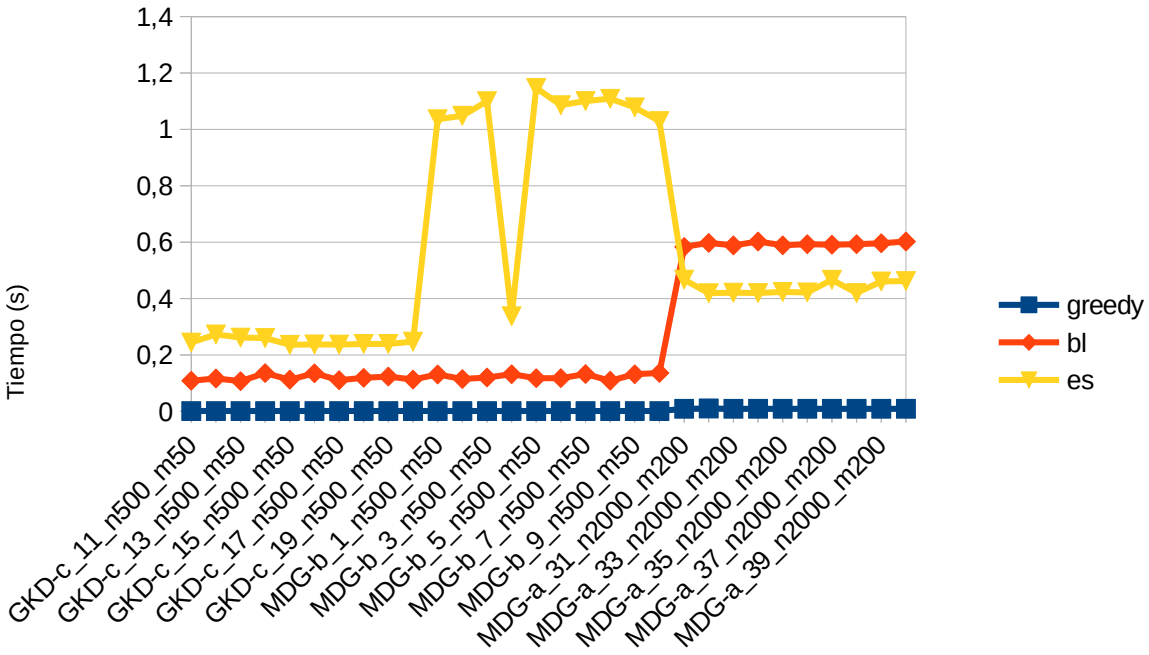
Tabla de Resultados Global

Algoritmo	**Desv**	**Tiempo**
Greedy	1,3654	0,0036
BL	0,9471	0,2784
Greedy + BL	0,9905	0,259
ES	1,3217	0,5641
BMB	0,7266	0,5801
ILS	0,5282	0,5187
ILS-ES	0,9215	0,9939

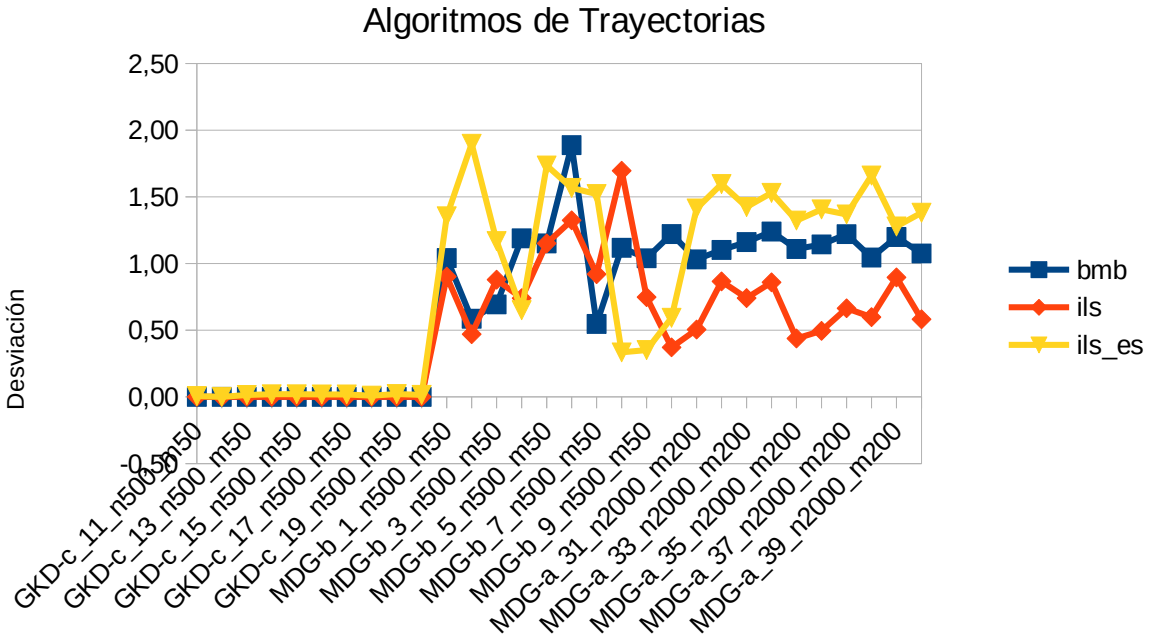
El ES tiene en media peores resultados que la BL. El esquema de enfriamiento usado para el MDP no consigue un buen equilibrio intensificación/diversificación, especialmente en los casos de ejecución más grandes en los que se necesitan muchas mejoras para llegar a converger como podemos ver en el siguiente gráfico de evolución de la desviación:



También al no depender de una heurística que puede funcionar mejor o peor en algunos casos y quedar menos atrapado en óptimos locales consigue buenos resultados en comparación con el greedy y la BL en los casos MDG-b, teniendo mayor dependencia del tamaño del problema.



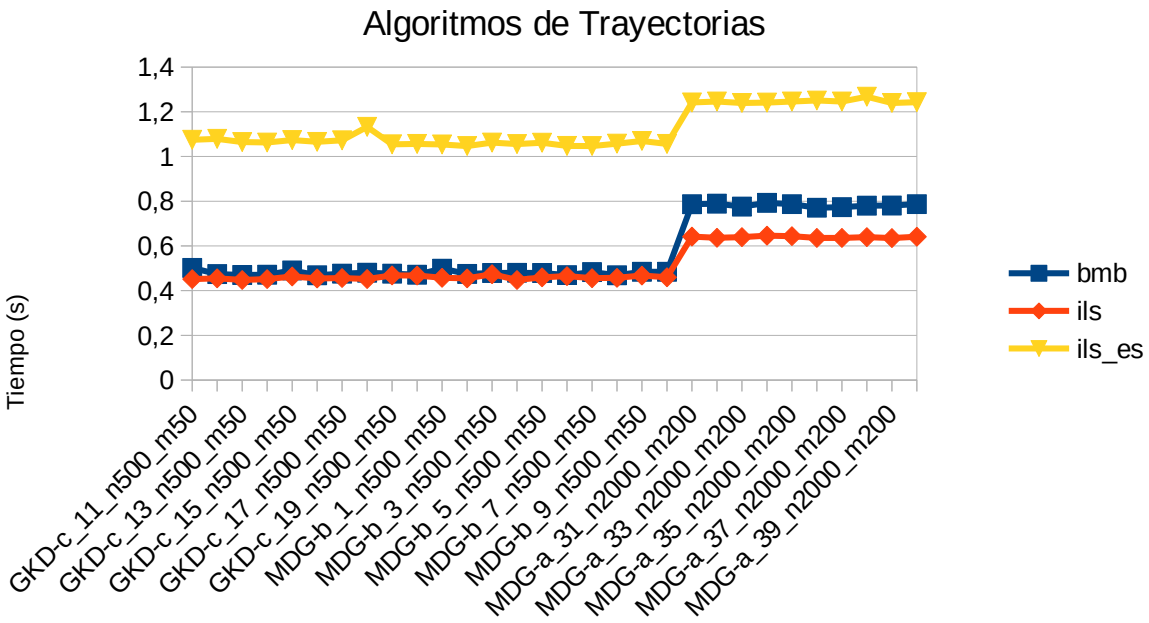
En la comparación de tiempos el greedy sigue siendo el más rápido con diferencia, y la BL es en general algo más rápida que el ES. Sin embargo el ES suele tener mejor tiempo para los casos más grandes, tal vez porque al no converger fácilmente llegue rápidamente a la condición del máximo de éxitos, el caso contrario parece darse en los casos MDG-b



Ahora vamos a analizar los algoritmos de trayectorias, todos presentan mejores resultados en media que los algoritmos de búsqueda simples o el algoritmo greedy, aun teniendo disponibles menos evaluaciones de la función objetivo en cada búsqueda. Aunque para ello necesitan algo más de tiempo que la BL, pero teniendo en cuenta la diferencia y que todos los tiempos están por debajo de 1 segundo no es un compromiso muy negativo.

La BL se queda atrapada en óptimos locales por lo que no aprovecha todas sus evaluaciones de la función objetivo disponibles, por lo que la BMB es una mejora simple pero efectiva sobre esta al empezar desde distintos puntos aleatorios.

El algoritmo que mejores resultados obtiene es el ILS ya que obtiene un buen equilibrio entre intensificación / diversificación explorando distintos entornos de búsqueda. El operador de mutación fuerte es bueno para aportar diversidad y salir de un óptimo local, la búsqueda local en cambio es muy buena intensificando permitiendo obtener el óptimo local. Con estas consideración parece lógico que el ILS-ES obtenga peores resultados, ya que se sustituye la búsqueda local por el Enfriamiento Simulado que no aporta tanta intensificación como la BL.



En tiempo vemos un paralelismo, los algoritmos que obtienen mejores resultados también son los que menos tiempo utilizan, teniendo los 3 algoritmos de trayectorias parámetros muy similares.

El algoritmo ILS es el que mejor resultados ha proporcionado en las 3 prácticas de la asignatura, superando a los algoritmos genéticos y meméticos tradicionales donde el mejor de todos tenía una desviación media de 0.67 y un tiempo medio de 2 segundos.