Práctica 3.a

Búsquedas por Trayectorias para el Problema de la Máxima Diversidad

Alejandro Palencia Blanco

DNI: 77177568X

alepalenc@correo.ugr.es

Grupo y horario de prácticas: Jueves 17:30-19:30

Índice

1. Formulación del problema

El **Problema de la Máxima Diversidad** (*Maximum Diversity Problem*, MDP) es un problema de optimización combinatoria que pertenece a la clase de complejidad NP-completo, luego su resolución es compleja. Partiendo de un conjunto inicial S de n elementos, el problema general consiste en seleccionar un subconjunto Sel de m elementos que maximice la diversidad entre los elementos escogidos. La diversidad se calcula a partir de las distancias entre los elementos, las cuales se almacenan en una matriz $D = (d_{ij})$ de dimensión $n \times n$.

Existen distintas variantes del problema que dependen de la forma en que se calcula la diversidad:

- *MaxSum*: La diversidad se calcula como la suma de las distancias entre cada par de elementos seleccionados
- *MaxMin*: La diversidad se calcula como la distancia mínima entre los pares de elementos seleccionados
- *Max Mean Model*: La diversidad se calcula como el promedio de las distancias entre los pares de elementos seleccionados
- *Generalized Max Mean*: Existen pesos asociados a los elementos empleados en el denominador al calcular el promedio de distancias (hay un orden de importancia de los elementos)

En esta práctica, trabajaremos con el criterio *MaxSum*, luego el problema se puede formular de la siguiente forma:

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

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

$$x_i \in \{0, 1\} \; \forall i \in \{1, \ldots, n\}$$

Donde x es un vector binario con n componentes que indica los m elementos seleccionados y d_{ij} es la distancia entre los elementos i y j.

2. Consideraciones comunes a los algoritmos empleados al problema

Los algoritmos que emplearemos en esta práctica comparten una serie de características comunes que serán descritas en esta sección. Concretamente, aquí explicaremos el esquema de representación de las soluciones, la función objetivo y los distintos operadores comunes.

El lenguaje utilizado para la implementación de la práctica ha sido `C++` y he usado las siguientes bibliotecas:

- `<iostream>`
- `<ctime>` para medir tiempos
- `<cmath>` para lsa funciones log y exp
- `<cstdlib>` para las funciones `rand` y `srand`
- `<stl>` para la estructura de datos `vector`
- `<algorithm>` para la función `swap` y `random_shuffle`

2.1. Esquema de representación de soluciones

En esta práctica volvemos a usar únicamente el esquema de soluciones de la primera práctica, que consiste en representar los elementos de S mediante números enteros del 0 al $n-1$. Por tanto, una solución factible está formada por un subconjunto $Sel \subset S = \{0, \ldots, n-1\}$ tal que $|Sel| = m$.

En todos los algoritmos se trabaja con una estructura `Solucion`, que está constituida por los siguientes elementos:

- Un vector de enteros `selected` en el que se almacenan los m elementos de S que determinan una solución del problema.
- Un vector de enteros `not_selected` en el que se almacenan los $n-m$ elementos de S no seleccionados.
- Un flotante `fitness`, que guarda el *fitness* de la solución.

Como el orden de los elementos en los vectores `selected` y `not_selected` es irrelevante, esto será aprovechado en determinados puntos de la implementación para eliminar elementos en ellos de una forma más rápida, o para elegir aleatoriamente subconjuntos de los mismos (barajando el vector y eligiendo los elementos que se encuentren en las primeras posiciones).

2.2. Función objetivo

La función objetivo que he implementado recibe como parámetros una solución y la matriz de distancias y simplemente actualiza el valor de fitness asociado a dicha solución. Para ello, inicializa su fitness a 0 y calcula su fitness sumando las distancias entre cada para de elementos seleccionados.

Algorithm 1: evaluateFitness

```
Data: sol : solución,
      matrix : matriz de distancias
begin
    sol.fitness ← 0
    for element_1, element_2 ∈ sol.selected do
        sol.fitness ← sol.fitness + matrix[element_1][element_2]
    end
end
```

Además de ésta, también haré uso de otra función que calcule la contribución de un elemento a una solución. Esta contribución se obtiene sumando las distancias de dicho elemento a cada uno de los elementos seleccionados, y será usada en la factorización del movimiento de intercambio en los algoritmos de Búsqueda Local y Enfriamiento Simulado.

Algorithm 2: contribution

```
Data: sol : solución,
      element : elemento cuya contribución se quiere calcular,
      matrix : matriz de distancias
Result: contrib : contribución del elemento
begin
    contrib ← 0
    for sel_element ∈ sol.selected do
        contrib ← contrib + matrix[element][sel_element]
    end
end
```

2.3. Generación de solución aleatoria

El operador de generación de solución aleatoria recibe como parámetros una solución, el número total de elementos (n) y el número de elementos que una solución debe tener seleccionados para ser factible (m). Al finalizar, devuelve una solución factible. Para ello, introduce primero todos los elementos en el vector de no seleccionados, luego baraja dicho vector y, por último, mueve aleatoriamente m elementos al vector de seleccionados (inicialmente vacío).

Algorithm 3: generateRandomSolution

Data: sol : solución
n : número total de elementos,
m : número de elementos que una solución debe tener seleccionados para ser factible

begin
$sol.selected \leftarrow \{\}$
///// Introduzco todos los elementos en $not\_selected$ /////
$sol.not\_selected \leftarrow \{0, 1, \ldots, n-1\}$
///// Barajo el vector de elementos no seleccionados /////
$random\_shuffle(sol.not\_selected)$
///// Muevo aleatoriamente m elementos de $not\_selected$ a $selected$ /////
for $i \in \{0, \ldots, m-1\}$ **do**
$random\_element \leftarrow rand() \,\%\, |sol.not\_selected|$
$swap(sol.not\_selected[n-1], sol.not\_selected[random\_element])$
$sol.selected \leftarrow sol.selected \cup \{sol.not\_selected[n-1]\}$
$sol.not\_selected \leftarrow sol.not\_selected - \{sol.not\_selected[n-1]\}$
end
end

3. Algoritmos

3.1. Enfriamiento simulado

El algoritmo de Enfriamiento Simulado está diseñado con el objetivo de intentar evitar óptimos locales de baja calidad. Para ello, este algoritmo permite la aceptación de soluciones en el vecindario de la solución actual que sean peores que ésta con una determinada probabilidad. Esto se implementa a través de temperaturas con un esquema de enfriamiento.

El algoritmo empieza generando aleatoriamente una solución inicial, *sol*, y calculando la temperatura inicial con la que empezará a ejecutarse:

$$T_0 = \frac{\mu \cdot sol.fitness}{-\log(\phi)}, \qquad \text{donde } \phi = \mu = 0.3$$

Esta temperatura disminuirá progresivamente al final de cada iteración del bucle externo hasta que se llegue al límite de evaluaciones fijado o si no se ha aceptado ninguna nueva solución durante la iteración actual.

Para aceptar nuevas soluciones, tenemos un bucle interno en el que se hará uso de dos constantes:

- $max_{vecinos} = 10 \cdot m$: número máximo de vecinos a generar en el bucle interno.
- $max_{exitos} = 0.1 \cdot max_{vecinos}$: número máximo de vecinos aceptados en el bucle interno.

En cada iteración del bucle interno, se generará un nuevo vecino mediante el operador de intercambio (intercambiando aleatoriamente un elemento seleccionado por otro no seleccionado) y se comparará con la solución actual. Si el vecino v es mejor que la solución actual *sol*, siempre es aceptado. Si no, se aceptará si un número generado en una distribución uniforme $[0, 1]$ es menor o igual que $\exp((v.fitness - sol.fitness)/T_k)$, donde T_k es la temperatura actual.

El esquema de enfriamiento usado en esta práctica es el de Cauchy modificado. Fijada una temperatura final $T_f = 10^{-3}$ y un número de enfriamientos M, el algoritmo inicializa una constante β que depende de M y cada vez que se produce un enfriamiento se modifica la temperatura actual mediante la siguiente regla:

$$T_{k+1} = \frac{T_k}{1 + \beta \cdot T_k}, \qquad \text{donde } M = \frac{max_{evaluaciones}}{max_{vecinos}} \text{ y } \beta = \frac{T_0 - T_f}{M \cdot T_0 \cdot T_f}$$

Algorithm 4: ES

```
Data: matrix : matriz de distancias,
      m : número de elementos que una solución debe tener seleccionados para ser factible
begin
    ///// Genero una solución inicial aleatoriamente /////
    generateRandomSolution(sol, n, m)
    evaluateFitness(sol, matrix)
    best_sol ← sol
    ///// Inicializo constantes /////
    MAX_VECINOS ← 10 · m
    MAX_EXITOS ← m
    MAX_EVALUATIONS ← 100000
    M ← MAX_EVALUATIONS/MAX_VECINOS
    PHI ← 0.3
    MU ← 0.3
    T_0 ← (MU · sol.fitness)/(−log(PHI))
    T_f ← 0.001
    while T_f ≥ T_0 do
        T_f ← 0.1 · T_f
    end
    BETA ← (T_0 − T_f)/(M · T_0 · T_f)
    ///// Aplico enfriamiento simulado /////
    t ← T_0
    evaluations ← 1
    do
        num_vecinos ← 0
        num_exitos ← 0
        while num_vecinos < MAX_VECINOS and
              num_exitos < MAX_EXITOS and
              evaluations < MAX_EVALUATIONS do
            i ← rand() %|sol.selected|
            sel_element ← sol.selected[i]
            j ← rand() %|sol.not_selected|
            not_sel_element ← sol.not_selected[j]
            inc_fitness ← (contribution(sol, not_sel_element, matrix)
                          −matrix[sel_element][not_sel_element])
                          −contribution(sol, sel_element, matrix)
            u ← rand()/RAND_MAX
            if inc_fitness > 0 or u ≤ exp(inc_fitness/t) then
                swap(sol.selected[i], sol.not_selected[j])
                sol.fitness ← sol.fitness + inc_fitness
                num_exitos ← num_exitos + 1
                if sol.fitness > best_sol.fitness then
                    best_sol ← sol
                end
            end
            num_vecinos ← num_vecinos + 1
            evaluations ← evaluations + 1
        end
        t ← t/(1 + BETA · t)
    while evaluations < MAX_EVALUATIONS and num_exitos > 0;
end
```

3.2. Búsqueda Multiarranque Básica

Un gran problema del algoritmo de Búsqueda Local es su alta dependencia de la solución inicial aleatoriamente generada, pues ésta deterina en gran medida el óptimo local que será alcanzado. La Búsqueda Multiarranque Básica intenta reducir esta dependencia ejecutando Búsqueda Local sucesivas vecces sobre distintas soluciones iniciales generadas aleatoriamente. El algoritmo devolverá la mejor solución encontrada.

El número de evaluaciones se reparte de de forma equitativa entre las suucesivas ejecuciones de Búsqueda Local. Como aplicaremos 10 iteraciones de Búsqueda Local y el número total de evaluaciones es 100000, el número de evaluaciones que tendrá como máximo cada BL será 10000.

Algorithm 5: BMB

```
Data: matrix : matriz de distancias,
      m : número de elementos que una solución debe tener seleccionados para ser factible
begin
    ///// Inicializo constantes /////
    ITERATIONS ← 10
    MAX_EVALUATIONS_LS ← 10000
    ///// Aplico BMB /////
    generateRandomSolution(best_sol, n, m)
    evaluateFitness(best_sol, matrix)
    localSearch(best_sol, MAX_EVALUATIONS_LS, matrix)
    for i ∈ {1, ..., ITERATIONS − 1} do
        generateRandomSolution(sol, n, m)
        evaluateFitness(sol, matrix)
        localSearch(sol, MAX_EVALUATIONS_LS, matrix)
        if sol.fitnesss > best_sol.fitness then
            swap(best_sol, sol)
        end
    end
end
```

Aplicaremos el algoritmo de Búsqueda Local del primer mejor desarrollado en la práctica 1. Este algoritmo finaliza cuando se alcanza el número máximo de evaluaciones o cuando no se encuentre una solución mejor en todo el vecindario.

La exploración del vecindario se realiza ordenando los elementos seleccionados de menor a mayor contribución e iterando en este orden sobre ellos. A continuación, se recorren los elementos no seleccionados aleatoriamente. Para cada par de elementos seleccionado y no seleccionado, se aplica una factorización de la función objetivo que consiste en calcular las contribuciones ambos elementos a la solución actual. Si la contribución del elemento no seleccionado es mayor, entonces se aplica el operador de intercambio de vecino sobre la solución (que intercambia el elemento seleccionado por el no seleccionado).

Algorithm 6: localSearch

```
Data: sol : solución sobre la que se aplica búsqueda local,
      max_evaluations : número máximo de evaluaciones,
      matrix : matriz de distancias
begin
    evaluations ← 0
    do
        exchange ← false
        for i ∈ {0, ..., |selected| − 1} and !exchange and evaluations < max_evaluations do
                        ///// Encuentro el siguiente elemento seleccionado con menor contribución /////
            worst_element_contrib ← contribution(selected, selected[i], matrix)
            for j ∈ {i + 1, ..., |selected| − 1} do
                element_contrib ← contribution(selected, selected[j], matrix)
                if element_contrib < worst_element_contrib then
                    swap(selected[i], selected[j])
                    worst_element_contrib ← element_contrib
                end
            end
            for j ∈ {0, ..., |not_selected| − 1} and !exchange and evaluations < 400 do
                            ///// Calculo la contribución del siguiente elemento no seleccionado /////
                element_contrib ← contribution(selected, not_selected[j], matrix)
                                  −matrix[selected[i]][not_selected[j]]
                evaluations ← evaluations + 1
                    ///// Si tiene mayor contribución que el elemento seleccionado, los intercambio /////
                if element_contrib > worst_element_contrib then
                    swap(selected[i], not_selected[j])
                    sol.fitness ← sol.fitness + element_contrib − worst_element_contrib
                    exchange ← true
                end
            end
        end
    while exchange and evaluations < max_evaluations;
end
```

3.3. Búsqueda Local Iterativa

El algoritmo de Búsqueda Local Iterativa es otro algoritmo multiarranque que, al igual que BMB, empieza generando una solución inicial aleatoriamente y aplicando sobre ella Búsqueda Local con un máximo de 10000 evaluaciones. La diferencia con BMB se encuentra en el resto de iteraciones, pues en vez de generar aleatoriamente otra solución, aplicamos un operador de mutación sobre la mejor solución encontrada hasta el momento, y luego una Búsqueda Local. Si la solución resultante es mejor que la mejor solución encontrada hasta el momento, entonces actualizamos la mejor solución. Por último, aplicamos el criterio del mejor como criterio de aceptación, es decir, usamos la mejor solución como aquella a la que aplicar mutación y Búsqueda Local en la siguiente iteración.

Algorithm 7: ILS

Data: matrix : matriz de distancias,
m : número de elementos que una solución debe tener seleccionados para ser factible

begin

///// Inicializo constantes /////

$ITERATIONS \leftarrow 10$
$MAX\_EVALUATIONS\_LS \leftarrow 10000$
$ELEMENTS\_TO\_MUTATE \leftarrow m/10$

///// Genero solución aleatoria /////

$generateRandomSolution(sol, n, m)$
$evaluateFitness(sol, matrix)$
$localSearch(sol, MAX\_EVALUATIONS\_LS, matrix)$
$best\_sol \leftarrow sol$

///// Aplico ILS /////

for $i \in \{1, \ldots, ITERATIONS - 1\}$ **do**
  $mutation(sol, ELEMENTS\_TO\_MUTATE, matrix)$
  $localSearch(sol, MAX\_EVALUATIONS\_LS, matrix)$
  if $sol.fitnesss > best\_sol.fitness$ **then**
    $best\_sol \leftarrow sol$
  else
    $sol \leftarrow best\_sol$
  end
end

end

El operador de mutación simplemente aplica el operador de intercambio $t = 0.1 \cdot m$ veces sobre elementos distintos de los conjuntos de elementos seleccionados y no seleccionados. El objetivo de ello es provocar un cambio brusco en la solución actual que luego pueda ser aprovechado por Búsqueda Local. Para ello, barajo ambos conjuntos de elementos e intercambio las primeras t posiciones de ambos.

Algorithm 8: mutation

Data: sol : solución sobre la que se aplica búsqueda local,
elements_to_mutate : número de elementos a mutar,
matrix : matriz de distancias

begin

///// Barajo los vectores de elementos seleccionados y no seleccionados /////

$random\_shuffle(sol.selected)$
$random\_shuffle(sol.not\_selected)$

///// Muto los primeros elementos de ambos conjuntos y actualizo el fitness /////

for $i \in \{0, \ldots, elements\_to\_mutate\}$ **do**
  $sol.fitness \leftarrow (contribution(sol, sol.not\_selected[i], matrix)$
    $-matrix[sol.selected[i]][sol.not\_selected[i]])$
    $-contribution(sol, sol.selected[i], matrix)$
  $swap(sol.selected[i], sol.not\_selected[i])$
end

end

El algoritmo ILS-ES es similar a ILS, con la única diferencia de que se sustituye el algoritmo de Búsqueda Local empleado hasta ahora por el de Enfriamiento Simulado.

4. Procedimiento de desarrollo de la práctica

Cada algoritmo ha sido implementado en un fichero independiente que incluye todas las funciones que necesita. Los códigos fuentes se encuentran en el directorio `src`. Las 30 instancias proporcionadas para este problema se encuentran en el directorio `input`, (por motivos de espacio, en la entrega de la práctica este directorio está

inicialmente vacío, pero si se quiere ejecutar los algoritmos sobre dichas instancias solo hay que introducirlas en él). Además, el proyecto también dispone de un directorio `bin` para los binarios y otro llamado `output` para guardar los resultados devueltos por los algoritmos. Todos estos directorios se encuentran dentro del directorio `software`.

Para automatizar todo el proceso, he creado un script en `bash` llamado `executeAlgorithms.sh` y un archivo `makefile`. El primero ejecuta todos los algoritmos con cada una de las 30 instancias del problema y guarda los resultados obtenidos por cada algoritmo en un fichero `.dat` dentro del directorio `output`. Por otro lado, el archivo `makefile` dispone de los siguientes comandos:

- `bin/<algoritmo>`: compila el código fuente con el algoritmo especificado (ES, BMB, ILS, ILS-ES)
- `example<algoritmo>`: compila el algoritmo especificado y lo ejecuta con tres instancias del problema con distintos tamaños, mostrando los resultados por la terminal
- `compile_all`: compila todos los algoritmos
- `all`: compila todos los algoritmos y ejecuta el script `executeAlgorithms.sh`
- `clean`: elimina el contenido de los directorios `bin` y `output`

El ordenador en el que se han realizado los experimentos tiene 13.7 GiB de memoria RAM y un procesador AMD Ryzen 7 3700u. El sistema operativo que tiene instalado es Ubuntu 20.04.2.

5. Experimentos y análisis de resultados

5.1. Experimentos

Para evaluar el rendimiento de cada algoritmo y compararlos entre sí, los ejecutamos sobre los 30 casos proporcionados. Estos casos se pueden clasificar en 3 grupos en función del comienzo del nombre del fichero que los contiene. Cada uno de estos grupos está formado por 10 casos y tienen las siguientes características:

- MDG-a: $n = 500$, $m = 50$, distancias racionales
- MDG-b: $n = 2000$, $m = 200$, distancias racionales
- MDG-c: $n = 3000$, $m \in \{300, 400, 500, 600\}$, distancias enteras

Como podemos ver, con cada letra los valores de n y m aumentan, luego también es esperable que aumente el tiempo de ejecución.

En cuanto a las semillas utilizadas, he utilizado la sentencia `srand(1)` para fijar la semilla en todos los algoritmos.

5.2. Resultados

Podemos ver los tiempos y las desviaciones obtenidas por cada algoritmo en las tablas 1 (ES), 2 (BMB), 3 (ILS) y 4 (ILS-ES). Por último, en la tabla 5 se recogen los resultados medios de desviación y tiempo para todos los algoritmos considerados.

ES		
Caso	**Desv**	**Tiempo**
MDG-a_1_n500_m50	4.42	0.001257
MDG-a_2_n500_m50	3.58	0.002094
MDG-a_3_n500_m50	4.78	0.001636
MDG-a_4_n500_m50	4.11	0.001815
MDG-a_5_n500_m50	4.20	0.001338
MDG-a_6_n500_m50	2.50	0.001992
MDG-a_7_n500_m50	5.06	0.000904
MDG-a_8_n500_m50	2.13	0.002258
MDG-a_9_n500_m50	2.60	0.002248
MDG-a_10_n500_m50	3.19	0.001560
MDG-b_21_n2000_m200	1.12	0.301306
MDG-b_22_n2000_m200	1.36	0.247538
MDG-b_23_n2000_m200	1.85	0.132758
MDG-b_24_n2000_m200	1.72	0.109667
MDG-b_25_n2000_m200	1.48	0.138355
MDG-b_26_n2000_m200	1.39	0.175318
MDG-b_27_n2000_m200	1.36	0.150594
MDG-b_28_n2000_m200	1.44	0.162471
MDG-b_29_n2000_m200	1.66	0.145042
MDG-b_30_n2000_m200	1.50	0.142441
MDG-c_1_n3000_m300	1.06	0.336533
MDG-c_2_n3000_m300	1.12	0.464051
MDG-c_8_n3000_m400	0.83	0.613833
MDG-c_9_n3000_m400	1.07	0.608590
MDG-c_10_n3000_m400	0.84	0.612016
MDG-c_13_n3000_m500	0.79	0.755266
MDG-c_14_n3000_m500	0.54	0.746921
MDG-c_15_n3000_m500	0.68	0.745031
MDG-c_19_n3000_m600	0.58	0.867780
MDG-c_20_n3000_m600	0.60	0.861054

Cuadro 1: ES

BMB		
Caso	**Desv**	**Tiempo**
MDG-a_1_n500_m50	4.49	0.013815
MDG-a_2_n500_m50	3.89	0.013750
MDG-a_3_n500_m50	4.06	0.013822
MDG-a_4_n500_m50	3.84	0.013881
MDG-a_5_n500_m50	3.83	0.013729
MDG-a_6_n500_m50	3.49	0.013758
MDG-a_7_n500_m50	3.92	0.013730
MDG-a_8_n500_m50	3.18	0.014093
MDG-a_9_n500_m50	3.16	0.013897
MDG-a_10_n500_m50	3.90	0.013568
MDG-b_21_n2000_m200	7.73	0.366589
MDG-b_22_n2000_m200	7.61	0.405210
MDG-b_23_n2000_m200	7.56	0.383877
MDG-b_24_n2000_m200	7.81	0.388696
MDG-b_25_n2000_m200	7.57	0.411305
MDG-b_26_n2000_m200	7.62	0.400185
MDG-b_27_n2000_m200	7.82	0.389223
MDG-b_28_n2000_m200	7.58	0.331120
MDG-b_29_n2000_m200	7.54	0.325514
MDG-b_30_n2000_m200	7.68	0.389257
MDG-c_1_n3000_m300	7.13	2.029940
MDG-c_2_n3000_m300	7.30	1.891329
MDG-c_8_n3000_m400	6.24	3.378783
MDG-c_9_n3000_m400	6.25	3.533600
MDG-c_10_n3000_m400	6.21	3.695669
MDG-c_13_n3000_m500	5.24	5.042132
MDG-c_14_n3000_m500	5.35	5.070244
MDG-c_15_n3000_m500	5.18	5.318703
MDG-c_19_n3000_m600	4.88	6.970146
MDG-c_20_n3000_m600	4.84	6.646882

Cuadro 2: BMB

ILS		
Caso	**Desv**	**Tiempo**
MDG-a_1_n500_m50	2.57	0.006191
MDG-a_2_n500_m50	2.45	0.006330
MDG-a_3_n500_m50	1.94	0.006378
MDG-a_4_n500_m50	1.66	0.006257
MDG-a_5_n500_m50	2.55	0.006336
MDG-a_6_n500_m50	1.20	0.006364
MDG-a_7_n500_m50	0.81	0.009534
MDG-a_8_n500_m50	1.45	0.006148
MDG-a_9_n500_m50	1.19	0.006588
MDG-a_10_n500_m50	1.74	0.006113
MDG-b_21_n2000_m200	1.53	0.292548
MDG-b_22_n2000_m200	1.87	0.339452
MDG-b_23_n2000_m200	1.81	0.325409
MDG-b_24_n2000_m200	1.68	0.342774
MDG-b_25_n2000_m200	1.64	0.319158
MDG-b_26_n2000_m200	1.79	0.350028
MDG-b_27_n2000_m200	1.85	0.378004
MDG-b_28_n2000_m200	1.72	0.350357
MDG-b_29_n2000_m200	1.85	0.346172
MDG-b_30_n2000_m200	1.85	0.347050
MDG-c_1_n3000_m300	2.43	1.481418
MDG-c_2_n3000_m300	2.31	1.658405
MDG-c_8_n3000_m400	2.32	2.819906
MDG-c_9_n3000_m400	2.27	3.090409
MDG-c_10_n3000_m400	2.44	2.850241
MDG-c_13_n3000_m500	2.34	4.969077
MDG-c_14_n3000_m500	2.29	4.948011
MDG-c_15_n3000_m500	2.17	4.770283
MDG-c_19_n3000_m600	2.18	6.817747
MDG-c_20_n3000_m600	2.11	6.470598

Cuadro 3: ILS

ILS-ES		
Caso	**Desv**	**Tiempo**
MDG-a_1_n500_m50	2.33	0.014818
MDG-a_2_n500_m50	2.48	0.013207
MDG-a_3_n500_m50	1.50	0.014503
MDG-a_4_n500_m50	1.58	0.014606
MDG-a_5_n500_m50	2.10	0.016733
MDG-a_6_n500_m50	0.63	0.015642
MDG-a_7_n500_m50	2.65	0.013812
MDG-a_8_n500_m50	2.13	0.014266
MDG-a_9_n500_m50	1.73	0.016286
MDG-a_10_n500_m50	2.70	0.014100
MDG-b_21_n2000_m200	2.19	0.315643
MDG-b_22_n2000_m200	2.22	0.314147
MDG-b_23_n2000_m200	2.08	0.312742
MDG-b_24_n2000_m200	2.33	0.282180
MDG-b_25_n2000_m200	2.39	0.278828
MDG-b_26_n2000_m200	2.05	0.280177
MDG-b_27_n2000_m200	2.29	0.278543
MDG-b_28_n2000_m200	1.97	0.281728
MDG-b_29_n2000_m200	1.99	0.282365
MDG-b_30_n2000_m200	2.25	0.282592
MDG-c_1_n3000_m300	2.19	0.521579
MDG-c_2_n3000_m300	2.09	0.520619
MDG-c_8_n3000_m400	1.93	0.624625
MDG-c_9_n3000_m400	1.81	0.616829
MDG-c_10_n3000_m400	1.94	0.627544
MDG-c_13_n3000_m500	1.36	0.762844
MDG-c_14_n3000_m500	1.43	0.760389
MDG-c_15_n3000_m500	1.50	0.835893
MDG-c_19_n3000_m600	1.40	0.974397
MDG-c_20_n3000_m600	1.42	0.960059

Cuadro 4: ILS-ES

Algoritmo	**Global**		**Grupo a**		**Grupo b**		**Grupo c**	
	Desv	**Tiempo**	**Desv**	**Tiempo**	**Desv**	**Tiempo**	**Desv**	**Tiempo**
Greedy	9.20	0.525801	12.18	0.001005	9.05	0.385110	6.36	1.617179
BL	3.59	0.730803	1.82	0.004678	4.56	0.539808	4.40	2.226739
ES	1.98	0.277789	3.66	0.001710	1.49	0.243831	0.81	0.742380
BMB	5.76	1.583548	3.78	0.013804	7.65	0.993535	5.86	5.182482
ILS	1.93	1.444443	1.75	0.006624	1.76	0.839327	2.29	4.845195
ILS-ES	1.96	0.342057	1.98	0.014797	2.18	0.363324	1.71	0.791136

Cuadro 5: Resultados Globales

5.3. Análisis

5.3.1. Análisis por calidad

Empezamos viendo que los algoritmos que obtienen las mejores desviaciones son ILS, ILS-ES y ES en este orden, con desviaciones muy similares entre ellos (solo 5 centésimas de diferencia entre el primero y el tercero). Todos ellos se encuentran muy por encima de Búsqueda Local en términos de calidad, y considero que esto está provocado por el aumento de la diversidad. El esquema de enfriamiento de ES añade la posibilidad de avanzar hacia peores soluciones en las primeras iteraciones del algoritmo, lo cual permite reconducir la búsqueda hacia otras zonas a las que no se podría llegar únicamente mediante Búsqueda Local. Por otro lado, el mecanismo que tiene ILS para aumentar la diversidad es mutar la mejor solución encontrada hasta el momento justo antes de aplicarle BL o ES.

También es importante destacar que ILS-ES no aporta una gran mejora con respecto a ES, e incluso es ligeramente superado por ILS. Por tanto, parece que el hecho de combinar los mecanismos de aumento de diversidad aportados por ES e ILS no nos permite obtener un mejor algoritmo.

En cuanto a BMB, vemos que sus resultados son incluso peores que Búsqueda Local. A pesar de que BMB reduzca la dependencia con respecto a la solución inicial (lo cual puede verse como un aumento de la diversidad), creo que es superado por BL precisamente porque apenas es capaz de explotar las soluciones iniciales generadas en cada iteración. Esto se debe a que repartimos el número de evaluaciones totales entre las 10 iteraciones, por lo tanto queda un número demasiado reducido de evaluaciones para la Búsqueda Local que ejecutamos en cada iteración.

Si analizamos las desviaciones por grupos, podemos ver que en el *Grupo a* (instancias con menor tamaño), el algoritmo que obtiene los mejores resultados es ILS, seguido muy de cerca por BL e ILS-ES. No es extraño que BL sea el segundo mejor algoritmo, pues al estar ejecutándose sobre instancias de menor tamaño, los espacios de búsqueda no son muy grandes, luego la baja diversidad de BL no se ve tan penalizada. Sin embargo, los resultados de ILS son ligeramente mejores porque consigue introducir algo de diversidad con cada mutación sin que esto sea contraproducente para el algoritmo.

En el *Grupo b* (instancias con tamaño intermedio), comprobamos que ahora ES es el mejor algoritmo, seguido por ILS con resultados ligeramente peores. Considero que ES tiene un mejor desempeño que ILS porque, a pesar de que ambos sean algoritmos con mecanismos que aportan diversidad, ES consigue darle un mayor peso a ésta en las primeras iteraciones mientras que en las últimas da mayor prioridad a la explotación (se comporta casi igual que una Búsqueda Local). Sin embargo, ILS añade diversidad en todas las iteraciones mediante el mismo procedimiento de mutación, lo cual creo que puede llegar a ser incluso contraproducente en las últimas iteraciones.

Por último, para el *Grupo c* (instancias con mayor tamaño) vemos que las diferencias se acentúan aun más. Por lo ya comentado para el *Grupo b*, ES consigue los mejores resultados con diferencia. También podemos ver que el segundo mejor es ILS-ES, justo por delante de ILS, el tercero mejor.

5.3.2. Análisis por tiempos

El algoritmo que obtiene el mejor tiempo medio es ES. El segundo mejor es ILS-ES, con menos de una décima de diferencia. Sorprendentemente, Greedy solo llega a ser el tercero mejor con casi dos décimas más que ILS-ES. Por tanto, parece que globalmente los algoritmos que hacen uso de ES son los más rápidos.

Si nos restringimos al *Grupo a*, vemos que los mejores algoritmos son Greedy y ES, habiendo entre ellos menos de una milésima de diferencia.

Para el *Grupo b*, se tiene que ES es el mejor, sacando una décima de diferencia tanto a Greedy como a ILS-ES.

Por último, en el *Grupo c*, la rápidez de los algoritmos ES e ILS-ES es muy clara, pues ambos obtienen tiempo inferiores al segundo, además de que superan con casi un segundo de diferencia a Greedy, el tercero mejor.

5.3.3. Conclusiones

En términos de calidad, podemos concluir que ILS junto con BL son los mejores algoritmos en instancias de MDP con tamaños pequeños (*Grupo a*), mientras que para instancias con tamaños intermedios o grandes (*Grupo b* y *Grupo c*) el algoritmo que obtiene los mejores resultados es ES.

Atendiendo a los tiempos, ES es el algoritmo con los mejores resultados, obteniendo mayores diferencias con respecto al resto de algoritmos cuanto mayor es el tamaño de la instancia sobre la que se ejecutan. A pesar de ello, como para tamaños de instancia pequeños las diferencias en tiempos son del orden de milésimas, considero que en estos casos es mucho más recomendable usar ILS, o incluso BL.