Metaheurísticas: Práctica 3

Búsquedas por Trayectorias
para el Problema de la Máxima Diversidad

Pilar Navarro Ramírez - 76592479H
pilarnavarro@correo.ugr.es
Grupo 2: Viernes de 17:30 a 19:30

6 de junio de 2021

Índice

1. Descripción del problema

El **Problema de la Máxima Diversidad (Maximum Diversity Problem, MDP)**, es un problema NP-completo de optimización combinatoria. Consiste en seleccionar un subconjunto de m elementos de un conjunto inicial de n elementos (con $n > m$) de forma que se maximice la diversidad entre los elementos escogidos.

Además de esto, se dispone de una matriz $D = (d_{ij})$ de dimensión $n \times n$ que contiene las distancias entre todos los n elementos. Así, en la posición (i, j) de la matriz, se encuentra la distandia entre el elemento i-ésimo y el j-ésimo ($\forall i, j = 1, ..., n$), siendo $d_{ii} = 0 \; \forall i = 1, ..., n$. Por lo tanto, se trata de una matriz simétrica cuya diagonal está formada por ceros.

Existen varias formas de calcular la diversidad, pero la que nosotros usaremos consiste en calcular la suma de las distancias entre cada par de elementos de los m seleccionados.

El problema MDP se puede formular matemáticamente como sigue:

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

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

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

donde x es una solución al problema, esto es, es un vector binario de longitud n que indica los m elementos seleccionados, donde la posición i-ésima es 1 si se ha seleccionado el elemento i-ésimo.

2. Descripción de la aplicación de los algoritmos

Describimos aquí las consideraciones comunes a los algoritmos estudiados.

Todos los algoritmos parten de una matriz de distancias D de tamaño $n \times n$, como ya hemos comentado (que nosotros llamaremos simplemente *matriz* en nuestras implementaciones). Dicha matriz es construida leyendo las distancias de los ficheros de datos que se nos proporcionan en cada caso (de lo cual se encarga la función readInput). Se considera como entrada además el número de elementos a seleccionar m, también indicado en cada fichero.

2.1. Representación de la soluciones

Una solución vendrá dada como un contenedor de enteros que contiene los m elementos seleccionados, en vez de un vector binario como se indica en la descripción del problema. Esta última representación es menos eficiente, pues hay que tener en cuenta n elementos con sus distancias en vez de m a la hora de calcular la bondad de la solución (*fitness*), así como en cualquier otra operación que involucre recorrer la solución completa.

En el caso del algoritmo Greedy una solución será un conjunto (set) de enteros correspondientes a los elementos elegidos, que pueden tomar los valores de entre 1 y n, sin aparecer ninguno de ellos repetido. El tamaño de este conjunto será de m. Se usa aquí esta estructura de datos por ser el número de operaciones de consulta en la implementación del algoritmo muy pequeño en comparación con el número de operaciones de inserción y borrado, como veremos en la siguiente sección.

Para el algoritmo de la búsqueda local y los demás algoritmos basados en trayectorias, tomamos un vector de enteros en lugar de un conjunto (por realizarse un mayor número de operaciones de consulta en este algoritmo que en greedy) cumpliendo exactamente las mismas condiciones que el conjunto (elementos no repetidos, tamaño m, enteros con valores entre 1 y n), junto con el valor de fitness asociado a la solución. Concretamente, consideramos la siguiente estructura:

```
struct solucion {
        vector<int> elements;
        double fitness;
};
```

para la cual se ha sobrecargado el operador de asignación, de manera que al asignar una solución a otra lo que se hace es llamar al operador de asignación de cada una de las componentes del **struct**.

Aunque en un vector y en un conjunto los elementos aparecen ordenados, cabe mencionar que nosotros no tendremos en cuenta este orden, es decir, dos conjuntos o vectores con los mismos enteros pero en distinto orden son considerados la misma solución.

2.2. Contribución de un elemento

Definimos una función **contribution**, que calcula la contribución de un determinado elemento al coste de la solución que se le pasa como parámetro. Esto es, suma las distancias de ese elemento a cada uno de los elementos que se encuentran en la solución indicada, la cual puede ser un conjunto o un vector de enteros, como ya hemos comentado.

El elemento para el cual se quiere calcular la contribución puede formar parte o no del conjunto solución. En caso de que el elemento se encuentre en dicho conjunto determina la contribución de ese elemento a la solución. Si no forma parte, esta función permite saber cómo contribuiría el elemento en caso de estar incluido en la misma.

El pseudocódigo de esta función es el siguiente:

Algorithm 1: CONTRIBUTION

```
Input: conjunto de enteros, matriz de distancias, entero element
Output: contribucion del entero element en conjunto
begin
    sum ← 0
    for i in conjunto do
        sum ← sum + matriz[ element, i ]
    end
    return sum
end
```

2.3. Función objetivo

Como ya explicamos en el punto anterior, la función objetivo a maximizar en este problema es

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

que está definida en la función fitness , cuyo pesudocódigo es el siguiente:

Algorithm 2: FITNESS

```
Input: conjunto de enteros, matriz de distancias
Output: valor de la función objetivo para la solución dada en conjunto
begin
    sum ← 0
    for it_1 = conjunto.begin to conjunto.end do
        for it_2 = it_1 to conjunto.end do
            sum ← sum + matriz[conjunto(it_1), conjunto(it_2)]
        end
    end
    return sum
end
```

Así, esta función permite evaluar la solución dada en *conjunto*, de manera que cuanto mayor sea el valor devuelto por esta función mejor será la solución. Como para la función contribution, el parámetro *conjunto* que contiene los enteros que determinan una solución, puede ser un conjunto/set o un vector, según si se usa en el algoritmo greedy o en el de la búsqueda local.

2.4. Generación de soluciones aleatorias

Todos los algoritmos estudiados en esta práctica (menos el algoritmo Greedy) hacen uso de soluciones generadas aleatoriamente. Por lo tanto, consideramos la siguiente función, que se encarga de generar una solución aleatoria válida y de su posterior evaluación:

Algorithm 3: RANDOMSOLUTION

```
Input: tamaño de la solución m, matriz de distancias
Output: solución válida del problema MDP junto con su fitness
begin
    sol ← ∅ ;                                   // Partimos de la solución vacía
    while |sol| < m do
        random ← elemento aleatorio de {0, ..., n − 1}
        if random ∉ sol then
            sol ← sol ∪ random ;          // Si el elemento aleatorio considerado
                                           // no está ya en la solución, se añade
        end
    end
    return sol, fitness(sol, matriz)
end
```

2.5. Determinación de los elementos no seleccionados

Dada una solución válida, es importante determinar los elementos que no pertenecen a esa solución, los cuales serán candidatos a ser elegidos para incluirlos en la solución a la hora de modificar la misma. La función validElements se encargará de esto:

Algorithm 4: VALIDELEMENTS

```
Input: vector de enteros seleccionados, sel
Input: número total de elementos del problema, n
Output: vector de enteros no seleccionados
begin
    no_sel ← ∅ ;              // Partimos del vector de no seleccionados vacío
    elem ← 0
    while |no_sel| < n − |sel| do
        if elem ∉ sel then
            // Si el elemento considerado en la iteración actual no se
            // encuentra en el conjunto de seleccionados, se añade
            // al vector de no seleccionados
            no_sel ← no_sel ∪ elem
        end
        elem ← elem + 1;
    end
    return no_sel
end
```

Esta función se usará en todos los algoritmos, menos en el caso de Greedy.

3. Descripción de los algoritmos

Pasamos ya a explicar los algoritmos implementados.

3.1. Algoritmo Greedy

Para este algoritmo consideramos dos conjuntos de elementos (enteros): el conjunto de los elementos que han sido seleccionados para formar parte de la solución, Sel, y el conjunto de elementos que no han sido seleccionados, $NoSel$.

El algoritmo empieza con el conjunto Sel vacío y añade a él en primer lugar el elemento más alejado al resto, esto es, aquel cuya suma de las distancias a todos los demás elementos es la mayor. Para determinar este elemento, nosotros hemos implementado la función furthestElement. Lo que hace es llamar a la función contribution (descrita en el apartado anterior) para cada uno de los elementos del problema y con un conjunto que contiene a todos los elementos, es decir, calcula la contribución de cada uno de los elementos a dicho conjunto total, y devuelve el elemento cuya contibución es la mayor.

Algorithm 5: FURTHESTELEMENT

```
Input: matriz de distancias
Output: elemento más alejado del resto, el de mayor contribución
begin
    NoSel ← {0, 1, ..., n − 1} ;                    // Inicializo el conjunto de no
                                                     // seleccionados a los n elementos del problema
    furthest ← -1
    max_sum_dist ← -1
    for i in NoSel do
        contrib ← contribution (NoSel, matriz, i )
        if contrib > max_sum_dist then
            max_sum_dist ← contrib
            furthest ← i
        end
    end
    return furthest
end
```

Una vez añadido a Sel el elemento más alejado a todos los demás, el algoritmo continúa introduciendo en cada iteración el elemento no seleccionado que está más alejado al conjunto de elementos seleccionados, hasta alcanzar el tamaño máximo que puede tener la solución, m. Definimos la distancia de un elemento a un conjunto como el mínimo de las distancias de ese elemento a los elementos del conjunto:

$$Dist(e, Sel) = \min_{s \in Sel} d(s, e)$$

La función distanceToSet se encarga de calcular esta distancia:

Algorithm 6: DISTANCETOSET

```
Input: conjunto de enteros, matriz de distancias, entero element
Output: distancia de element a conjunto
begin
    min_dist ← ∞
    for i in conjunto do
        dist ← matriz[element,i]
        if dist < min_dist then
            min_dist ← dist
        end
    end
    return min_dist
end
```

Para determinar en cada iteración del algoritmo cuál es el elemento no seleccionado más alejado de Sel, en el sentido de que maximiza la distancia a dicho conjunto, hacemos uso de la función furthestToSel, cuyo pseudocódigo se muestra a continuación:

Algorithm 7: FURTHESTTOSEL

```
Input: conjunto de enteros seleccionados Sel
Input: conjunto de enteros no seleccionados NoSel
Input: matriz de distancias
Output: elemento de NoSel más alejado de Sel
begin
    furthest ← -1
    max_dist ← -1
    for i in NoSel do
        dist ← distanceToSet(Sel, matriz, i)
        if dist > max_dist then
            max_dist ← dist
            furthest ← i
        end
    end
    return furthest
end
```

Podemos ya ver el pseudo-código del algoritmo Greedy completo, donde se hace uso de las

funciones anteriores en el sentido que hemos ido explicando:

Algorithm 8: GREEDY

Input: *matriz* de distancias, tamaño de la solución m
Output: solución válida del problema MDP junto con su fitness
begin
  $NoSel \leftarrow \{0, 1, \ldots, n-1\}$
  $Sel \leftarrow \emptyset$
  furthest $\leftarrow$ furthestElement($matriz$)
  $Sel \leftarrow Sel \cup \{furthest\}$
  $NoSel \leftarrow NoSel \backslash \{furthest\}$
  while $|Sel| < m$ **do**
    furthest $\leftarrow$ furthestToSel($Sel, NoSel, matriz$)
    $Sel \leftarrow Sel \cup \{furthest\}$
    $NoSel \leftarrow NoSel \backslash \{furthest\}$
  end
  return Sel, fitness($Sel, matriz$)
end

3.2. Búsqueda Local del Primer Mejor

Este algoritmo parte de una solución generada aleatoriamente y en cada iteración se generan soluciones del entorno (soluciones vecinas) hasta que se encuentra una que es mejor que la actual, la cual es entonces sustituida por la nueva solución generada. El algoritmo termina cuando se explora todo el vecindario y no se encuentra ninguna solución mejor o, para nuestro caso, cuando se han evaluado 100000 soluciones diferentes.

Para generar las soluciones vecinas, lo que se hace es escoger un elemento de la solución e intercambiarlo por otro elemento que no se encuentre en la misma, es decir, un elemento del conjunto devuelto por la función recién introducida. Se puede asegurar que este intercambio, cumpliendo las condiciones descritas, da siempre lugar a una solución válida.

Una solución vecina será aceptada si mejora a la solución actual, en otro caso se rechaza y se genera otra solución. La función improvement se encarga de hacer esto. Es decir, determina si un cierto intercambio en la solución produce una mejora o no y en caso afirmativo actualiza la solución cambiando el elemento viejo por el nuevo y calculando el fitness de la nueva solución. Este cálculo resulta más eficiente si se factoriza, esto es, en vez de volver a considerar las distancias entre todos los elementos de la nueva solución, basta con sustraer del fitness antiguo la contribución del elemento eliminado y añadirle la contribución del nuevo elemento a la solución.

Para que se produzca una mejora, se debe cumplir que el nuevo elemento introducido tenga una mayor contribución a la solución que el elemento eliminado. Así, no hay que calcular la bondad de la nueva solución para compararla con la antigua, sino que es suficiente con determinar la contribución del elemento nuevo a la solución y compararla con la del elemento anterior.

Veamos ya el pseudo-código que lleva a cabo todas estas consideraciones:

Algorithm 9: IMPROVEMENT

Input: sol: solución
Input: pos: posición de *sol* cuyo elemento se va a cambiar
Input: *old_cont*: contribución a la solución del elemento que se encuentra en *pos*
Input: elem: nuevo elemento que se va a introducir en la posición *pos* de *sol*
Input: matriz: matriz de distancias
Output: mejora: booleano que indica si la solución mejora o no
Output: sol: nueva solución si se produce mejora o la antigua si no se mejora
begin
    mejora ← false
    `// Solución auxiliar que es copia de la solución considerada`
    $nueva \leftarrow sol$
    `// Elemento de la solución que se va a intercambiar`
    $old\_elem \leftarrow sol[pos]$
    $nueva[pos] \leftarrow elem$
    `// Contribución del nuevo elemento a la solución actualizada`
    $new\_cont \leftarrow \text{contribution}(nueva, matriz, elem)$
    `// Si la contribución del nuevo elemento es mayor que la del antiguo,`
    `   se produce mejora y se actualiza la solución`
    if $new\_cont > old\_cont$ **then**
        `// Factorización de la función objetivo`
        $nueva.fitness \leftarrow sol.fitness - old\_cont + new\_cont$
        $sol \leftarrow nueva$
        mejora ← true
    end
    return mejora, *sol*
end

El elemento a intercambiar de la solución no se escoge de manera aleatoria, sino que se lleva a cabo una exploración inteligente del entorno de soluciones, enfocándonos en zonas donde se pueden obtener soluciones mejores. Concretamente, lo que se hace es calcular la contribución de cada elemento de la solución a la bondad de la misma, y seleccionar para intercambiar el elemento que menos contribuye. La función lowestContribution se encarga de esto:

Algorithm 10: LOWESTCONTRIBUTION

```
Input: sol: vector de enteros que determinan una solución
Input: matriz: matriz de distancias
Output: pos_min: posición en la solución sol del elemento que menos contribuye
Output: min_contrib: contribución del elemento que menos contribuye
begin
    pos_min ← −1
    min_contrib ← ∞
    for i in indices of sol do
        cont ← contribution(sol, matriz, sol[i])
        if cont < min_contrib then
            pos_min ← i
            min_contrib ← cont
        end
    end
    return pos_min, min_contrib
end
```

Sólo nos queda un detalle del algoritmo por explicar y es qué elemento de entre los no seleccionados se introduce en la posición del elemento que menos contribuye para generar una solución vecina. Pues en este caso sí es totalmente aleatorio. Por ello, lo que hacemos es barajar en cada iteración el conjunto de elementos que no forman parte de la solución.

Presentamos finalmente el algoritmo de la búsqueda local, que hace uso de todas estas

funciones explicadas:

Algorithm 11: localSearch

```
Input: m: tamaño de solución
Input: matriz: matriz de distancias
Output: solución válida del problema MDP junto con su fitness
begin
    num_eval ← 0
    mejora ← true
    // Empezamos con una solución aleatoria
    sol ← randomSolution(m, matriz)
    // Elementos válidos para el intercambio
    valid_elements ← validElements(sol, matriz.size)
    // Elemento que menos contribuye y su contribución
    min_contrib ← lowestContribution(sol, matriz)
    // Iteramos mientras la solución mejore y no se haya superado el
       número máximo de evaluaciones de la función objetivo
    while mejora and num_eval < 100000 do
        mejora ← false
        // min_contrib contiene tanto la posición como la contribución del
           elemento que menos contribuye
        min_pos ← min_contrib.pos
        // Guardamos el elemento antiguo que vamos a cambiar
        old_elem ← sol[min_pos]
        shuffle(valid_elements)
        // /Intercambiamos el elemento que menos contribuye por todos los
           posibles hasta que se produzca una mejora
        for k ∈ valid_elements and mejora is false and num_eval < 100000 do
            mejora ← improvement(sol, min_pos, min_contrib.contrib, k, matriz)
            num_eval ← num_eval + 1
        end
        if mejora then
            // Actualizamos los elementos válidos, cambiando el elemento
               nuevo por el antiguo
            valid_elements ← valid_elements\{k}
            valid_elements ← valid_elements ∪ {old_elem}
            // Determinamos el elemento que menos contribuye en la nueva
               solución
            min_contrib ← lowestContribution(sol, matriz)
        end
    end
    return sol.elements, sol.fitness
end
```

3.3. Enfriamiento Simulado

El algoritmo de Enfriamiento Simulado (Simulated Annealing) es un método de búsqueda por entornos que usa un criterio de aceptación de soluciones vecinas que va cambiando a lo largo de su ejecución, para evitar que la búsqueda se quede atrapada en un óptimo local, permitiendo que se acepten soluciones peores. En concreto, controla la frecuencia de los movimientos de escape de *óptimos locales* mediante una función de probabilidad que hará disminuir la probabi-

lidad de estos movimientos hacia soluciones peores conforme avanza la búsqueda. Así, se aplica la filosofía de diversificar al principio e intensificar al final.

Este algoritmo hace uso de una variable llamada Temperatura, T, cuyo valor determina en qué medida pueden ser aceptadas soluciones vecinas peores que la actual. Dicha variable se inicializa a un valor alto, denominado Temperatura inicial, T_0, y se va reduciendo en cada iteración mediante un mecanismo de enfriamiento de la temperatura hasta alcanzar una Temperatura final, T_f.

Establecemos el valor de la temperatura inicial a

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

donde $C(S_0)$ es el coste de la solución inicial y $\phi \in [0, 1]$ es la probabilidad de aceptar una solución un μ por 1 peor que la inicial. Así, la temperatura inicial dependerá de la solución aleatoria de la que parte la ejecución.

Consideramos $\mu = \phi = 0{,}3$ y $T_f = 10^{-3}$.

Para la modificación del valor de la temperatura hacemos uno del esquema de Cauchy modificado,

$$T_{k+1} = \frac{T_k}{1 + \beta_k}; \beta_k = \frac{T_0 - T_f}{M T_0 T_f}$$

donde T_k es el valor de la temperatura en la iteración k-ésima del algoritmo y M es el número de enfriamientos (o iteraciones) que se realizarán.

En cada iteración del algoritmo se genera un número concreto de vecinos, que podrá ser como máximo $max\_vecinos = 0{,}1n$. Por lo tanto, debemos implementar un operador de generación de vecinos. Este se encargará de, dada una solución factible, seleccionar aleatoriamente un elemento de la misma y cambiarlo por otro que no forme parte de ella, esto es, algún elemento aleatorio de los devueltos por validElements. El pseudocódigo de esta función es el siguiente:

Algorithm 12: changeSolution

Input: sol: solución, n:número total de elementos del problema
Input: matriz: matriz de distancias
Output: solución vecina a la original
begin
    // Posición aleatoria de sol
    pos1 ← elemento aleatorio en $\{1, ..., m\}$ // m es la longitud de una solución
    // Elemento de la solución que se va a intercambiar
    $old\_elem \leftarrow sol[pos]$
    // Contribución del elemento a cambiar en la solución
    $old\_cont \leftarrow$ contribution$(sol, matriz, old_elem)$
    // Conjunto de elementos válidos para añadir a la solución
    $valid \leftarrow$ validElements(sol, n)
    // Posición aleatoria del vector de elementos válidos
    pos2 ← elemento aleatorio en $\{1, ..., n - m\}$
    // Elemento no seleccionado a introducir en la nueva solución
    $new\_elem \leftarrow valid[pos2]$
    $sol[pos1] \leftarrow new\_elem$
    // Contribución del nuevo elemento a la solución actualizada
    $new\_cont \leftarrow$ contribution$(sol, matriz, new_elem)$
    // Caculamos el fitness de la nueva solución de manera factorizada
    $sol.fitness \leftarrow sol.fitness - old\_cont + new\_cont$
    return *sol*
end

Tras cada generación de una solución vecina se aplica el criterio de aceptación para ver si la solución generada sustituye a la actual. Si la solución generada presenta mayor valor de fitness que la actual se acepta automáticamente. En caso contrario, la nueva solución se aceptará con una probabilidad de $e^{\frac{\Delta f}{T}}$, siendo Δf la diferencia de coste entre la solución nueva y la anterior. Esto permite, como ya hemos comentado, que el algoritmo no se quede atrapado en óptimos locales y pueda salir de estos.

Notamos que $e^{\frac{\Delta f}{T}}$ depende tanto de la diferencia de coste entre las soluciones como de la temperatura, de manera que a mayores temperaturas (primeras iteraciones) más probable es que se acepten soluciones peores y, del mismo modo, a menor diferencia de fitness entre las soluciones, hay mayor probabilidad de aceptación.

Tras generar el máximo número de vecinos permitidos en una iteración o tras aceptar un determinado número de soluciones, que será $max\_exitos = 0{,}1 * max\_vecinos$, se enfría la temperatura según el esquema de Cauchy y se pasa a la siguiente iteración.

Este proceso se repite hasta que se hayan realizado 100000 evaluaciones de la función objetivo o hasta que el número de soluciones aceptadas en una iteración sea 0, en cuyo caso se habrá encontrado una solución suficientemenete buena o el algoritmos se habrá quedado atrapado en un óptimo local del cual no es capaz de salir y no tendría sentido seguir iterando.

Así, el pseudocódigo final de este algoritmo queda como sigue:

Algorithm 13: ES

Input: matriz: matriz de distancias, m:longitud de una solución
Input: n:número total de elementos del problema
Output: solución válida del problema MDP junto con su fitness

```
num_eval ← 1                    // Número de evaluaciones de la función objetivo
                                // Empieza siendo 1 para contar la solución aleatoria inicial
μ ← 0,3
φ ← 0,3
max_vecinos ← 10n
max_exitos ← 0,1 * max_vecinos
M ← 100000/max_vecinos
T_f ← 10^-3
exitos ← 1                      // Para que entre la primera vez en el bucle
sol ← randomSolution(m, matriz)
T_0 ← −μ · sol.fitness/ log φ   // Temperatura inicial
beta ← (T_0 − T_f)/(M · T · T_f)
// Si la temperatura final es mayor que la final, disminuimos la última
while T_f > T_0 do
    T_f ← T_f/10
end
T ← T_0
best_sol ← sol
while exitos > 0 and num_eval < 100000 do
    exitos ← 0                  // Número de soluciones aceptadas
    neighbors ← 0               // Número de vecinos generados
    while neighbors < max_vecinos and exitos < max_exitos and num_eval < 100000
     do
        aux ← sol               // Solución auxiliar
        // Se genera una solución vecina a la actual
        aux← changeSolution(aux, n, matriz)
        neighbors ← neighbors + 1
        num_eval ← num_eval + 1
        Δf ← aux.fitness − sol.fitness
        random ← número aleatorio en (0, 1)
        if Δf > 0 or random ≤ exp(Δf/T) then
            exitos ← exitos + 1
            sol ← aux           // Se acepta la solución vecina generada
            if sol.fitness > best_sol.fitness then
                best_sol ← sol
            end
        end
    end
    T ← T/(1 + beta · T)        // Se enfría la temperatura
end
return best_sol
```

3.4. Búsqueda Multiarranque básica

Este algoritmo consiste en aplicar la búsqueda local a varias soluciones generadas aleatoriamente, guardando y devolviendo la mejor de las soluciones obtenidas. En concreto, en nuestro algoritmo se generarán 10 soluciones aleatorias, a las cuales se les aplicará la búsqueda local durante un máximo de 10000 evaluaciones de la función objetivo.

Consideramos unas ligeras modificaciones en el pseudo-código de la búsqueda local, presentado en el apartado correspondiente, pues ahora esta no partirá de una solución aleatoria cualquiera, sino de la solución que se le proporcione como parámetro. Además, debemos establecer el límite de evaluaciones a 10000, en vez de 100000 como teníamos antes:

Algorithm 14: localSearch

```
Input: sol: solución de partida
Input: matriz: matriz de distancias
Output: solución válida del problema MDP junto con su fitness
begin
    num_eval ← 0
    mejora ← true
    // Elementos válidos para el intercambio
    valid_elements ← validElements(sol, matriz.size)
    // Elemento que menos contribuye y su contribución
    min_contrib ← lowestContribution(sol, matriz)
    // Iteramos mientras la solución mejore y no se haya superado el
       número máximo de evaluaciones de la función objetivo
    while mejora and num_eval < 10000 do
        mejora ← false
        // min_contrib contiene tanto la posición como la contribución del
           elemento que menos contribuye
        min_pos ← min_contrib.pos
        // Guardamos el elemento antiguo que vamos a cambiar
        old_elem ← sol[min_pos]
        shuffle(valid_elements)
        // /Intercambiamos el elemento que menos contribuye por todos los
           posibles hasta que se produzca una mejora
        for k ∈ valid_elements and mejora is false and num_eval < 10000 do
            mejora ← improvement(sol, min_pos, min_contrib.contrib, k, matriz)
            num_eval ← num_eval + 1
        end
        if mejora then
            // Actualizamos los elementos válidos, cambiando el elemento
               nuevo por el antiguo
            valid_elements ← valid_elements\{k}
            valid_elements ← valid_elements ∪ {old_elem}
            // Determinamos el elemento que menos contribuye en la nueva
               solución
            min_contrib ← lowestContribution(sol, matriz)
        end
    end
    return sol.elements, sol.fitness
end
```

Podemos ya ver el pseudo-código de la BMB:

Algorithm 15: BMB

Input: matriz: matriz de distancias
Input: m: tamaño de una solución
Output: solución válida del problema MDP
$best\_sol.fitness \leftarrow -1$
for $i \in \{0, \ldots, 9\}$ **do**
    $sol \leftarrow$ randomSolution$(m, matriz)$
    $sol \leftarrow$ localSearch$(sol, matriz)$
    if $sol.fitness > best\_sol.fitness$ **then**
        $best\_sol \leftarrow sol$
    end
end
return $best\_sol$

3.5. Iterative Local Search

Este algoritmo está basado en la aplicación repetida de la búsqueda local a una solución que se obtiene mutando un óptimo local previamente encontrado. Se parte de una solución generada aleatoriamente, a la cual se le aplica la búsqueda local para mejorarla. A continuación se muta la mejor solución encontrada hasta el momento, y también se le aplica BL. Este último paso se repite durante 9 iteraciones, almacenando y devolviendo la mejor solución encontrada en todas ellas.

El operador de mutación de una solución consiste en cambiar aleatoriamente el 10 % de los elementos de la misma por otros elementos que no formen parte de dicha solución. Así, el pseudocódigo de dicho operador queda como sigue:

Algorithm 16: MUTATE

Input: Solución a mutar, *sol*, número de elementos a mutar, *num_mut*, *matriz* de distancias
Output: Solución *sol* modificada

```
// Posiciones de la solución cuyos elementos se van a cambiar
to_change ← ∅
elems ← ∅                    // Nuevos elementos a introducir en la solución
do
pos ← elemento aleatorio de {1, ..., m}          // m es el tamaño de la solución
// Si la posición no está ya seleccionada
if pos ∉ to_change then
    // Se añade al conjunto de posiciones a cambiar
    to_change ← to_change ∪ {pos}
end
while to_change.size < num_mut
valid ← validElements(sol, matriz.size)
do
pos ← elemento aleatorio de {1, ..., n − m}
// Si el elemento no está ya seleccionado
if valid[pos] ∉ elems then
    elems ← elems ∪ {valid[pos]}   // Se añade al conjunto de nuevos elementos
end
while elems.size < num_mut
// Cambiamos los elementos de la solución
for i ∈ {0, ..., num_mut − 1} do
    sol[to_change[i]] ← elems[i]
end
// Actualizamos el fitness de la solución
sol.fitness ← fitness(sol, matriz)
return sol
```

Y el pseudo-código principal de ILS es el siguiente:

Algorithm 17: ILS

Input: *matriz* de distancias, tamaño de una solución m
Output: Solución válida del problema MDP

```
num_mut ← 0,1 · m
best_sol ← randomSolution(m, matriz)
best_sol ← localSearch(best_sol, matriz)
for i ∈ {0, ..,9} do
    sol ← best_sol
    sol ← mutate(num_mut, sol, matriz)
    sol ← localSearch(sol, matriz)
    if sol.fitness > best_sol.fitness then
        best_sol ← sol
    end
end
return best_sol
```

donde localSearch es la modificación de la BL cuyo pseudo-código aparece en el apartado

anterior (de BMB).

3.6. Algoritmo Híbrido ILS-ES

Este algoritmo presenta exactamente el mismo funcionamiento que el anterior salvo que, en lugar de la búsqueda local, usa el enfriamiento simulado para mejorar las soluciones.

Consideramos una ligera modificación en el pseudo-código del ES presentado en el apartado correspondiente, pues ahora este no partirá de una solución aleatoria, sino de la solución que se le pase como parámetro. Además, el número máximo de evaluaciones permitido de la función objetivo será 10000, en vez de 100000:

Algorithm 18: ES

Input: sol: solución de partida
Input: matriz: matriz de distancias, m:longitud de una solución
Input: n:número total de elementos del problema
Output: solución válida del problema MDP junto con su fitness

$num\_eval \leftarrow 0$ `// Número de evaluaciones de la función objetivo`
$\mu \leftarrow 0{,}3$
$\phi \leftarrow 0{,}3$
$max\_vecinos \leftarrow 10n$
$max\_exitos \leftarrow 0{,}1 * max\_vecinos$
$M \leftarrow 10000/max\_vecinos$
$T_f \leftarrow 10^{-3}$
$exitos \leftarrow 1$ `// Para que entre la primera vez en el bucle`
$T_0 \leftarrow -\mu \cdot sol.fitness/\log\phi$ `// Temperatura inicial`
$beta \leftarrow (T_0 - T_f)/(M \cdot T \cdot T_f)$
`// Si la temperatura final es mayor que la final, disminuimos la última`
while $T_f > T_0$ **do**
  $T_f \leftarrow T_f/10$
end
$T \leftarrow T_0$
$best\_sol \leftarrow sol$
while $exitos > 0$ ***and*** $num\_eval < 10000$ **do**
  $exitos \leftarrow 0$ `// Número de soluciones aceptadas`
  $neighbors \leftarrow 0$ `// Número de vecinos generados`
  while $neighbors < max\_vecinos$ ***and*** $exitos < max\_exitos$ ***and*** $num\_eval < 10000$ **do**
    aux $\leftarrow$ sol `// Solución auxiliar`
    `// Se genera una solución vecina a la actual`
    aux$\leftarrow$ changeSolution$(aux, n, matriz)$
    $neighbors \leftarrow neighbors + 1$
    $num\_eval \leftarrow num\_eval + 1$
    $\Delta f \leftarrow aux.fitness - sol.fitness$
    random $\leftarrow$ número aleatorio en $(0, 1)$
    if $\Delta f > 0$ ***or*** $random \leq \exp(\Delta f/T)$ **then**
      $exitos \leftarrow exitos + 1$
      sol $\leftarrow$ aux `// Se acepta la solución vecina generada`
      if $sol.fitness > best\_sol.fitness$ **then**
        $best\_sol \leftarrow sol$
      end
    end
  end
  $T \leftarrow T/(1 + beta \cdot T)$ `// Se enfría la temperatura`
end
return $best\_sol$

Por tanto, el pseudo-código de ILS-ES es idéntico al del apartado anterior, salvo que hay que cambiar donde aparece localSearch por el ES que acabamos de presentar.

3.7. ES y ILS-ES con esquema de enfriamiento proporcional

Consideramos una modificación de los algoritmos de enfriamiento simulado y ILS-ES, en los que, para el enfriamiento de la temperatura, se usa un esquema proporcional en lugar del esquema de Cauchy modificado, esto es, tomamos $T_{k+1} = 0{,}9 \cdot T_k$ siendo T_k el valor de la temperatura en el enfiramiento k-ésimo.

Así, el pseudo-código para estos dos algoritmo será el mismo que el de sus algoritmos correspondientes cambiando el enfriamiento de la temperatura por $T \leftarrow 0{,}9 \cdot T$. Además, no es necesario calcular las variables β y μ.

3.8. ES versión 2

Este algoritmo será otra modificación del ES, en el que se toman los valores $\phi = 0{,}5$ y $\mu = 0{,}3$, esto es, se aceptan soluciones un 30 % peores que la acutal con una probabilidad del 50 %, de manera que se aumenta la diversificación.

3.9. ES versión 3

En este caso lo que hacemos es cambiar el máximo número de vecinos que se pueden generar en cada iteración, pasando de $10m$ a m. Así, el número de vecinos que se generan en una iteración se ve reducido, y el número de iteraciones que se llevarán a cabo es mayor.

3.10. ILS versión 2

Modificamos la búsqueda local iterativa para que se muten más soluciones y el límite de evaluaciones de la función objetivo en cada aplicación de la búsqueda local sea menor, de manera que se favorece la diversificación y se disminuye la intensificación. En concreto, se mutará la mejor solución encontrada hasta el momento 99 veces, y se aplicará la búsqueda local a la solución resultante durante un máximo de 1000 evaluaciones de la función objetivo. Cabe notar que la BL se aplica en total 99+1=100 veces contando la solución aleatoria inicial. Estudiaremos en el apartado de Análisis de los resultados cómo afecta esta modificación a los resultados.

4. Procedimiento para el desarrollo de la práctica

La implementación de todos los algoritmos ha sido llevada a cabo usando el lenguaje C++ y la librería STL, de la cual usamos los tipos de estructuras de datos **set** y **vector**, como ya hemos comentado. Además, se utilizan las siguientes funciones:

- clock de la librería time.h para medir el tiempo de ejecución
- shuffle y find de la librería algorithm
- rand, para generar números pseudo-aleatorios y srand, para fijar una semilla, de stdlib.h
- numeric_limits<**double**>::infinity() de la librería limits para inicializar los valores mínimos a infinito

4.1. Manual de usuario

Los ejecutables de cada uno de los algoritmos estudiados se encuentran en la carpeta **bin** del proyecto. Disponemos de los siguientes archivos:

- greedy → algoritmo greedy
- localSearch → algoritmo de búsqueda local del primer mejor
- ES → algoritmo de enfriamiento simulado
- ES-Prop → algoritmo de enfriamiento simulado con esquema proporcional
- ES2 → algoritmo de enfriamiento simulado segunda versión
- ES3 → algoritmo de enfriamiento simulado tercera versión
- BMB → algoritmo de la búsqueda multiarranque básica
- ILS → algoritmo de la búsqueda local iterativa
- ILS2 → algoritmo de la búsqueda local iterativa segunda versión
- ILS-ES → algoritmo de la búsqueda local iterativa con enfriamiento simulado
- ILS-ES-Prop → algoritmo de la búsqueda local iterativa con enfriamiento simulado y esquema proporcional

Todos muestran los resultados por pantalla en el formato: *Fitness, Tiempo de ejecución (s)* pero podemos redirigir la salida al fichero que queramos. Además, la semilla se le pasa como parámetro (en el caso de los algoritmos de la búsqueda local) y leen los datos de la entrada estándar. Así, para ejecutar el algoritmo de la búsqueda local del primer mejor con una semilla de 4, con el fichero de datos de entrada $MDG-a\_1\_n500\_m50.txt$ y con salida en el fichero *localSearch.csv*, basta con escribir en consola la siguiente sentencia:

```
bin/localSearch 4 < data/MDG-a_1_n500_m50.txt >> salida/localSearch.csv
```

Para el algoritmo Greedy la sentencia sería igual pero sin incorporar la semilla.

Para automatizar el proceso de ejecución de cada algoritmo sobre los distintos casos de estudio, se dispone del script execute.sh. La semilla se fija dentro de este archivo en la variable

semilla, por lo que para ejecutar los algoritmos con todos los ficheros de datos con una semilla diferente, solo hay que cambiar el valor de dicha variable y ejecutar el script.

Por otra parte, como era de esperar, el fichero makefile se encarga de la compilación. Al escribir en consola la orden make se compilan todos los ficheros de código fuente y se ejecuta el script execute.sh.

5. Experimentos y análisis de resultados

Los experimentos han sido realizados en el mismo ordenador, que tiene las siguientes características: sistema operativo Ubuntu 20.04.1 64 bits, procesador Intel Core i7-6500U 2.50GHz, memoria RAM 8GB DDR3 L.

Los resultados han sido obtenidos fijando la semilla:

7413

Los casos del problema considerados son 30, elegidos de los casos recopilados en la biblioteca **MDPLib**. Concretamente, se estudia el grupo de casos **MDG**, del que se han seleccionado las 10 primeras instancias del *tipo a* (matrices $n \times n$ con distancias enteras aleatorias en $\{0, 10\}$, n=500 y m=50), 10 instancias (entre la 21 y la 30) del *tipo b* (matrices $n \times n$ con distancias reales aleatorias en [0,1000], n=2000 y m=200) y otras 10 instancias (1,2,8,9,10,13,14,15,19,20) del *tipo c* (matrices $n \times n$ con distancias enteras aleatorias en $\{00, 1000\}$, n=3000 y $m = \{300, 400, 500, 600\}$).

5.1. Resultados obtenidos

Presentamos a continuación los valores de coste, desviación y tiempo de ejecución obtenidos para cada uno de los algoritmos considerados y para cada caso de estudio.

Algoritmo Greedy

Caso	Coste obtenido	Desv	Tiempo (s)
MDG-a_1_n500_m50	6865.94	12.36	0.00431
MDG-a_2_n500_m50	6754.02	13.09	0.004396
MDG-a_3_n500_m50	6741.6	13.12	0.004332
MDG-a_4_n500_m50	6841.59	11.95	0.004125
MDG-a_5_n500_m50	6740.34	13.09	0.004243
MDG-a_6_n500_m50	7013.94	9.77	0.004313
MDG-a_7_n500_m50	6637.46	14.59	0.004386
MDG-a_8_n500_m50	6946.28	10.38	0.004014
MDG-a_9_n500_m50	6898.01	11.22	0.004446
MDG-a_10_n500_m50	6853.68	11.91	0.00442
MDG-b_21_n2000_m200	10314568.35	8.72	0.450164
MDG-b_22_n2000_m200	10283328.5	8.89	0.448588
MDG-b_23_n2000_m200	10224214.16	9.52	0.444544
MDG-b_24_n2000_m200	10263575.47	9.10	0.456051
MDG-b_25_n2000_m200	10250090.79	9.26	0.438881
MDG-b_26_n2000_m200	10196189.88	9.71	0.535054
MDG-b_27_n2000_m200	10358195.61	8.38	0.652858
MDG-b_28_n2000_m200	10277383.17	8.89	0.514529
MDG-b_29_n2000_m200	10291258.67	8.90	0.453689
MDG-b_30_n2000_m200	10263859.33	9.14	0.437068
MDG-c_1_n3000_m300	22943111	7.80	1.557347
MDG-c_2_n3000_m300	22982398	7.72	1.703191
MDG-c_8_n3000_m400	40434465	6.91	2.50232
MDG-c_9_n3000_m400	40488295	6.79	2.391048
MDG-c_10_n3000_m400	40455410	6.95	2.641655
MDG-c_13_n3000_m500	63170811	5.73	3.631593
MDG-c_14_n3000_m500	62817710	6.21	3.497278
MDG-c_15_n3000_m500	63066444	5.86	3.515948
MDG-c_19_n3000_m600	90566205	5.30	4.681146
MDG-c_20_n3000_m600	90602264	5.27	5.020458

Tabla 1: Resultados para el algoritmo Greedy

Búsqueda local

Caso	Coste obtenido	Desv	Tiempo (s)
MDG-a_1_n500_m50	7599.76	2.99	0.002025
MDG-a_2_n500_m50	7679.05	1.19	0.001863
MDG-a_3_n500_m50	7636.37	1.59	0.001918
MDG-a_4_n500_m50	7589.15	2.33	0.001854
MDG-a_5_n500_m50	7588.68	2.15	0.002375
MDG-a_6_n500_m50	7589.2	2.37	0.001465
MDG-a_7_n500_m50	7616.8	1.99	0.001889
MDG-a_8_n500_m50	7570.99	2.32	0.001829
MDG-a_9_n500_m50	7650.44	1.54	0.001972
MDG-a_10_n500_m50	7623.53	2.02	0.001726
MDG-b_21_n2000_m200	11194345.39	0.93	0.078849
MDG-b_22_n2000_m200	11198330.26	0.78	0.121368
MDG-b_23_n2000_m200	11182727.52	1.04	0.090211
MDG-b_24_n2000_m200	11184415.56	0.94	0.125306
MDG-b_25_n2000_m200	11202715.92	0.83	0.134078
MDG-b_26_n2000_m200	11152433.18	1.24	0.116869
MDG-b_27_n2000_m200	11189891.7	1.02	0.119383
MDG-b_28_n2000_m200	11157321.43	1.09	0.106994
MDG-b_29_n2000_m200	11192932.07	0.92	0.104435
MDG-b_30_n2000_m200	11152329.79	1.28	0.078128
MDG-c_1_n3000_m300	24648263	0.95	0.501001
MDG-c_2_n3000_m300	24676154	0.92	0.512707
MDG-c_8_n3000_m400	43098299	0.78	0.898523
MDG-c_9_n3000_m400	43141730	0.68	1.175382
MDG-c_10_n3000_m400	43201539	0.63	1.046369
MDG-c_13_n3000_m500	66668600	0.52	1.660269
MDG-c_14_n3000_m500	66693391	0.43	1.673518
MDG-c_15_n3000_m500	66783597	0.31	1.794494
MDG-c_19_n3000_m600	95307787	0.34	3.095673
MDG-c_20_n3000_m600	95315225	0.34	3.067012

Tabla 2: Resultados para el algoritmo de búsqueda local del primer mejor

Enfriamiento Simulado

Caso	Coste obtenido	Desv	Tiempo (s)
MDG-a_1_n500_m50	7678.11	1.99	0.361629
MDG-a_2_n500_m50	7640.55	1.69	0.14189
MDG-a_3_n500_m50	7538.7	2.84	0.241213
MDG-a_4_n500_m50	7616.87	1.97	0.418492
MDG-a_5_n500_m50	7638.39	1.51	0.370235
MDG-a_6_n500_m50	7632.39	1.82	0.265257
MDG-a_7_n500_m50	7571.32	2.58	0.359476
MDG-a_8_n500_m50	7612.92	1.78	0.444353
MDG-a_9_n500_m50	7630.04	1.80	0.255594
MDG-a_10_n500_m50	7572.18	2.68	0.238489
MDG-b_21_n2000_m200	11130280.84	1.50	13.053819
MDG-b_22_n2000_m200	11103673.87	1.62	12.767416
MDG-b_23_n2000_m200	11128548.35	1.52	12.969726
MDG-b_24_n2000_m200	11123723.52	1.48	12.808461
MDG-b_25_n2000_m200	11181858.71	1.01	12.962768
MDG-b_26_n2000_m200	11126914.76	1.46	12.721742
MDG-b_27_n2000_m200	11155542.02	1.33	12.743572
MDG-b_28_n2000_m200	11149355.34	1.16	12.741256
MDG-b_29_n2000_m200	11106235.88	1.69	12.799816
MDG-b_30_n2000_m200	11160546.52	1.20	13.07157
MDG-c_1_n3000_m300	24611445	1.10	25.508213
MDG-c_2_n3000_m300	24568380	1.35	25.310284
MDG-c_8_n3000_m400	43053944	0.88	32.055538
MDG-c_9_n3000_m400	43118902	0.73	32.415083
MDG-c_10_n3000_m400	43011653	1.07	32.028375
MDG-c_13_n3000_m500	66515370	0.74	38.722601
MDG-c_14_n3000_m500	66575781	0.60	38.919961
MDG-c_15_n3000_m500	66596130	0.59	38.91442
MDG-c_19_n3000_m600	95020722	0.64	44.861724
MDG-c_20_n3000_m600	95078923	0.59	46.077711

Tabla 3: Resultados del Enfriamiento Simulado

Enfriamiento simulado con esquema proporcional

Caso	Coste obtenido	Desv	Tiempo (s)
MDG-a_1_n500_m50	7734.5	1.27	0.907588
MDG-a_2_n500_m50	7702.51	0.89	1.029182
MDG-a_3_n500_m50	7701.5	0.75	1.015719
MDG-a_4_n500_m50	7732.14	0.49	0.966112
MDG-a_5_n500_m50	7734.96	0.26	0.976293
MDG-a_6_n500_m50	7759.33	0.18	0.881734
MDG-a_7_n500_m50	7728.63	0.55	0.88912
MDG-a_8_n500_m50	7680.42	0.91	0.95001
MDG-a_9_n500_m50	7681.24	1.14	1.062548
MDG-a_10_n500_m50	7741.3	0.50	1.005495
MDG-b_21_n2000_m200	10157254.21	10.11	12.710014
MDG-b_22_n2000_m200	10173287.92	9.87	12.980063
MDG-b_23_n2000_m200	10166166.32	10.03	12.734856
MDG-b_24_n2000_m200	10166892.44	9.95	12.753764
MDG-b_25_n2000_m200	10176082.91	9.91	12.720688
MDG-b_26_n2000_m200	10192410.98	9.74	12.76846
MDG-b_27_n2000_m200	10147864.97	10.24	13.03998
MDG-b_28_n2000_m200	10144825.34	10.06	12.703971
MDG-b_29_n2000_m200	10159694.61	10.07	12.799469
MDG-b_30_n2000_m200	10163366.59	10.03	12.760804
MDG-c_1_n3000_m300	22667884	8.91	25.413164
MDG-c_2_n3000_m300	22662773	9.00	25.304935
MDG-c_8_n3000_m400	40241308	7.36	32.115399
MDG-c_9_n3000_m400	40211514	7.43	32.500695
MDG-c_10_n3000_m400	40174381	7.59	32.443536
MDG-c_13_n3000_m500	62733372	6.39	39.012691
MDG-c_14_n3000_m500	62751395	6.31	39.110124
MDG-c_15_n3000_m500	62741296	6.35	38.84062
MDG-c_19_n3000_m600	90322433	5.55	45.344314
MDG-c_20_n3000_m600	90343588	5.54	63.080852

Tabla 4: Resultados del Enfriamiento Simulado con esquema proporcional

Enfriamiento simulado versión 2

Caso	Coste obtenido	Desv	Tiempo (s)
MDG-a_1_n500_m50	7634.29	2.55	0.482379
MDG-a_2_n500_m50	7588.5	2.36	0.445368
MDG-a_3_n500_m50	7538.7	2.84	0.235949
MDG-a_4_n500_m50	7646.88	1.59	0.242228
MDG-a_5_n500_m50	7638.39	1.51	0.37175
MDG-a_6_n500_m50	7644.29	1.66	0.352385
MDG-a_7_n500_m50	7588	2.36	0.352015
MDG-a_8_n500_m50	7579.85	2.21	0.298304
MDG-a_9_n500_m50	7630.04	1.80	0.242226
MDG-a_10_n500_m50	7561.45	2.81	0.465259
MDG-b_21_n2000_m200	11154434.35	1.29	12.799825
MDG-b_22_n2000_m200	11162903.71	1.10	12.752653
MDG-b_23_n2000_m200	11150240.06	1.32	12.749326
MDG-b_24_n2000_m200	11156425.4	1.19	12.759103
MDG-b_25_n2000_m200	11181858.71	1.01	12.752216
MDG-b_26_n2000_m200	11164008.41	1.14	12.67755
MDG-b_27_n2000_m200	11135236.78	1.51	12.745947
MDG-b_28_n2000_m200	11149355.34	1.16	13.023176
MDG-b_29_n2000_m200	11151755.87	1.29	12.743956
MDG-b_30_n2000_m200	11143741.49	1.35	12.800738
MDG-c_1_n3000_m300	24600543	1.14	25.435441
MDG-c_2_n3000_m300	24663905	0.97	25.29303
MDG-c_8_n3000_m400	43053944	0.88	32.142121
MDG-c_9_n3000_m400	43118902	0.73	32.07991
MDG-c_10_n3000_m400	43011653	1.07	32.295398
MDG-c_13_n3000_m500	66515370	0.74	38.65491
MDG-c_14_n3000_m500	66575781	0.60	38.604845
MDG-c_15_n3000_m500	66596130	0.59	38.822558
MDG-c_19_n3000_m600	95096407	0.56	45.03496
MDG-c_20_n3000_m600	95078923	0.59	46.95559

Tabla 5: Resultados del Enfriamiento Simulado segunda versión

Enfriamiento simulado versión 3

Caso	Coste obtenido	Desv	Tiempo (s)
MDG-a_1_n500_m50	7503.68	4.21	0.035391
MDG-a_2_n500_m50	7616.18	2.00	0.065185
MDG-a_3_n500_m50	7390.9	4.75	0.03146
MDG-a_4_n500_m50	7450.74	4.11	0.02706
MDG-a_5_n500_m50	7447.15	3.97	0.049311
MDG-a_6_n500_m50	7459.87	4.04	0.051728
MDG-a_7_n500_m50	7519.62	3.24	0.04686
MDG-a_8_n500_m50	7554.37	2.54	0.056807
MDG-a_9_n500_m50	7490.46	3.60	0.0433
MDG-a_10_n500_m50	7433.08	4.46	0.049834
MDG-b_21_n2000_m200	11153490.31	1.30	7.555939
MDG-b_22_n2000_m200	11110020.94	1.57	4.011747
MDG-b_23_n2000_m200	11121529.78	1.58	4.991886
MDG-b_24_n2000_m200	11135886.02	1.37	6.997175
MDG-b_25_n2000_m200	11125404.86	1.51	7.063434
MDG-b_26_n2000_m200	11200346.41	0.81	12.525884
MDG-b_27_n2000_m200	11123444.28	1.61	7.543947
MDG-b_28_n2000_m200	11152601.32	1.13	12.732243
MDG-b_29_n2000_m200	11153613.26	1.27	10.775145
MDG-b_30_n2000_m200	11095525.94	1.78	5.503966
MDG-c_1_n3000_m300	24638530	0.99	25.429986
MDG-c_2_n3000_m300	24611980	1.18	19.501333
MDG-c_8_n3000_m400	43067327	0.85	32.352137
MDG-c_9_n3000_m400	43053409	0.89	32.407405
MDG-c_10_n3000_m400	42939077	1.24	29.310709
MDG-c_13_n3000_m500	66582253	0.64	38.903866
MDG-c_14_n3000_m500	66592291	0.58	38.663022
MDG-c_15_n3000_m500	66566317	0.64	38.627055
MDG-c_19_n3000_m600	95089913	0.57	45.109599
MDG-c_20_n3000_m600	95082076	0.59	47.259715

Tabla 6: Resultados del Enfriamiento Simulado tercera versión

Búsqueda multiarranque básica

Caso	Coste obtenido	Desv	Tiempo (s)
MDG-a_1_n500_m50	7712.41	1.55	0.016881
MDG-a_2_n500_m50	7679.05	1.19	0.017119
MDG-a_3_n500_m50	7713.09	0.60	0.017717
MDG-a_4_n500_m50	7676.25	1.21	0.016254
MDG-a_5_n500_m50	7678.14	0.99	0.018175
MDG-a_6_n500_m50	7645.05	1.66	0.015232
MDG-a_7_n500_m50	7669.83	1.31	0.015869
MDG-a_8_n500_m50	7688.22	0.81	0.018087
MDG-a_9_n500_m50	7650.44	1.54	0.019071
MDG-a_10_n500_m50	7690.85	1.15	0.016993
MDG-b_21_n2000_m200	11185219.43	1.01	0.643558
MDG-b_22_n2000_m200	11164624.55	1.08	0.645764
MDG-b_23_n2000_m200	11181096.56	1.05	0.669538
MDG-b_24_n2000_m200	11162933.03	1.13	0.658882
MDG-b_25_n2000_m200	11180764.24	1.02	0.664427
MDG-b_26_n2000_m200	11184247.51	0.96	0.656049
MDG-b_27_n2000_m200	11176658.75	1.14	0.657572
MDG-b_28_n2000_m200	11165811.19	1.01	0.658754
MDG-b_29_n2000_m200	11181414	1.02	0.665456
MDG-b_30_n2000_m200	11174992.53	1.07	0.655536
MDG-c_1_n3000_m300	24655615	0.92	3.726733
MDG-c_2_n3000_m300	24624066	1.13	3.591083
MDG-c_8_n3000_m400	43095630	0.79	7.42543
MDG-c_9_n3000_m400	43035076	0.93	7.424318
MDG-c_10_n3000_m400	43075071	0.92	7.560634
MDG-c_13_n3000_m500	66556484	0.68	12.758743
MDG-c_14_n3000_m500	66523292	0.68	12.779268
MDG-c_15_n3000_m500	66557652	0.65	12.714156
MDG-c_19_n3000_m600	95022407	0.64	18.942945
MDG-c_20_n3000_m600	95006643	0.67	20.507988

Tabla 7: Resultados de la Búsqueda multiarranque básica

Iterative Local Search

Caso	Coste obtenido	Desv	Tiempo (s)
MDG-a_1_n500_m50	7619.6	2.73	0.007738
MDG-a_2_n500_m50	7703.68	0.87	0.008425
MDG-a_3_n500_m50	7692.22	0.87	0.008906
MDG-a_4_n500_m50	7673.3	1.25	0.006773
MDG-a_5_n500_m50	7642.03	1.46	0.008469
MDG-a_6_n500_m50	7698.95	0.96	0.006153
MDG-a_7_n500_m50	7619.52	1.96	0.007598
MDG-a_8_n500_m50	7706.72	0.57	0.01111
MDG-a_9_n500_m50	7696.93	0.94	0.006665
MDG-a_10_n500_m50	7687.11	1.20	0.007095
MDG-b_21_n2000_m200	11221592.83	0.69	0.432231
MDG-b_22_n2000_m200	11212350.98	0.66	0.473368
MDG-b_23_n2000_m200	11257630.56	0.37	0.398343
MDG-b_24_n2000_m200	11224057.74	0.59	0.392823
MDG-b_25_n2000_m200	11220210.11	0.67	0.401019
MDG-b_26_n2000_m200	11205205.11	0.77	0.406654
MDG-b_27_n2000_m200	11228103.43	0.69	0.403328
MDG-b_28_n2000_m200	11221052.61	0.52	0.404951
MDG-b_29_n2000_m200	11218843.76	0.69	0.398336
MDG-b_30_n2000_m200	11192911.8	0.92	0.383134
MDG-c_1_n3000_m300	24776309	0.43	1.529762
MDG-c_2_n3000_m300	24760019	0.58	1.538959
MDG-c_8_n3000_m400	43287246	0.35	2.941266
MDG-c_9_n3000_m400	43227355	0.48	3.056526
MDG-c_10_n3000_m400	43281674	0.45	2.944639
MDG-c_13_n3000_m500	66848792	0.25	5.223967
MDG-c_14_n3000_m500	66836311	0.21	5.359053
MDG-c_15_n3000_m500	66875251	0.18	5.432664
MDG-c_19_n3000_m600	95387035	0.26	8.284411
MDG-c_20_n3000_m600	95374556	0.28	7.346346

Tabla 8: Resultados de la búsqueda local iterativa

Iterative Local Search versión 2

Caso	Coste obtenido	Desv	Tiempo (s)
MDG-a_1_n500_m50	7693.11	1.80	0.041519
MDG-a_2_n500_m50	7708.74	0.81	0.046397
MDG-a_3_n500_m50	7739.15	0.26	0.048144
MDG-a_4_n500_m50	7758.61	0.15	0.056866
MDG-a_5_n500_m50	7698.13	0.74	0.044648
MDG-a_6_n500_m50	7715.01	0.76	0.04294
MDG-a_7_n500_m50	7741.22	0.39	0.047083
MDG-a_8_n500_m50	7684.64	0.85	0.046416
MDG-a_9_n500_m50	7699.43	0.91	0.041623
MDG-a_10_n500_m50	7713.42	0.86	0.047057
MDG-b_21_n2000_m200	11149053.61	1.33	1.269768
MDG-b_22_n2000_m200	11154982.15	1.17	1.141834
MDG-b_23_n2000_m200	11193201.93	0.94	1.437461
MDG-b_24_n2000_m200	11149474.99	1.25	1.105185
MDG-b_25_n2000_m200	11177149.27	1.05	1.168833
MDG-b_26_n2000_m200	11133512.77	1.41	1.161558
MDG-b_27_n2000_m200	11203357.86	0.91	1.339872
MDG-b_28_n2000_m200	11133588.62	1.30	1.160689
MDG-b_29_n2000_m200	11169465.09	1.13	1.088349
MDG-b_30_n2000_m200	11147321.52	1.32	1.108581
MDG-c_1_n3000_m300	24542832	1.37	6.801242
MDG-c_2_n3000_m300	24541173	1.46	7.3322
MDG-c_8_n3000_m400	42833103	1.39	14.093004
MDG-c_9_n3000_m400	42909525	1.22	14.109673
MDG-c_10_n3000_m400	42931477	1.25	14.357467
MDG-c_13_n3000_m500	66328542	1.02	25.07074
MDG-c_14_n3000_m500	66291310	1.03	24.124481
MDG-c_15_n3000_m500	66300128	1.03	23.047678
MDG-c_19_n3000_m600	94712889	0.96	34.765983
MDG-c_20_n3000_m600	94679068	1.01	35.629243

Tabla 9: Resultados de la búsqueda local iterativa segunda versión

Iterative Local Search con enfriamiento simulado

Caso	Coste obtenido	Desv	Tiempo (s)
MDG-a_1_n500_m50	7621.39	2.71	1.044327
MDG-a_2_n500_m50	7640.43	1.69	1.036032
MDG-a_3_n500_m50	7612.31	1.90	0.968841
MDG-a_4_n500_m50	7632.12	1.78	1.067518
MDG-a_5_n500_m50	7605.3	1.93	1.071288
MDG-a_6_n500_m50	7601.34	2.22	1.031699
MDG-a_7_n500_m50	7594.72	2.28	0.951498
MDG-a_8_n500_m50	7584.74	2.14	0.978588
MDG-a_9_n500_m50	7622.81	1.90	0.984065
MDG-a_10_n500_m50	7675.95	1.34	0.982734
MDG-b_21_n2000_m200	10983733.01	2.80	12.215443
MDG-b_22_n2000_m200	11003615.06	2.51	12.439122
MDG-b_23_n2000_m200	11002424.42	2.63	12.096516
MDG-b_24_n2000_m200	11006395.73	2.52	12.098864
MDG-b_25_n2000_m200	11009798.02	2.53	12.200119
MDG-b_26_n2000_m200	11004446.44	2.55	11.828054
MDG-b_27_n2000_m200	11026676.55	2.47	11.609973
MDG-b_28_n2000_m200	10990952.26	2.56	11.745558
MDG-b_29_n2000_m200	11020075.32	2.45	11.270717
MDG-b_30_n2000_m200	11033151.03	2.33	11.285024
MDG-c_1_n3000_m300	24236309	2.60	26.621994
MDG-c_2_n3000_m300	24277132	2.52	26.350929
MDG-c_8_n3000_m400	42468845	2.23	33.910087
MDG-c_9_n3000_m400	42450200	2.27	33.336485
MDG-c_10_n3000_m400	42488597	2.27	34.194774
MDG-c_13_n3000_m500	65793728	1.82	41.117654
MDG-c_14_n3000_m500	65761926	1.82	40.506943
MDG-c_15_n3000_m500	65775859	1.82	41.201347
MDG-c_19_n3000_m600	94120232	1.58	46.815538
MDG-c_20_n3000_m600	94083932	1.63	47.482251

Tabla 10: Resultados de la búsqueda local iterativa con ES

Iterative Local Search con enfriamiento simulado y esquema proporcional

Caso	Coste obtenido	Desv	Tiempo (s)
MDG-a_1_n500_m50	6335.43	19.13	1.087402
MDG-a_2_n500_m50	6301.49	18.92	1.102967
MDG-a_3_n500_m50	6280.08	19.06	1.04335
MDG-a_4_n500_m50	6413.57	17.46	0.996363
MDG-a_5_n500_m50	6301.21	18.75	1.045147
MDG-a_6_n500_m50	6265.55	19.40	0.999544
MDG-a_7_n500_m50	6253.84	19.53	0.999302
MDG-a_8_n500_m50	6207.24	19.92	1.099883
MDG-a_9_n500_m50	6409.63	17.51	1.023551
MDG-a_10_n500_m50	6260.44	19.54	1.112417
MDG-b_21_n2000_m200	10021922.56	11.31	12.16016
MDG-b_22_n2000_m200	9985499.19	11.53	11.168514
MDG-b_23_n2000_m200	9977116.28	11.71	11.70662
MDG-b_24_n2000_m200	9995844.09	11.47	11.441743
MDG-b_25_n2000_m200	10032679.79	11.18	11.659981
MDG-b_26_n2000_m200	10019692.39	11.27	11.34339
MDG-b_27_n2000_m200	10026706.83	11.31	11.412562
MDG-b_28_n2000_m200	10029215.75	11.09	11.602571
MDG-b_29_n2000_m200	10025558.06	11.26	11.369525
MDG-b_30_n2000_m200	10002716.94	11.45	12.194783
MDG-c_1_n3000_m300	22489689	9.62	26.611388
MDG-c_2_n3000_m300	22526006	9.55	26.843193
MDG-c_8_n3000_m400	40066986	7.76	33.682657
MDG-c_9_n3000_m400	39980385	7.96	34.485605
MDG-c_10_n3000_m400	40018196	7.95	34.107439
MDG-c_13_n3000_m500	62597915	6.59	40.202515
MDG-c_14_n3000_m500	62508742	6.67	39.664199
MDG-c_15_n3000_m500	62552343	6.63	39.752276
MDG-c_19_n3000_m600	90032871	5.86	46.577802
MDG-c_20_n3000_m600	90069830	5.83	46.334917

Tabla 11: Resultados de la búsqueda local iterativa con ES y esquema proporcional

5.2. Comparación entre los algoritmos

Mostramos ahora una tabla con la media de los estadísticos (desviación y tiempo de ejecución) para cada uno de los cuatro algoritmos:

Algoritmo	Desv	Tiempo (s)
Greedy	9.22	1.2
BL	1.22	0.55
ES	1.43	16.22
ES-Prop	5.91	17.03
ES2	1.40	16.22
ES3	1.97	14.26
BMB	1.02	3.81
ILS	0.76	1.59
ILS2	1.04	7.06
ILS-ES	2.19	16.68
ILS-ES-Prop	12.57	16.49

Tabla 12: Comparativa de estadísticos medios obtenidos por distintos algoritmos para el MDP

5.3. Análisis de los resultados

Para tener una visión global de los resultados obtenidos por los distintos algoritmos empezamos observando la Tabla 12. Nos damos cuenta de que el algoritmo de ILS-ES-Prop ofrece una desviación exageradamente alta, seguido por el algoritmo Greedy. Ya sabemos que este último no explora el espacio de soluciones con tanta profundidad como lo hacen el resto de algoritmos considerados, de ahí que presente una deviación alta. Por otra parte, el algoritmo ILS presenta el mejor de los resultados, con una desviación menor que 1, seguido por la BMB y la segunda versión de ILS.

En cuanto al tiempo de ejecución, la BL sigue siendo la más rápida, debido a la factorización que se hace de la función objetivo en todas las evaluaciones de las soluciones, seguida por Greedy y la ILS.

Analizamos estos aspectos con más detalle a continuación.

Comenzamos analizando los resultados del **enfriamiento simulado**. Este algoritmo, a diferencia de la búsqueda local, acepta soluciones peores a la actual, de manera que permite salir de óptimos locales, en los que la BL se quedaría atrapada. Así, la diversificación en este algoritmo es mayor que en la BL. Por otra parte, el operador de generación de soluciones vecinas en el ES cambia posiciones aleatorias de la solución por un elemento aleatorio no seleccionado en la misma, mientras que en el caso de la BL eran los elementos que menos contribuían los que se cambiaban por los elementos no seleccionados más prometedores. Esto hace que en la búqueda local la convergencia hacia óptimos locales sea más rápida que en el enfriamiento simulado, de modo que en BL hay una mayor explotación de las soluciones.

Podemos ver que el ES proporciona un valor de la desviación algo más alto que la BL. Aunque ES permite salir de óptimos locales y presenta mayor diversificación, la convergencia de la búsqueda local es más rápida, de modo que es probable que ES necesite más iteraciones para converger a un óptimo local adecuado, es decir, que sea necesario aumentar la intensificación frente a la diversificación en el mismo. El esquema de enfriamiento de Cauchy modificado permite que el enfriamiento sea más rápido si el número de iteraciones es pequeño, buscando así un equilibrio entre intensificación y diversificación. Sin embargo, puede ser que el número de iteraciones no sea suficiente para permitir una adecuada explotación de las soluciones en las últimas iteraciones, que es cuando la temperatura es baja y se aceptan muy pocas soluciones peores.

El hecho recién comentado también hace que los resultados proporcionados por ILS con búsqueda local sean mucho mejores que con el enfriamiento simulado. En este caso la diferencia entre ambas versiones es bastante considerable, pues pasa de una desviación de 0.76 con la búsqueda local a 2.19 con el enfriamiento simulado. En ILS se dejan incluso menos iteraciones para el ES, lo que hace que la diferencia en este caso sea más acusada.

La BL es mucho más rápida que el ES. Esto puede ser debido a que en el operador de vecino del ES se deben generar dos posiciones aleatorias cada vez que se usa, y esto es costoso, mientras que en la generación de vecinos en la BL esto no era necesario. La factorización de la función objetivo se lleva acabo de la misma forma en ES y en BL, por lo que esto no debería suponer un aumento o disminución del tiempo de ejecución entre los dos algoritmos.

La segunda versión considerada del ES presenta una desviación prácticamente igual que el ES original estudiado. En este caso lo que hacemos es aumentar la probabilidad de elegir soluciones peores, de manera que aumenta aún más la diversifación, pudiendo el algoritmo salta a soluciones peores con una mayor frecuencia. No obstante esto no hace que mejores los resultados, ni tampoco que empeores.

En la tercera versión se generan menos vecinos como máximo, pasando de $10m$ en el original a m, de manera que M también aumenta, pues recordemos que $M = 100000/max\_vecinos$, y entonces el enfriamiento es más lento. Esto implica que haya aún más diversificación y menos intensificación, motivo por el cual los resultados con esta tercera versión empeoran. El equilibrio entre exploración y explotación (diversificación e intensificación) se rompe a favor de la primera.
´

Todas las versiones del enfriamineto simulado tienen aproximadamente el mismo tiempo de ejecución, pues no se realizan cambios significativos que afecten a la eficiencia del algoritmo.

En el caso en el que se usa el esquema de enfriamiento proporcional los resultados son realmente malos. En el esquema de Cauchy modificado la temperatura se enfría muy rápidamente al principio y más lento al final, cuando las soluciones son mejores y se aboga más por la explotación. Además, permite adaptar la temperatura al número de iteraciones M, el cual depende a su vez del número de evaluaciones límite de la función objetivo y del número máximo de vecinos (que depende del tamaño de las soluciones, m). En cambio, en el esquema proporcional, el decrecimiento de la temperatura es lineal, se realiza según un parámetro fijo α y no se ve afectado por el tamaño de las soluciones o por el número de evaluaciones límite de la función objetivo.

Este hecho hace que para los casos de estudio del *tipo a*, donde m es menor, el esquema proporcional con parámetro $\alpha = 0{,}9$ proporcione mejores resultados que el esquema de Cauchy modificado, como podemos ver en las Tablas 3 y 4. Sin embargo, para los casos del *tipo b y c*, donde m es más grande, el esquema proporcional ofrece mayores desviaciones que el de Cauchy modificado, pues el primero no es capaz de adaptarse al tamaño del problema.

Es posible que se pueda determinar un valor de α para el cual el esquema proporcional sea mejor, en función del tamaño del problema, pues hemos visto que para los ejemplos del *tipo a* $\alpha = 0{,}9$ proporciona resultados satisfactorios.

Por otra parte, dado que con el esquema proporcional la temperatura disminuye más gradualmente, se aceptan más soluciones peores que con Cauchy modificado, de manera que hay una diversificación aún mayor y menor intensificación, por lo que la convergencia se hace aún más lenta y al algoritmo no le da tiempo a encontrar un óptimo local lo suficientemente bueno, sobre todo para los problemas de mayor tamaño como los del *tipo b ó c*

En el algoritmo híbrido ILS con ES observamos en las Tablas 10 y 11 que para todos

los casos de estudio el esquema proporcional presenta las mayores desviaciones, siendo estas considerablemente grandes, incluso para los casos del *tipo a*, por lo que deducimos que $\alpha = 0{,}9$ no es adecuado en este algoritmo para ninguno de los casos de estudio y quizás debería ajustarse para que ofrezca resultados decentes.

Por lo tanto, podemos decir que el esquema de enfriamiento de Cauchy modificado es más adecuado en general que el esquema proporcional, pues se adapta a cualquier problema, mientras que el proporcional debería ser configurado a mano.

Pasamos ahora a analizar la **búsqueda multiarranque básica**. En este caso se aplicaba la búsqueda local a 10 soluciones generadas aleatoriamente. Puesto que hay que generar las soluciones aleatorias, lo cual ya sabemos que es costoso, el tiempo de ejecución de este algoritmo es ligeramente superior al de la búsqueda local y al de ILS. Por otro lado vemos que presenta una desviación menor que la búsqueda local pero mayor que ILS. Esto es debido a que con BMB se le añade algo más de diversificación a la BL, sin perder la propiedad de intensificación con la que esta cuenta, ya que, aunque se disminuyen el número máximo de evaluaciones a 10000 en la BL, la rápida convergencia de ésta hace que la disminución no sea notoria.

Por su parte la **búsqueda local iterativa** es similar a la BMB, salvo que en vez de considerar soluciones aleatorias, se mutan considerablemenete las mejores soluciones encontradas hasta el momento, de modo que hay diversificación, pues el algoritmo se aleja lo suficiente de la mejor solución, lo que permite escapar de óptimos locales. Pero también está presente la intensificación de la búsqueda local y el hecho de que se cambian soluciones buenas, y no cualesquiera aleatorias, como es el caso de BMB. Esto hace que ILS supere en resultados a BMB. Así, se añade más exploración a la búsqueda local, de manera que los óptimos que se encuentran son mejores, pero también tenemos una considerable explotación de las mejores soluciones. Todo esto da lugar a unas muy buenas soluciones, con una desviación media menor a 1, y posiciona al algoritmo ILS como el mejor de los estudiados hasta la fecha para el problema MDP.

En cuanto al tiempo de ejecución, ILS es algo más lenta que la BL, pues en la mutación de las soluciones estas se evalúan en su totalidad, y no de manera factorizada como ocurría en BL. Sin embargo, como en ILS sólo se genera una solución aleatoria y no 10, ILS es más rápida que BMB, en la cual la evaluación de las soluciones aleatorias tampoco se lleva a cabo de manera factorizada, otro aspecto que hace que BMB sea maś lenta que la BL.

Analizamos finalmente la segunda versión de ILS. Aquí le dejamos menos iteraciones a cada aplicación de la búsqueda local, pues esta converge bastante rápido, y consideramos más mutaciones de las soluciones mejores, de manera que se fomenta la exploración del entorno un poco más frente a la explotación. No obstante la idea no ha sido demasiado buena, pues los resultados han empeorado ligeramente. Podemos deducir así que al favorecer la diversificación las soluciones empeoran, pues quizás la BL no tiene ahora suficientes iteraciones y la intensificación no es tan notoria como antes, a pesar de la rápida convergencia de la BL.

Como en esta nueva versión hay más mutaciones y, por lo tanto, más evaluaciones de soluciones completas, el tiempo de ejecución aumenta bastante frente al de la versión original que consideramos para ILS.