Metaheurísticas: Práctica 2

Técnicas de Búsqueda basadas en Poblaciones
para el Problema de la Máxima Diversidad

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

12 de mayo 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 distintos algoritmos.

Todos los algoritmos parten de una matriz de distancias D de tamaño $n \times n$, como ya hemos comentado (que nosotros llamaremos simplemente *matrix* 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. Práctica 1

2.1.1. 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, 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 solution {
        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.

2.1.2. Contribución de un elemento

Definimos para ambos algoritmos 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

2.1.3. 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.

2.2. Práctica 2

2.2.1. Representación de las soluciones

Para esta práctica se considera una nueva representación de las soluciones, basada en un vector binario con tantas posiciones como elementos hay en el problema (n). En concreto, tendremos un vector de booleanos, donde la posición i-ésima de dicho vector será *true* (equivalentemente 1) si el elemento i-ésimo forma parte de la solución y *false* (ó 0) si no forma parte de la misma. Asociado a este vector tendremos dos parámetros adicionales: el *fitness* de la solución determinada por el vector y un booleano que servirá para indicar si la solución ha sido o no evaluada (se ha calculado su fitness), de manera que se evite evaluar varias veces la misma solución. Así, la estructura de la solución será la siguiente:

```
struct solution {
        vector<bool> elements;
        double fitness;
        bool evaluated;
};
```

Por otra parte, se representa una población como un vector de soluciones, con un tamaño igual al de la población, esto es, el número de soluciones que tiene esa población. Junto al vector de soluciones se guarda también el fitness de la mejor solución de la población y la posición en el vector de la misma:

```
struct population {
        vector<solution> solutions;
        double best_fitness;     //Fitness de la mejor solucion
        int best_sol;            //Posicion de la mejor solucion
};
```

Para ambas estructuras sobrecargamos el operador de asignación, como en la práctica anterior.

Cabe destacar que la representación de las soluciones de la BL y esta nueva representación tienen el mismo nombre (solution), pero se usan en algoritmos diferentes, por lo que esto no supone ningún problema. Para los algoritmos meméticos, en cambio, sí se usan las dos representaciones a la vez, por lo que en este caso notaremos a la estructura de la solución binaria como solution_bin.

2.2.2. Contribución de un elemento

Debido a esta nueva representación de las soluciones, la función que determina la contribución de un elemento a una solución varía ligeramente de la usada en la práctica anterior. En este caso, se suma la distancia del elemento considerado a otro elemento de la solución sólo si el valor de la posición correspondiente a ese elemento en el vector de booleanos es *true*:

Algorithm 3: CONTRIBUTION

Input: vector de booleanos *sol*, *matrix* de distancias, entero *element*
Output: contribucion del entero *element* a la solución dada en *sol*
begin
  sum ← 0
  for *i* ***in*** $[0, sol.size)$ **do**
    if *sol[i]* **then**
      sum ← sum + matrix[element, i]
    end
  end
  return sum
end

2.2.3. Función objetivo

La función objetivo también se ve modificada parcialmente, de forma que queda como sigue:

Algorithm 4: FITNESS

Input: vector de booleanos *sol*, *matrix* de distancias
Output: valor de la función objetivo para la solución dada en *sol*
begin
  sum ← 0
  for *i* ***in*** $[0, sol.size)$ **do**
    for $j = i + 1$ ***to*** $sol.size$ **do**
      if *sol[i]* ***and*** *sol[j]* **then**
        sum ← sum + matrix[i,j]
      end
    end
  end
  return sum
end

Así, se suma la distancia entre dos elementos sólo si estos forman parte de la solución, es decir, si sus

respectivos valores son *true* en el vector *sol*.

2.2.4. Evaluación de una población

Con el objetivo de evaluar una población de soluciones implementamos la función evaluatePopulation, que se encarga de calcular el fitness de todas las soluciones de la población que aún no han sido evaluadas y determinar la posición y el fitness de la mejor solución contenida en la población:

Algorithm 5: EVALUATEPOPULATION

```
Input: población pop a ser evaluda, matrix de distancias,
número de evaluaciones evaluations
Output: población pop evaluada
begin
    best_pos ← pop.best_sol
    best_fit ← pop.best_fitness
    for sol in pop.solutions do
        if sol is not evaluated then
            evaluations ++
            sol.fitness ← fitness(sol, matrix)
            sol.evaluated ← true
            if best_fit < sol.fitness then
                best_fit ← sol.fitness
                best_pos ← index_of(sol)
            end
        end
    end
    pop.best_fitness ← best_fit
    pop.best_sol ← best_pos
    return pop
end
```

Notamos que tras cada llamada a la función *fitness* para una solución, se aumenta el número de evaluaciones de dicha función (*evaluations*)

2.2.5. Generación de una población aleatoria

En todos los algoritmos genéticos se parte de una población generada aleatoriamente. Para generar las soluciones aleatorias usamos la función randomSolution:

Algorithm 6: randomSolution

Input: tamaño de la solución m, *matrix* de distancias
Output: solución válida del problema MDP
begin
  $sol \leftarrow [0, 0, ...^{n)}, 0]$ `// Partimos de una solución con todos los elementos // sin seleccionar`
  chosen $\leftarrow 0$ `// Los elementos elegidos son 0`
  while $chosen < m$ **do**
    rand $\leftarrow$ elemento aleatorio de $\{0, ..., n-1\}$
    if !$sol[rand]$ **then**
      $sol[rand] \leftarrow$ true `// Si la posición aleatoria considerada // no está elegida, se añade a la solución`
      chosen++
    end
  end
  sol.evaluated $\leftarrow$ false
  return sol
end

La función randomPopulation se encarga de generar la población inicial aleatoriamente haciendo uso de la función anterior como se muestra a continuación:

Algorithm 7: randomPopulation

Input: tamaño de la población *size_pop*, tamaño de una solución m, *matrix* de distancias, número de evaluaciones de la función objetivo *evaluations*
Output: población de soluciones válidas del problema MDP correctamente evaluada
begin
  for i ***in*** $[0, size\_pop)$ **do**
    sol $\leftarrow$ randomSolution$(m, matrix)$
    pop.solutions $\leftarrow$ pop.solutions $\cup \{sol\}$
  end
  $pop.best\_sol \leftarrow$ -1
  $pop.best\_fitness \leftarrow 0$
  evaluatePopulation$(pop, matrix, evaluations)$
  return pop
end

2.2.6. Operadores comunes de los algoritmos genéticos

Describimos a continuación cada uno de los operadores que tienen en común los algoritmos genéticos implementados.

En todos los casos se sigue la estrategia de selección por torneo, en concreto por torneo binario, en el que se escogen dos soluciones aleatorias de la población con reemplazamiento y se selecciona (para el posterior cruce) aquella que tiene un valor de fitness mayor de entre las dos elegidas. El pseudo-código

de la función que lleva a cabo este proceso es el siguiente:

Algorithm 8: BinaryCompetition

Input: Población *pop*
Output: Posición en la población de la solución con mayor fitness de entre dos elegidas aleatoriamente
begin
  $rand1 \leftarrow$ elemento aleatorio de $\{0, ..., size\_pop - 1\}$
  $rand2 \leftarrow$ elemento aleatorio de $\{0, ..., size\_pop - 1\}$
  if $pop.solutions[rand1].fitness > pop.solutions[rand2].fitness$ **then**
    return $rand1$
  else
    return $rand2$
  end
end

El proceso de selección depende en cada caso del algoritmo concreto, generacional o estacionario.

Para la mutación, hacemos uso en todos los algoritmos de la función mutateSolution, que elige dos posiciones aleatorias distintas de la solución que se le pasa como parámetro, una con el valor 1 y otra con el valor 0, e intercambia sus valores, de manera que se obtiene otra solución válida.

Algorithm 9: mutateSolution

Input: solución *sol*, *matrix* de distancias
Output: solución modificada tras mutar dos posiciones
begin
  do
  `// Posición aleatoria de la solución que tenga el valor 1 (true)`
  pos1 $\leftarrow$ elemento aleatorio de $\{0, ..., size\_pop - 1\}$
  while $!sol.elements[pos1]$
  do
  `// Posición aleatoria de la solución que tenga el valor 0 (false)`
  pos2 $\leftarrow$ elemento aleatorio de $\{0, ..., size\_pop - 1\}$
  while $sol.elements[pos2]$
  $sol.elements[pos1] \leftarrow$ false
  $sol.elements[pos2] \leftarrow$ true
  sol.evaluated $\leftarrow$ false
end

Veamos ahora los distintos operadores de cruce usados en los algoritmos genéticos, que han sido dos: operador de cruce basado en posición y operador de cruce uniforme.

El primero de ellos genera una nueva solución factible a partir de otras dos soluciones padre, la cual comparte con los padres los valores de las posiciones que tienen el mismo valor en ambos padres. El resto de las posiciones del hijo toman un valor aleatorio de los valores restantes de uno de los padres (que serán los mismos en los dos padres por ser soluciones factibles). El pseudocódigo de este operadror es el siguiente:

Algorithm 10: positionalCross

Input: solución *father*, solución *mother*
Output: nueva solución *son*
begin
  `// Posiciones con valores no comunes en father y mother`
  $to\_shuffle, to\_change \leftarrow \emptyset$
  `// Determinamos las posiciones que tienen valores iguales en father y mother`
  `// y las que no`
  foreach $i \in \{0, ..., mother.size() - 1\}$ **do**
    if *father[i] = mother[i]* **then**
      son[i] ← mother[i]
    else
      $to\_shuffle \leftarrow to\_shuffle \cup \{i\}$
      $to\_change \leftarrow to\_change \cup \{i\}$
    end
  end
  `// Barajamos las posiciones que no tienen valores comunes`
  to_shuffle ← randomShuffle(to_shuffle)
  `// En las posiciones que no tienen valor aún, se introduce`
  `// un valor aleatorio de los restantes de un padre`
  foreach $j \in \{0, ..., to\_change.size() - 1\}$ **do**
    son[to_change[j]] ← mother[to_shuffle[j]]
  end
  son.evaluated ← false
  return son
end

El operador de cruce uniforme, como el anterior, conserva en el hijo los valores de las posiciones que presentan el mismo valor en ambos padres. Sin embargo, en este caso las posiciones restantes del hijo se rellenan con un valor aleatorio (0 ó 1), de manera que la solución resultante puede no ser factible. Así, es necesario aplicar un operador de reparación sobre el hijo, que determina el número de elementos seleccionados (número de posiciones que tienen el valor true) y, en caso de que sea diferente a m (número de elementos seleccionados que debe haber en una solución), añade (si faltan elementos) o elimina (si sobran) el elemento que más contribuye a la solución (en caso de que forme parte de ella) o que más contribuiría si formara parte de la misma, respectivamente, hasta que el tamaño de la solución sea el adecuado (m).

Algorithm 11: REPAIR

Input: solución *sol*, tamaño de una solución m, *matrix* de distancias
Output: solución *sol* reparada
begin
  selected ← nº de posiciones con valor true en *sol*
  while *selected* $> m$ **do**
    max_pos ← posición de *sol* con valor *true* de mayor contribución
    sol[max_pos]← false
    selected - -
  end
  while *selected* $< m$ **do**
    max_pos ← posición de *sol* con valor $false$ que más contribuiría a la solución
    sol[max_pos]← true
    selected ++
  end
  return sol
end

Algorithm 12: UNIFORMCROSS

Input: solución *father*, solución *mother*, tamaño de una solución m, *matrix* de distancias
Output: nueva solución *son*
begin
  `// Posiciones con valores no comunes en father y mother`
  $to\_change \leftarrow \emptyset$
  `// Determinamos las posiciones que tienen valores iguales en father y mother`
  `// y las que no`
  foreach $i \in \{0, ..., mother.size() - 1\}$ **do**
    if *father[i] = mother[i]* **then**
      son[i] ← mother[i]
    else
      $to\_change \leftarrow to\_change \cup \{i\}$
    end
  end
  `// En las posiciones que no tienen valor aún,`
  `// se introduce un valor aleatorio`
  foreach $j \in \{0, ..., to\_change.size() - 1\}$ **do**
    son[to_change[j]] ← valor aleatorio 0 ó 1
  end
  repair($son, m, matrix$)
  son.evaluated ← false
  return son
end

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 13: 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 14: 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 15: 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 16: 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 ← furthestElement($matriz$)
  $Sel \leftarrow Sel \cup \{furthest\}$
  $NoSel \leftarrow NoSel \backslash \{furthest\}$
  while $|Sel| < m$ **do**
    furthest ← 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 la solución aleatoria de partida consideramos la siguiente función:

Algorithm 17: 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
```

Definimos otra función validElements, que determina los elementos que son válidos para ser añadidos a un solución, es decir, aquellos elementos que no se encuentran ya en la misma:

Algorithm 18: 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
```

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 nue-

va 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 19: 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 $\leftarrow$ 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 $\leftarrow$ 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 20: 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 21: 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. Algoritmos genéticos generacionales

En este esquema, se selecciona una nueva población de soluciones a partir de otra población antigua, con tantas soluciones como tamaño tenga la población de la que se parte. Para ello, se usa el torneo binario, como ya describimos en la sección de aplicación de los algoritmos, aplicándolo tantas veces como sea necesario para obtener el tamaño de población buscado. Puesto que en el torneo binario las soluciones se eligen aleatoriamente, puede ocurrir que la nueva población de soluciones seleccionadas presente varias veces la misma solución. El pseucódigo de este proceso es el siguiente:

Algorithm 22: SELECTION

Input: Población *old_pop*
Output: Nueva población de soluciones *new_pop*
begin
  $new\_pop.solutions \leftarrow \emptyset$
  $new\_pop.best\_fitness \leftarrow 0$
  $new\_pop.best\_sol \leftarrow -1$
  foreach $i \in \{0, ..., old\_pop.solutions.size() - 1\}$ **do**
    $pos \leftarrow$ binaryCompetition(old_pop)
    new_pop.solutions $\leftarrow new\_pop.solutions \cup old\_pop.solutions[pos]$
  end
  return new_pop
end

Las soluciones de esta nueva población se cruzarán por parejas, con una probabilidad de 0,7 para cada pareja. Así, el número esperado de cruces será $0{,}7 \times \frac{size\_pop}{2} = 17{,}5$, siendo *size_pop* el tamaño de la población (50 en nuestro caso) y $\frac{size\_pop}{2} = 25$ el número de parejas que se pueden formar. Para la implementación se toma la parte entera, siendo así 17 cruces los llevados a cabo en cada generación. Además, como las posiciones de las soluciones en la nueva población seleccionada son aleatorias (gracias al torneo binario), basta cruzar la solución $2i$ con la $2i + 1$, $\forall i \in \{0, ..., 17\}$. Como resultado de cada cruce se generarán dos nuevas soluciones (serán igual a los padres si estos son la misma solución), que sustituyen a los padres en la población. Por lo tanto, el cruce en estos algoritmos queda como sigue:

Algorithm 23: CROSS

Input: Población *pop*, probabilidad de cruce por pareja *cross_prob*
begin
  num_cross $\leftarrow cross\_prob \times pop.size()/2$
  foreach $i \in \{0, ..., num\_cross - 1\}$ **do**
    sol1 $\leftarrow$ PCross/UCross($pop.solutions[2i], pop.solutions[2i + 1]$) `// Cruce uniforme`
    sol2 $\leftarrow$ PCross/UCross($pop.solutions[2i], pop.solutions[2i + 1]$) `// o posicional`
    $pop.solutions[2i] \leftarrow$ sol1
    $pop.solutions[2i + 1] \leftarrow$ sol2
  end
end

A continuación, se lleva a cabo una mutación aleatoria de algunos de los valores de ciertas soluciones de la nueva población. En concreto, cada valor se muta con una probabilidad de $0{,}1/m$, donde m es el tamaño de un vector solución. Por tanto, el número esperado de mutaciones en una población será $0{,}1/m \times size\_pop \times m = 0{,}1 \times size\_pop = 5$, teniendo en cuenta que $size\_pop = 50$ en nuestro estudio. Se seleccionan entonces aleatoriamente tantas soluciones como número esperado de mutaciones (puede elegirse varias veces la misma solución) y se intercambian los valores de dos posiciones aleatorias de cada

solución seleccionada haciendo uso de mutateSolution.

Algorithm 24: MUTATION

Input: Población *pop*, probabilidad de mutación por solución *mut_prob*, *matrix* de distancias
begin
  num_mut ← $mut\_prob \times pop.size()$
  foreach $i \in \{0, ..., num\_mut - 1\}$ **do**
    pos ← número aleatorio entre 0 y pop.size()-1
    pop.solutions[pos] ← mutateSolution($pop.solutions[pos], matrix$)
  end
end

Finalmente, hay que reemplazar la población antigua (de la iteración anterior) por esta nueva población obtenida tras el cruce y mutación. Para ello, usamos el operador de reemplazamiento que simplemente sustituye la población antigua por la nueva, y, en caso de que la mejor solución de la población nueva no mejore a la mejor solución de la población anterior (en términos de un mayor fitness), se busca la posición de la peor solución presente en la población nueva (de lo cual se encarga la función worstSolution, cuyo pseudocódigo no se incluye por no tener mayor interés) y se inserta en ella la mejor solución de la población antigua.

Algorithm 25: REPLACE

Input: Dos poblaciones de soluciones *new_pop* y *old_pop*, *matrix* de distancias
Output: Población *old_pop* reemplazada por *new_pop*
begin
  if *old_pop.best_fitness* > *new_pop.best_fitness* **then**
    new_pop.best_fitness ← old_pop.best_fitness
    pos ← worstSolution(*new_pop*)
    new_pop.solutions[pos] ← old_pop.solutions[old_pop.best_sol]
    new_pop.best_sol ← pos
  end
  old_pop ← new_pop
  return old_pop
end

La función principal que llama a todos estos operadores y representa el esquema de evolución es la

siguiente:

Algorithm 26: AGG

```
Input: matrix de distancias, número de elementos a seleccionar en una solución m
Output: Población evolucionada, obtenida tras varias iteraciones del algoritmo
begin
    evaluations ← 0
    generations ← 1
    size_pop ← 50
    mut_prob ← 0.1                          // Probabilidad de mutación por solución
    cross_prob ← 0.7                        // Probabilidad de cruce por solución
    old_pop ← randomPopulation(size_pop,m,matrix,evaluations)        // Partimos de
                                                                     // una población aleatoria
    while evaluations < 100000 do
        new_pop ← selection(old_pop)
        cross(new_pop)                      // Cruce uniforme o posicional, según el caso
        mutation(new_pop)
        // Evaluamos las soluciones de la nueva población
        new_pop ← evaluatePopulation(new_pop,matrix,evaluations)
        replace(old_pop,new_pop,matrix)
        generations ++
    end
    return old_pop
end
```

Como vemos, se parte de una población inicial de soluciones generada aleatoriamente y se van generando durante varias iteraciones nuevas poblaciones completas, mediante la aplicación de los operadores de cruce y mutación, que sustituyen a la población de la iteración anterior en cada caso gracias al operador de reemplazamiento. El algoritmo para cuando se han realizado 100000 evaluaciones de la función fitness. Puede ocurrir que se supere este número, pues no se realiza la comprobación $evaluations < 100000$ en mitad de una iteración, pero no en más de 40, ya que el máximo número de evaluaciones que se pueden llevar a cabo en una iteración es de 36(soluciones hijas) + 5 (mutaciones) = 41. Se imprime por pantalla el fitness de la mejor solución de la población obtenida en la última iteración, el tiempo en segundos que tarda en ejecutarse el algoritmo, el número de iteraciones (generaciones producidas) y el número de evaluaciones de la función objetivo (para comprobar que no se superan las 100040).

3.4. Algoritmos genéticos estacionarios

En estos algoritmos se seleccionan únicamente dos soluciones de entre una población dada (usando torneo binario), las cuales serán cruzadas posteriormente para generar dos nuevas soluciones que las sustituyen. Es posible que las dos soluciones seleccionadas sean la misma, en cuyo caso los hijos serán iguales al padre. La selección queda en este caso como sigue:

Algorithm 27: pairSelection

```
Input: Población de solucionespop
Output: Dos soluciones de pop
begin
    pos1 ← binaryCompetition(pop)
    pos2 ← binaryCompetition(pop)
    sol1 ← pop.solutions[pos1]
    sol2 ← pop.solutions[pos2]
    return sol1,sol2
end
```

Estas dos soluciones se cruzan (usando el operador de cruce basado en posición o el uniforme) y

generan dos hijos:

Algorithm 28: pairCross

```
Input: Dos solucinoes padre, p1 y p2
Output: Dos nuevas soluciones hijas, s1 y s2
begin
    s1 ← cross(p1, p2)                    // Cruce uniforme o posicional, según el caso
    s2 ← cross(p1, p2)
    return s1,s2
end
```

Para la mutación, en este caso sólo tenemos dos posibles soluciones que mutar, luego el número esperado de mutaciones en una generación será de $0{,}1/m \times 2 \times m = 0{,}1 \times 2 = 0{,}2$. Así, con probabilidad 0,2, se selecciona aleatoriamente una de las dos soluciones y se mutan dos valores aleatorios de la misma usando mutateSolution.

Algorithm 29: pairMutation

```
Input: Dos soluciones s1 y s2, probabilidad de mutación por solución mut_prob,
       matrix de distancias
begin
    if número aleatorio entre 0 y 1 < 2*mut_prob then
        pos ← número aleatorio 0 ó 1
        if pos = 0 then
            mutateSolution(s1,matrix)
        else
            mutateSolution(s2,matrix)
        end
    end
end
```

Como en este esquema sólo se generan dos nuevas soluciones en cada iteración y no una nueva población completa, el reemplazamiento consiste aquí en introducir en la población anterior las dos nuevas soluciones en las posiciones de las dos peores soluciones de dicha población, siempre y cuando estas nuevas soluciones sean mejores que las dos peores soluciones de la población (o sólo una es mejor). Es decir, las nuevas soluciones compiten con las dos peores de la población. Para determinar las posiciones de las dos peores soluciones se usa la función worstSolutions, cuyo pseudocódigo no es importante y no se

inlcuye en la memoria.

Algorithm 30: REPLACE

```
Input: Población pop, dos soluciones s1 y s2
Output: Población pop modificada
begin
    worst1, worst2 ← worstSolutions(pop) ;   // Índices de las dos peores soluciones de
                                             // la población siendo worst1 la peor de las dos
    worst_sol ← solución con menor fitness entre s1 y s2
    best_sol ← solución con mejor fitness entre s1 y s2
    // Si las soluciones s1 y s2 son mejores que las peores de la población,
    // las intercambiamos
    if pop[worst1].fitness < worst_sol.fitness and pop[worst2].fitness < best_sol.fitness then
        pop[worst1] ← worst_sol;
        pop[worst2] ← best_sol;
        // Si el fitness de la mejor solución introducida supera al de
        // la poblacion, actualizamos la mejor solución de la población
        if best_sol.fitness > pop.best_fitness then
            pop.best_fitness ← best_sol.fitness
            pop.best_sol ← worst2
        end
    else
        // La mejor solución es mejor que la peor de la población
        if pop[worst1].fitness < best_sol.fitness then
            pop[worst1] ← best_sol;
            if best_sol.fitness > pop.best_fitness then
                pop.best_fitness ← best_sol.fitness
                pop.best_sol ← worst2
            end
        end
    end
end
```

Ya podemos ver el pseudocódigo de la función principal:

Algorithm 31: AGE

```
Input: matrix de distancias, número de elementos a seleccionar en una solución m
Output: Población evolucionada, obtenida tras varias iteraciones del algoritmo
begin
    evaluations ← 0
    size_pop ← 50
    mut_prob ← 0.1                                  // Probabilidad de mutación por solución
    pop ← randomPopulation(size_pop,m,matrix,evaluations)                  // Partimos de
                                                                  // una población aleatoria
    while evaluations < 100000 do
        p1,p2 ← pairSelection(pop)
        s1,s2 ← pairCross(p1,p2)            // Cruce uniforme o posicional, según el caso
        pairMutation(s1,s2,mut_prob,matrix)
        // Evaluamos las dos soluciones obtenidas
        s1.fitness ← fitness(s1,matrix)
        s1.evaluated ← true
        s2.fitness ← fitness(s2,matrix)
        s2.evaluated ← true
        evaluations ← evaluations + 2
        replace(pop,s1,s2)
    end
    return pop
end
```

En este caso, sólo se puede superar el límite de evaluaciones de la función objetivo en 1. La función imprime por pantalla el fitness de la mejor solución de la población obtenida en la última iteración y el tiempo de ejecución en segundos.

3.5. Algoritmos meméticos

Estos algoritmos combinan el algoritmo genético generacional con operador de cruce uniforme (pues como veremos es el que presenta mejores resultados) y el algoritmo de la búsqueda local. Este último se aplicará cada 10 iteraciones(generaciones) del algoritmo genético a ciertas soluciones de la población obtenida en esa iteración.

Cabe destacar que es necesario transformar las soluciones en binario a soluciones de enteros y viceversa, pues cada uno de los algoritmos (AGG y BL) trabaja con una estructura diferente de solución. De esto se encargan las funciones BinToInt e IntToBin, respectivamente, cuyo pseudocódigo no mostramos por no tener interés.

Para estos algoritmos modificamos ligeramente el pseudocódigo de la búsqueda local, pues estableceremos que en cada ejecución de la misma no se superen las 400 evaluaciones de la función objetivo y además hay que tener en cuenta que en total no se pueden sobrepasar las 100000 evaluaciones. Así, consideramos dos variables: num_eval (número de evaluaciones en esa ejecución de BL) y evaluations (evaluaciones totales del fitness llevadas a cabo hasta ese momento por el algoritmo memético).

Algorithm 32: localSearch

Input: m: tamaño de solución, matriz: matriz de distancias,
evaluations: número de evaluaciones de la función objetivo,
sol: solución de partida para el algoritmo
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 < 400 and evaluations < 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 < 400 and
         evaluations < 100000 do
            mejora ← improvement(sol, min_pos, min_contrib.contrib, k, matriz)
             num_eval ← num_eval + 1
            evaluations ← evaluations + 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
end
```

Estudiamos distintas versiones de estos algoritmos, según las soluciones a las que se aplica búsqueda local.

Todos los algoritmos imprimen por pantalla el fitness de la mejor solución de la población obtenida en la última iteración, el tiempo en segundos que tarda en ejecutarse el algoritmo, el número de iteraciones (generaciones producidas) y el número de evaluaciones de la función objetivo.

3.5.1. AM-(10,1)

En esta versión se aplica la búsqueda local cada 10 generaciones a todas las soluciones de la población.

Algorithm 33: AM1

```
Input: matrix de distancias, número de elementos a seleccionar en una solución m
Output: Población evolucionada, obtenida tras varias iteraciones del algoritmo
begin
    evaluations ← 0
    generations ← 1
    size_pop ← 50
    mut_prob ← 0.1                          // Probabilidad de mutación por solución
    cross_prob ← 0.7                        // Probabilidad de cruce por solución
    old_pop ← randomPopulation(size_pop,m,matrix,evaluations)   // Partimos de
                                                                // una población aleatoria
    while evaluations < 100000 do
        new_pop ← selection(old_pop)
        cross(new_pop)                                          // Cruce uniforme
        mutation(new_pop)
        // Evaluamos las soluciones de la nueva población
        new_pop ← evaluatePopulation(new_pop,matrix,evaluations)
        // Se ejecuta cada 10 generaciones
        if generations mód 10 = 0 then
            for i ∈ [0, size_pop) and evaluations < 100000 do
                sol ← BinToInt(new_pop.solutions[i])    // Transformamos la solución
                                                        // de binario a enteros
                sol ← localSearch(matrix,sol,evaluations,m)
                new_pop.solutions[i] ← IntToBin(sol)    // Transformamos la solución
                                                        // de enteros a binario
                // Se actualiza la mejor solución de la nueva población
                updateBest(new_pop)
            end
        end
        replace(old_pop,new_pop,matrix)
        generations ++
    end
    return old_pop
end
```

3.5.2. AM-(10,0.1)

Ahora aplicamos la búsqueda local (cada 10 generaciones) a un subconjunto de soluciones aleatorias seleccionadas con probabilidad 0.1. Por tanto, el número de soluciones a las que se aplicará la búsqueda local será de $0{,}1 \times size\_pop = 5$. Lo que hacemos entonces es seleccionar aleatoriamente 5 soluciones de la población obtenida y aplicar sobre ellas la búsqueda local.

Algorithm 34: AM2

```
Input: matrix de distancias, número de elementos a seleccionar en una solución m
Output: Población evolucionada, obtenida tras varias iteraciones del algoritmo
begin
    evaluations ← 0
    generations ← 1
    size_pop ← 50
    mut_prob ← 0.1                          // Probabilidad de mutación por solución
    cross_prob ← 0.7                        // Probabilidad de cruce por solución
    num_local ← 0.1*size_pop                // Número de soluciones a las que se aplica BL
    old_pop ← randomPopulation(size_pop,m,matrix,evaluations)   // Partimos de
                                                                // una población aleatoria
    while evaluations < 100000 do
        new_pop ← selection(old_pop)
        cross(new_pop)                                          // Cruce uniforme
        mutation(new_pop)
        // Evaluamos las soluciones de la nueva población
        new_pop ← evaluatePopulation(new_pop,matrix,evaluations)
        // Se ejecuta cada 10 generaciones
        if generations mód 10 = 0 then
            for i ∈ [0, num_local) and evaluations < 100000 do
                rand_pos ← número aleatorio en [0, size_pop)
                sol ← BinToInt(new_pop.solutions[rand_pos])   // Transformamos la solución
                                                              // de binario a enteros
                sol ← localSearch(matrix,sol,evaluations,m)
                new_pop.solutions[rand_pos] ← IntToBin(sol)   // Transformamos la solución
                                                              // de enteros a binario
                // Se actualiza la mejor solución de la nueva población
                updateBest(new_pop)
            end
        end
        replace(old_pop,new_pop,matrix)
        generations ++
    end
    return old_pop
end
```

3.5.3. AM-(10,0.1mejores)

En esta ocasión consideramos las $0{,}1 \times size\_pop = 5$ mejores soluciones de la población para aplicarles la búsqueda local cada 10 generaciones. Para ello, se ordenan todas las soluciones de la población de mayor a menor fitness y aplica la búsqueda local a las 5 primeras.

Algorithm 35: AM3

Input: *matrix* de distancias, número de elementos a seleccionar en una solución m
Output: Población evolucionada, obtenida tras varias iteraciones del algoritmo

```
begin
    evaluations ← 0
    generations ← 1
    size_pop ← 50
    mut_prob ← 0.1                          // Probabilidad de mutación por solución
    cross_prob ← 0.7                        // Probabilidad de cruce por solución
    num_local ← 0.1*size_pop                // Número de soluciones a las que se aplica BL
    old_pop ← randomPopulation(size_pop,m,matrix,evaluations)   // Partimos de
                                                                // una población aleatoria
    while evaluations < 100000 do
        new_pop ← selection(old_pop)
        cross(new_pop)                                          // Cruce uniforme
        mutation(new_pop)
        // Evaluamos las soluciones de la nueva población
        new_pop ← evaluatePopulation(new_pop,matrix,evaluations)
        // Se ejecuta cada 10 generaciones
        if generations mód 10 = 0 then
            // Vector con parejas (fitness,posición) asociadas a cada solución de
               la población
            solutions ← ∅
            foreach i ∈ [0, size_pop) do
                solutions ← solutions∪{(new_pop[i].fitness, i)}
            end
            sort(solutions)          // Se ordenan las soluciones de mayor a menor fitness
            for j ∈ [0, num_local) and evaluations < 100000 do
                sol ← BinToInt(new_pop[solutions[j].second])    // Transformamos la solución
                                                                // de binario a enteros
                sol ← localSearch(matrix,sol,evaluations,m)
                new_pop[solutions[j].second] ← IntToBin(sol)    // Transformamos la solución
                                                                // de enteros a binario
            end
            // Se actualiza la mejor solución de la nueva población
            updateBest(new_pop)
        end
        replace(old_pop,new_pop,matrix)
        generations ++
    end
    return old_pop
end
```

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 , find y sort 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
- AGGPos → algoritmo genético generacional con operador de cruce basado en posición
- AGGPos-ternary → algoritmo genético generacional con operador de cruce basado en posición y selección por torneo de tamaño 3
- AGGPos-bestcross → algoritmo genético generacional con operador de cruce basado en posición modificado, donde siempre se cruza la mejor solución
- AGEPos → algoritmo genético estacionario con operador de cruce basado en posición
- AGEPos-ternary → algoritmo genético estacionario con operador de cruce basado en posición y selección por torneo de tamaño 3
- AGEUnif → algoritmo genético estacionario con operador de cruce uniforme
- AGGUnif → algoritmo genético generacional con operador de cruce uniforme
- AGGUnif-2 → algoritmo genético estacionario con operador de cruce uniforme modificado
- AM1 → primera versión de los algoritmos meméticos AM-(10,1)
- AM2 → segunda versión de los algoritmos meméticos AM-(10,0.1)
- AM2-2 → segunda versión de los algoritmos meméticos modificada AM-(1,0.1)
- AM3 → tercera versión de los algoritmos meméticos AM-(10,0.1mejores)
- AM3-peores → tercera versión de los algoritmos meméticos modificada AM-(10,0.1peores)

Todos muestran los resultados por pantalla en el formato *Fitness, Tiempo de ejecución (s)*, y para algunos algoritmos *Fitness, Tiempo de ejecución (s), Generaciones, Evaluaciones* (en los que ya hemos ido comentando) pero podemos redirigir la salida al fichero que queramos. Además, la semilla se le pasa como parámetro y leen los datos de la entrada estándar. Así, para ejecutar el algoritmo AM-(10,1) por ejemplo, con el fichero de datos de entrada $MDG-a\_1\_n500\_m50.txt$, con semilla 4 y con salida en el fichero *AM1.csv*, basta con escribir en consola la siguiente sentencia:

```
bin/AM1 4 < data/MDG−a_1_n500_m50.txt >> salida/AM1.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 de los script execute_p1.sh, execute_genetic.sh, execute_memetic.sh, execute_extras.sh, que se encargan de la ejecución de los algoritmos de la práctica 1, los algoritmos genéticos, los algoritmos meméticos y los algoritmos extra, respectivamente. La semilla se fija dentro de estos archivos 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 correspondiente.

Por otra parte, como era de esperar, el fichero makefile se encarga de la compilación. Al escribir en consola la orden make all se compilan todos los ficheros de código fuente. Con las órdenes make execute_p1, make execute_genetic, make execute_memetic y make execute_extras se compilan los ficheros necesarios y se ejecuta el script del mismo nombre.

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.

5.1.1. 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

5.1.2. Búsqueda local del primer mejor

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

5.1.3. AGG con operador de cruce basado en posición

Caso	Coste obtenido	Desv	Tiempo (s)
MDG-a_1_n500_m50	7640.3000	2.47	6.3353
MDG-a_2_n500_m50	7551.9700	2.83	6.2692
MDG-a_3_n500_m50	7574.3100	2.38	6.2851
MDG-a_4_n500_m50	7580.1600	2.45	6.1060
MDG-a_5_n500_m50	7443.9500	4.01	6.2518
MDG-a_6_n500_m50	7505.3600	3.45	6.4323
MDG-a_7_n500_m50	7585.5200	2.40	6.3148
MDG-a_8_n500_m50	7537.2600	2.76	6.2330
MDG-a_9_n500_m50	7541.3700	2.94	6.1885
MDG-a_10_n500_m50	7528.3700	3.24	6.1245
MDG-b_21_n2000_m200	11005496.3700	2.61	98.5493
MDG-b_22_n2000_m200	11021074.4900	2.35	96.2491
MDG-b_23_n2000_m200	11004088.4300	2.62	97.3611
MDG-b_24_n2000_m200	10999027.4100	2.58	96.8508
MDG-b_25_n2000_m200	11021001.9300	2.44	95.3023
MDG-b_26_n2000_m200	11009952.5400	2.50	96.0952
MDG-b_27_n2000_m200	11011009.4600	2.61	95.6791
MDG-b_28_n2000_m200	10969735.8000	2.75	96.8783
MDG-b_29_n2000_m200	10992141.5100	2.70	104.6651
MDG-b_30_n2000_m200	11001403.8500	2.61	108.2584
MDG-c_1_n3000_m300	24283029.0000	2.42	256.6622
MDG-c_2_n3000_m300	24295419.0000	2.45	255.5318
MDG-c_8_n3000_m400	42496401.0000	2.17	276.8820
MDG-c_9_n3000_m400	42454312.0000	2.26	275.5431
MDG-c_10_n3000_m400	42524681.0000	2.19	291.5733
MDG-c_13_n3000_m500	65724464.0000	1.92	311.0767
MDG-c_14_n3000_m500	65753905.0000	1.83	303.0759
MDG-c_15_n3000_m500	65892345.0000	1.64	306.4597
MDG-c_19_n3000_m600	94125235.0000	1.58	314.3004
MDG-c_20_n3000_m600	94163288.0000	1.55	324.7335

Tabla 3: Resultados para el algoritmo genético generacional con operador de cruce basado en posición

5.1.4. AGG con operador de cruce uniforme

Caso	Coste obtenido	Desv	Tiempo (s)
MDG-a_1_n500_m50	7599.7100	2.99	7.1859
MDG-a_2_n500_m50	7589.6700	2.34	7.2879
MDG-a_3_n500_m50	7586.2300	2.23	7.0329
MDG-a_4_n500_m50	7637.7500	1.71	7.2787
MDG-a_5_n500_m50	7598.8400	2.02	7.3794
MDG-a_6_n500_m50	7670.3100	1.33	7.4429
MDG-a_7_n500_m50	7696.2100	0.97	7.5719
MDG-a_8_n500_m50	7650.0500	1.30	7.1203
MDG-a_9_n500_m50	7735.8400	0.44	7.1916
MDG-a_10_n500_m50	7617.1300	2.10	7.3131
MDG-b_21_n2000_m200	11138998.9300	1.42	145.0090
MDG-b_22_n2000_m200	11165941.8900	1.07	149.0917
MDG-b_23_n2000_m200	11153701.8200	1.29	154.4304
MDG-b_24_n2000_m200	11144229.3900	1.30	150.2561
MDG-b_25_n2000_m200	11147015.0000	1.32	147.2957
MDG-b_26_n2000_m200	11119564.5900	1.53	152.2065
MDG-b_27_n2000_m200	11161459.9600	1.28	151.1558
MDG-b_28_n2000_m200	11109143.0300	1.51	146.3838
MDG-b_29_n2000_m200	11134650.7900	1.44	151.9206
MDG-b_30_n2000_m200	11141441.5700	1.37	148.1965
MDG-c_1_n3000_m300	24608485.0000	1.11	393.2946
MDG-c_2_n3000_m300	24632809.0000	1.09	392.2716
MDG-c_8_n3000_m400	42986288.0000	1.04	497.4414
MDG-c_9_n3000_m400	42961690.0000	1.10	459.4445
MDG-c_10_n3000_m400	42959967.0000	1.19	445.6626
MDG-c_13_n3000_m500	66406940.0000	0.91	521.6510
MDG-c_14_n3000_m500	66473974.0000	0.75	517.3427
MDG-c_15_n3000_m500	66441799.0000	0.82	517.5912
MDG-c_19_n3000_m600	94940170.0000	0.73	559.3752
MDG-c_20_n3000_m600	94887804.0000	0.79	581.2187

Tabla 4: Resultados para el algoritmo genético generacional con operador de cruce uniforme

5.1.5. AGE con operador de cruce basado en posición

Caso	Coste obtenido	Desv	Tiempo (s)
MDG-a_1_n500_m50	7584.0100	3.19	6.6380
MDG-a_2_n500_m50	7519.5400	3.24	6.5950
MDG-a_3_n500_m50	7525.6100	3.01	6.4858
MDG-a_4_n500_m50	7559.1700	2.72	6.7525
MDG-a_5_n500_m50	7521.1500	3.02	6.4310
MDG-a_6_n500_m50	7546.7500	2.92	6.4193
MDG-a_7_n500_m50	7472.5300	3.85	6.4477
MDG-a_8_n500_m50	7574.1800	2.28	6.5106
MDG-a_9_n500_m50	7625.6900	1.86	6.6205
MDG-a_10_n500_m50	7591.1900	2.43	6.4016
MDG-b_21_n2000_m200	10976951.1500	2.86	98.5636
MDG-b_22_n2000_m200	10970372.8900	2.80	97.9045
MDG-b_23_n2000_m200	11008968.4700	2.57	97.8373
MDG-b_24_n2000_m200	10988903.0900	2.67	96.8395
MDG-b_25_n2000_m200	10991849.5000	2.69	97.7293
MDG-b_26_n2000_m200	10990358.9800	2.67	97.4258
MDG-b_27_n2000_m200	10955690.4800	3.10	97.4216
MDG-b_28_n2000_m200	10963812.9700	2.80	98.1521
MDG-b_29_n2000_m200	10973169.1300	2.87	98.2144
MDG-b_30_n2000_m200	11015277.7800	2.49	97.1605
MDG-c_1_n3000_m300	24193096.0000	2.78	225.8768
MDG-c_2_n3000_m300	24202565.0000	2.82	228.9723
MDG-c_8_n3000_m400	42506675.0000	2.14	251.7806
MDG-c_9_n3000_m400	42500618.0000	2.16	250.6708
MDG-c_10_n3000_m400	42462823.0000	2.33	252.9400
MDG-c_13_n3000_m500	65843142.0000	1.75	277.9741
MDG-c_14_n3000_m500	65830741.0000	1.72	280.1346
MDG-c_15_n3000_m500	65788119.0000	1.80	277.2306
MDG-c_19_n3000_m600	94062218.0000	1.64	302.0920
MDG-c_20_n3000_m600	94007359.0000	1.71	309.6937

Tabla 5: Resultados para el algoritmo genético estacionario con operador de cruce basado en posición

5.1.6. AGE con operador de cruce uniforme

Caso	Coste obtenido	Desv	Tiempo (s)
MDG-a_1_n500_m50	7596.1600	3.03	7.3520
MDG-a_2_n500_m50	7601.1500	2.19	7.2460
MDG-a_3_n500_m50	7504.4300	3.29	7.1593
MDG-a_4_n500_m50	7660.1100	1.42	7.3927
MDG-a_5_n500_m50	7541.7200	2.75	7.4196
MDG-a_6_n500_m50	7596.2800	2.28	7.5483
MDG-a_7_n500_m50	7671.7800	1.29	7.5733
MDG-a_8_n500_m50	7607.7600	1.85	7.3822
MDG-a_9_n500_m50	7621.3900	1.91	7.5032
MDG-a_10_n500_m50	7639.1500	1.81	7.2798
MDG-b_21_n2000_m200	11063704.0100	2.09	131.2819
MDG-b_22_n2000_m200	11066523.8400	1.95	125.1071
MDG-b_23_n2000_m200	11104517.6600	1.73	129.8289
MDG-b_24_n2000_m200	11067495.0200	1.98	129.0687
MDG-b_25_n2000_m200	11078996.0600	1.92	127.6533
MDG-b_26_n2000_m200	11092375.3100	1.77	127.9010
MDG-b_27_n2000_m200	11049667.0000	2.26	123.8305
MDG-b_28_n2000_m200	11101019.8400	1.59	124.5623
MDG-b_29_n2000_m200	11067019.6800	2.04	125.0926
MDG-b_30_n2000_m200	11068015.0500	2.02	127.9364
MDG-c_1_n3000_m300	24482532.0000	1.61	310.7571
MDG-c_2_n3000_m300	24515288.0000	1.57	305.3671
MDG-c_8_n3000_m400	42854798.0000	1.34	389.7618
MDG-c_9_n3000_m400	42813615.0000	1.44	360.4363
MDG-c_10_n3000_m400	42780235.0000	1.60	350.1306
MDG-c_13_n3000_m500	66322030.0000	1.03	405.4928
MDG-c_14_n3000_m500	66216803.0000	1.14	387.8432
MDG-c_15_n3000_m500	66214576.0000	1.16	395.0711
MDG-c_19_n3000_m600	94685351.0000	0.99	455.7575
MDG-c_20_n3000_m600	94711478.0000	0.97	430.7895

Tabla 6: Resultados para el algoritmo genético estacionario con operador de cruce uniforme

5.1.7. AM-(10,1)

Caso	Coste obtenido	Desv	Tiempo (s)
MDG-a_1_n500_m50	7722.4400	1.42	0.7676
MDG-a_2_n500_m50	7667.9700	1.33	0.7692
MDG-a_3_n500_m50	7637.5400	1.57	0.7313
MDG-a_4_n500_m50	7710.4800	0.77	0.6419
MDG-a_5_n500_m50	7729.9300	0.33	0.6836
MDG-a_6_n500_m50	7660.9800	1.45	0.6801
MDG-a_7_n500_m50	7600.7700	2.20	0.6018
MDG-a_8_n500_m50	7652.4800	1.27	0.6230
MDG-a_9_n500_m50	7676.5200	1.20	0.6132
MDG-a_10_n500_m50	7704.9900	0.97	0.7335
MDG-b_21_n2000_m200	11096533.5500	1.80	34.1228
MDG-b_22_n2000_m200	11124752.3000	1.44	32.3317
MDG-b_23_n2000_m200	11102635.1800	1.75	33.2903
MDG-b_24_n2000_m200	11103623.4100	1.66	30.5755
MDG-b_25_n2000_m200	11129061.6100	1.48	31.8156
MDG-b_26_n2000_m200	11124099.6000	1.49	29.8145
MDG-b_27_n2000_m200	11111502.1800	1.72	30.3296
MDG-b_28_n2000_m200	11114466.8500	1.47	31.1305
MDG-b_29_n2000_m200	11131466.8500	1.47	29.1715
MDG-b_30_n2000_m200	11136221.6300	1.42	28.7041
MDG-c_1_n3000_m300	24517103.0000	1.47	98.6396
MDG-c_2_n3000_m300	24462365.0000	1.78	90.5566
MDG-c_8_n3000_m400	42858685.0000	1.33	112.5296
MDG-c_9_n3000_m400	42846734.0000	1.36	122.6820
MDG-c_10_n3000_m400	42768325.0000	1.63	120.0775
MDG-c_13_n3000_m500	66227422.0000	1.17	155.4455
MDG-c_14_n3000_m500	66212307.0000	1.15	141.3751
MDG-c_15_n3000_m500	66233469.0000	1.13	144.2372
MDG-c_19_n3000_m600	94539077.0000	1.14	154.6568
MDG-c_20_n3000_m600	94671428.0000	1.02	153.4724

Tabla 7: Resultados para la primera versión de los algoritmos meméticos

5.1.8. AM-(10,0.1)

Caso	Coste obtenido	Desv	Tiempo (s)
MDG-a_1_n500_m50	7674.1600	2.04	1.6272
MDG-a_2_n500_m50	7642.0900	1.67	1.4551
MDG-a_3_n500_m50	7577.3900	2.35	1.6110
MDG-a_4_n500_m50	7604.3900	2.13	1.5097
MDG-a_5_n500_m50	7612.3600	1.84	1.5642
MDG-a_6_n500_m50	7649.3800	1.60	1.5563
MDG-a_7_n500_m50	7628.8100	1.84	1.6205
MDG-a_8_n500_m50	7654.5600	1.24	1.8468
MDG-a_9_n500_m50	7718.5400	0.66	1.6992
MDG-a_10_n500_m50	7676.0400	1.34	1.7339
MDG-b_21_n2000_m200	11148288.0600	1.34	54.2756
MDG-b_22_n2000_m200	11196231.6200	0.80	58.4671
MDG-b_23_n2000_m200	11190341.2600	0.97	49.9339
MDG-b_24_n2000_m200	11189314.0100	0.90	67.4230
MDG-b_25_n2000_m200	11175032.0600	1.07	55.0096
MDG-b_26_n2000_m200	11211456.6900	0.72	52.8924
MDG-b_27_n2000_m200	11141507.0700	1.45	49.5905
MDG-b_28_n2000_m200	11179794.7500	0.89	60.4579
MDG-b_29_n2000_m200	11187212.9300	0.97	56.8724
MDG-b_30_n2000_m200	11167649.1800	1.14	46.4786
MDG-c_1_n3000_m300	24664724.0000	0.88	175.4363
MDG-c_2_n3000_m300	24657475.0000	1.00	173.2818
MDG-c_8_n3000_m400	43186536.0000	0.58	218.6578
MDG-c_9_n3000_m400	43103854.0000	0.77	200.8528
MDG-c_10_n3000_m400	43108976.0000	0.84	194.1992
MDG-c_13_n3000_m500	66673897.0000	0.51	265.6807
MDG-c_14_n3000_m500	66729937.0000	0.37	221.2810
MDG-c_15_n3000_m500	66768750.0000	0.33	271.5697
MDG-c_19_n3000_m600	95225625.0000	0.43	264.0697
MDG-c_20_n3000_m600	95265428.0000	0.40	283.2550

Tabla 8: Resultados para la segunda versión de los algoritmos meméticos

5.1.9. AM-(10,0.1mejores)

Caso	Coste obtenido	Desv	Tiempo (s)
MDG-a_1_n500_m50	7626.3900	2.65	1.6267
MDG-a_2_n500_m50	7574.5900	2.54	1.4842
MDG-a_3_n500_m50	7624.3300	1.74	1.6898
MDG-a_4_n500_m50	7587.0800	2.36	1.4680
MDG-a_5_n500_m50	7585.1400	2.19	1.6609
MDG-a_6_n500_m50	7667.9700	1.36	1.5686
MDG-a_7_n500_m50	7590.9000	2.33	1.5818
MDG-a_8_n500_m50	7633.1500	1.52	1.5919
MDG-a_9_n500_m50	7673.6500	1.24	1.5337
MDG-a_10_n500_m50	7654.0400	1.62	1.5922
MDG-b_21_n2000_m200	11143376.4400	1.39	50.2011
MDG-b_22_n2000_m200	11124002.9300	1.44	53.8830
MDG-b_23_n2000_m200	11172449.3600	1.13	46.3347
MDG-b_24_n2000_m200	11177988.2400	1.00	58.6131
MDG-b_25_n2000_m200	11147166.6900	1.32	45.8337
MDG-b_26_n2000_m200	11173896.9900	1.05	51.3627
MDG-b_27_n2000_m200	11142883.3400	1.44	50.5609
MDG-b_28_n2000_m200	11139484.6500	1.24	45.0440
MDG-b_29_n2000_m200	11193433.8200	0.92	55.9717
MDG-b_30_n2000_m200	11190439.5100	0.94	55.3895
MDG-c_1_n3000_m300	24715984.0000	0.68	193.0204
MDG-c_2_n3000_m300	24658766.0000	0.99	173.0503
MDG-c_8_n3000_m400	43097761.0000	0.78	207.3779
MDG-c_9_n3000_m400	43107079.0000	0.76	212.5781
MDG-c_10_n3000_m400	43135160.0000	0.78	203.9789
MDG-c_13_n3000_m500	66627115.0000	0.58	239.6138
MDG-c_14_n3000_m500	66705078.0000	0.41	267.5730
MDG-c_15_n3000_m500	66651289.0000	0.51	241.3988
MDG-c_19_n3000_m600	95200194.0000	0.45	255.6300
MDG-c_20_n3000_m600	95269047.0000	0.39	256.6493

Tabla 9: Resultados para la tercera versión de los algoritmos meméticos

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 algoritmos:

Algoritmo	**Desv**	**Tiempo (s)**
Greedy	9.22	1.2
BL del Primer Mejor	1.22	0.55
AGG con operador basado en posición	2.49	132.14
AGG con operador uniforme	1.35	215.13
AGE con operador basado en posición	2.56	123.33
AGE con operador uniforme	1.80	171.25
AM-(10,1)	1.38	53.73
AM-(10,0.1)	1.10	94.53
AM-(10,0.1mejores)	1.26	92.66

Tabla 10: 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 observamos la Tabla 10. Nos damos cuenta de que los algoritmos genéticos no consiguen superar los resultados que presenta la búsqueda local, pues esta tiene una desviación media y un tiempo de ejecución más bajo. Además el algortimo Greedy sigue siendo el peor de todos los algoritmos, pues, como ya dijimos en su momento, este no explora el espacio de soluciones con tanta profundidad como lo hacen el resto de algoritmos considerados. Por otro lado, los algoritmos meméticos ofrecen mejores desviaciones medias que los genéticos, y unos tiempos de ejecución medios también menores, uno de los cuales, AM-(10,0.1), mejora incluso los resultados de la BL. Comentamos estos aspectos con más detalle a continuación.

Empezamos analizando los resultados de los **algoritmos genéticos**. Notamos que tanto en el esquema estacionario como en el generacional, se obtienen mejores soluciones usando el operador de cruce uniforme que el operador de cruce basado en posición. En ambos operadores de cruce, los valores comunes de las dos soluciones padre se mantienen en el hijo, de manera que los elementos seleccionados prometedores se mantienen y, si las soluciones padre son buenas, es bastante probable que el hijo también sea una buena solución. Sin embargo, mientras que en el operador de cruce basado en posición el resto de posiciones del hijo se rellenan con los valores restantes de un padre barajados aleatoriamente, con el operador uniforme se introducen valores totalmente aleatorios en dichas posiciones, lo que hace que la solución no sea factible necesariamente. Se usa entonces el operador de reparación, que transforma una solución no válida en una solución factible y es gracias a este operador por el que los resultados mejoran aquí. En efecto, ya vimos que lo que hace la reparación es añadir a la solución hija, en caso de que a esta le falten elementos seleccionados, el elemento no elegido que más contribuye a esa solución, con lo que esta mejorará considerablemente. Además, en el caso de que sobren elementos en la solución, se elimina de la misma el elemento que más contribuye, lo cual, por otro lado, evita que la solución se estanque en algún óptimo local, y amplía el espacio de búsqueda. Así, las soluciones hijas con el operador de cruce uniforme serán más prometedoras y la población descendiente tendrá una mayor diversidad, es decir, se explora el espacio de soluciones en mayor profundidad. De ahí el hecho de que obtengamos mejores soluciones con este tipo de operador de cruce.

En cuanto al tiempo, el operador de reparación conlleva la búsqueda de los elementos que más contribuyen a la solución para cada una de las soluciones hija generadas en el cruce, y esto es costoso, por lo que es de esperar que el tiempo medio de ejecución de los algoritmos que usan este operador sea más elevado, tal y como vemos en la tabla 10.

Por otra parte, vemos que los algoritmos genéticos estacionarios presentan soluciones ligeramente peores que los generacionales. Esto puede ser debido a que en el esquema estacionario en cada iteración (generación) se generan como mucho únicamente dos nuevas soluciones, que pueden o no pasar a formar

parte de la población de la generación anterior (es posible incluso que la población se mantenga invariante), mientras que en el esquema generacional, en cada generación puede llegar a modificarse la población entera, de manera que la exploración de soluciones es más profunda en este último esquema. Por tanto, esto propicia que las soluciones obtenidas por el algoritmo generacional sean mejores y más diversas que las encontradas por el estacionario. Además, el esquema generacional necesita menos generaciones que el estacionario para llegar a encontrar buenas soluciones.

Analizamos ahora las distintas versiones de los **algoritmos meméticos**. Podemos observar que estos algoritmos presentan resultados muy buenos, mejores que los algoritmos genéticos, aunque aparecen diferencias entre los resultados de las tres versiones consideradas.

Puesto que los algoritmos meméticos hacen uso del AGG uniforme, que es el mejor de los algoritmos genéticos estudiados para este problema, y además mejoran las soluciones con la BL, era de esperar que estos algoritmos mejoraran los resultados de los algoritmos genéticos en general. Además, su tiempo medio de ejecución es bastante inferior, ya que en la BL la evaluación de las soluciones (que es lo más costoso de los algoritmos) se hace de forma factorizada, y esta lleva a cabo gran parte de las evaluaciones totales, como ahora comentaremos.

La versión que ofrece una desviación media más alta es la primera, donde se aplica la búsqueda local a todas las soluciones de la población obtenida cada 10 generaciones. Este hecho es debido a que en el algoritmo genético sólo se generan 51 poblaciones (contando la primera generada aleatoriamente). En efecto, como cada 10 generaciones se usa BL sobre cada solución durante un máximo de 400 evaluaciones de la función objetivo, al disponer de 50 soluciones en cada población, se llevarán a cabo 20000 evaluaciones en la búsqueda local, lo cual supone $\frac{1}{5}$ del límite de evaluaciones. Es decir, cada 10 generaciones se realiza un quinto del número total de evaluaciones, lo cual nos lleva a las 50 poblaciones generadas. De esta forma, no hay un buen equilibrio entre exploración y explotación, pues hay una gran explotación por parte de la búsqueda local (que se aplica sobre todas las soluciones) y la exploración llevada a cabo por el algoritmo genético (búsqueda global) es muy pequeña. Esto da lugar a que las soluciones queden atrapadas en óptimos locales, y que, por tanto, sean peores, ya que la exploración del espacio total de soluciones se ve disminuida.

El hecho explicado también da lugar a que la primera versión de los algoritmos meméticos sea la más rápida, pues un quinto de las evaluaciones totales de la función fitness se realizan de manera factorizada en la búsqueda local, y, por lo tanto, más rápida que en el algoritmo genético.

Notamos que esta versión no mejora (las otras dos sí) la desviación obtenida por el algoritmo AGG uniforme, que era la menor de los algoritmos genéticos, a pesar de incluir al mismo. Esto puede ser debido a la poca profundidad de exploración de esta versión del algoritmo memético frente al genético uniforme, que sí genera más poblaciones y por tanto explora más el espacio de búsqueda. En cambio, sí que supera las desviaciones medias del resto de algoritmos genéticos, por los motivos ya comentados.

En la segunda y tercera versión de los algoritmos meméticos, se generan un total de 431 poblaciones (incluyendo la población aleatoria inicial), ya que sólo se aplica la búsqueda local a
$0{,}1 \times size\_pop = 0{,}1 \times 50 = 5$ soluciones cada 10 generaciones. Por lo tanto, en estos casos hay un mayor equilibrio entre exploración y explotación, siendo la exploración del espacio de soluciones en estas versiones mayor que en la versión primera. De ahí que la desviación media proporcionada por estas dos variantes sea mejor que la de la primera.

Por otro lado, notamos que la segunda variante presenta un mejor resultado que la tercera. Esto se debe a que en la tercera versión sólo se mejoran con BL las mejores soluciones de la población, con lo que estas pueden caer en óptimos locales y llegar a un punto donde la búsqueda local no las pueda mejorar más. En cambio, la segunda versión mejora con BL cualquier solución aleatoria, tanto buenas como malas soluciones, luego las soluciones malas tienen una mayor probabilidad de ser mejoradas. Así se obtienen soluciones más diversas, se lleva a cabo una mayor exploración del espacio de soluciones y se evita que las mejores soluciones se queden atrapadas en óptimos locales y no se pueda seguir explorando su entorno, que podría ser prometedor. Sin embargo, cuando se encuentra un óptimo local bueno, la solución puede ser adecuada y la convergencia a ese óptimo se produciría bastante rápido gracias a la aplicación de la búsqueda local sólo a las mejores soluciones.

Estas dos últimas versiones tienen un tiempo medio de ejecución parecido y mayor al de la primera

versión, pues ahora se llevan a cabo en la búsqueda local muchas menos evaluaciones que antes (aunque siguen siendo bastantes), siendo una gran parte de las soluciones evaluadas por el algoritmo genético, que es más lento. Dado que en la tercera versión se tienen que ordenar las soluciones de la población según su fitness, su tiempo medio de ejecución debería ser algo más elevado, pero no es el caso (por ejemplo porque el ordenador tuviera menos carga cuando se ejecutó este algoritmo, por la función `sort()` usada para llevar a cabo la ordenación, etc).

Finalmente, comparamos todos estos nuevos algoritmos implementados en la práctica 2 con **la búsqueda local**. Podemos ver en la Tabla 10 que sólo la segunda versión de los algoritmos meméticos consigue mejorar la desviación media que presenta la búsqueda local. El resto de algoritmos tienen una desviación media mayor que la BL.

Los algoritmos genéticos llevan a cabo una exploración global del espacio de soluciones, mientras que la búsqueda local, como su nombre indica, sólo explora locamente el entorno de las soluciones. Así, con la búsqueda local se pueden llegar a mejorar bastante las soluciones de partida, mientras que con los algoritmos genéticos se va saltando de un sitio a otro del espacio, sin llegar a explotar lo suficiente el entorno de las soluciones, lo cual hace que estos puedan quedar atascados en soluciones no muy buenas. Como la exploración con los algoritmos genéticos es más profunda, quizás necesitan un mayor número de iteraciones para converger a una solución tan buena como la encontrada por BL. La convergencia en la BL a una solución buena es más rápida que en los algortimos genéticos.

Al incluir la búsqueda local en los algoritmos genéticos, se aumenta la explotación del entorno de las soluciones, lo cual hace que se puedan llegar a encontrar soluciones buenas más rápidamente, de manera que los algoritmos meméticos encuentran mejores soluciones en general que los genéticos, como ya hemos comentado. La segunda versión de los meméticos presenta un buen equilibrio entre exploración global del entorno, gracias al algoritmo genético, y explotación, gracias a la búsqueda local. Así, lleva a cabo una exploración del entorno más amplia que la BL y es por ello que este algoritmo consigue mejorar la desviación media ofrecida por BL.

Respecto al tiempo medio de ejecución, la búsqueda local es muchísimo más rápida que todos los algoritmos genéticos y meméticos. Esto es debido principalmente al hecho, ya comentado, de que la evaluación de las soluciones en la BL se hace de forma factorizada, mientras que en los algoritmos genéticos se calcula el fitness de cada solución completa, y es aquí donde se necesita el mayor tiempo. Además, al usar representación binaria de las soluciones en los algoritmos genéticos, para recorrer las soluciones hay que iterar sobre un vector de longitud mucho mayor que en la búsqueda local, donde la representación es con enteros y las soluciones tienen longitud m (frente a n en la representación binaria).

Queda claro entonces que una estrategia de tipo evolutivo en nuestro problema quizás no merece mucho la pena, pues obtenemos tiempos de ejecución mucho mayores y no se consigue mejorar las soluciones en general. Sin embargo, puede que para otros problema, otros conjuntos de datos mayores y un límite de evaluaciones mayor, este tipo de algoritmos ofrezcan mejores resultados que la BL, pero no es nuestro caso. Si lo que se busca es obtener las mejores soluciones y el tiempo no es un factor importante, una buena estrategia a seguir sería hacer uso de los algoritmos meméticos, buscando un buen equilibrio entre exploración y explotación que permita mejorar los resultados, tal y como hemos explicado.

6. Algoritmos extra

6.1. AGG con operador de cruce uniforme modificado

Como primera modificación de los algoritmos estudiados, vamos a considerar el algoritmo genético generacional con operador de cruce uniforme. Lo que haremos es cambiar la función de reparación usada en el operador de cruce para que, si sobran elementos seleccionados en la solución generada, en vez de eliminar el elemento que más contribuye a la solución se elimine el que menos contribuye, dando lugar así a soluciones más prometedoras. El pseudocódigo del nuevo operador de reparación es prácticamente el mismo que el del operador antiguo, quedando como sigue:

Algorithm 36: REPAIR

Input: solución *sol*, tamaño de una solución m, *matrix* de distancias
Output: solución *sol* reparada
begin
  selected ← nº de posiciones con valor true en *sol*
  while $selected > m$ **do**
    min_pos ← posición de *sol* con valor $true$ de menor contribución
    sol[min_pos]← false
    selected - -
  end
  while $selected < m$ **do**
    max_pos ← posición de *sol* con valor $false$ que más contribuiría a la solución
    sol[max_pos]← true
    selected ++
  end
  return sol
end

Al aplicar este nuevo algoritmo con el operador de reparación modificado, los resultados que obtenemos son los siguientes:

Caso	Coste obtenido	Desv	Tiempo (s)
MDG-a_1_n500_m50	7670.83	2.08	6.877278
MDG-a_2_n500_m50	7627.58	1.85	6.943295
MDG-a_3_n500_m50	7564.49	2.51	6.660707
MDG-a_4_n500_m50	7716.84	0.69	6.865681
MDG-a_5_n500_m50	7625.28	1.68	6.703208
MDG-a_6_n500_m50	7618.7	1.99	6.668023
MDG-a_7_n500_m50	7546.63	2.90	6.814342
MDG-a_8_n500_m50	7616.9	1.73	6.973793
MDG-a_9_n500_m50	7640.99	1.66	6.741009
MDG-a_10_n500_m50	7641.31	1.79	6.763375
MDG-b_21_n2000_m200	11197487.25	0.91	132.438107
MDG-b_22_n2000_m200	11154821.81	1.17	131.047321
MDG-b_23_n2000_m200	11184296.84	1.02	134.339525
MDG-b_24_n2000_m200	11176067.62	1.02	130.073282
MDG-b_25_n2000_m200	11143099.54	1.35	133.95065
MDG-b_26_n2000_m200	11191251.7	0.89	133.005439
MDG-b_27_n2000_m200	11178058.86	1.13	136.016861
MDG-b_28_n2000_m200	11145015.08	1.20	130.780709
MDG-b_29_n2000_m200	11154197.47	1.27	128.175937
MDG-b_30_n2000_m200	11186315.9	0.97	130.798987
MDG-c_1_n3000_m300	24651168	0.94	338.496244
MDG-c_2_n3000_m300	24660725	0.98	347.561528
MDG-c_8_n3000_m400	43148965	0.66	387.585997
MDG-c_9_n3000_m400	43136394	0.69	393.055177
MDG-c_10_n3000_m400	43212255	0.61	396.471386
MDG-c_13_n3000_m500	66716093	0.44	455.939878
MDG-c_14_n3000_m500	66608718	0.55	415.73203
MDG-c_15_n3000_m500	66662601	0.49	444.644123
MDG-c_19_n3000_m600	95291574	0.36	509.55044
MDG-c_20_n3000_m600	95179741	0.48	516.911111

Tabla 11: Resultados para AGG con operador de cruce uniforme modificado

Algoritmo	Desv	Tiempo (s)
BL del Primer Mejor	1.22	0.55
AGG con operador uniforme	1.35	215.13
AGG con operador uniforme modificado	1.20	186.49

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

Podemos observar que la desviación típica media mejora con respecto al otro operador de reparación, incluso supera ligeramente a la búsqueda local. Esto es debido a que en el operador de reparación se mejoran todas las soluciones que no son facitibles, eliminando el peor elemento o añadiendo el mejor, mientras que en el caso anterior podíamos llegar a empeorar las soluciones si se eliminaba el elemento que más contribuía a la solución. Al mejorar las soluciones, estamos incluyendo información del entorno de las mismas, de forma parecida a como lo hace la búsqueda local, aumentando por lo tanto la explotación del entorno de las soluciones en el algoritmo genético. Así, las soluciones de las poblaciones hijas serán mejores que las obtenidas con el antiguo operador de reparación. Podría ser que de esta forma perdiéramos diversidad en las soluciones y fuera más probable quedarse estancado en un óptimo local, pero nos damos cuenta de que no es un problema, pues aquí no se mejora la solución completa como en la búsqueda local quedándonos con la mejor del entorno, sino que solamente se le añade o elimina el elemento más adecuado en cada caso.

En cuanto al tiempo de ejecución, debería ser el mismo en ambos reparadores, pues simplemente se busca el elemento que más o que menos contribuye, lo cual tiene el mismo coste. La diferencia en los tiempos puede ser debida, por ejemplo, a la carga del ordenador en el momento de la ejecución.

6.2. AG-Pos con selección por torneo de tamaño 3

Consideramos ahora los algoritmos genéticos con operador de cruce basado en posición y una estrategia de selección por torneo de tamaño 3, a diferencia del torneo binario que venimos usando hasta ahora.

La única modificación en la implementación de estos algoritmos es entonces la función que lleva a cabo el torneo, que quedaría como sigue:

Algorithm 37: TernaryCompetition

Input: Población *pop*
Output: Posición en la población de la solución con mayor fitness de entre tres elegidas aleatoriamente

```
begin
    rands ← ∅                                          // Vector de números aleatorios
    best_fitness ← 0          // Mejor fitness encontrado en las soluciones elegidas
    for i ∈ [0, 3) do
        rands ← rands∪ {número aleatorio en [0, size_pop)}
        if pop[rands[i]].fitness > best_fitness then
            best_fitness ← pop[rands[i]].fitness
            best_pos ← rands[i]
        end
    end
    return best_pos
end
```

Notemos que esta función podría cambiarse para obtener un torneo de cualquier tamaño, simplemente cambiando el 3 por el tamaño deseado.

Usamos esta nueva estrategia de selección tanto en el esquema generacional como en el estacionario y los resultados obtenidos han sido los siguientes:

Caso	Coste obtenido	Desv	Tiempo (s)
MDG-a_1_n500_m50	7545.96	3.67	9.191111
MDG-a_2_n500_m50	7550.21	2.85	8.887951
MDG-a_3_n500_m50	7530.63	2.95	8.807654
MDG-a_4_n500_m50	7552.63	2.80	9.93795
MDG-a_5_n500_m50	7597.96	2.03	8.902931
MDG-a_6_n500_m50	7587.97	2.39	9.11007
MDG-a_7_n500_m50	7524.53	3.18	8.806744
MDG-a_8_n500_m50	7553.94	2.54	9.514392
MDG-a_9_n500_m50	7604.87	2.13	9.424579
MDG-a_10_n500_m50	7609.03	2.20	9.421083
MDG-b_21_n2000_m200	11038817.09	2.31	142.971788
MDG-b_22_n2000_m200	11025787.57	2.31	122.799947
MDG-b_23_n2000_m200	11019831.79	2.48	113.369495
MDG-b_24_n2000_m200	10988812.31	2.68	114.905014
MDG-b_25_n2000_m200	11020558.9	2.44	120.705959
MDG-b_26_n2000_m200	11010339.81	2.50	122.609482
MDG-b_27_n2000_m200	11047254.31	2.29	121.711127
MDG-b_28_n2000_m200	11027101.55	2.24	116.459197
MDG-b_29_n2000_m200	11040806.3	2.27	126.288953
MDG-b_30_n2000_m200	10978083.01	2.82	118.154906
MDG-c_1_n3000_m300	24306954	2.32	284.28966
MDG-c_2_n3000_m300	24312000	2.38	249.065336
MDG-c_8_n3000_m400	42584567	1.96	273.333283
MDG-c_9_n3000_m400	42575129	1.99	277.347906
MDG-c_10_n3000_m400	42637461	1.93	277.229978
MDG-c_13_n3000_m500	65896122	1.67	298.819363
MDG-c_14_n3000_m500	65842012	1.70	298.331112
MDG-c_15_n3000_m500	65905082	1.62	305.36861
MDG-c_19_n3000_m600	94246603	1.45	331.271994
MDG-c_20_n3000_m600	94125103	1.59	328.069765

Tabla 13: Resultados para AGG con operador de cruce basado en posición y torneo de tamaño 3

Caso	Coste obtenido	Desv	Tiempo (s)
MDG-a_1_n500_m50	7517.01	4.04	6.175289
MDG-a_2_n500_m50	7550.09	2.85	6.392706
MDG-a_3_n500_m50	7613.57	1.88	6.393504
MDG-a_4_n500_m50	7454.46	4.06	6.539565
MDG-a_5_n500_m50	7567.45	2.42	6.18279
MDG-a_6_n500_m50	7558.2	2.77	6.5125
MDG-a_7_n500_m50	7523.2	3.20	6.112938
MDG-a_8_n500_m50	7506.01	3.16	6.215489
MDG-a_9_n500_m50	7600.47	2.18	6.172668
MDG-a_10_n500_m50	7559.58	2.84	6.103371
MDG-b_21_n2000_m200	10981495.99	2.82	94.098611
MDG-b_22_n2000_m200	11001285.96	2.53	93.097359
MDG-b_23_n2000_m200	11012018.42	2.55	95.828485
MDG-b_24_n2000_m200	11025219.82	2.35	93.02934
MDG-b_25_n2000_m200	11006108.76	2.57	92.209933
MDG-b_26_n2000_m200	10999599.72	2.59	94.014263
MDG-b_27_n2000_m200	11031343.3	2.43	93.193665
MDG-b_28_n2000_m200	10986255.21	2.60	94.779364
MDG-b_29_n2000_m200	10996816.08	2.66	93.525412
MDG-b_30_n2000_m200	10960514.28	2.97	93.040551
MDG-c_1_n3000_m300	24276498	2.44	216.521896
MDG-c_2_n3000_m300	24233822	2.70	219.760588
MDG-c_8_n3000_m400	42487194	2.19	241.821356
MDG-c_9_n3000_m400	42499354	2.16	240.105493
MDG-c_10_n3000_m400	42510291	2.22	242.117392
MDG-c_13_n3000_m500	65840571	1.75	268.763779
MDG-c_14_n3000_m500	65802515	1.76	262.91381
MDG-c_15_n3000_m500	65871801	1.67	265.624341
MDG-c_19_n3000_m600	94093687	1.61	283.989535
MDG-c_20_n3000_m600	94095250	1.62	286.013736

Tabla 14: Resultados para AGE con operador de cruce basado en posición y torneo con tamaño 3

Algoritmo	Desv	Tiempo (s)
AGG-Pos	2.49	132.14
AGG-Pos con selección por torneo de tamaño 3	2.32	141.17
AGE-Pos	2.56	123.33
AGE-Pos con selección por torneo de tamaño 3	2.52	117.57

Tabla 15: Comparativa de estadísticos medios

Nos damos cuenta de que las desviaciones medias mejoran ligeramente con esta nueva estrategia, pero la diferencia no es muy grande. Si nos fijamos en las tablas 12 y 3, por ejemplo, vemos que para algunos casos, como el primero y el último, el torneo binario presenta una menor desviación que el 'ternario', y para otros casos, como MDG-a-5 y MDG-c-8, es el ternario el que encuentra mejores soluciones.

Al aumentar el tamaño del torneo, lo que hacemos es poner más presión selectiva, de manera que sólo los individuos relativamente buenos tienen posibilidades de ser elegidos para el cruce (los dos peores nunca serán elegidos si el tamaño es 3), habiendo así una mayor influencia del elitismo. Esto disminuiría la diversidad de las generaciones posteriores, limitando más la exploración del espacio. Sin embargo, vemos que al dar una mayor oportunidad de reproducción a los individuos mejores, las soluciones mejoran en algunos casos, pues puede que aquí se encuentren zonas 'buenas' del espacio y al centrar ahí la búsqueda se llega a soluciones mejores.

La diferencia, sin embargo, entre las dos estrategias no es muy significativa, aunque es posible que si se aumentara aún más la presión selectiva la diferencia se hiciera más notoria. Habría que encontrar un tamaño de torneo adecuado para cada caso.

Los tiempos medios de ejecución son parecidos en todos los casos, por lo que el tamaño del torneo no afecta demasiado al tiempo de ejecución.

6.3. AGG con operador de cruce basado en posición modificado

En esta ocasión modificamos el cruce, de manera que siempre es la mejor solución de la población actual la que se cruza con una solución aleatoria de la población seleccionada y el hijo sustituye a esa solución aleatoria. De esta forma garantizamos que en la población hija va a haber algunas soluciones parecidas a la mejor solución de la población (tendrán valores en común con la misma), conservando así las selecciones prometedoras de la mejor solución en las poblaciones descendientes. El pseudocódigo para el cruce sería el siguiente:

Algorithm 38: CROSS

Input: Población *pop*, probabilidad de cruce por pareja *cross_prob*
begin
  num_cross ← $cross\_prob \times pop.size()/2$ `// Número esperado de cruces`
  foreach $i \in \{0, ..., num\_cross - 1\}$ **do**
    updateBest(pop) `// Se actualiza la mejor solución de la población pop`
    sol ← positionalCross($pop.solutions[pop.best\_sol], pop.solutions[2i]$)
    $pop.solutions[2i]$ ← sol
  end
end

Notamos que ahora sólo se genera un hijo como resulado de cada cruce, por lo que habrá menos soluciones hijas en la población descendiente.

Como las soluciones en la población seleccionada tienen posiciones aleatorias, en lugar de la posición 2i podríamos haber elegido la posición i en cada caso como segunda solución padre, pero esto no influiría en los resultados, por lo que se toma esa posición por similitud con el cruce usado en las otras versiones.

Los resultados obtenidos se muestran a continuación:

Caso	Coste obtenido	Desv	Tiempo (s)
MDG-a_1_n500_m50	7628.63	2.62	8.84832
MDG-a_2_n500_m50	7577.52	2.50	9.012987
MDG-a_3_n500_m50	7545.23	2.76	8.407234
MDG-a_4_n500_m50	7650.27	1.54	8.763967
MDG-a_5_n500_m50	7494.75	3.36	8.104967
MDG-a_6_n500_m50	7565.79	2.67	8.307673
MDG-a_7_n500_m50	7637.2	1.73	8.898012
MDG-a_8_n500_m50	7651.5	1.28	8.403199
MDG-a_9_n500_m50	7654.2	1.49	8.693488
MDG-a_10_n500_m50	7570.01	2.70	9.161015
MDG-b_21_n2000_m200	11063133.82	2.10	135.161278
MDG-b_22_n2000_m200	11077643.48	1.85	115.267638
MDG-b_23_n2000_m200	11071537.47	2.02	118.135294
MDG-b_24_n2000_m200	11079430.97	1.87	107.786727
MDG-b_25_n2000_m200	11116054.38	1.59	108.000094
MDG-b_26_n2000_m200	11085635.31	1.83	120.490777
MDG-b_27_n2000_m200	11100802.76	1.81	114.916376
MDG-b_28_n2000_m200	11082809.4	1.75	114.004996
MDG-b_29_n2000_m200	11042428.96	2.26	118.649995
MDG-b_30_n2000_m200	11051476.44	2.17	113.759387
MDG-c_1_n3000_m300	24399734	1.95	264.431763
MDG-c_2_n3000_m300	24414094	1.97	243.683412
MDG-c_8_n3000_m400	42767463	1.54	262.149503
MDG-c_9_n3000_m400	42755035	1.57	262.957088
MDG-c_10_n3000_m400	42778424	1.61	261.529897
MDG-c_13_n3000_m500	66204849	1.21	280.913845
MDG-c_14_n3000_m500	66199775	1.16	282.926002
MDG-c_15_n3000_m500	66195205	1.19	289.012164
MDG-c_19_n3000_m600	94512280	1.17	330.327593
MDG-c_20_n3000_m600	94675696	1.01	306.29762

Tabla 16: Resultados para AGG con operador de cruce basado en posición modificado

Algoritmo	Desv	Tiempo (s)
AGG-Pos	2.49	132.14
AGG-Pos con selección por torneo de tamaño 3	2.32	141.17
AGG con operador posicional modificado	1.88	134.17
AGG con operador uniforme	1.35	215.13

Tabla 17: Comparativa de estadísticos medios

Nos damos cuenta de que esta modificación presenta resultados buenos, mejorando las soluciones que ofrecen los otros algoritmos genéticos generacionales que usan el cruce basado en posición, pero no llega a superar al algoritmo que usa el operador de cruce uniforme.

Este hecho se debe a que las soluciones de las poblaciones hijas van a tener elementos seleccionados buenos, al ser siempre la mejor solución de la población la que toma parte en el cruce. Así, las soluciones mejoran considerablemente y en cada generación se producirán soluciones aún mejores, al ir añadiéndose en cada caso las posiciones prometedoras a las soluciones. Cabe notar que eso no disminuye la diversidad de la población tanto como sería de esperar, pues, debido a que la probabilidad de cruce es de 0.7, no serán todas las soluciones de las poblaciones descendientes hijas de la mejor solución. Además, el hecho de que la mejor solución se cruce con cualquier otra aleatoria, también introduce diversidad.

El tiempo de ejecución es parecido para todos los AGG con operador de cruce basado en posición, lo cual era de esperar.

6.4. AM-(1,0.1)

Modificamos la segunda versión estudiada de los aloritmos meméticos, aplicando ahora la búsqueda local en todas las generaciones a 5 individuos elegidos aleatoriamente. Así, el pseudocódigo es exactamente el mismo que el ya explicado para AM-(10,0.1), salvo porque la sentencia if desaparece. En las siguientes tablas aparecen los resultados obtenidos con esta modificación:

Caso	Coste obtenido	Desv	Tiempo (s)
MDG-a_1_n500_m50	7762.53	0.91	0.80711
MDG-a_2_n500_m50	7699.98	0.92	0.670934
MDG-a_3_n500_m50	7722.91	0.47	0.858086
MDG-a_4_n500_m50	7767.03	0.04	0.741739
MDG-a_5_n500_m50	7670.8	1.09	0.742876
MDG-a_6_n500_m50	7703.83	0.90	0.642476
MDG-a_7_n500_m50	7706.45	0.84	1.015232
MDG-a_8_n500_m50	7655.59	1.23	0.891856
MDG-a_9_n500_m50	7744.74	0.33	0.728914
MDG-a_10_n500_m50	7672.34	1.39	0.702153
MDG-b_21_n2000_m200	11188569.29	0.99	40.293898
MDG-b_22_n2000_m200	11137593.79	1.32	38.852803
MDG-b_23_n2000_m200	11136582.07	1.45	39.225565
MDG-b_24_n2000_m200	11150532.89	1.24	28.277517
MDG-b_25_n2000_m200	11170802.06	1.11	40.17034
MDG-b_26_n2000_m200	11175202.48	1.04	34.731421
MDG-b_27_n2000_m200	11185140.53	1.07	28.693629
MDG-b_28_n2000_m200	11133082.23	1.30	36.323246
MDG-b_29_n2000_m200	11168654.5	1.14	35.368849
MDG-b_30_n2000_m200	11189369.65	0.95	37.397898
MDG-c_1_n3000_m300	24589833	1.18	105.137038
MDG-c_2_n3000_m300	24530311	1.51	100.958068
MDG-c_8_n3000_m400	43015330	0.97	131.437467
MDG-c_9_n3000_m400	42940727	1.14	151.285856
MDG-c_10_n3000_m400	42997116	1.10	145.157449
MDG-c_13_n3000_m500	66480455	0.80	142.888046
MDG-c_14_n3000_m500	66548597	0.64	150.212986
MDG-c_15_n3000_m500	66562362	0.64	169.677096
MDG-c_19_n3000_m600	95028625	0.63	140.609118
MDG-c_20_n3000_m600	95030894	0.64	161.111014

Tabla 18: Resultados para AM-(1,0.1)

Algoritmo	Desv	Tiempo (s)
BL del Primer Mejor	1.22	0.55
AM-(10,1)	1.38	53.73
AM-(10,0.1)	1.10	94.53
AM-(1,0.1)	0.97	58.85
AM-(10,0.1mejores)	1.26	92.66

Tabla 19: Comparativa de estadísticos medios

Vemos que la desviación media obtenida es muy pequeña con este algoritmo, es de sólo 0.97, superando así a todas las otras versiones de los algoritmos meméticos y a la búsqueda local. En la tabla 15 podemos ver que para todos los casos las desviaciones con respecto al óptimo son bastante pequeñas y en ningún caso se supera el 1.5.

Al aplicar la búsqueda local en todas las generaciones, la influencia de la misma es bastante significativa, y ya hemos visto que la búsqueda local nos da buenos resultados. Se van mejorando 5 soluciones de todas las poblaciones, que pueden ser buenas o malas, dando oportunidad a todas las soluciones a ser mejoradas por la búsqueda local, pudiendo ser la misma solución mejorada varias veces. Así, se explota el entorno de algunas soluciones en todas las generaciones, lo que, junto a la exploración global llevada a cabo por el algoritmo genético, hace que esta nueva versión ofrezca muy buenos resultados.

El tiempo de ejecución también se ve disminuido, pues al aplicar BL con más frecuencia, se llevan a cabo más evaluaciones de la función objetivo dentro de la misma de manera factorizada, siendo por tanto el cálculo del fitness de las soluciones más rápido y disminuyéndose así el tiempo total de ejecución.

6.5. AM-(10,0.1peores)

En este caso cambiamos la tercera versión de los algoritmos meméticos, aplicando la búsqueda local a las 5 peores soluciones de la población en lugar de a las 5 mejores, para lo cual simplemente hay que ordenar las soluciones de menor a mayor fitness y escoger las 5 primeras. Los resultados obtenidos han sido los siguientes:

Caso	Coste obtenido	Desv	Tiempo (s)
MDG-a_1_n500_m50	7684.96	1.90	1.982638
MDG-a_2_n500_m50	7610.8	2.07	1.805557
MDG-a_3_n500_m50	7617.22	1.83	2.387658
MDG-a_4_n500_m50	7642.69	1.64	2.024255
MDG-a_5_n500_m50	7611.9	1.85	2.031721
MDG-a_6_n500_m50	7732.61	0.53	1.987931
MDG-a_7_n500_m50	7671.88	1.28	1.729454
MDG-a_8_n500_m50	7644.34	1.37	2.103747
MDG-a_9_n500_m50	7670.47	1.28	1.934887
MDG-a_10_n500_m50	7658.4	1.57	1.904369
MDG-b_21_n2000_m200	11191628.16	0.96	71.260824
MDG-b_22_n2000_m200	11191593.25	0.84	64.267928
MDG-b_23_n2000_m200	11185087.13	1.02	66.437607
MDG-b_24_n2000_m200	11199050.05	0.81	87.90317
MDG-b_25_n2000_m200	11198344.76	0.87	59.748716
MDG-b_26_n2000_m200	11220821.5	0.63	72.00934
MDG-b_27_n2000_m200	11183620.59	1.08	76.289367
MDG-b_28_n2000_m200	11140101.78	1.24	75.998972
MDG-b_29_n2000_m200	11195670.8	0.90	82.044276
MDG-b_30_n2000_m200	11162823.77	1.18	61.928646
MDG-c_1_n3000_m300	24703801	0.72	229.183004
MDG-c_2_n3000_m300	24646462	1.04	216.916618
MDG-c_8_n3000_m400	43063760	0.86	251.668098
MDG-c_9_n3000_m400	43128555	0.71	289.513087
MDG-c_10_n3000_m400	43115644	0.83	255.694975
MDG-c_13_n3000_m500	66628054	0.58	265.158416
MDG-c_14_n3000_m500	66623677	0.53	276.959503
MDG-c_15_n3000_m500	66539792	0.68	272.988358
MDG-c_19_n3000_m600	95181690	0.47	334.672693
MDG-c_20_n3000_m600	95171504	0.49	300.528017

Tabla 20: Resultados para AM-(10,0.1peores)

Algoritmo	Desv	Tiempo (s)
BL del Primer Mejor	1.22	0.55
AM-(10,1)	1.38	53.73
AM-(10,0.1)	1.10	94.53
AM-(1,0.1)	0.97	58.85
AM-(10,0.1mejores)	1.26	92.66
AM-(10,0.1peores)	1.06	114.37

Tabla 21: Comparativa de estadísticos medios

Notamos que esta idea de aplicar la BL a las peores soluciones nos da muy buenos resultados, mejorando a casi todas las versiones de los algoritmos meméticos (menos la del apartado anterior por aplicarse allí BL con mayor frecuencia que en este caso) y a la búsqueda local. Al explotar el entorno de las peores soluciones con BL, le damos una oportunidad a las mismas para mejorar, pues alrededor de ellas también puede haber óptimos buenos, que en el caso anterior era más complicado alcanzar. Si se mejoraban sólo las mejores soluciones, como ya comentamos, podía llegar un punto en el que estas no mejorasen más por haber caído en un óptimo local. Sin embargo, el aplicarla a las peores, es una garantía de que estas van a mejorar casi siempre, de manera que la búsqueda local siempre va a ser efectiva. De esta forma se obtiene también una mayor diversidad de la población que en AM-(10,0.1mejores), pudiendo explorarse el entorno más profundamente y evitando los óptimos locales.

Podemos ver que esta nueva versión tarda algo más de tiempo en ejecutarse, lo cual podríamos decir que no es debido al algoritmo, pues la única diferencia es que las soluciones se ordenan ahora en orden creciente del fitness en vez de decreciente, luego los tiempos de ejecución deberían ser iguales.

6.6. Tabla comparativa de todos los algoritmos

Por último, mostramos una tabla resumen, con los valores medios de las desviaciones y los tiempos medios de ejecución de todos los algoritmos estudiados en esta práctica:

Algoritmo	**Desv**	**Tiempo (s)**
Greedy	9.22	1.2
BL del Primer Mejor	1.22	0.55
AGG-Pos	2.49	132.14
AGG-Pos con selección por torneo de tamaño 3	2.32	141.17
AGG con operador posicional modificado	1.88	134.17
AGG con operador uniforme	1.35	215.13
AGG con operador uniforme modificado	1.20	186.49
AGE-Pos	2.56	123.33
AGE-Pos con selección por torneo de tamaño 3	2.52	117.57
AGE con operador uniforme	1.80	171.25
AM-(10,1)	1.38	53.73
AM-(10,0.1)	1.10	94.53
AM-(1,0.1)	0.97	58.85
AM-(10,0.1mejores)	1.26	92.66
AM-(10,0.1peores)	1.06	114.37

Tabla 22: Comparativa de estadísticos medios obtenidos por distintos algoritmos para el MDP