Práctica 2.a

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

Alejandro Palencia Blanco

DNI: 77177568X

alepalenc@correo.ugr.es

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

Índice

1. Formulación del problema

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

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

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

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

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

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

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

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

2. Consideraciones comunes a los algoritmos empleados al problema

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

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

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

2.1. Esquema de representación de soluciones

En esta práctica usaremos dos esquemas distintos para representar las soluciones. El esquema principal consistirá en un vector binario con n posiciones en el que almacenaremos un 1 si el elemento asociado a esa posición ha sido seleccionado, y un 0 en caso contrario. Por tanto, una solución con este esquema será factible si tiene

exactamente m elementos a 1 y el resto a 0. Este será el esquema que usaremos en todos los algoritmos genéticos que se implementen.

Por otro lado, también usaremos el esquema de representación de soluciones usado en la práctica anterior. En él, los elementos de S se representan mediante números enteros del 0 al $n-1$. Por tanto, una solución factible consistirá en un subconjunto $Sel \subset S = \{0, \ldots, n-1\}$ tal que $|Sel| = m$. Este esquema solamente será usado cuando se aplique búsqueda local en los algoritmos meméticos.

En todos los algoritmos se trabaja con una estructura `Solucion`, que consiste en un vector de booleanos `genes` que almacena una solución en codificación binaria y un flotante `fitness` que almacena el valor que devuelve la función objetivo para esa solución.

2.2. Función objetivo

La función objetivo que he implementado recibe como parámetros una solución y la matriz de distancias y simplemente actualiza el valor de fitness asociado a dicha solución. Para ello, inicializa su fitness a 0, calcula la representación de la solución en conjunto de enteros (un vector de enteros *selected*) y usa esta representación para calcular su fitness.

Algorithm 1: evaluateFitness

```
Data: sol : solución,
      matrix : matriz de distancias
begin
    sol.genes ← {0, }
    selected ← {}
    for i ∈ {0, ..., n − 1} do
        if sol.genes[i] == 1 then
            selected ← selected ∪ {i}
        end
    end
    for gen1, gen2 ∈ selected do
        sol.fitness ← sol.fitness + matrix[gen1][gen2]
    end
end
```

2.3. Operadores

Los operadores usados en los algoritmos genéticos son la generación de solución aleatoria, la selección, el cruce uniforme (junto con el operador de reparación), el cruce por posición, la mutación.

2.3.1. Generación de solución aleatoria

El operador de generación de solución aleatoria recibe como parámetros una solución con todos los genes a 0 y el número de genes que deben estar a 1 para que ésta sea factible. Al finalizar, devuelve una solución factible. Para ello, sustituye de forma aleatoria m ceros por unos, y luego, baraja el vector de genes con la función `random_shuffle`.

Algorithm 2: generateRandomSolution

```
Data: sol : solución con todos los genes a 0,
      m : número de genes que deben estar a 1 para que la solución sea factible
begin
    for i ∈ {0, ..., m − 1} do
        ///// Genero una posición aleatoriamente /////
        random_position ← rand() % |sol.genes|
        ///// Incremento la posición hasta encontrar una con un 0 /////
        while sol.genes[random_position] == 1 do
            random_position ← (random_position + 1) % |sol.genes|
        end
        ///// Cambio el 0 por un 1 /////
        sol.genes[random_position] ← 1
    end
    ///// Barajo el vector de genes /////
    random_shuffle(sol.genes)
end
```

2.3.2. Selección

El operador de selección recibe una población (un vector de soluciones) y devuelve un conjunto de soluciones seleccionadas (vector con las posiciones que ocupan dichas soluciones en la población). La selección se lleva a cabo mediante torneo binario, es decir, se eligen aleatoriamente dos individuos de la población y se selecciona aquel que tenga el mayor fitness.

Algorithm 3: selection

```
Data: population : vector de soluciones,
      selected_indivs : vector de individuos seleccionados
begin
    for i ∈ {0, ..., |selected_indivs| − 1} do
        ///// Elijo a dos individuos aleatoriamente /////
        first_indiv ← rand() % |population|
        second_indiv ← rand() % |population|
        ///// Selecciono por torneo binario a aquel con mayor fitness /////
        if population[first_indiv].fitness >= population[second_indiv].fitness then
            selected_indivs[i] ← first_indiv
        else
            selected_indivs[i] ← second_indiv
        end
    end
end
```

2.3.3. Cruce uniforme

El operador de cruce uniforme recibe como parámetros la población actual, el vector de individuos seleccionados, el número de genes a 1 que debe tener una solución para ser factible y la matriz de distancias. Al finalizar, devolverá una nueva población en la que el 70 % de los individuos son el resultado de cruzar a dos individuos de la población original y el 30 % restante son individuos que no han sufrido ningún cambio.

En primer lugar, para el algoritmo generacional, calculamos el número de parejas a cruzar, que será la siguiente:

$$couples\_to\_cross = \frac{|population|}{2} \cdot 0{,}7$$

Cruzaremos las *couples_to_cross* primeras parejas del vector de individuos seleccionados, es decir, el primer

individuo con el segundo, el tercero con el cuarto, y así sucesivamente. Si estamos en el caso estacionario, esto no es necesario, pues cruzaremos a los dos hijos seleccionados con una probabilidad del 100 %.

Para cada pareja de padres, generaremos dos hijos, los cuales tendrán su valor de fitness inicialmente a -1 (esto significa que todavía no ha sido calculado, se calculará posteriormente en la fase de reemplazamiento). En el cruce uniforme, aquellas posiciones que contengan el mismo valor en ambos padres se mantienen en los hijos. Luego, el resto de posiciones se completan aleatoriamente tomando cada gen de un padre o de otro.

Algorithm 4: cross_uniform

```
Data: population : vector de soluciones,
      new_population : nuevo vector de soluciones,
      selected_indivs : vector de individuos seleccionados,
      m : número de genes a 1 que debe tener una solución para ser factible,
      matrix : matriz de distancias
begin
    couples_to_cross ← (|population|/2) · 0,7
    foreach father_1, father_2 ∈ selected_indivs[0 : 2 · couples_to_cross − 1] do
        child_1.fitness ← −1
        child_2.fitness ← −1
        for j ∈ {0, ..., |father_1.genes| − 1} do
            if father_1.genes[j] == father_2.genes[j] then
                ///// Si los valores de un gen en los padres son iguales, pasan a los hijos /////
                child_1.genes[j] ← father_1.genes[j]
                child_2.genes[j] ← father_1.genes[j]
            else
                ///// Si no, completo aleatoriamente con alguno de los padres /////
                k ← rand() % 2
                child_1.genes[j] ← father_k.genes[j]
                k ← rand() % 2
                child_2.genes[j] ← father_k.genes[j]
            end
        end
        ///// Reparo ambos hijos /////
        repair(child_1, m, matrix)
        repair(child_2, m, matrix)
        ///// Añado los hijos a la nueva población /////
        new_population ← new_population ∪ {child_1, child_2}
    end
    ///// Completo la nueva población con el resto de individuos sin cruzar /////
    foreach indiv ∈ selected_indivs[2 · couples_to_cross :] do
        new_population ← new_population ∪ {indiv}
    end
end
```

Finalmente, como los hijos obtenidos pueden ser soluciones no factibles, es necesario aplicarles un operador de reparación. Este operador convierte una solución no factible en factible añadiendo o eliminando según sea necesario. Primero, crea un vector en el que introduce todas las posiciones en las que hay un uno (elementos seleccionados). Mientras sobren elementos, calcula el elemento en la solución que aporta una mayor contribución y lo elimina de la solución. Por otro lado, mientras falten elementos, calcula el elemento no seleccionado que aporte una mayor contribución a la solución y lo añade a ésta.

Algorithm 5: repair

```
Data: solution : solución a reparar,
      m : número de genes a 1 que debe tener una solución para ser factible,
      matrix : matriz de distancias
begin
    selected ← {}
                                        ///// Creo vector de genes seleccionados (posiciones con un 1) /////
    for i ∈ {0, ..., |sol.genes| − 1} do
        if sol.genes[i] == 1 then
            selected ← selected ∪ {i}
        end
    end
              ///// Mientras haya exceso de 1's, elimino el elemento que aporte una mayor contrib. /////
    while |selected| > m do
        best_contrib ← −1
        for i ∈ {0, ..., |selected| − 1} do
            contrib ← calculateContribution(selected, selected[i], matrix)
            if contrib > best_contrib then
                best_gen ← i
                best_contrib ← contrib
            end
        end
        sol.genes[selected[best_gen]] ← 0
        selected ← selected − {selected[best_gen]}
    end
                    ///// Mientras haya falta de 1's, añado el elemento que aporte una mayor contrib. /////
    while |selected| > m do
        best_contrib ← −1
        for gen ∈ {0, ..., |sol.genes| − 1} do
            if sol.genes[gen] == 0 then
                contrib ← calculateContribution(selected, gen, matrix)
                if contrib > best_contrib then
                    best_gen ← gen
                    best_contrib ← contrib
                end
            end
        end
        sol.genes[selected[best_gen]] ← 1
        selected ← selected ∪ {best_gen}
    end
end
```

La contribución que aporta cada elemento a una solución se calcula como la suma de las distancias de ese elemento a los elementos seleccionados.

Algorithm 6: calculateContribution

```
Data: selected : conjuntos de elementos seleccionados en una solución,
      gen : gen o elemento al que se le calcula la contribución,
      matrix : matriz de distancias
Result: contribution : contribución del gen o elemento
begin
    contribution ← 0
    for selected_gen ∈ selected do
        contribution ← contribution + matrix[gen][selected_gen]
    end
end
```

2.3.4. Cruce por posición

El operador de cruce por posición recibe los mismos parámetros que el cruce uniforme y devuelve una población con el mismo porcentaje de individuos cruzados. Solo se diferencia en la política que sigue a la hora de cruzar dos padres y obtener de ellos dos hijos.

Empieza de forma similar, manteniendo en los hijos los valores de aquellas posiciones cuyos valores son iguales en los padres. Los valores de las posiciones restantes para completar cada hijo se toman del primer padre y se asignan en un orden aleatorio distinto. Para ello, guardo en un vector los genes restantes del primer padre, y en otro, las posiciones asociadas a esos genes. Luego, para cada hijo, barajo el vector de restos del primer padre y asigno los genes de dicho vector en las posiciones indicadas por el vector de posiciones a completar.

Este operador ya genera por sí solo soluciones factibles, luego no hay necesidad de aplicar ningún reparador.

Algorithm 7: cross_position

```
Data: population : vector de soluciones,
      new_population : nuevo vector de soluciones,
      selected_indivs : vector de individuos seleccionados,
      m : número de genes a 1 que debe tener una solución para ser factible,
      matrix : matriz de distancias
begin
    couples_to_cross ← (|population|/2) · 0,7
    foreach father_1, father_2 ∈ selected_indivs[0 : 2 · couples_to_cross − 1] do
        child_1.fitness ← −1
        child_2.fitness ← −1
        first_father_remains ← {}
        positions_to_complete ← {}
        for j ∈ {0, ..., |father_1.genes| − 1} do
            if father_1.genes[j] == father_2.genes[j] then
                ///// Si los valores de un gen en los padres son iguales, pasan a los hijos /////
                child_1.genes[j] ← father_1.genes[j]
                child_2.genes[j] ← father_1.genes[j]
            else
                ///// Si no, guardo el valor en el primer padre y su posición /////
                first_father_remains ← first_father_remains ∪ {father_1.genes[j]}
                positions_to_complete ← positions_to_complete ∪ {j}
            end
        end
        ///// Barajo los genes restantes y completamos con ellos al primer hijo /////
        random_shuffle(first_father_remains)
        child_1[positions_to_complete] ← first_father_remains
        ///// Barajo los genes restantes y completamos con ellos al segundo hijo /////
        random_shuffle(first_father_remains)
        child_2[positions_to_complete] ← first_father_remains
        ///// Añado los hijos a la nueva población /////
        new_population ← new_population ∪ {child_1, child_2}
    end
    ///// Completo la nueva población con el resto de individuos sin cruzar /////
    foreach indiv ∈ selected_indivs[2 · couples_to_cross :] do
        new_population ← new_population ∪ {indiv}
    end
end
```

2.3.5. Mutación

El operador de mutación recibe como parámetros la nueva población cuyos individuos ya han sido cruzados, el número de genes a 1 que debe tener una solución para ser factible y la matriz de distancias. Al finalizar, devolverá una nueva población en la que un porcentaje de los individuos han sido mutados y el resto no han

sufrido ningún cambio.

En primer lugar, para el algoritmo generacional, calculamos el número de genes a mutar, que será el siguiente:

$$genes\_to\_mutate = |population| \cdot genes\_per\_solution \cdot \frac{0{,}1}{genes\_per\_solution} = |population| \cdot 0{,}1$$

Si estamos en el caso estacionario, simplemente mutaremos cada uno de los dos hijos con una probabilidad del 10 %.

Cada vez que se va a mutar un gen, se elige aleatoriamente un individuo de la población, y sobre él, se eligen aleatoriamente dos genes con valores distintos para intercambiarlos. La forma que he elegido para elegir estos genes consiste en generar dos números aleatorios, $random\_gene_1 \in \{1, \ldots, m\}$ y $random\_gene_0 \in \{1, \ldots, n-m\}$ (siendo m el número de unos y $n - m$ el número de ceros). A continuación, nos quedamos con el gen que tiene el $random\_gene_1$-ésimo 1 que encontramos al recorrer el vector de genes de izquierda a derecha, y procedemos análogamente para el 0. Esos serán los genes cuyos valores intercambiaremos.

Algorithm 8: mutation

```
Data: new_population : nuevo vector de soluciones obtenido por el cruce,
      m : número de genes a 1 que debe tener una solución para ser factible,
      matrix : matriz de distancias
begin
    genes_to_mutate = |population| · 0,1
    for k ∈ {0, ..., genes_to_mutate − 1} do
                                                  ///// Elijo un individuo aleatoriamente /////
        random_indiv ← rand() % |new_population|
                                                  ///// Elijo dos genes con valores distintos aleatoriamente /////
        random_gene_1 ← (rand() % m) + 1
        random_gene_0 ← (rand() % |new_population[random_indiv].genes| − m) + 1
        i ← 0
        while random_gene_1 + random_gene_0 > 0 do
            if new_population[random_indiv].genes == 1 then
                if random_gene_1 > 0 then
                    random_gene_1 ← random_gene_1 − 1
                    if random_gene_1 == 0 then
                        gene1_to_mutate ← i
                    end
                end
            else
                if random_gene_0 > 0 then
                    random_gene_0 ← random_gene_0 − 1
                    if random_gene_0 == 0 then
                        gene0_to_mutate ← i
                    end
                end
            end
            i ← i + 1
        end
                                                  ///// Intercambio los valores de los genes /////
        new_population[random_indiv].genes[gene1_to_mutate] ← 0
        new_population[random_indiv].genes[gene0_to_mutate] ← 1
    end
end
```

3. Algoritmos

3.1. Algoritmos genéticos generacionales

Para cada uno de los dos operadores de cruce descritos anteriormente, he desarrollado un algoritmo genético con un esquema de evolución generacional con elitismo. Este esquema consiste en seleccionar en cada iteración del algoritmo una población de padres del mismo tamaño que la población actual. Sobre esta población se aplicará el operador de cruce correspondiente, seguido del operador de mutación. La iteración termina con el reemplazamiento de la población actual por la nueva población de hijos.

Al haber elitismo, siempre se conserva la mejor solución encontrada hasta el momento. Para ello, es necesario comprobar antes de efectuar el reemplazamiento si la mejor solución de la población actual supera a la mejor solución de la población de hijos. Si esto ocurre, se sustituirá al peor hijo por la mejor solución de la población actual, pasando así a la siguiente generación.

En el operador de reemplazamiento desarrollado para el esquema generacional, primero se evalúa el fitness de todas las soluciones que no lo tengan calculado (su fitness vale -1), y luego se realiza el reemplazamiento garantizando el elitismo. El operador recibe como parámetros ambas poblaciones, la mejor solución de la población actual (pasado por referencia para actualizarlo durante el reemplazamiento) y la matriz de distancias, y devuelve el número de evaluaciones de la función objetivo que se han llevado a cabo.

Algorithm 9: replacement_AGG

```
Data: population : población actual,
      new_population : población de hijos,
      best_solution_population : mejor solución de la población actual,
      matrix : matriz de distancias
Result: inc_evaluations : número de evaluaciones de la función objetivo realizadas durante el
        reemplazamiento
begin
                                  ///// Evalúo el fitness de las soluciones que no lo tengan calculado /////
    inc_evaluations ← 0
    for sol ∈ new_population do
        if sol.fitness < 0 then
            evaluateFitness(sol, matrix)
            inc_evaluations ← inc_evaluations + 1
        end
    end
                                                  ///// Calculo la mejor sol. de la nueva pobl. /////
    best_solution_new_population ← 0
    for i ∈ {1, ..., |new_population| − 1} do
        if new_population[i].fitness > new_population[best_solution_new_population].fitness
         then
            best_solution_new_population ← i
        end
    end
                                                 ///// Garantizo elitismo si es necesario /////
    if new_population[best_solution_new_population].fitness <
     population[best_solution_population].fitness then
                                                  ///// Calculo la peor sol. de la nueva pobl. /////
        worst_solution_new_population ← 0
        for i ∈ {1, ..., |new_population| − 1} do
            if
             new_population[i].fitness < new_population[worst_solution_new_population].fitness
             then
                worst_solution_new_population ← i
            end
        end
                        ///// La mejor sol. de la pobl. actual sustituye a la peor sol. de la nueva pobl. /////
        new_population[worst_solution_new_population] ← population[best_solution_population])
        best_solution_population ← worst_solution_new_population
    else
        best_solution_population ← best_solution_new_population
    end
                                        ///// Sustituyo la población actual por la nueva población /////
    population ← new_population
end
```

Los dos algoritmos generacionales empiezan generando una población aleatoria de 50 individuos, calculando sus fitness, inicializando el número de evaluaciones a 50 y hallando la mejor solución dentro de esta población. A continuación, ejecutan la secuencia de operadores de selección, cruce, mutación y reemplazamiento hasta que el número de evaluaciones llegue a 100000. Finalmente, se devuelve la mejor solución de la población final. La única diferencia que tienen ambos algoritmos se encuentra en la llamada al operador de cruce, uno llama a $cross\_uniform$ y otro a $cross\_position$.

Algorithm 10: AGG

Data: matrix : matriz de distancias,
m : número de genes a 1 que debe tener una solución para ser factible,
population_size : tamaño de la población (fijado a 50)
Result: population[best_solution] : mejor solución encontrada
begin
$sol0 \leftarrow Solution(genes = \{0, \overset{(n)}{?\ldots?}, 0\}, fitness = -1)$
$population \leftarrow \{sol0, \overset{(population\_size)}{\ldots}, sol0\}$
$new\_population \leftarrow \{sol0, \overset{(population\_size)}{\ldots}, sol0\}$
$selected\_indivs \leftarrow \{0, \overset{(population\_size)}{\ldots}, 0\}$
///// genero soluciones aleatorias /////
for $sol \in population$ **do**
$generateRandomSolution(sol, m)$
$evaluateFitness(sol, matrix)$
end
$evaluations \leftarrow population\_size$
///// hallo la mejor solución /////
$best\_solution \leftarrow 0$
for $i \in \{1, \ldots, |population| - 1\}$ **do**
if $population[i].fitness > population[best\_solution].fitness$ **then**
$best\_solution \leftarrow i$
end
end
///// aplico algoritmo genético generacional /////
while $evaluations < 100000$ **do**
$selection(population, selected\_indivs)$
$cross(population, new\_population, selected\_indivs, m, matrix)$
$mutation(new\_population, m, matrix)$
$evaluations \leftarrow evaluations + replacement(population, new\_population, best\_solution, matrix)$
end
end

3.2. Algoritmos genéticos estacionarios

Análogamente, he desarrollado un algoritmo genético con un esquema de evolución estacionario para cada operador de cruce. Este esquema consiste en seleccionar únicamente dos padres en cada iteración del algoritmo, sobre los que se aplicará el operador de cruce correspondiente, seguido del operador de mutación. La iteración termina con la inserción de los dos hijos obtenidos en la población actual, en caso de ser mejores que las dos peores soluciones de ésta.

El operador de reemplazamiento desarrollado para el esquema estacionario empieza evaluando el fitness de los dos hijos obtenidos. A continuación, halla la peor solución de la población y, si ésta es peor el mejor hijo, éste la sustituye. Si se ha efectuado la sustitución anterior, se vuelve a calcular la peor solución y, de nuevo, es sustituida por el otro hijo si ésta es peor. El operador recibe como parámetros la población actual, los dos hijos y la matriz de distancias, y no devuelve nada.

Algorithm 11: replacement_AGE

```
Data: population : población actual,
      new_population : población de hijos (en este caso son solo 2),
      matrix : matriz de distancias
begin
    ///// Evalúo el fitness de los hijos /////
    for sol ∈ new_population do
        evaluateFitness(sol, matrix)
    end
    ///// Los intercambio si el segundo es mejor que el primero /////
    if new_population[1].fitness > new_population[0].fitness then
        swap(new_population[0], new_population[1])
    end
    ///// Calculo la peor solución de la población /////
    worst_solution ← 0
    for i ∈ {1, ..., |population| − 1} do
        if population[i].fitness < population[worst_solution].fitness then
            worst_solution ← i
        end
    end
    ///// Si el primer hijo es mejor que la peor solución, la sustituye /////
    if new_population[0].fitness < population[worst_solution].fitness then
        population[worst_solution] ← new_population[0]
        ///// Calculo la peor solución de la población /////
        worst_solution ← 0
        for i ∈ {1, ..., |population| − 1} do
            if population[i].fitness < population[worst_solution].fitness then
                worst_solution ← i
            end
        end
        ///// Si el segundo hijo es mejor que la peor solución, la sustituye /////
        if new_population[1].fitness < population[worst_solution].fitness then
            population[worst_solution] ← new_population[1]
        end
    end
end
```

Los dos algoritmos generacionales empiezan, al igual que antes, generando una población aleatoria de 50 individuos, calculando sus fitness e inicializando el número de evaluaciones a 50. A continuación, ejecutan la secuencia de operadores de selección, cruce, mutación y reemplazamiento hasta que el número de evaluaciones llegue a 100000. En este esquema, el incremento de evaluaciones al final de cada iteración siempre es 2. Finalmente, se devuelve la mejor solución de la población final. De nuevo, la única diferencia que tienen ambos algoritmos se encuentra en la llamada al operador de cruce.

Algorithm 12: AGE

Data: matrix : matriz de distancias,
m : número de genes a 1 que debe tener una solución para ser factible,
population_size : tamaño de la población (fijado a 50)
Result: population[best_solution] : mejor solución encontrada
begin
$sol0 \leftarrow Solution(genes = \{0, \overset{(n)}{?}, 0\}, fitness = -1)$
$population \leftarrow \{sol0, \overset{(population\_size)}{\ldots}, sol0\}$
$new\_population \leftarrow \{sol0, sol0\}$
$selected\_indivs \leftarrow \{0, 0\}$
///// genero soluciones aleatorias /////
for $sol \in population$ **do**
$generateRandomSolution(sol, m)$
$evaluateFitness(sol, matrix)$
end
$evaluations \leftarrow population\_size$
///// aplico algoritmo genético estacionario /////
while $evaluations < 100000$ **do**
$selection(population, selected\_indivs)$
$cross(population, new\_population, selected\_indivs, m, matrix)$
$mutation(new\_population, m, matrix)$
$replacement(population, new\_population, best\_solution, matrix)$
$evaluations \leftarrow evaluations + 2$
end
///// hallo la mejor solución /////
$best\_solution \leftarrow 0$
for $i \in \{1, \ldots, |population| - 1\}$ **do**
if $population[i].fitness > population[best\_solution].fitness$ **then**
$best\_solution \leftarrow i$
end
end
end

3.3. Algoritmos meméticos

Para desarrollar los algoritmos meméticos, hibridamos el algoritmo genético generacional que mejor resultado ha proporcionado con la búsqueda local de la práctica anterior. En mi caso, el mejor resultado lo proporciona el algoritmo que usa el operador de cruce uniforme.

Para implementar la búsqueda local, primero será necesario pasar de la codificación binaria usada en los algoritmos genéticos a la codificación de conjunto de enteros. A partir de aquí, la búsqueda local funciona de la misma forma que en la práctica anterior, pero teniendo como criterio de parada no encontrar una mejora en todo el entorno o alcanzar las 400 evaluaciones. Por último, será necesario pasar la solución encontrada por búsqueda local a codificación binaria.

Algorithm 13: localSearch

```
Data: sol : solución sobre la que se aplica búsqueda local,
      matrix : matriz de distancias
begin
                                                  ///// paso a codificación de conjunto de enteros /////
    selected ← {}
    not_selected ← {}
    for i ∈ {0, ..., |sol.genes| − 1} do
        if sol.genes[i] == 1 then
            selected ← selected ∪ {i}
        else
            not_selected ← not_selected ∪ {i}
        end
    end
                                                  ///// aplico búsqueda local /////
    evaluations ← 0
    do
        exchange ← false
        for i ∈ {0, ..., |selected| − 1}  and  !exchange  and  evaluations < 400 do
                                  ///// encuentro el siguiente elemento seleccionado con menor contribución /////
            worst_element_contrib ← calculateContribution(selected, selected[i], matrix)
            for j ∈ {i + 1, ..., |selected| − 1} do
                element_contrib ← calculateContribution(selected, selected[j], matrix)
                if element_contrib < worst_element_contrib then
                    swap(selected[i], selected[j])
                    worst_element_contrib ← element_contrib
                end
            end
            for j ∈ {0, ..., |not_selected| − 1}  and  !exchange  and  evaluations < 400 do
                                  ///// calculo la contribución del siguiente elemento no seleccionado /////
                element_contrib ← calculateContribution(selected, not_selected[j], matrix) −
                  matrix[selected[i]][not_selected[j]]
                evaluations ← evaluations + 1
                    ///// si tiene mayor contribución que el elemento seleccionado, los intercambio /////
                if element_contrib > worst_element_contrib then
                    swap(selected[i], not_selected[j])
                    exchange ← true
                end
            end
        end
    while exchange  and  evaluations < 400;
                                                  ///// paso a codificación binaria /////
    for i ∈ {0, ..., |selected| − 1} do
        sol.genes[selected[i]] ← 1
    end
    for i ∈ {0, ..., |not_selected| − 1} do
        sol.genes[not_selected[i]] ← 0
    end
                                                  ///// actualizo fitness /////
    sol.fitness ← 0
    for i ∈ {0, ..., |selected| − 1} do
        for j ∈ {i + 1, ..., |selected| − 1} do
            sol.fitness ← sol.fitness + matrix[selected[i]][selected[j]]
        end
    end
end
```

La estructura de los tres algoritmos meméticos es la misma: Aplicamos el algoritmo genético durante 10 generaciones y, a continuación, aplicamos búsqueda local sobre las soluciones. Lo único que cambia es la política

de aplicación de la búsqueda local (denotado por *applyLocalSearch* en el pseudocódigo).

Algorithm 14: AM

Data: matrix : matriz de distancias,
m : número de genes a 1 que debe tener una solución para ser factible,
population_size : tamaño de la población (fijado a 50)
Result: population[best_solution] : mejor solución encontrada
begin
$sol0 \leftarrow Solution(genes = \{0, \overset{(n)}{\ldots}, 0\}, fitness = -1)$
$population \leftarrow \{sol0, \overset{(population\_size)}{\ldots}, sol0\}$
$new\_population \leftarrow \{sol0, \overset{(population\_size)}{\ldots}, sol0\}$
$selected\_indivs \leftarrow \{0, \overset{(population\_size)}{\ldots}, 0\}$
///// genero soluciones aleatorias /////
for $sol \in population$ **do**
$generateRandomSolution(sol, m)$
$evaluateFitness(sol, matrix)$
end
$evaluations \leftarrow population\_size$
///// hallo la mejor solución /////
$best\_solution \leftarrow 0$
for $i \in \{1, \ldots, |population| - 1\}$ **do**
if $population[i].fitness > population[best\_solution].fitness$ **then**
$best\_solution \leftarrow i$
end
end
///// aplico algoritmo memético /////
while $evaluations < 100000$ **do**
for $k \in \{0, \ldots, 9\}$ *and* $evaluations < 100000$ **do**
$selection(population, selected\_indivs)$
$cross\_uniform(population, new\_population, selected\_indivs, m, matrix)$
$mutation(new\_population, m, matrix)$
$evaluations \leftarrow$
$evaluations + replacement(population, new\_population, best\_solution, matrix)$
end
$applyLocalSearch()$
end
end

3.3.1. AM-(10,1.0)

La primera versión de algoritmo memético, cada vez que aplica búsqueda local, lo hace sobre todas las soluciones de la población.

Algorithm 15: applyLocalSearch (AM1)

begin
for $i \in \{0, \ldots, |population| - 1\}$ *and* $evaluations < 100000$ **do**
$evaluations \leftarrow evaluations + localSearch(population[i], matrix)$
///// actualizo la posición de la mejor solución si es necesario /////
if $population[i].fitness > population[best\_solution].fitness$ **then**
$best\_solution \leftarrow i$
end
end
end

3.3.2. AM-(10,0.1)

La segunda versión, cada vez que aplica búsqueda local, lo hace sobre un 10 % población seleccionado aleatoriamente. Lo he implementado tomando un vector con enteros de 0 a 49 (las posiciones de las soluciones en la población), barajándolo y aplicando búsqueda local sobre las 5 primeras soluciones (el 10 % de 50).

Algorithm 16: applyLocalSearch (AM2)

```
begin
    population_positions ← {0, ..., |population| − 1}
    random_shuffle(population_positions)
                                  ///// aplico BL sobre el primer 10 % de soluciones del vector barajado /////
    for i ∈ {0, ..., 0,1 · |population| − 1} and evaluations < 100000 do
        evaluations ← evaluations + localSearch(population[population_positions[i]], matrix)
                                  ///// actualizo la posición de la mejor solución si es necesario /////
        if population[population_positions[i]].fitness > population[best_solution].fitness then
            best_solution ← population_positions[i]
        end
    end
end
```

3.3.3. AM-(10,0.1mej)

La tercera versión, cada vez que aplica búsqueda local, lo hace sobre las $0{,}1 \cdot |population|$ mejores soluciones de la población actual. Para ello, primero busco las $0{,}1 \cdot |population|$ mejores soluciones y las coloco en las primeras posiciones de la población. A continuación, aplico búsqueda local solamente sobre ellas.

Algorithm 17: applyLocalSearch (AM3)

```
begin
                        ///// coloco las mejores soluciones en las primeras posiciones de la población /////
    swap(population[best_solution], population[0])
    for i ∈ {1, ..., 0,1 · |population| − 1} do
        best_solution ← i
        for j ∈ {i + 1, ..., 0,1 · |population| − 1} do
            if population[best_solution].fitness < population[j].fitness then
                best_solution ← j
            end
            swap(population[best_solution], population[i])
        end
    end
    best_solution ← 0
                                                  ///// aplico BL sobre las mejores soluciones /////
    for i ∈ {0, ..., 0,1 · |population| − 1} and evaluations < 100000 do
        evaluations ← evaluations + localSearch(population[i], matrix)
                                  ///// actualizo la posición de la mejor solución si es necesario /////
        if population[i].fitness > population[best_solution].fitness then
            best_solution ← i
        end
    end
end
```

4. Procedimiento de desarrollo de la práctica

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

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

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

- `bin/<algoritmo>`: compila el código fuente con el algoritmo especificado (AGGuniform, AGGposition, AGEuniform, AGEposition, AM1, AM2, AM3)
- `example<algoritmo>`: compila el algoritmo especificado y lo ejecuta con tres instancias del problema con distintos tamaños, mostrando los resultados por la terminal
- `compile_all`: compila todos los algoritmos
- `all`: compila todos los algoritmos y ejecuta el script `executeAlgorithms.sh`
- `clean`: elimina el contenido de los directorios `bin` y `output`

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

5. Experimentos y análisis de resultados

5.1. Experimentos

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

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

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

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

5.2. Resultados

A continuación, podemos ver los tiempos y las desviaciones obtenidas por cada algoritmo:

Por último, en la tabla 8 se recogen los resultados medios de desviación y tiempo para todos los algoritmos considerados. Los resultados medios de búsqueda local no son los mismos de la práctica anterior, ya que he corregido el código agregando el criterio de parada de las 100000 evaluaciones y lo he vuelto a ejecutar.

AGGuniform		
Caso	**Desv**	**Tiempo**
MDG-a_1_n500_m50	1.19	0.814040
MDG-a_2_n500_m50	1.99	0.754790
MDG-a_3_n500_m50	1.86	0.765043
MDG-a_4_n500_m50	1.26	0.769825
MDG-a_5_n500_m50	2.24	0.828254
MDG-a_6_n500_m50	3.63	0.727737
MDG-a_7_n500_m50	2.56	0.777321
MDG-a_8_n500_m50	1.50	0.735060
MDG-a_9_n500_m50	1.84	0.799064
MDG-a_10_n500_m50	1.19	0.767439
MDG-b_21_n2000_m200	1.06	21.703165
MDG-b_22_n2000_m200	0.75	20.241446
MDG-b_23_n2000_m200	0.82	21.737553
MDG-b_24_n2000_m200	0.87	20.169540
MDG-b_25_n2000_m200	1.08	19.388049
MDG-b_26_n2000_m200	1.09	19.407117
MDG-b_27_n2000_m200	1.06	18.795398
MDG-b_28_n2000_m200	0.81	19.485889
MDG-b_29_n2000_m200	1.31	18.400062
MDG-b_30_n2000_m200	1.37	18.909820
MDG-c_1_n3000_m300	1.08	56.843499
MDG-c_2_n3000_m300	1.10	53.985827
MDG-c_8_n3000_m400	0.87	85.675737
MDG-c_9_n3000_m400	0.75	91.529975
MDG-c_10_n3000_m400	0.90	90.359767
MDG-c_13_n3000_m500	0.67	108.145062
MDG-c_14_n3000_m500	0.55	111.106732
MDG-c_15_n3000_m500	0.50	115.684725
MDG-c_19_n3000_m600	0.74	124.194071
MDG-c_20_n3000_m600	0.63	131.800769

Cuadro 1: AGGuniform

AGGposition		
Caso	**Desv**	**Tiempo**
MDG-a_1_n500_m50	3.40	0.472603
MDG-a_2_n500_m50	3.86	0.493042
MDG-a_3_n500_m50	2.00	0.440347
MDG-a_4_n500_m50	3.21	0.450523
MDG-a_5_n500_m50	3.05	0.440250
MDG-a_6_n500_m50	2.71	0.507060
MDG-a_7_n500_m50	3.19	0.441592
MDG-a_8_n500_m50	3.01	0.496859
MDG-a_9_n500_m50	2.43	0.449557
MDG-a_10_n500_m50	2.04	0.468577
MDG-b_21_n2000_m200	2.54	3.653694
MDG-b_22_n2000_m200	2.22	3.718302
MDG-b_23_n2000_m200	2.58	3.721227
MDG-b_24_n2000_m200	2.75	3.682594
MDG-b_25_n2000_m200	2.67	3.593467
MDG-b_26_n2000_m200	2.49	3.662391
MDG-b_27_n2000_m200	2.88	3.662841
MDG-b_28_n2000_m200	2.20	3.525714
MDG-b_29_n2000_m200	2.61	3.762359
MDG-b_30_n2000_m200	2.61	3.641250
MDG-c_1_n3000_m300	2.48	10.745056
MDG-c_2_n3000_m300	2.54	11.862815
MDG-c_8_n3000_m400	1.92	26.051449
MDG-c_9_n3000_m400	2.02	25.429321
MDG-c_10_n3000_m400	2.21	25.532542
MDG-c_13_n3000_m500	1.75	39.067775
MDG-c_14_n3000_m500	1.66	38.325787
MDG-c_15_n3000_m500	1.63	38.400953
MDG-c_19_n3000_m600	1.45	48.332032
MDG-c_20_n3000_m600	1.56	47.997017

Cuadro 2: AGGposition

AGEuniform		
Caso	**Desv**	**Tiempo**
MDG-a_1_n500_m50	2.50	0.795903
MDG-a_2_n500_m50	3.94	0.803522
MDG-a_3_n500_m50	1.45	0.805167
MDG-a_4_n500_m50	3.26	0.796446
MDG-a_5_n500_m50	3.36	0.767390
MDG-a_6_n500_m50	1.82	0.780485
MDG-a_7_n500_m50	2.69	0.789929
MDG-a_8_n500_m50	2.77	0.802505
MDG-a_9_n500_m50	1.85	0.803048
MDG-a_10_n500_m50	2.32	0.801402
MDG-b_21_n2000_m200	1.82	13.750168
MDG-b_22_n2000_m200	1.75	15.224447
MDG-b_23_n2000_m200	2.20	16.366386
MDG-b_24_n2000_m200	2.45	17.024631
MDG-b_25_n2000_m200	1.92	13.906019
MDG-b_26_n2000_m200	1.98	13.512376
MDG-b_27_n2000_m200	2.19	13.904464
MDG-b_28_n2000_m200	1.87	16.868825
MDG-b_29_n2000_m200	1.92	16.079131
MDG-b_30_n2000_m200	2.05	15.787427
MDG-c_1_n3000_m300	1.78	24.210146
MDG-c_2_n3000_m300	1.77	23.687862
MDG-c_8_n3000_m400	1.23	43.240817
MDG-c_9_n3000_m400	1.40	44.435686
MDG-c_10_n3000_m400	1.45	46.603064
MDG-c_13_n3000_m500	0.98	64.634440
MDG-c_14_n3000_m500	0.99	62.382837
MDG-c_15_n3000_m500	1.09	63.347990
MDG-c_19_n3000_m600	0.98	74.037329
MDG-c_20_n3000_m600	0.97	74.593469

Cuadro 3: AGEuniform

AGEposition		
Caso	**Desv**	**Tiempo**
MDG-a_1_n500_m50	3.21	0.590372
MDG-a_2_n500_m50	3.50	0.576865
MDG-a_3_n500_m50	1.76	0.632360
MDG-a_4_n500_m50	4.65	0.632680
MDG-a_5_n500_m50	3.03	0.579404
MDG-a_6_n500_m50	2.58	0.585874
MDG-a_7_n500_m50	1.96	0.579503
MDG-a_8_n500_m50	2.10	0.580978
MDG-a_9_n500_m50	2.38	0.581494
MDG-a_10_n500_m50	3.37	0.578158
MDG-b_21_n2000_m200	2.56	5.356617
MDG-b_22_n2000_m200	2.50	4.785483
MDG-b_23_n2000_m200	2.69	5.191796
MDG-b_24_n2000_m200	2.67	4.840733
MDG-b_25_n2000_m200	2.75	4.856372
MDG-b_26_n2000_m200	2.66	3.379678
MDG-b_27_n2000_m200	2.89	4.733798
MDG-b_28_n2000_m200	2.34	5.299156
MDG-b_29_n2000_m200	2.67	5.532262
MDG-b_30_n2000_m200	2.54	5.127958
MDG-c_1_n3000_m300	2.38	16.276895
MDG-c_2_n3000_m300	2.56	22.816067
MDG-c_8_n3000_m400	1.84	54.107420
MDG-c_9_n3000_m400	2.02	54.014267
MDG-c_10_n3000_m400	2.03	56.517120
MDG-c_13_n3000_m500	1.81	85.522973
MDG-c_14_n3000_m500	1.82	88.143424
MDG-c_15_n3000_m500	1.70	85.665349
MDG-c_19_n3000_m600	1.52	110.132993
MDG-c_20_n3000_m600	1.39	111.923544

Cuadro 4: AGEposition

AM-(10,1.0)		
Caso	**Desv**	**Tiempo**
MDG-a_1_n500_m50	1.98	0.070282
MDG-a_2_n500_m50	2.48	0.054285
MDG-a_3_n500_m50	3.44	0.061563
MDG-a_4_n500_m50	1.44	0.081171
MDG-a_5_n500_m50	1.77	0.082578
MDG-a_6_n500_m50	2.06	0.081511
MDG-a_7_n500_m50	1.75	0.055971
MDG-a_8_n500_m50	1.62	0.055840
MDG-a_9_n500_m50	1.80	0.074269
MDG-a_10_n500_m50	1.08	0.092080
MDG-b_21_n2000_m200	3.43	5.512489
MDG-b_22_n2000_m200	3.30	4.620877
MDG-b_23_n2000_m200	2.81	5.521977
MDG-b_24_n2000_m200	3.87	4.237251
MDG-b_25_n2000_m200	3.42	4.877300
MDG-b_26_n2000_m200	3.23	5.123272
MDG-b_27_n2000_m200	3.40	5.034545
MDG-b_28_n2000_m200	3.07	5.543876
MDG-b_29_n2000_m200	3.19	4.906408
MDG-b_30_n2000_m200	2.94	4.984883
MDG-c_1_n3000_m300	3.23	17.145767
MDG-c_2_n3000_m300	3.09	14.160857
MDG-c_8_n3000_m400	3.23	23.102365
MDG-c_9_n3000_m400	2.94	22.100279
MDG-c_10_n3000_m400	3.10	24.283897
MDG-c_13_n3000_m500	2.63	23.511748
MDG-c_14_n3000_m500	2.61	25.764075
MDG-c_15_n3000_m500	2.55	26.138551
MDG-c_19_n3000_m600	2.52	30.235119
MDG-c_20_n3000_m600	2.51	28.848457

Cuadro 5: AM-(10,1.0)

AM-(10,0.1)		
Caso	**Desv**	**Tiempo**
MDG-a_1_n500_m50	1.09	0.185021
MDG-a_2_n500_m50	1.22	0.225064
MDG-a_3_n500_m50	1.63	0.170501
MDG-a_4_n500_m50	2.66	0.216106
MDG-a_5_n500_m50	2.05	0.189955
MDG-a_6_n500_m50	1.49	0.176410
MDG-a_7_n500_m50	1.07	0.196836
MDG-a_8_n500_m50	1.86	0.173510
MDG-a_9_n500_m50	1.52	0.190738
MDG-a_10_n500_m50	0.71	0.219248
MDG-b_21_n2000_m200	1.36	12.229355
MDG-b_22_n2000_m200	1.11	11.748460
MDG-b_23_n2000_m200	1.38	12.895893
MDG-b_24_n2000_m200	1.41	11.173985
MDG-b_25_n2000_m200	1.25	12.341223
MDG-b_26_n2000_m200	1.22	11.593821
MDG-b_27_n2000_m200	1.12	14.681689
MDG-b_28_n2000_m200	1.00	11.635903
MDG-b_29_n2000_m200	1.06	11.062632
MDG-b_30_n2000_m200	1.04	11.061899
MDG-c_1_n3000_m300	0.90	39.053288
MDG-c_2_n3000_m300	1.09	38.118750
MDG-c_8_n3000_m400	0.97	48.424242
MDG-c_9_n3000_m400	0.89	53.447973
MDG-c_10_n3000_m400	1.06	52.789269
MDG-c_13_n3000_m500	0.86	54.903659
MDG-c_14_n3000_m500	0.72	58.939140
MDG-c_15_n3000_m500	0.56	60.657723
MDG-c_19_n3000_m600	0.76	62.611614
MDG-c_20_n3000_m600	0.82	68.704060

Cuadro 6: AM-(10,0.1)

AM-(10,0.1mej)		
Caso	**Desv**	**Tiempo**
MDG-a_1_n500_m50	1.24	0.200193
MDG-a_2_n500_m50	1.12	0.188062
MDG-a_3_n500_m50	1.35	0.181626
MDG-a_4_n500_m50	1.63	0.197720
MDG-a_5_n500_m50	2.13	0.151076
MDG-a_6_n500_m50	1.09	0.179198
MDG-a_7_n500_m50	1.22	0.176590
MDG-a_8_n500_m50	1.60	0.151532
MDG-a_9_n500_m50	2.63	0.164586
MDG-a_10_n500_m50	1.84	0.164784
MDG-b_21_n2000_m200	1.52	9.428342
MDG-b_22_n2000_m200	1.28	10.545795
MDG-b_23_n2000_m200	1.66	9.211574
MDG-b_24_n2000_m200	1.18	10.151755
MDG-b_25_n2000_m200	1.12	10.338780
MDG-b_26_n2000_m200	0.95	9.917696
MDG-b_27_n2000_m200	1.57	9.629785
MDG-b_28_n2000_m200	1.19	9.751008
MDG-b_29_n2000_m200	1.04	10.156745
MDG-b_30_n2000_m200	1.08	10.766242
MDG-c_1_n3000_m300	0.90	35.794718
MDG-c_2_n3000_m300	0.91	32.524801
MDG-c_8_n3000_m400	0.97	49.562503
MDG-c_9_n3000_m400	0.72	50.926882
MDG-c_10_n3000_m400	1.06	45.725144
MDG-c_13_n3000_m500	0.80	53.162213
MDG-c_14_n3000_m500	0.64	57.306982
MDG-c_15_n3000_m500	0.65	58.834663
MDG-c_19_n3000_m600	0.81	62.892318
MDG-c_20_n3000_m600	0.83	63.812622

Cuadro 7: AM-(10,0.1mej)

Algoritmo	Desv	Tiempo
Greedy	9.20	0.525801
Local Search	3.59	0.730803
AGGuniform	1.24	39.176759
AGGposition	2.45	11.767633
AGEuniform	1.96	22.718110
AGEposition	2.46	24.671386
AM-(10,1.0)	2.68	9.545451
AM-(10,0.1)	1.20	22.000599
AM-(10,0.1mej)	1.22	20.406531

Cuadro 8: Resultados Globales

5.3. Análisis

5.3.1. Análisis por calidad

En primer lugar, vemos que todos los algoritmos de esta práctica superan con bastante diferencia a la búsqueda local, que era el mejor algoritmo de la práctica anterior.

Entre los algoritmos genéticos con el operador de cruce uniforme, vemos que aquel que obtiene los mejores resultados es el que usa un esquema generacional. Por otro lado, entre los que usan el operador de cruce basado en posición, no se aprecian grandes diferencias en la calidad de sus soluciones al usar un esquema generacional o estacionario.

También podemos ver que los dos algoritmos que usan el operador de cruce uniforme obtienen resultados de una calidad mayor que aquellos que usan el operador basado en posición. Teniendo en cuenta que ambos operadores empiezan copiando en los hijos los genes que son iguales en ambos padres, considero que los mejores resultados del operador uniforme se deben a su operador de reparación. Si lo analizamos en mayor profundidad, podemos observar que cuando la solución no es factible por falta de unos, añade aquellos cuya contribución sea mayor, lo cual favorece la explotación. Por otro lado, si no es factible por exceso de unos, elimina aquellos cuya contribución sea mayor, lo cual favorece la exploración. Sin embargo, el operador basado en posición simplemente completa el resto de genes aleatoriamente garantizando la factibilidad del hijo, y creo que es esta aleatoriedad la que provoca que sus resultados no sean tan buenos como los del operador uniforme.

Entre los algoritmos meméticos, el que obtiene los mejores resultados en calidad es el segundo, seguido muy de cerca por el tercero. Ambos algoritmos aplican búsqueda local sobre el 10 % de las soluciones. Su diferencia consiste en que el segundo selecciona estas soluciones aleatoriamente y el tercero toma aquellas que tienen mejor fitness. A pesar de que la política de selección de soluciones del tercero parezca la mejor a priori (pues explota las soluciones que parecen más prometedoras), considero que la aleatoriedad del segundo puede llegar a ser muy beneficiosa, ya que es posible que una solución con un fitness no tan bueno se encuentre cerca de un buen máximo local, el cual puede ser alcanzado gracias a la búsqueda local. Esto justifica que los resultados del segundo algoritmo se encuentren ligeramente por encima de los del tercero.

En cuanto al primer algoritmo memético, vemos que sus resultados en calidad se encuentran muy por debajo de los dos anteriores. Creo que esto se debe a que, cada vez que aplica búsqueda local, lo hace sobre todas las soluciones. Considero que esto consume demasiadas evaluaciones en búsqueda local (explotación), dejando pocas evaluaciones para el algoritmo genético (exploración). Es decir, no consigue explorar el espacio de soluciones tan exhaustivamente como las otras dos versiones.

Por último, al comparar algoritmo genético y el algoritmo memético que obtienen mejores resultados en calidad (AGGuniform y AM-(10,0.1), respectivamente), me ha sorprendido comprobar que el genético obtiene resultados muy parecidos al memético, a pesar de que este último tenga un mayor equilibrio entre exploración y explotación.

5.3.2. Análisis por tiempos

Como era esperable, empezamos viendo que todos los algoritmos desarrollados en esta práctica obtienen tiempos mucho más altos que los de la práctica anterior. Concretamente, tanto Greedy como Búsqueda Local

tienen un tiempo medio inferior al segundo, mientras que los tiempos medios de todos los algoritmos genéticos y meméticos son superiores a los 9 segundos.

El esquema estacionario vemos que obtiene tiempos muy similares para ambos operadores de cruce, mientras que para el esquema generacional, los tiempos del operador uniforme son mucho más superiores que los del operador basado en posición. Tiene sentido que el operador uniforme tarde más que el basado en posición, pues necesita aplicar el reparador sobre cada hijo, el cual puede consumir mucho tiempo si el número de unos a añadir o a eliminar en el hijo es muy elevado.

Por otro lado, los algoritmos genéticos experimentan una gran reducción en sus tiempos medios en comparación con AGGuniform (el algoritmo genético que se hibrida para obtenerlos). La justifiicación de este hecho se encuentra precisamente en la hibridación con la búsqueda local, pues en los meméticos, una parte de las evaluaciones son consumidas por la búsqueda local, que necesita invertir mucho menos tiempo que el algoritmo genético. Precisamente el algoritmo memético que consume más evaluaciones mediante búsqueda local, AM-(10,1.0), es el que obtiene mejores tiempos.

5.3.3. Conclusiones

Llegamos a la conclusión de que, al menos para este problema, los algoritmos genéticos y meméticos consiguen mejorar de forma muy significativa la calidad de los resultados obtenidos por Greedy y Búsqueda Local. Sin embargo, estos algoritmos necesitan invertir mucho más tiempo para calcular sus soluciones realizando el mismo número de evaluaciones de la función objetivo.

El operador de cruce uniforme obtiene resultados de mucha mayor calidad que el operador basado en posición independientemente del esquema usado, aunque sus tiempos aumentan de forma considerable en el esquema generacional.

Por último, a pesar de que los algoritmos meméticos no consigan mejorar de forma significativa la calidad de los resultados del algoritmo genético sobre el que se hibridan (de hecho, el primer memético incluso empeora), los tiempos que obtienen son mucho más bajos. Esto demuestra que es mucho más recomendable aplicar la estrategia de los algoritmos meméticos, basada en un equilibrio entre exploración y explotación, que la de los genéticos, la cual se centra únicamente en la exploración.