# UNIVERSIDAD DE GRANADA

MH

Metaheurísticas

**Curso:** 3º **Grupo:** 1

# Práctica 2
# MDP

**Autor:** Mario Carmona Segovia
**DNI:** 45922466E **E-mail:** mcs2000carmona@correo.ugr.es

**Profesor:** Daniel Molina


Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

—

Curso 2020 - 2021

# Índice

# Índice de figuras

# Índice de tablas

# 1. Descripción del problema

El Problema de la Máxima Diversidad (en ingles, Maximum Diversity Problem, MDP) consiste en la elección de un subconjunto de M puntos, de un conjunto mayor de N puntos. Este subconjunto debe maximizar la diversidad, Z; es decir, maximizar la distancia entre los elementos del subconjunto.

La fórmula para obtener la diversidad de un subconjunto es la siguiente:

$$Z = \sum_{i=1}^{m-1} \sum_{j=i+1}^{m} d_{ij}$$

Siendo:

- i y j índices de los elementos del conjunto
- $d_{ij}$ la distancia entre los puntos con índice i y j

# 2. Descripción de los aspectos comunes

## 2.1. Representación entera

### 2.1.1. Representación de las soluciones

Cada elemento del conjunto total se ha etiquetado con un número entero positivo, empezando por el cero.

Como una solución a este problema es un subconjunto de este conjunto total, he representado el subconjunto como un lista de enteros.

### 2.1.2. Función objetivo

La función objetivo valora como de buena es una solución. La calidad se mide como la suma total de todas las distancias entre los elementos que forman la solución. Una solución es de mayor calidad cuanto mayor es la distancia total, ya que a mayor distancia mayor diversidad en la solución.

**Algorithm 1** Función objetivo

**Input:** *solucion*: lista con la etiquetas de los elementos que forman la solución.
**Input:** *distancias*: matriz que guarda la distancia entre cada par de elementos del conjunto total.
**Output:** *distancia_total*: es la distancia total que existe en la *solucion*.

```
1:
2: distancia_total ← 0.0
3: /* Calculamos la suma de todas la distancias entre los elementos
4:  de la solución */
5: solucion_size ← número de elementos en la solucion
6: for i ∈ {0, ..., solucion_size − 1} do
7:     for j ∈ {i + 1, ..., solucion_size} do
8:         distancia_total ← sumar distancia entre i y j
9: return distancia_total
```

Además de esta versión de la función objetivo se ha creado otra función objetivo que además de calcular la distancia total de una solución, calcula la contribución de cada elemento en la distancia total, es decir, calcula la distancia total de cierto elemento con el resto de elementos que conforman la solución. La distancia entre el elemento x e y será considerada tanto para la contribución de x como de y, de esta manera se aprovechan las operaciones realizadas para calcular la función objetivo.

**Algorithm 2** Función objetivo (extendida)

**Input:** *solucion*: lista con la etiquetas de los elementos que forman la solución.
**Input:** *distancias*: matriz que guarda la distancia entre cada par de elementos del conjunto total.
**Output:** *distancia_total*: es la distancia total que existe en la *solucion*.
**Output:** *contribuciones*: vector de flotantes que indican la contribución de cada elemento a la *distancia_total*.

```
2: distancia_total ← 0.0
   /* Calculamos la suma de todas la distancias entre los elementos
4:  de la solución */
   solucion_size ← número de elementos en la solucion
6: for i ∈ {0, ..., solucion_size − 1} do
       for j ∈ {i + 1, ..., solucion_size} do
8:         distancia_total ← sumar distancia entre i y j
           contribuciones ← sumar contribución al elemento i
10:        contribuciones ← sumar contribución al elemento j
   return distancia_total, contribuciones
```

## 2.2. Representación binaria

### 2.2.1. Representación de las soluciones

A diferencia de la representación entera, en este caso para representar una solución hace falta una lista de 0's y 1's del tamaño del número total de elementos. La lista debe tener una casilla por cada uno de los elementos, para indicar cual está activo y cual no. Se indica que está activo, o dicho de otra manera a sido seleccionados para la solución, con un 1; y se indica que no está activo con un 0. Para saber a que entero corresponde cada casilla, se tiene en cuenta la posición que ocupa dentro de la lista la casilla, por ejemplo la cuarta casilla corresponde al elemento que en la representación entera tenía un tres, se ha restado uno porque los enteros empiezan desde el cero.

### 2.2.2. Función objetivo

La función objetivo sigue calculando de la misma forma, con la única diferencia de que se debe extraer a que entero corresponde la casilla activada.

**Algorithm 3** Función objetivo binaria

**Input:** *solucion*: lista de 0's y 1's de un tamaño igual al número de elementos del problema.
**Input:** *distancias*: matriz que guarda la distancia entre cada par de elementos del conjunto total.
**Output:** *distancia_total*: es la distancia total que existe en la *solucion*.

```
    distancia_total ← 0.0
 3: /* Obtenemos los elementos seleccionados en la solución */
    solucion_size ← número de elementos en la solucion
    posiciones ← Vacío
 6: for i ∈ {0, ..., solucion_size − 1} do
        valor ← Contenido de la posición i en la solución
        if valor == 1 then
 9:         posiciones ← Añadir posición

    /* Calculamos la suma de todas la distancias entre los elementos
12:  de la solución */
    posiciones_size ← número de elementos en la lista posiciones
    for i ∈ {0, ..., posiciones_size − 1} do
15:     for j ∈ {i + 1, ..., posiciones_size} do
            distancia_total ← sumar distancia entre i y j
    return distancia_total
```

## 2.3. Lectura de datos

Los datos utilizados para el problema están guardados en un archivo de texto plano. Este archivo contiene los siguiente datos:

- Nº de elementos del conjunto
- Nº de elementos que debe tener la solución
- Distancia entre cada par de elementos del conjunto

El formato de la información es el siguiente:

```
1 {Número de elementos del conjunto} {Número de elementos que debe tener la solución}
2 {Elemento 0} {Elemento 1} {Distancia entre elemento 0 y elemento 1}
3 {Elemento 0} {Elemento 2} {Distancia entre elemento 0 y elemento 2}
4 ...
```

Para obtener esta información y colocarla en una estructura de datos más manejable para el programa, he creado la siguiente función:

**Algorithm 4** Leer archivo

**Input:** *nombre_archivo*: nombre del archivo de texto plano que contiene la información.
**Output:** *num_elem_selec*: número de elementos que debe tener la solución al problema.
**Output:** *distancias*: matriz de distancias entre pares de elementos del conjunto.

```
    archivo ← abrir el archivo con nombre "nombre_archivo"
    numElemTotal ← obtener el número de elementos del conjunto completo
4:  num_elem_selec ← obtener el número de elemento que debe tener la solución
    distancias ← crear la matriz de distancias
    while !final_archivo do
        i, j ← obtener el par de elementos
8:      distancias ← añadir distancia entre el par de elementos
    return num_elem_selec, distancias
```

## 2.4. Generación de soluciones aleatorias

He creado una clase llamada Individuo que contiene un vector de genes, el fitness y una variable que indica si el fitness está actualizado respecto de los genes del individuo.

Al crear la población inicial, se crea una lista de individuos de cierto tamaño. Al crear cada elemento de la lista se llama al contructor de la clase Individuo que genera de forma aleatoria los genes. Este constructor se encarga de que la solución sea factible.

**Algorithm 5** Generación aleatoria de una solución

**Input:** *tam*: número de elementos que hay en el problema.
**Input:** *genesActivos*: número de elementos que debe tener la solución al problema.
**Input:** *distancias*: matriz de distancias entre pares de elementos del conjunto.

```
    genes ← Generar una lista de 0's con un tamaño igual a tam

    /* Generar los genes del individuo */
5:  numGenes ← 0
    while numGenes != genesActivos do
        nuevaPos ← Número aleatorio entre 0 y tam − 1
        valor ← Valor de la posición nuevaPos en la lista de genes
        if valor == 0 then
10:         genes[nuevaPos] ← 1
            numGenes ← numGenes + 1

    /* Calcular el fitness de la solución */
    fitness ← Calcular fitness de la lista de genes
```

## 2.5. Mecanismo de selección del modelo generacional

Para generar una nueva población a partir de la población actual se selecciona un número de individuos igual al tamaño de la población actual. La selección se realiza mediante el torneo binario.

**Algorithm 6** Torneo binario

**Input:** *poblacion*: lista de individuos.
**Input:** *tam*: tamaño de la población que se tendrá en cuenta en el torneo.
**Input:** *tamTorneo*: número de individuos que realizan el torneo.
**Output:** *mejorIndividuo*: individuo ganador del torneo.

```
    /* Elegir los individuos que van a participar en el torneo */
    /* Se crea un set para que no haya posiciones repetidas y estén ordenadas las posiciones */
    elegidos ← Vacío
    while tamaño de  elegidos != tamTorneo do
 6:     elegido ← Número aleatorio entre 0 y tam − 1
        elegidos ← Añadir número elegido

    /* Comprobar que individuo es mejor */
    mejorFitness ← −1
    mejorIndividuo ← Vacío
12: elegidos_size ← tamañodelalistadeelegidos
    for i ∈ {0, ..., elegidos_size} do
        fitness ← Obtener fitness del individuo de la posición i
        if fitness ¿mejorFitness then
            mejorFitness ← fitness
            mejorIndividuo ← Individuo de la posición i
18: return mejorIndividuo
```

**Algorithm 7** Operador de selección en el modelo generacional

**Input:** *poblacion*: lista de individuos.
**Output:** *nuevaPoblacion*: lista de individuos seleccionados.

```
nuevaPoblacion ← poblacion
tam ← tamañodelapoblacion
/* Generar la nueva población */
for i ∈ {0, ..., tam − 1} do
    seleccionado ← Realizar un torneo binario con la población
    nuevaPoblacion ← Añadir el individuo seleccionado al final de la población

/* Eliminar los individuos de la población anterior */
for i ∈ {0, ..., tam − 1} do
    nuevaPoblacion ← Eliminar el primer individuo de la población
return nuevaPoblacion
```

## 2.6. Mecanismo de selección del modelo estacionario

Para seleccionar los individuos se utiliza el mismo torneo binario, la única diferencia es que la nueva población sólo tendrá un tamaño de dos individuos.

**Algorithm 8** Operador de selección en el modelo estacionario

**Input:** *poblacion*: lista de individuos.
**Input:** *tamNuevaPoblacion*: tamaño de la nueva población.
**Output:** *nuevaPoblacion*: lista de individuos seleccionados.

```
nuevaPoblacion ← Vacío
/* Generar la nueva población */
for i ∈ {0, ..., tamNuevaPoblacion − 1} do
    seleccionado ← Realizar un torneo binario con la población
    nuevaPoblacion ← Añadir el individuo seleccionado al final de la población
return nuevaPoblacion
```

## 2.7. Operador de mutación

En los distintos algoritmos genéticos y meméticos se realiza una mutación en la población en función de una probabilidad. La mutación de un gen en un cromosoma, consiste en intercambiar el valor del gen que se va a mutar por el valor de otro gen del mismo cromosoma, que se elige aleatoriamente, que tenga un valor contrario al gen que se está mutando; es decir, se realiza un intercambio de los valores. La mutación se hace de esta forma para mantener la factibilidad de la solución.

La probabilidad que se tiene en cuenta es la probabilidad de que mute un gen.

**Algorithm 9** Operador de mutación

**Input:** *poblacion*: lista de individuos.
**Input:** *probabilidadMutacion*: probabilidad de que mute un gen.
**Output:** *nuevaPoblacion*: lista de individuos mutados.

```
    nuevaPoblacion ← poblacion
    total ← Se calcula el número total de genes de la población
    numMutaciones ← probabilidadMutacion * total
    /* Se obtienen las distintas mutaciones */
    posiciones ← Vacío
    for i ∈ {0, ..., numMutaciones} do
        posicion ← Se obtiene un posición de la población de forma aleatoria
 9:     /* Se calcula a que cromosoma pertenece la posición */
        cromosoma ← posicion / Número de genes de un individuo
        /* Se calcula a que gen pertenece la posición */
        gen ← posicion % Número de genes de un individuo
        posiciones ← Añadir el par formado por cromosoma y gen

    /* Ordenar las posiciones de menor a mayor */
    posiciones ← Posiciones ordenadas de menor a mayor posición

18: /* Realizar las mutaciones */
    tamPosiciones ← Númerodeposiciones
    for i ∈ {0, ..., tamPosiciones} do
        valor ← Obtener el valor del gen situado en la posición i
        if valor == 0 then
            nuevaPoblacion ← Modificar el valor del gen situado en la posición i a 1
            /* Calcular aleatoriamente la posición de intercambio */
            encontrado ← False
            tam ← Tamaño de la lista de genes del individuo
27:         while encontrado == False do
                pos ← Número aleatorio entre 0 y tam − 1
                valor ← Valor del gen situado en la posición pos
                if valor == 1  pos != i then
                    nuevaPoblacion ← Modificar el valor del gen situado en la posición pos a 0
                    encontrado ← True
        else
            nuevaPoblacion ← Modificar el valor del gen situado en la posición i a 0
            /* Calcular aleatoriamente la posición de intercambio */
36:         encontrado ← False
            tam ← Tamaño de la lista de genes del individuo
            while encontrado == False do
                pos ← Número aleatorio entre 0 y tam − 1
                valor ← Valor del gen situado en la posición pos
                if valor == 0  pos != i then
                    nuevaPoblacion ← Modificar el valor del gen situado en la posición pos a 1
                    encontrado ← True
        cromosoma ← Obtener el cromosoma mutado
45:     cromosoma ← Indicar que no está actualizado el fitness
    return nuevaPoblacion
```

## 2.8. Operador de cruce uniforme

Este cruce consiste en a partir de dos individuos padre obtener un individuo hijo, que tendrá los mismos valores que los padres en los genes en que coincide el valor de ambos padres, y que tendrá en el resto un valor elegido al azar entre los dos valores que proporcionan los padres.

En el cruce se generan dos hijos a partir de los mismo padres, para la población siga teniendo el mismo tamaño.

**Algorithm 10** Operador de cruce uniforme

**Input:** *poblacion*: lista de individuos.
**Input:** *probabilidadMutacionCruce*: probabilidad de que un par de individuos se cruce.
**Input:** *numGenesFactible*: número de elementos para que una solución sea factible.
**Input:** *distancias*: matriz de distancias entre pares de elementos del conjunto.
**Output:** *nuevaPoblacion*: lista de individuos cruzados.

```
nuevaPoblacion ← poblacion
tam ← Tamaño de la población
numParejas ← tam / 2
numParejasCruce ← numParejas * probabilidadMutacionCruce
for i ∈ {0, ..., numParejasCruce − 1} do
    /* Obtener los padres */
    padre1 ← Obtener el primer individuo de la nueva población
    nuevaPoblacion ← Eliminar el primer individuo de la población
    padre2 ← Obtener el primer individuo de la nueva población
    nuevaPoblacion ← Eliminar el primer individuo de la población

    /* Generar los dos hijos */
    hijo ← Cruce uniforme entre el padre 1 y el padre 2
    hijo ← Convertir en una solución factible al hijo
    nuevaPoblacion ← Añadir el hijo al final de la nueva población
    hijo ← Cruce uniforme entre el padre 1 y el padre 2
    hijo ← Convertir en una solución factible al hijo
    nuevaPoblacion ← Añadir el hijo al final de la nueva población
return nuevaPoblacion
```

**Algorithm 11** Cruce uniforme

**Input:** *padre1*: Primer padre.
**Input:** *padre2*: Segundo padre.
**Output:** *hijo*: Individuo resultado del cruce.

```
hijo ← padre1hijo ← Indicar que el fitness del hijo no está actualizado

tam ← Tamañodelalistadegenesdelhijo
for i ∈ {0, ..., tam − 1} do
    valorPadre1 ← Valor del gen situado en la posición i en el padre 1
    valorPadre2 ← Valor del gen situado en la posición i en el padre 2

    /* Elegir de que padre coger el valor del gen */
    /* Sólo se tiene en cuenta si son distintos, porque sin son iguales al igualar el padre1 al
hijo, el hijo ya tendrá el valor correcto */
11: if valorPadre1 != valorPadre2 then
        elegido ← Número aleatorio entre 0 y 1
        /* Si sale 1 quiere decir que se ha elegido el gen del padre2 */
        if elegido == 1 then
            hijo ← Cambiar valor del gen situado en la posición i, al valor del gen del padre
situado en la misma posición
return hijo
```

## 2.9. Operador de reparación

Dado que el individuo hijo resultado del cruce uniforme es posible que sea una solución no factible, es necesario comprobarlo y si se da el caso de que no es factible solucionarlo, añadiendo o eliminando elementos seleccionados.

Como criterio para eliminar o añadir elementos seleccionados se utiliza la contribución. Los primeros elementos que se añaden o eliminan son los que tienen mayor contribución.

**Algorithm 12** Operador de reparación

**Input:** *hijo*: Hijo resultado del cruce uniforme.
**Input:** *numGenesFactible*: número de elementos necesarios para que una solución sea factible.
**Input:** *distancias*: matriz de distancias entre pares de elementos del conjunto.
**Output:** *hijoReparado*: Individuo resultado de la reparación.

```
hijoReparado ← hijo
/* Obtener el número de genes activos */
numGenesActivos ← 0
tam ← Tamaño de la lista de genes del hijo
for i ∈ {0, ..., tam − 1} do
    valor ← Valor del gen del hijo situado en la posición i
    if valor == 1 then
        numGenesActivos ← numGenesActivos + 1

/* Obtener la diferencia en el estado actual y el estado factible */
diferencia ← numGenesActivos − numGenesFactible

/* Convertir en factible al hijoReparado */
if diferencia > 0 then
    hijoReparado ← Eliminar tantos elementos seleccionados como indique diferencia
else
    if diferencia < 0 then
        hijoReparado ← Seleccionar tantos elementos no seleccionados como indique diferencia
return hijoReparado
```

**Algorithm 13** Eliminar genes activos

**Input:** *hijo*: Hijo resultado del cruce uniforme.
**Input:** *numElementos*: Número de elementos a eliminar.
**Input:** *distancias*: matriz de distancias entre pares de elementos del conjunto.
**Output:** *hijoReparado*: Individuo resultado de la reparación.

```
hijoReparado ← hijo
/* Inicializar las contribuciones de los elementos seleccionados */
tam ← Tamaño de la lista de genes del hijo
seleccionados ← Vacío
contribuciones ← Vacío
for i ∈ {0, ..., tam − 1} do
    valor ← Obtener el valor del gen del hijo situado en la posición i
    if valor == 1 then
        seleccionados ← Añadir i a la lista
        contribuciones ← Añadir el par formado por i y 0.0

/* Calcular las contribuciones */
tam ← Número de elementos en la lista seleccionados
for i ∈ {0, ..., tam − 2} do
    for j ∈ {i + 1, ..., tam − 1} do
        elem_i ← Obtener el elemento de seleccionados situado en la posición i
        elem_j ← Obtener el elemento de seleccionados situado en la posición j
        distancia ← Obtener la distancia entre elem_i y elem_j
        contribuciones ← Sumar distancia a la contribución de elem_i
        contribuciones ← Sumar distancia a la contribución de elem_j

/* Ordenar las contribuciones de mayor a menor contribución */
contribuciones ← Ordenar de mayor a menor contribución

/* Eliminar los genes activos */
for i ∈ {0, ..., numElementos − 1} do
    elegido ← Obtener la posición del elemento con mayor contribución
    hijoReparado ← Cambiar el valor del gen situado en la posición elegido a 0
    contribuciones ← Eliminar el primer elemento de la lista
return hijoReparado
```

**Algorithm 14** Añadir genes activos

**Input:** *hijo*: Hijo resultado del cruce uniforme.
**Input:** *numElementos*: Número de elementos a eliminar.
**Input:** *distancias*: matriz de distancias entre pares de elementos del conjunto.
**Output:** *hijoReparado*: Individuo resultado de la reparación.

```
hijoReparado ← hijo
/* Inicializar las contribuciones de los elementos seleccionados */
tam ← Tamaño de la lista de genes del hijo
seleccionados ← Vacío
noSeleccionados ← Vacío
contribuciones ← Vacío
for i ∈ {0, ..., tam − 1} do
    valor ← Obtener el valor del gen del hijo situado en la posición i
    if valor == 0 then
        noSeleccionados ← Añadir i a la lista
        contribuciones ← Añadir el par formado por i y 0.0
    else
14:     seleccionados ← Añadir i a la lista

/* Calcular las contribuciones */
tamNoSel ← Número de elementos en la lista noSeleccionados
tamSel ← Número de elementos en la lista seleccionados
for i ∈ {0, ..., tamNoSel − 1} do
    for j ∈ {0, ..., tamSel − 1} do
        elem_i ← Obtener el elemento de seleccionados situado en la posición i
        elem_j ← Obtener el elemento de seleccionados situado en la posición j
        distancia ← Obtener la distancia entre elem_i y elem_j
        contribuciones ← Sumar distancia a la contribución de elem_i

/* Ordenar las contribuciones de mayor a menor contribución */
contribuciones ← Ordenar de mayor a menor contribución

28: /* Añadir los genes activos */
for i ∈ {0, ..., numElementos − 1} do
    elegido ← Obtener la posición del elemento con mayor contribución
    hijoReparado ← Cambiar el valor del gen situado en la posición elegido a 1
    contribuciones ← Eliminar el primer elemento de la lista
return hijoReparado
```

## 2.10. Operador de cruce posición

Este cruce consiste en a partir de dos individuos padres obtener un individuo hijo, que tendrá los mismo valores que los padres en los genes en que coincide el valor de ambos padres, y que en

el resto de genes tendrá un valor al azar entre los valores de los genes que todavía quedan por fijar. El valor de los genes sin fijar se coge del primer padre en mi implementación.

En el cruce se generan dos hijos a partir de los mismo padres, para la población siga teniendo el mismo tamaño.

**Algorithm 15** Operador de cruce posición

**Input:** *poblacion*: lista de individuos.
**Input:** *probabilidadMutacionCruce*: probabilidad de que un par de individuos se cruce.
**Output:** *nuevaPoblacion*: lista de individuos cruzados.

```
nuevaPoblacion ← poblacion
tam ← Tamaño de la población
numParejas ← tam / 2
numParejasCruce ← numParejas * probabilidadMutacionCruce
for i ∈ {0, ..., numParejasCruce − 1} do
    /* Obtener los padres */
    padre1 ← Obtener el primer individuo de la nueva población
    nuevaPoblacion ← Eliminar el primer individuo de la población
    padre2 ← Obtener el primer individuo de la nueva población
    nuevaPoblacion ← Eliminar el primer individuo de la población

    /* Generar los dos hijos */
    hijo ← Cruce posición entre el padre 1 y el padre 2
    nuevaPoblacion ← Añadir el hijo al final de la nueva población
    hijo ← Cruce posición entre el padre 1 y el padre 2
    nuevaPoblacion ← Añadir el hijo al final de la nueva población
return nuevaPoblacion
```

**Algorithm 16** Cruce posición

**Input:** *padre1*: Primer padre.
**Input:** *padre2*: Segundo padre.
**Output:** *hijo*: Individuo resultado del cruce.

```
hijo ← padre1hijo ← Indicar que el fitness del hijo no está actualizado

/* Obtener las posiciones de los genes y sus valores, que tienen un valor distinto en los padres
*/
posACambiar ← Vacío
valores ← Vacío
tam ← Tamaño de la lista de genes del hijo
for i ∈ {0, ..., tam − 1} do
    valorPadre1 ← Valor del gen situado en la posición i en el padre 1
    valorPadre2 ← Valor del gen situado en la posición i en el padre 2

    if valorPadre1 != valorPadre2 then
        posACambiar ← Añadir la posición i
        valores ← Añadir valorPadre1

/* Fijar valor de los genes que nos son iguales */
numElem ← Tamaño de la lista valores
while numElem != 0 do
    elegido ← Número aleatorio entre 0 y numElem
    val ← Obtener el elemento de valores situado en la posición elegido
    pos ← Obtener el primer elementos de a lista posACambiar
    hijo ← Cambiarelvalordelgensituadoenlaposiciónposaval
    posACambiar ← Eliminar el primer elemento de la lista
    valores ← Eliminar el elemento situado en la posición elegido
return hijo
```

# 3. Descripción de la Búsqueda Local

## 3.1. Funciones comunes

### 3.1.1. Operador de generación de vecino

Como operador de generación de vecino se ha utilizado el intercambio, que consiste en elegir un elemento de la solución y otro elemento no seleccionado para la solución, e intercambiarlos de listas. Su implementación en pseudocódigo es la siguiente:

**Algorithm 17** Intercambio

**Input:** *solucion_actual*: lista con las etiquetas de los elementos que forman la solución actual.
**Input:** *elemento_a_sustituir*: índice del elemento que se quiere sustituir.
**Input:** *no_seleccionados*: lista con las etiquetas de los elementos que no forman parte de la solución.
**Input:** *elemento_a_incluir*: índice del elemento que se quiere incluir en la solución.
**Input:** *distancias*: matriz que guarda la distancia entre cada par de elementos del conjunto total.
**Input:** *contribuciones_actual*: vector de flotantes que contiene las contribuciones de cada elemento de la solución actual.
**Output:** *nueva_solucion*: lista con las etiquetas de los elementos que forman la nueva solución.
**Output:** *nuevas_contribuciones*: vector de flotantes que contiene las contribuciones de cada elemento de la nueva solución.

```
contri_elem_a_incluir ← 0
nuevas_contribuciones ←contribuciones_actual
/* Modificar las contribuciones de los elementos que no son
intercambiados */
for i ∈ solucion_actual do
    if i != elemento_a_sustituir then
        nuevas_contribuciones ← restar distancia del elemento i al elemento_a_sustituir
        nuevas_contribuciones ← sumar distancia del elemento i al elemento_a_incluir
        contri_elem_a_incluir ← añadir la distancia sumada a la contribución del nuevo elemento
nueva_solucion ← intercambiamos el nuevo elemento por el elemento elegido para ser sustituido
nuevas_contribuciones ← intercambiar también las contribuciones de los elementos intercambiados
return nueva_solucion, nuevas_contribuciones
```

### 3.1.2. Factorización de la BL

Dado que que la creación de un nuevo vecino sólo modifica un elemento de la solución original, por lo que el coste de la solución sólo es alterado por las distancias de ambos elementos involucrados en el operador de intercambio, parece mejor opción realizar una factorización de la función objetivo; es decir, en vez de volver a calcular todas las distancias de la nueva solución, sólo tenemos que restar las distancias del elemento sustituido al resto de elementos de la solución y

sumar las distancias del elemento incluido al resto de elementos de la solución. Su implementación en pseudocódigo es la siguiente:

**Algorithm 18** Función objetivo factorizada

**Input:** *solucion*: lista con las etiquetas de los elementos que forman la solución.
**Input:** *elemento_sustituido*: índice del elemento que se ha sustituido.
**Input:** *elemento_incluido*: índice del elemento que se ha incluido en la solución.
**Input:** *distancias*: matriz que guarda la distancia entre cada par de elementos del conjunto total.
**Input:** *coste_actual*: suma de la distancia total de la solución actual.
**Output:** *nuevo_coste*: suma de la distancia total de la solución modificada.
**Output:** *mejorado*: variable booleana que indica si ha mejorado el coste de la función.
**Output:** *elemento_incluido*: índice del elemento que se ha incluido en la solución.

```
    nuevo_coste ←coste_actual
    /* Modificar el coste de la nueva solución */
    for i ∈ solucion do
5:      if i != elemento_sustituido then
            nuevo_coste ← restar distancia del elemento i al elemento_sustituido
            nuevo_coste ← sumardistanciadelelemento i al elemento_incluido)
    /* Indicar si se ha mejorado el coste o no */
    if nuevo_coste > coste_actual then
10:     mejorado ← true
    else
        mejorado ← false
    return mejorado, elemento_incluido, nuevo_coste
```

### 3.1.3. Generación de soluciones aleatorias

Tanto las soluciones de la Búsqueda Local del Primero Mejor como de la Búsqueda Local del Mejor se crean de forma aleatoria.

En el caso del Mejor se crea de forma aleatoria la solución inicial, y no se crea de forma aleatoria las soluciones vecinas, ya que en este algoritmo se examina todo el entorno, por lo que es absurdo gastar tiempo en crear una solución aleatoria cuando se van a examinar todas.

Para crear la solución inicial se van eligiendo elementos del conjunto de forma aleatoria hasta tener el número de elementos necesario para forma una solución. El resto de elementos del conjunto se introducen en la lista de no seleccionados.

En el caso del Primero Mejor se crea de forma aleatoria tanto la solución inicial, como las soluciones vecinas, ya que en este caso se cogerá como vecina la primera solución que mejora la actual, por lo que no se exploran siempre todos los vecinos.

La solución inicial se genera de la misma forma que en del Mejor. La solución vecina se crea sustituyendo el elemento que menos aporta por uno de los elementos no seleccionados, para que la creación sea aleatoria antes de empezar a generar vecinos para ver cuál mejora la solución

actual, se baraja de forma aleatoria los elementos no seleccionados, para que a la hora de recorre los elementos de forma iterativa sean aleatorios los elementos que nos vayamos encontrando, esta acción se realiza con la función *shuffle*.

## 3.2. Búsqueda Local del Primero Mejor

### 3.2.1. Método de exploración del entorno

En esta variante de la Búsqueda Local se van generando vecinos hasta encontrar uno que mejore la solución actual, si se encuentra uno que la mejore se continua con la búsqueda; si no se encuentra un vecino que la mejore, se para el proceso de búsqueda y nos quedamos con la solución actual. Además se puede parar el proceso si se llega a un límite de iteraciones, y se incrementa una iteración por cada llamada a al función objetivo.

**Algorithm 19** Método de exploración del entorno ($BL\_PM$)

```
solucion_actual ← generar solución inicial aleatoria
iteraciones ← 0
hay_mejora ← true
while iteraciones ≤ 100000 & hay_mejora do
    noSeleccionadosNuevo ← barajar aleatoriamente la lista de elementos no seleccionados
    hay_mejora, elegido ← obtener el primer vecino que mejora el coste de la solución
    /* Si se encuentra un vecino que mejore el coste */
    if hay_mejora then
        solucion_modificada ← intercambiar el elemento elegido por el elemento de la solución
que menos aporte
        contribuciones_modi ← actualizar contribuciones con el nuevo elemento de la solución
```

## 3.3. Búsqueda Local del Mejor

### 3.3.1. Método de exploración del entorno

En esta variante de la Búsqueda Local se genera todo el entorno de una solución, y se elige la solución que consiga una mayor mejora de la solución actual, si no se alcanza ninguna mejor se para el proceso de búsqueda. Además se puede parar el proceso si se llega a un límite de iteraciones, y se incrementa una iteración por cada llamada a al función objetivo. Su implementación en pseudocódigo es la siguiente:

**Algorithm 20** Método de exploración del entorno ($BL\_M$)

*solucion_actual* ← *generar solución inicial aleatoria*
iteraciones ← 0
*hay_mejora* ← *true*
**while** iteraciones ≤ 100000 & *hay_mejora* **do**
    noSeleccionadosNuevo ← *barajar aleatoriamente la lista de elementos no seleccionados*
    *hay_mejora*, elegido ← *obtener el vecino que mejora en mayor medida el coste de la solución*
    /* Si se encuentra un vecino que mejore el coste */
    **if** *hay_mejora* **then**
        *solucion_modificada* ← *intercambiar el elemento elegido por el elemento de la solución que menos aporte*
        *contribuciones_modi* ← *actualizar contribuciones con el nuevo elemento de la solución*

# 4. Descripción del Greedy

## 4.1. Algoritmo de comparación

Para elegir que elemento incluir en la solución en cada momento se tiene en cuenta sus distancias con el resto de elementos.

En primer lugar al insertar el primer elemento se tiene en cuenta que el elemento insertado sea el que mayor distancia acumulada tenga.

**Algorithm 21** Elegir primer elemento a insertar en la solución

**Input:** *distancias*: matriz que guarda la distancia entre cada par de elementos del conjunto total.
**Input:** *noSeleccionados*: lista de elementos no incluidos en la solución.
**Output:** *elegido*: elemento elegido para ser insertado.

```
distanciasAcu ← inicializar vector a 0.0
noSeleccionados_size ← número de elementos no seleccionados
/* Calcular la distancia acumulada de cada elemento no seleccionado */
for i ∈ {0, ..., noSeleccionados_size − 1} do
    for j ∈ {i + 1, ..., noSeleccionados_size} do
        distanciasAcu ← aumentar distancia acumulada del elemento i
        distanciasAcu ← aumentar distancia acumulada del elemento j
distAcuMax ← 0.0
elegido ← vacio
/* Elegir el elemento con mayor distancia acumulada */
for i ∈ {noSeleccionados} do
    if distAcuMax < {distancia acumulada de i} then
        disAcuMax ← asigna la distancia acumulada del elemento i
        elegido ← i
return elegido
```

En el resto de elementos a insertar lo que que se tiene en cuenta es que tenga la mayor distancia de las mínimas distancias entre los elementos de la solución y los elementos no seleccionados.

**Algorithm 22** Elegir el resto de elementos a insertar en la solución

**Input:** *distancias*: matriz que guarda la distancia entre cada par de elementos del conjunto total.
**Input:** *solucion*: lista de elementos incluidos en la solución.
**Input:** *noSeleccionados*: lista de elementos no incluidos en la solución.
**Output:** *elegido*: elemento elegido para ser insertado.

```
    distanciaMin ← inicializar vector al máximo valor de un flotante
    /* Calcular la distancia mínima entre cada elemento de la solución y el resto de elementos
    no seleccionados */
    for i ∈ solucion do
        for j ∈ noSeleccionados do
            distanciaMin ← quedarse con lamínima distancia al elemento i
    distMax ← −1
    elegido ← vacio
    /* Elegir el elemento con mayor distancia mínima */
 9: for i ∈ solucion do
        if distMax < {distancia mínima de i} then
            distMax ← asigna la distancia mínima del elemento i
            elegido ← i
    return elegido
```

Se sigue el proceso de inserción de elementos hasta que la solución alcanza su tamaño necesario para ser válida como solución.

# 5. Descripción del Algoritmo Genético

## 5.1. Esquema de evolución y reemplazamiento

Para implementar el algoritmo genético hemos utilizado dos modelos:

### 5.1.1. Generacional

**Algorithm 23** Modelo generacional

**Input:** *poblacionIni*: Primer padre.
**Input:** *distancias*: matriz de distancias entre pares de elementos del conjunto.
**Input:** *numGenesFactible*: número de elementos necesarios para que la solución sea factible.
**Input:** *probabilidadCruce*: probabilidad de que una pareja de individuos se cruce.
**Input:** *probabilidadMutacion*: probabilidad de que un gen mute.
**Output:** *fitness*: mejor fitness encontrado en la ejecución del algoritmo.

```
poblacionActual ← poblacionIni
iteraciones ← 0
while iteraciones < 100000 do
    mejorPadre ← Primer elemento de la población actual
    poblacionActual ← Realizar el operador de selección generacional
    poblacionActual ← Realizar el operador de cruce
    poblacionActual ← Realizar el operador demutación
    poblacionActual ← Calcular el fitness de la población
    poblacionActual ← Realizar el operador de reemplazo generacional
    iteraciones ← iteraciones + número de evaluaciones
return fitness
```

**Algorithm 24** Operador de reemplazo generacional

**Input:** *poblacion*: lista de individuos.
**Input:** *mejorPadre*: mejor individuo de la población anterior.
**Output:** *poblacionActual*: lista de individuos después del reemplazo.

```
poblacionActual ← poblacion
poblacionActual ← Añadir el mejorPadre
poblacionActual ← Ordenardemayoramenorfitness

/* Búsqueda del mejor padre */
tam ← Tamaño de poblacionActual
encontrado ← 0
for i ∈ {0, ..., tam − 1} do
    if encontrado == 2 then
        break
    fitness ← Obtener el fitness del cromosoma situado en la posición i de la poblacionActual
    fitnessPadre ← Obtener el fitness del mejorPadre
    if fitness < fitnessPadre then
        break

    if fitness == fitnessPadre then
18:     cromosoma ← Obtener el cromosoma situado en la posición i de la poblacionActual
        if cromosoma == mejorPadre then
            encontrado ← encontrado + 1
            encontradoCromo ← cromosoma

if encontrado == 2 then
    poblacionActual ← Eliminar cromosoma encontradoCromo
else
    if encontrado == 1 then
        poblacionActual ← Eliminar un elemento de la población que no sea el mejorPadre
return poblacionActual
```

### 5.1.2. Estacionario

**Algorithm 25** Modelo estacionario

**Input:** *poblacionIni*: Primer padre.
**Input:** *distancias*: matriz de distancias entre pares de elementos del conjunto.
**Input:** *numGenesFactible*: número de elementos necesarios para que la solución sea factible.
**Input:** *probabilidadCruce*: probabilidad de que una pareja de individuos se cruce.
**Input:** *probabilidadMutacion*: probabilidad de que un gen mute.
**Output:** *fitness*: mejor fitness encontrado en la ejecución del algoritmo.

```
poblacionActual ← poblacionIni
iteraciones ← 0
while iteraciones < 100000 do
    nuevaPoblacion ← Vacío
    nuevaPoblacion ← Realizar el operador de selección estacionario
    nuevaPoblacion ← Realizar el operador de cruce
    nuevaPoblacion ← Realizar el operador de mutación
    nuevaPoblacion ← Calcular el fitness de la población
    poblacionActual ← Realizar el operador de reemplazo estacionario
    iteraciones ← iteraciones + número de evaluaciones
return fitness
```

**Algorithm 26** Operador de reemplazo estacionario

**Input:** *poblacionActual*: lista de individuos.
**Input:** *nuevaPoblacion*: lista de individuos.
**Output:** *poblacionActual*: lista de individuos después del reemplazo.

```
poblacionActual ← Añadir toda la nuevaPoblacion
poblacionActual ← Ordenar de mayor a menor fitness

/* Eliminar los peores individuos */
tam ← Tamaño de nuevaPoblacion
for i ∈ {0, ..., tam − 1} do
    poblacionActual ← Eliminar el último elemento de la lista
return poblacionActual
```

# 6. Descripción del Algoritmo Memético

## 6.1. Esquema de búsqueda

Todos los esquemas implementados se ejecutan cada 10 generaciones del algoritmo genético, y con una intensidad máxima de 400 iteraciones. Lo único que los diferencia es el número de cromosomas sobre los que se realiza la búsqueda local.

### 6.1.1. Primer esquema

En este esquema se realiza sobre toda la población.

**Algorithm 27** Primer esquema de búsqueda

**Input:** *poblacionIni*: Primer padre.
**Input:** *distancias*: matriz de distancias entre pares de elementos del conjunto.
**Input:** *numGenesFactible*: número de elementos necesarios para que la solución sea factible.
**Input:** *probabilidadCruce*: probabilidad de que una pareja de individuos se cruce.
**Input:** *probabilidadMutacion*: probabilidad de que un gen mute.
**Output:** *fitness*: mejor fitness encontrado en la ejecución del algoritmo.

```
poblacionActual ← poblacionIni
iteraciones ← 0
generacion ← 0
while iteraciones < 100000 do
    nuevaPoblacion ← Vacío
    nuevaPoblacion ← Realizar el operador de selección estacionario
    nuevaPoblacion ← Realizar el operador de cruce uniforme
    nuevaPoblacion ← Realizar el operador de mutación
    nuevaPoblacion ← Calcular el fitness de la población
    poblacionActual ← Realizar el operador de reemplazo estacionario
    iteraciones ← iteraciones + número de evaluaciones
    generacion ← generacion + 1
    if generacion == 10 then
        intensidad ← 400
        tam ← Tamaño de la poblacionActual
        for i ∈ {0, ..., tam − 1} do
            cromosoma ← Obtener el cromosoma de la poblacionActual situado en la posición i
            cromosoma ← Realizar una búsqueda local
        generacion ← 0
return fitness
```

### 6.1.2. Segundo esquema

En este esquema se realiza sobre el 10 % de la población.

**Algorithm 28** Segundo esquema de búsqueda

**Input:** *poblacionIni*: Primer padre.
**Input:** *distancias*: matriz de distancias entre pares de elementos del conjunto.
**Input:** *numGenesFactible*: número de elementos necesarios para que la solución sea factible.
**Input:** *probabilidadCruce*: probabilidad de que una pareja de individuos se cruce.
**Input:** *probabilidadMutacion*: probabilidad de que un gen mute.
**Output:** *fitness*: mejor fitness encontrado en la ejecución del algoritmo.

```
poblacionActual ← poblacionIni
iteraciones ← 0
generacion ← 0
while iteraciones < 100000 do
    nuevaPoblacion ← Vacío
    nuevaPoblacion ← Realizar el operador de selección estacionario
    nuevaPoblacion ← Realizar el operador de cruce uniforme
    nuevaPoblacion ← Realizar el operador de mutación
    nuevaPoblacion ← Calcular el fitness de la población
    poblacionActual ← Realizar el operador de reemplazo estacionario
    iteraciones ← iteraciones + número de evaluaciones
    generacion ← generacion + 1
    if generacion == 10 then
        intensidad ← 400
        tam ← Tamaño de la poblacionActual
        numCromosomas ← 0.1 * tam
        listaCromosomas ← Vacío
        tamLista ← 0
        while tamLista < numCromosomas do
            nuevo ← Número aleatorio entre 0 y tam − 1
22:         listaCromosomas ← Añadir nuevo
            tamLista ← tamLista + 1
        for i ∈ {0, ..., tamLista − 1} do
            cromosoma ← Obtener el cromosoma de la poblacionActual situado en la posición i
            cromosoma ← Realizar una búsqueda local
        generacion ← 0
return fitness
```

### 6.1.3. Tercer esquema

En este esquema se realiza sobre el 10 % de los mejores de la población.

**Algorithm 29** Tercer esquema de búsqueda

**Input:** *poblacionIni*: Primer padre.
**Input:** *distancias*: matriz de distancias entre pares de elementos del conjunto.
**Input:** *numGenesFactible*: número de elementos necesarios para que la solución sea factible.
**Input:** *probabilidadCruce*: probabilidad de que una pareja de individuos se cruce.
**Input:** *probabilidadMutacion*: probabilidad de que un gen mute.
**Output:** *fitness*: mejor fitness encontrado en la ejecución del algoritmo.

```
poblacionActual ← poblacionIni
iteraciones ← 0
generacion ← 0
while iteraciones < 100000 do
    nuevaPoblacion ← Vacío
    nuevaPoblacion ← Realizar el operador de selección estacionario
    nuevaPoblacion ← Realizar el operador de cruce uniforme
    nuevaPoblacion ← Realizar el operador de mutación
    nuevaPoblacion ← Calcular el fitness de la población
    poblacionActual ← Realizar el operador de reemplazo estacionario
    iteraciones ← iteraciones + número de evaluaciones
    generacion ← generacion + 1
    if generacion == 10 then
        intensidad ← 400
        tam ← Tamaño de la poblacionActual
        /* Obtener los cromosomas que se van a utilizar */
        numCromosomas ← 0.1 * tam
        /* Se recorre desde el principio porque la poblacionActual se encuentra ordenada de
mayor a menor fitness */
        for i ∈ {0, ..., numCromosomas − 1} do
            cromosoma ← Obtener el cromosoma de la poblacionActual situado en la posición i
            cromosoma ← Realizar una búsqueda local
        generacion ← 0
return fitness
```

# 7. Desarrollo de la práctica

La práctica la he implementado en C++, sin hacer uso de frameworks.

Para tomar los tiempos he utilizado el código del timer que se nos proporciona con la práctica. El resto de código está totalmente implementado por mi.

## 7.1. Compilación

Para compilar utilizo un archivo Makefile. Al ejecutar "make" en el terminal, se lanza su regla general, que genera todos los ejecutables, los ficheros objeto, y las carpetas necesarias. En el caso de los ejecutables crea dos por cada algoritmo, uno para depurar, y el otro para tomar tiempos con la opción de compilación -O2.

Para más información del Makefile abrir el archivo makefile y ver los comentarios y reglas que hay en él, el makefile se encuentra en la raíz del proyecto.

## 7.2. Limpiar ficheros derivados de la compilación

Para eliminar todos estos ficheros también hago uso del Makefile con la regla "make clean".

## 7.3. Obtener resultados

Para obtener los resultados he creado un script bash que ejecuta cada algoritmo con todos los casos y da como resultado un archivo CSV con los costes y tiempos del algoritmo en cada caso.

Para obtener los resultados de todos los algoritmos hay que ejecutar "./script_tiempos.sh" o ejecutar "make tiempos". Recomiendo esta última ya que da permisos al archivo para poder ejecutarse.

Y si se quiere obtener sólo el resultado de un algoritmo en concreto, se puede hacer ejecutando "./script_tiempos.sh <nombre del algoritmo sin extensión>".

Ejemplo:

- ./script_tiempos.sh BL_M
- ./script_tiempos.sh BL_PM
- ./script_tiempos.sh Greedy

Además de los CSV hay un archivo Excel de cada algoritmo que calcular la desviación de cada caso, la desviación media y el tiempo medio. Este archivo hay que modificarlo a mano, es decir, copiar y pegar los resultados del CSV.

Si se quiere ejecutar el archivo ejecutable de alguno de los algoritmos se debe ejecutar de la siguiente forma:

./<nombre del ejecutable> <seed> <nombre del fichero de datos con su ruta>

## 7.4. Distribución de carpetas

El proyecto está divido en las siguientes carpetas:

- src ← contiene los archivos fuente
- include ← contiene los archivos de cabecera
- data ← contiene los archivos de datos de los distintos casos
- tablas ← contiene los CSV y Excel con los resultados
- bin ← contiene los ejecutables
- obj ← contiene los ficheros objeto

Además en la raíz del proyecto se encuentra el Makefile y el script de bash.

# 8. Experimentos y Análisis de resultados

## 8.1. Descripción de los casos y valores de parámetros

Los casos utilizados se agrupan por diferente cantidad de elementos para el conjunto y cantidad de elementos para la solución. Para obtener los tiempos se han utilizado casos con los siguientes tamaños:

- 500 elementos en el conjunto, y 50 elementos para la solución
- 2000 elementos en el conjunto, y 200 elementos para la solución
- 3000 elementos en el conjunto, y 300, 400, 500, ó 600 elementos para la solución

Los argumentos pasados a los algoritmos son los siguientes:

- Nombre del caso
- Semilla

Para todas las ejecuciones de los algoritmos con los distintos casos se ha utilizado la semilla con valor 0.

## 8.2. Resultados obtenidos

Todos los tiempos están expresados en segundos.

### 8.2.1. Búsqueda Local

| **Algoritmo Búsqueda Local del Primero Mejor** | | |
|---|---|---|
| **Caso** | **Desv** | **Tiempo** |
| MDG-a_1_n500_m50 | 3,23 | 0,009096 |
| MDG-a_2_n500_m50 | 2,25 | 0,009857 |
| MDG-a_3_n500_m50 | 2,32 | 0,008824 |
| MDG-a_4_n500_m50 | 1,53 | 0,009175 |
| MDG-a_5_n500_m50 | 2,13 | 0,008875 |
| MDG-a_6_n500_m50 | 1,88 | 0,009707 |
| MDG-a_7_n500_m50 | 1,62 | 0,010312 |
| MDG-a_8_n500_m50 | 1,79 | 0,009197 |
| MDG-a_9_n500_m50 | 2,22 | 0,008761 |
| MDG-a_10_n500_m50 | 1,93 | 0,008628 |
| MDG-b_21_n2000_m200 | 0,73 | 0,217935 |
| MDG-b_22_n2000_m200 | 0,88 | 0,220469 |
| MDG-b_23_n2000_m200 | 1,20 | 0,221893 |
| MDG-b_24_n2000_m200 | 0,89 | 0,228545 |
| MDG-b_25_n2000_m200 | 0,87 | 0,224658 |
| MDG-b_26_n2000_m200 | 1,22 | 0,216504 |
| MDG-b_27_n2000_m200 | 1,31 | 0,219684 |
| MDG-b_28_n2000_m200 | 1,05 | 0,215919 |
| MDG-b_29_n2000_m200 | 1,21 | 0,221140 |
| MDG-b_30_n2000_m200 | 0,86 | 0,221838 |
| MDG-c_1_n3000_m300 | 0,68 | 0,434002 |
| MDG-c_2_n3000_m300 | 0,77 | 0,435650 |
| MDG-c_8_n3000_m400 | 0,41 | 0,617974 |
| MDG-c_9_n3000_m400 | 0,56 | 0,602094 |
| MDG-c_10_n3000_m400 | 0,60 | 0,590469 |
| MDG-c_13_n3000_m500 | 0,53 | 0,836042 |
| MDG-c_14_n3000_m500 | 0,34 | 0,805850 |
| MDG-c_15_n3000_m500 | 0,25 | 0,797442 |
| MDG-c_19_n3000_m600 | 0,39 | 1,009410 |
| MDG-c_20_n3000_m600 | 0,49 | 1,010500 |

Tabla 1: Resultados $BL\_PM$

| **Algoritmo Búsqueda Local del Mejor** | | |
|---|---|---|
| **Caso** | **Desv** | **Tiempo** |
| MDG-a_1_n500_m50 | 19,97 | 0,000028 |
| MDG-a_2_n500_m50 | 22,62 | 0,000029 |
| MDG-a_3_n500_m50 | 18,40 | 0,000030 |
| MDG-a_4_n500_m50 | 21,91 | 0,000028 |
| MDG-a_5_n500_m50 | 21,02 | 0,000052 |
| MDG-a_6_n500_m50 | 20,95 | 0,000032 |
| MDG-a_7_n500_m50 | 23,58 | 0,000029 |
| MDG-a_8_n500_m50 | 20,57 | 0,000023 |
| MDG-a_9_n500_m50 | 20,35 | 0,000022 |
| MDG-a_10_n500_m50 | 19,14 | 0,000031 |
| MDG-b_21_n2000_m200 | 11,67 | 0,000231 |
| MDG-b_22_n2000_m200 | 11,83 | 0,000226 |
| MDG-b_23_n2000_m200 | 12,05 | 0,000268 |
| MDG-b_24_n2000_m200 | 11,51 | 0,000235 |
| MDG-b_25_n2000_m200 | 10,95 | 0,000235 |
| MDG-b_26_n2000_m200 | 11,57 | 0,000227 |
| MDG-b_27_n2000_m200 | 11,92 | 0,000245 |
| MDG-b_28_n2000_m200 | 11,60 | 0,000245 |
| MDG-b_29_n2000_m200 | 12,08 | 0,000256 |
| MDG-b_30_n2000_m200 | 11,83 | 0,000223 |
| MDG-c_1_n3000_m300 | 9,96 | 0,000400 |
| MDG-c_2_n3000_m300 | 9,54 | 0,000401 |
| MDG-c_8_n3000_m400 | 8,02 | 0,000711 |
| MDG-c_9_n3000_m400 | 8,30 | 0,000595 |
| MDG-c_10_n3000_m400 | 7,94 | 0,000617 |
| MDG-c_13_n3000_m500 | 6,84 | 0,000879 |
| MDG-c_14_n3000_m500 | 6,81 | 0,000866 |
| MDG-c_15_n3000_m500 | 6,84 | 0,000859 |
| MDG-c_19_n3000_m600 | 6,04 | 0,001196 |
| MDG-c_20_n3000_m600 | 5,82 | 0,001183 |

Tabla 2: Resultados $BL\_M$

### 8.2.2. Greedy

| **Algoritmo Greedy** | | |
|---|---|---|
| **Caso** | **Desv** | **Tiempo** |
| MDG-a_1_n500_m50 | 24,63 | 0,00283 |
| MDG-a_2_n500_m50 | 21,77 | 0,00279 |
| MDG-a_3_n500_m50 | 23,50 | 0,00268 |
| MDG-a_4_n500_m50 | 23,39 | 0,00279 |
| MDG-a_5_n500_m50 | 24,98 | 0,00276 |
| MDG-a_6_n500_m50 | 22,74 | 0,00274 |
| MDG-a_7_n500_m50 | 24,36 | 0,00286 |
| MDG-a_8_n500_m50 | 24,61 | 0,00263 |
| MDG-a_9_n500_m50 | 19,67 | 0,00258 |
| MDG-a_10_n500_m50 | 26,99 | 0,00272 |
| MDG-b_21_n2000_m200 | 12,81 | 0,23355 |
| MDG-b_22_n2000_m200 | 12,53 | 0,23452 |
| MDG-b_23_n2000_m200 | 12,01 | 0,25347 |
| MDG-b_24_n2000_m200 | 12,48 | 0,24368 |
| MDG-b_25_n2000_m200 | 12,63 | 0,23630 |
| MDG-b_26_n2000_m200 | 12,21 | 0,26197 |
| MDG-b_27_n2000_m200 | 12,12 | 0,24787 |
| MDG-b_28_n2000_m200 | 12,58 | 0,22852 |
| MDG-b_29_n2000_m200 | 13,10 | 0,24468 |
| MDG-b_30_n2000_m200 | 12,53 | 0,22586 |
| MDG-c_1_n3000_m300 | 10,22 | 0,86770 |
| MDG-c_2_n3000_m300 | 10,55 | 0,82770 |
| MDG-c_8_n3000_m400 | 8,42 | 1,50415 |
| MDG-c_9_n3000_m400 | 8,47 | 1,53032 |
| MDG-c_10_n3000_m400 | 8,11 | 1,49027 |
| MDG-c_13_n3000_m500 | 7,15 | 2,24823 |
| MDG-c_14_n3000_m500 | 7,10 | 2,38319 |
| MDG-c_15_n3000_m500 | 7,21 | 2,18022 |
| MDG-c_19_n3000_m600 | 6,13 | 3,21624 |
| MDG-c_20_n3000_m600 | 6,24 | 3,22367 |

Tabla 3: Resultados Greedy

### 8.2.3. Algoritmo Genético

| **Algoritmo Genético Generacional Uniforme** | | |
|---|---|---|
| **Caso** | **Desv** | **Tiempo** |
| MDG-a_1_n500_m50 | 2,86 | 2,05134 |
| MDG-a_2_n500_m50 | 2,29 | 2,42187 |
| MDG-a_3_n500_m50 | 0,97 | 2,40216 |
| MDG-a_4_n500_m50 | 2,26 | 2,27867 |
| MDG-a_5_n500_m50 | 1,62 | 2,08868 |
| MDG-a_6_n500_m50 | 2,02 | 2,54159 |
| MDG-a_7_n500_m50 | 1,49 | 2,28424 |
| MDG-a_8_n500_m50 | 1,76 | 2,36072 |
| MDG-a_9_n500_m50 | 0,80 | 2,68403 |
| MDG-a_10_n500_m50 | 2,55 | 2,36472 |
| MDG-b_21_n2000_m200 | 1,27 | 52,48460 |
| MDG-b_22_n2000_m200 | 1,55 | 49,41550 |
| MDG-b_23_n2000_m200 | 1,14 | 55,40780 |
| MDG-b_24_n2000_m200 | 1,24 | 55,32120 |
| MDG-b_25_n2000_m200 | 1,53 | 44,71500 |
| MDG-b_26_n2000_m200 | 1,64 | 59,70840 |
| MDG-b_27_n2000_m200 | 1,29 | 55,97080 |
| MDG-b_28_n2000_m200 | 1,24 | 57,95830 |
| MDG-b_29_n2000_m200 | 1,92 | 49,19690 |
| MDG-b_30_n2000_m200 | 1,28 | 60,27940 |
| MDG-c_1_n3000_m300 | 1,15 | 169,7140 |
| MDG-c_2_n3000_m300 | 1,29 | 180,8270 |
| MDG-c_8_n3000_m400 | 0,87 | 262,0220 |
| MDG-c_9_n3000_m400 | 0,94 | 248,8020 |
| MDG-c_10_n3000_m400 | 1,28 | 295,1720 |
| MDG-c_13_n3000_m500 | 0,78 | 374,8730 |
| MDG-c_14_n3000_m500 | 0,96 | 376,7850 |
| MDG-c_15_n3000_m500 | 0,82 | 381,4910 |
| MDG-c_19_n3000_m600 | 0,70 | 555,2500 |
| MDG-c_20_n3000_m600 | 0,79 | 541,8560 |

Tabla 4: Resultados AGG-uniforme

| **Algoritmo Genético Generacional Posición** | | |
|---|---|---|
| **Caso** | **Desv** | **Tiempo** |
| MDG-a_1_n500_m50 | 3,12 | 0,68471 |
| MDG-a_2_n500_m50 | 3,90 | 0,68979 |
| MDG-a_3_n500_m50 | 2,89 | 0,69802 |
| MDG-a_4_n500_m50 | 2,84 | 0,69004 |
| MDG-a_5_n500_m50 | 3,18 | 0,69149 |
| MDG-a_6_n500_m50 | 2,48 | 0,68629 |
| MDG-a_7_n500_m50 | 1,79 | 0,68981 |
| MDG-a_8_n500_m50 | 2,57 | 0,68805 |
| MDG-a_9_n500_m50 | 1,41 | 0,68522 |
| MDG-a_10_n500_m50 | 3,46 | 0,69386 |
| MDG-b_21_n2000_m200 | 2,20 | 6,94798 |
| MDG-b_22_n2000_m200 | 2,78 | 6,86697 |
| MDG-b_23_n2000_m200 | 2,37 | 6,66475 |
| MDG-b_24_n2000_m200 | 2,22 | 5,96563 |
| MDG-b_25_n2000_m200 | 2,19 | 5,97550 |
| MDG-b_26_n2000_m200 | 2,40 | 5,96973 |
| MDG-b_27_n2000_m200 | 2,31 | 5,96078 |
| MDG-b_28_n2000_m200 | 2,53 | 5,99388 |
| MDG-b_29_n2000_m200 | 2,76 | 5,90010 |
| MDG-b_30_n2000_m200 | 2,19 | 7,05282 |
| MDG-c_1_n3000_m300 | 2,18 | 13,40250 |
| MDG-c_2_n3000_m300 | 2,45 | 13,64720 |
| MDG-c_8_n3000_m400 | 1,74 | 26,38530 |
| MDG-c_9_n3000_m400 | 1,81 | 23,31890 |
| MDG-c_10_n3000_m400 | 2,05 | 23,98740 |
| MDG-c_13_n3000_m500 | 1,63 | 38,72420 |
| MDG-c_14_n3000_m500 | 1,64 | 39,36490 |
| MDG-c_15_n3000_m500 | 1,56 | 40,33130 |
| MDG-c_19_n3000_m600 | 1,42 | 58,33060 |
| MDG-c_20_n3000_m600 | 1,48 | 57,41410 |

Tabla 5: Resultados AGG-posicion

| **Algoritmo Genético Estacionario Uniforme** | | |
|---|---|---|
| **Caso** | **Desv** | **Tiempo** |
| MDG-a_1_n500_m50 | 5,19 | 1,24700 |
| MDG-a_2_n500_m50 | 5,13 | 1,23490 |
| MDG-a_3_n500_m50 | 6,18 | 1,31716 |
| MDG-a_4_n500_m50 | 4,12 | 1,23839 |
| MDG-a_5_n500_m50 | 3,72 | 1,23140 |
| MDG-a_6_n500_m50 | 3,10 | 1,22887 |
| MDG-a_7_n500_m50 | 6,81 | 1,17513 |
| MDG-a_8_n500_m50 | 4,88 | 1,22157 |
| MDG-a_9_n500_m50 | 3,26 | 1,22962 |
| MDG-a_10_n500_m50 | 4,75 | 1,21637 |
| MDG-b_21_n2000_m200 | 5,60 | 12,09380 |
| MDG-b_22_n2000_m200 | 4,90 | 14,16980 |
| MDG-b_23_n2000_m200 | 4,25 | 12,09540 |
| MDG-b_24_n2000_m200 | 4,35 | 11,74830 |
| MDG-b_25_n2000_m200 | 4,00 | 13,95500 |
| MDG-b_26_n2000_m200 | 3,94 | 14,34370 |
| MDG-b_27_n2000_m200 | 4,02 | 12,72640 |
| MDG-b_28_n2000_m200 | 4,87 | 11,03590 |
| MDG-b_29_n2000_m200 | 3,98 | 12,71510 |
| MDG-b_30_n2000_m200 | 4,38 | 12,70990 |
| MDG-c_1_n3000_m300 | 4,98 | 25,88460 |
| MDG-c_2_n3000_m300 | 4,78 | 30,54290 |
| MDG-c_8_n3000_m400 | 3,47 | 45,42600 |
| MDG-c_9_n3000_m400 | 4,03 | 48,68960 |
| MDG-c_10_n3000_m400 | 3,65 | 47,76910 |
| MDG-c_13_n3000_m500 | 3,24 | 63,59920 |
| MDG-c_14_n3000_m500 | 3,30 | 62,87610 |
| MDG-c_15_n3000_m500 | 3,73 | 66,43930 |
| MDG-c_19_n3000_m600 | 2,97 | 88,10410 |
| MDG-c_20_n3000_m600 | 2,99 | 89,40250 |

Tabla 6: Resultados AGE-uniforme

| **Algoritmo Genético Estacionario Posición** | | |
|---|---|---|
| **Caso** | **Desv** | **Tiempo** |
| MDG-a_1_n500_m50 | 12,05 | 0,821339 |
| MDG-a_2_n500_m50 | 13,38 | 0,783013 |
| MDG-a_3_n500_m50 | 11,73 | 0,793680 |
| MDG-a_4_n500_m50 | 12,92 | 0,790551 |
| MDG-a_5_n500_m50 | 10,63 | 0,794508 |
| MDG-a_6_n500_m50 | 11,26 | 0,803923 |
| MDG-a_7_n500_m50 | 12,01 | 0,784719 |
| MDG-a_8_n500_m50 | 10,06 | 0,787745 |
| MDG-a_9_n500_m50 | 10,77 | 0,777115 |
| MDG-a_10_n500_m50 | 12,31 | 0,791747 |
| MDG-b_21_n2000_m200 | 8,37 | 6,986230 |
| MDG-b_22_n2000_m200 | 8,52 | 5,979240 |
| MDG-b_23_n2000_m200 | 8,33 | 6,991070 |
| MDG-b_24_n2000_m200 | 8,31 | 5,985400 |
| MDG-b_25_n2000_m200 | 8,18 | 5,953450 |
| MDG-b_26_n2000_m200 | 7,77 | 6,550930 |
| MDG-b_27_n2000_m200 | 8,05 | 6,354130 |
| MDG-b_28_n2000_m200 | 8,25 | 5,980130 |
| MDG-b_29_n2000_m200 | 8,82 | 5,952390 |
| MDG-b_30_n2000_m200 | 7,98 | 6,944240 |
| MDG-c_1_n3000_m300 | 7,19 | 15,23940 |
| MDG-c_2_n3000_m300 | 7,06 | 14,49210 |
| MDG-c_8_n3000_m400 | 6,42 | 24,42240 |
| MDG-c_9_n3000_m400 | 6,18 | 25,50000 |
| MDG-c_10_n3000_m400 | 5,95 | 25,19830 |
| MDG-c_13_n3000_m500 | 5,22 | 36,73110 |
| MDG-c_14_n3000_m500 | 5,22 | 38,16990 |
| MDG-c_15_n3000_m500 | 5,45 | 39,71850 |
| MDG-c_19_n3000_m600 | 4,86 | 52,45980 |
| MDG-c_20_n3000_m600 | 4,78 | 54,92730 |

Tabla 7: Resultados AGE-posicion

### 8.2.4. Algoritmo Memético

| **Algoritmo Memético (10,1.0)** | | |
|---|---|---|
| **Caso** | **Desv** | **Tiempo** |
| MDG-a_1_n500_m50 | 5,73 | 0,9600 |
| MDG-a_2_n500_m50 | 6,17 | 0,9432 |
| MDG-a_3_n500_m50 | 4,92 | 0,9652 |
| MDG-a_4_n500_m50 | 5,28 | 0,9502 |
| MDG-a_5_n500_m50 | 6,12 | 0,9410 |
| MDG-a_6_n500_m50 | 5,15 | 0,9308 |
| MDG-a_7_n500_m50 | 5,52 | 0,9514 |
| MDG-a_8_n500_m50 | 4,76 | 0,9153 |
| MDG-a_9_n500_m50 | 5,83 | 0,9425 |
| MDG-a_10_n500_m50 | 6,50 | 0,9690 |
| MDG-b_21_n2000_m200 | 4,20 | 10,6955 |
| MDG-b_22_n2000_m200 | 4,23 | 10,1560 |
| MDG-b_23_n2000_m200 | 4,08 | 10,3686 |
| MDG-b_24_n2000_m200 | 4,40 | 10,5511 |
| MDG-b_25_n2000_m200 | 3,93 | 10,2370 |
| MDG-b_26_n2000_m200 | 4,07 | 10,3353 |
| MDG-b_27_n2000_m200 | 3,63 | 10,6893 |
| MDG-b_28_n2000_m200 | 3,92 | 10,7476 |
| MDG-b_29_n2000_m200 | 4,10 | 10,4051 |
| MDG-b_30_n2000_m200 | 4,28 | 10,5951 |
| MDG-c_1_n3000_m300 | 3,38 | 22,1768 |
| MDG-c_2_n3000_m300 | 3,35 | 22,4184 |
| MDG-c_8_n3000_m400 | 3,04 | 29,3499 |
| MDG-c_9_n3000_m400 | 2,96 | 29,6037 |
| MDG-c_10_n3000_m400 | 3,03 | 29,7376 |
| MDG-c_13_n3000_m500 | 2,39 | 37,9994 |
| MDG-c_14_n3000_m500 | 2,20 | 37,3767 |
| MDG-c_15_n3000_m500 | 2,38 | 37,3277 |
| MDG-c_19_n3000_m600 | 2,11 | 46,6057 |
| MDG-c_20_n3000_m600 | 2,11 | 45,9050 |

Tabla 8: Resultados AM-(10,1.0)

| **Algoritmo Memético (10,0.1)** | | |
|---|---|---|
| **Caso** | **Desv** | **Tiempo** |
| MDG-a_1_n500_m50 | 2,42 | 1,3902 |
| MDG-a_2_n500_m50 | 1,72 | 1,5007 |
| MDG-a_3_n500_m50 | 2,85 | 1,2058 |
| MDG-a_4_n500_m50 | 1,42 | 1,6619 |
| MDG-a_5_n500_m50 | 2,39 | 1,3691 |
| MDG-a_6_n500_m50 | 1,51 | 1,8160 |
| MDG-a_7_n500_m50 | 1,36 | 1,4832 |
| MDG-a_8_n500_m50 | 1,81 | 1,4420 |
| MDG-a_9_n500_m50 | 1,71 | 1,8915 |
| MDG-a_10_n500_m50 | 2,54 | 1,5691 |
| MDG-b_21_n2000_m200 | 1,88 | 26,7251 |
| MDG-b_22_n2000_m200 | 2,03 | 27,3707 |
| MDG-b_23_n2000_m200 | 2,00 | 20,7778 |
| MDG-b_24_n2000_m200 | 1,84 | 26,9987 |
| MDG-b_25_n2000_m200 | 2,15 | 27,5935 |
| MDG-b_26_n2000_m200 | 1,79 | 24,4421 |
| MDG-b_27_n2000_m200 | 1,63 | 26,1138 |
| MDG-b_28_n2000_m200 | 2,06 | 25,9475 |
| MDG-b_29_n2000_m200 | 1,93 | 28,6723 |
| MDG-b_30_n2000_m200 | 1,75 | 30,0896 |
| MDG-c_1_n3000_m300 | 1,31 | 63,6325 |
| MDG-c_2_n3000_m300 | 1,61 | 68,9938 |
| MDG-c_8_n3000_m400 | 1,22 | 116,2590 |
| MDG-c_9_n3000_m400 | 1,63 | 100,8560 |
| MDG-c_10_n3000_m400 | 1,46 | 85,2201 |
| MDG-c_13_n3000_m500 | 1,38 | 132,2540 |
| MDG-c_14_n3000_m500 | 1,00 | 163,7750 |
| MDG-c_15_n3000_m500 | 1,07 | 120,2420 |
| MDG-c_19_n3000_m600 | 0,91 | 189,8850 |
| MDG-c_20_n3000_m600 | 1,09 | 162,3430 |

Tabla 9: Resultados AM-(10,0.1)

| **Algoritmo Memético (10,0.1) Mejores** | | |
|---|---|---|
| **Caso** | **Desv** | **Tiempo** |
| MDG-a_1_n500_m50 | 0,19 | 2,1148 |
| MDG-a_2_n500_m50 | 1,26 | 2,4121 |
| MDG-a_3_n500_m50 | 2,02 | 1,8862 |
| MDG-a_4_n500_m50 | 1,77 | 2,4470 |
| MDG-a_5_n500_m50 | 1,57 | 2,5367 |
| MDG-a_6_n500_m50 | 1,64 | 2,0857 |
| MDG-a_7_n500_m50 | 1,08 | 2,0600 |
| MDG-a_8_n500_m50 | 1,59 | 2,2310 |
| MDG-a_9_n500_m50 | 2,00 | 2,2897 |
| MDG-a_10_n500_m50 | 1,86 | 1,5884 |
| MDG-b_21_n2000_m200 | 2,06 | 32,2627 |
| MDG-b_22_n2000_m200 | 1,81 | 31,7925 |
| MDG-b_23_n2000_m200 | 1,92 | 38,8069 |
| MDG-b_24_n2000_m200 | 1,62 | 31,8356 |
| MDG-b_25_n2000_m200 | 1,74 | 39,0579 |
| MDG-b_26_n2000_m200 | 2,13 | 42,0403 |
| MDG-b_27_n2000_m200 | 1,65 | 38,5389 |
| MDG-b_28_n2000_m200 | 1,54 | 35,7289 |
| MDG-b_29_n2000_m200 | 1,10 | 50,1483 |
| MDG-b_30_n2000_m200 | 1,56 | 34,0905 |
| MDG-c_1_n3000_m300 | 1,09 | 131,4760 |
| MDG-c_2_n3000_m300 | 1,40 | 83,4747 |
| MDG-c_8_n3000_m400 | 1,35 | 142,3760 |
| MDG-c_9_n3000_m400 | 1,05 | 188,6070 |
| MDG-c_10_n3000_m400 | 1,42 | 134,5760 |
| MDG-c_13_n3000_m500 | 1,13 | 178,8560 |
| MDG-c_14_n3000_m500 | 0,79 | 189,8600 |
| MDG-c_15_n3000_m500 | 1,05 | 182,9620 |
| MDG-c_19_n3000_m600 | 0,91 | 234,7920 |
| MDG-c_20_n3000_m600 | 1,56 | 360,2740 |

Tabla 10: Resultados AM-(10,0.1,mej)

### 8.2.5. Resultados globales

| Algoritmo Greedy | Desv | Tiempo |
|---|---|---|
| Greedy | 14,71 | 0,73 |
| BL_PM | 1,2 | 0,31 |
| BL_M | 13,39 | 0,0003 |
| AGG-uniforme | 1,41 | 131,69 |
| AGG-posicion | 2,32 | 13,50 |
| AGE-uniforme | 4,29 | 23,62 |
| AGE-posicion | 8,6 | 13,28 |
| AM-(10,1.0) | 4,13 | 15,09 |
| AM-(10,0.1) | 1,72 | 49,45 |
| AM-(10,0.1,mej) | 1,46 | 74,11 |

Tabla 11: Resultados Globales

## 8.3. Análisis de resultados

### 8.3.1. Operadores de cruce

Para este apartado se van a utilizar las siguientes gráficas:

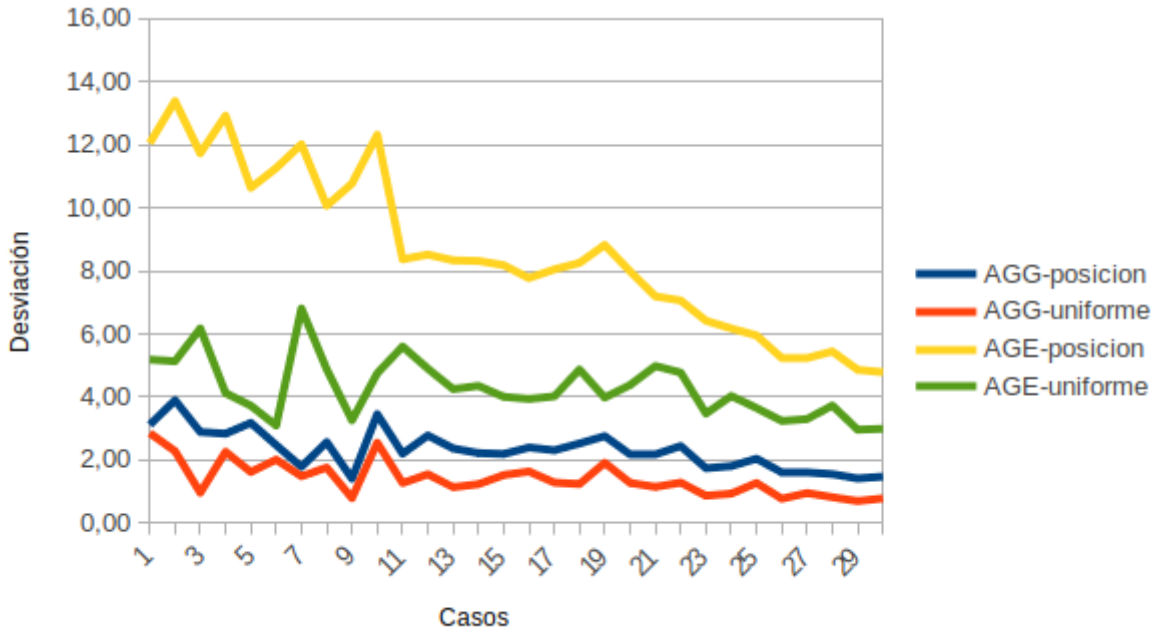


Figura 1: Evolución de la desviación - AG

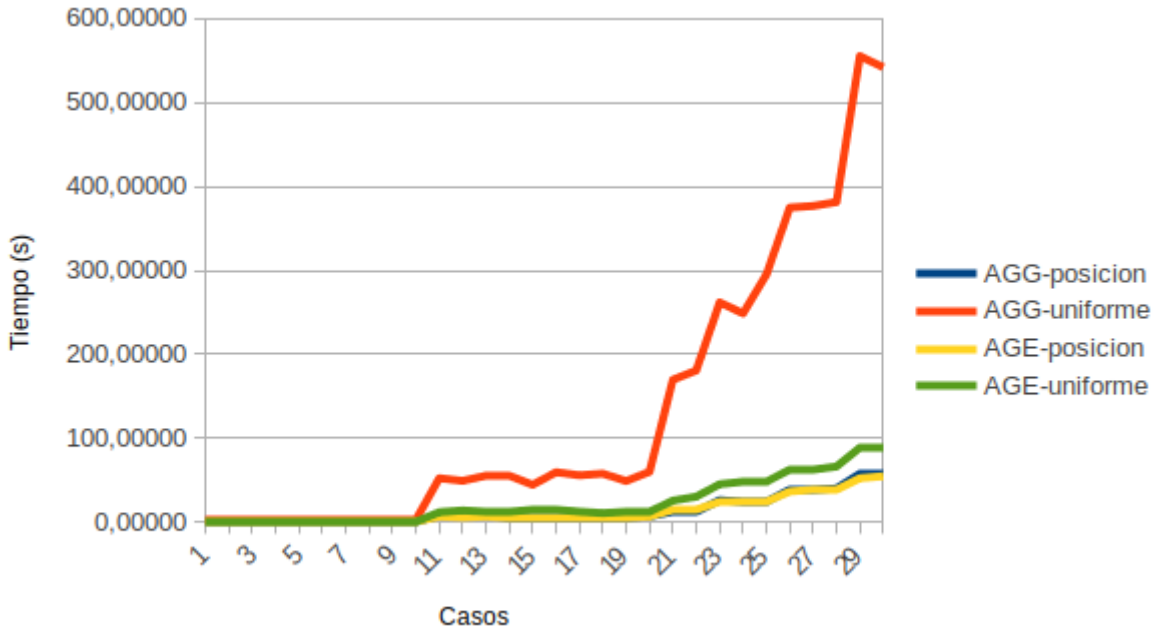


Figura 2: Evolución del tiempo - AG

En este punto se va a analizar el comportamiento de los dos operadores de cruce implementados junto con los dos modelo evolutivos utilizados.

En cuanto a la desviación, con la gráfica [1] se pueden extraer dos deducciones importantes.

En primer lugar independientemente del modelo evolutivo utilizado, el operador de cruce uniforme obtiene siempre una menor desviación respecto del operador de cruce posición, esto es debido al operador de reparación que utiliza el operador de cruce uniforme, ya que en este operador de reparación se utilizan las contribuciones para convertir en factible una solución, por lo tanto se está utilizando información del problema para seleccionar los elementos a eliminar o añadir. La utilización de esta información hace que el operador de cruce uniforme obtenga mejores resultados de fitness. Además se puede apreciar que cuanto mayor sea el tamaño de la nueva población por la que se reemplaza la actual, mejores resultados proporciona el operador de cruce uniforme.

En segundo lugar fijándonos en la influencia del modelo evolutivo en los resultados obtenidos, podemos comprobar como con el modelo evolutivo generacional se obtienen mejores resultados en todos los casos, aunque en algunos se queda cerca el otro modelo. Esto es debido a que en el modelo evolutivo generacional se reemplaza por completo la población por lo que en la selección habrá más posibilidades de elegir alguna solución prometedora, mientras que en el modelo evolutivo estacionario al seleccionar una población muy pequeña, en concreto de dos individuos, la probabilidad de que en esa población se encuentre una solución prometedora es menor, por lo que puede darse el caso de que no se seleccione una solución que podría dar buenos resultados, o que se selecciona cuando el algoritmo está cerca de terminar.

Una vez vista la desviación, ahora nos vamos a centrar en los tiempos obtenidos, con la

gráfica [2] y las tablas de los algoritmos que utilizan el operador de cruce uniforme y de cruce posición, se pueden extraer dos deducciones importantes.

En primer lugar viendo las tablas de resultados, se puede ver como los algoritmos que utilizan el operador de cruce tienen tiempos superiores que los que utilizan el operador de posición. Esto en parte se debe al operador de reparación que necesita el cruce uniforme, en el cual se deben calcular las contribuciones de algunos elementos, en concreto de los seleccionados a la hora de eliminar y de los no seleccionados a la hora de añadir; y debe calcularlas incluso cuando sólo hace falta eliminar o añadir un elemento a la solución.

En segundo lugar se puede ver como la diferencia entre ambos operadores es mayor con el modelo evolutivo generacional. Por lo que debe haber algo extra que haga que los tiempos aumenten, y la causa es el operador de reemplazo, ya que en el modelo generacional la nueva solución es de mayor tamaño lo que conlleva un mayor coste en el reemplazo, y además hay que tener en cuenta el elitismo en el reemplazo.

### 8.3.2. Mejor combinación de operador de cruce y modelo evolutivo

Viendo la gráfica [1] se puede ver que la mejor combinación es la del operador de cruce uniforme y el modelo evolutivo generacional. A pesar de que se obtienen los peores tiempos con esta combinación, esto es debido a lo poco optimizada que está la combinación. Teniendo todo en cuenta se elige esta combinación para implementar los algoritmo meméticos.

### 8.3.3. Mejoras de los AM's respecto del AG elegido

Para este apartado se van a utilizar las siguientes gráficas:

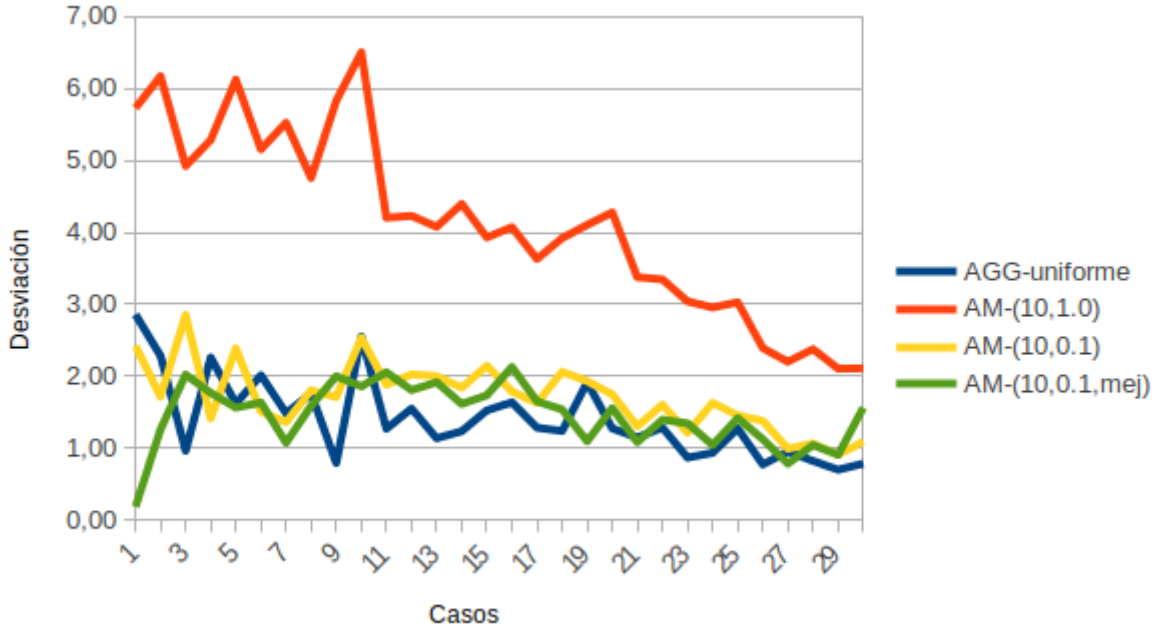


Figura 3: Evolución de la desviación - AG y AM

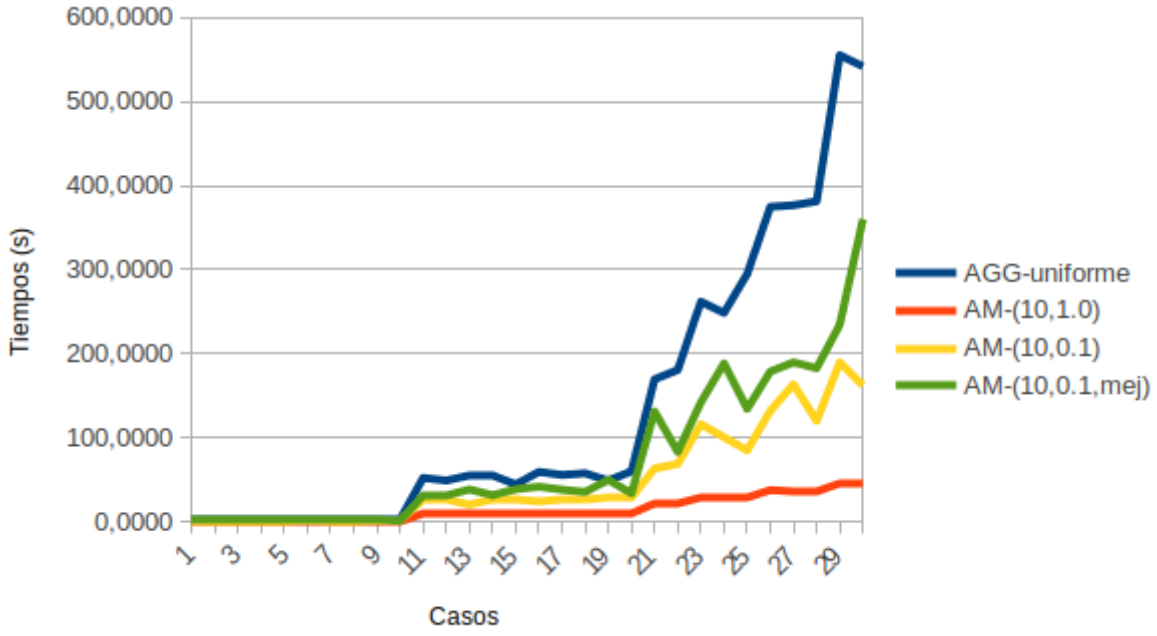


Figura 4: Evolución del tiempo - AG y AM

En este punto se va a comparar el comportamiento de los algoritmos meméticos y el algoritmo

genético en el que se basan los meméticos.

De la gráfica [3] se puede observar que el memético que realiza la búsqueda local sobre todos los cromosomas está muy separada del resto. La principal causa de esto es la limitación de evaluaciones que tienen los algoritmos, como se realiza la búsqueda local en todos los cromosomas, gastando evaluaciones en algunas soluciones que no tienen mucha mejora; se agotan muy rápido las evaluaciones y no da tiempo a llegar a una solución tan buena como el resto de algoritmos meméticos, por lo que la proporción exploración-explotación no está equilibrada perfectamente.

El resto de algoritmos están muy cercanos, y no sobresale ninguno sobre el resto, se van turnando como el algoritmo que mejor soluciones proporciona. Esto significa que estos meméticos tienen un buen equilibrio en la proporción exploración-explotación.

De la gráfica [4] se puede observar que todos los algoritmos meméticos mejoran en tiempos al algoritmo genético del que se basan. Esto es debido a que ahora parte de las evaluaciones son usadas por la búsqueda local, que es mucho más rápida que los operadores utilizados en el algoritmo genético generacional de cruce uniforme.

### 8.3.4. Comparación entre los AM's

Para este apartado se van a utilizar las siguientes gráficas:

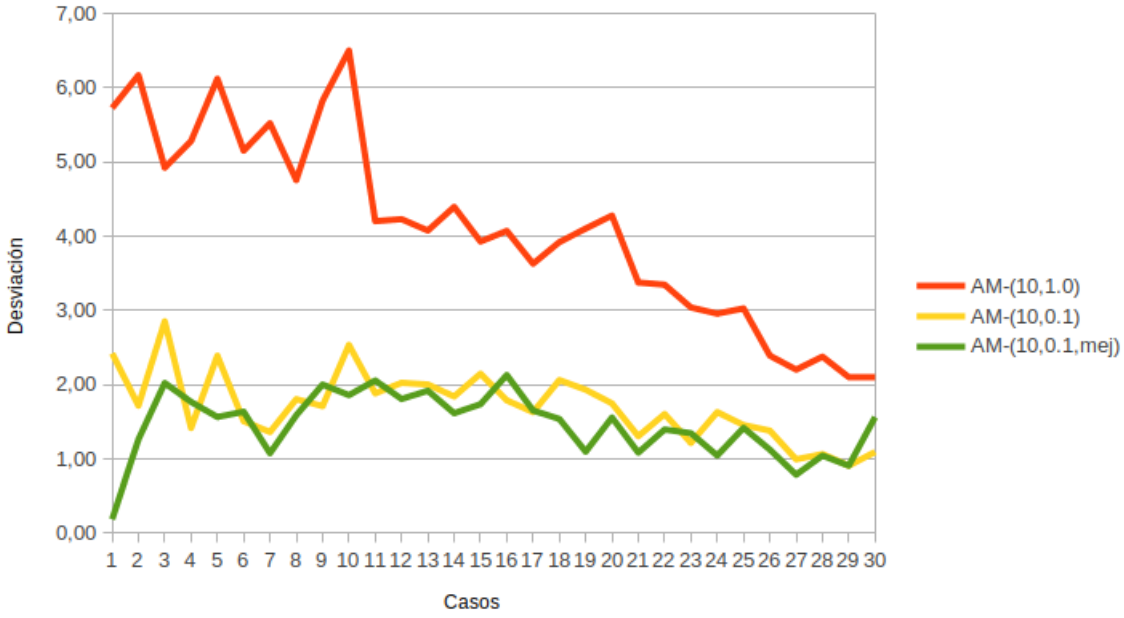


Figura 5: Evolución de la desviación - AM

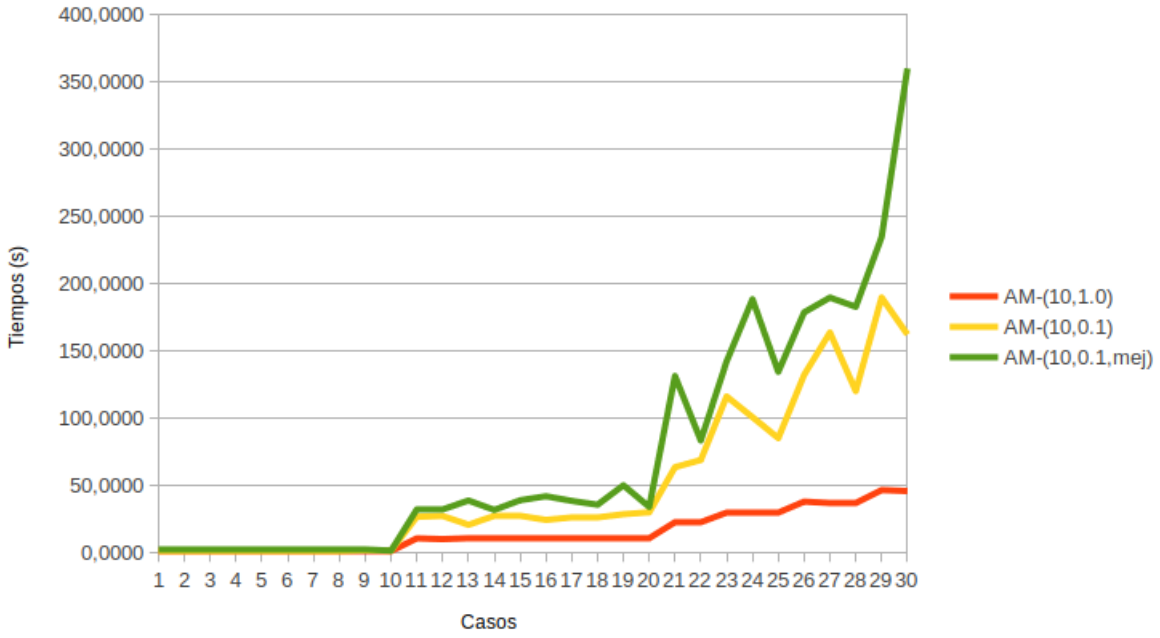


Figura 6: Evolución del tiempo - AM

De la gráfica [5] se puede extraer lo mismo que se extrajo con la gráfica [3] en el anterior apartado.

De la gráfica [6] se puede observar que hay una dependencia entre la proporción de exploración/exploración y los tiempos que se obtienen. Dado que la búsqueda local es mucho más rápida que el algoritmo genético generacional, los algoritmo meméticos que tengan un mayor número de evaluaciones gastadas en la explotación, obtendrá tiempos menores por esta razón. Por lo que tiene sentido que el memético que explota todos los cromosomas obtenga los menores tiempos, y los otros dos algoritmos al explotar sólo el 10 %, se obtiene tiempos mayores, y parecidos entre ambos algoritmos al ser la misma cantidad de cromosomas.

### 8.3.5. Comparación de BL con AG y AM

En base a los resultados de la tabla que contiene la media de la desviación y del tiempo de todos los algoritmos, se puede deducir que la búsqueda local obtiene mejores resultado tanto en la desviación como en los tiempos, pero creo que la comparación no se está realizando bajo las mismas reglas, ya que la búsqueda local está muy optimizada con la factorización de la función objetivo y los algoritmos genéticos y meméticos no están tan optimizados. Una comparación justa sería si se optimizase el AGG-uniforme, mejorando el operador de reemplazo y realizando alguna factorización en el cálculo del fitness. Pero en mi opinión creo que para los tamaños usados en las pruebas, aunque se optimizase el AGG-uniforme, es difícil que llegue a tiempos tan bajos como los de la BL; y aunque lo consiguiese la BL seguiría ganando en desviación media, ya que la optimización del AGG-uniforme no afectaría a la desviación obtenida.