UNIVERSIDAD DE GRANADA

Metaheurísticas

Práctica 2.a

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

Nombre: Francisco Javier Bolívar Expósito

DNI: ---

Email: ---

Grupo de prácticas: ---

Curso 2019-2020

Índice

1. Descripción del Problema

El problema de la máxima diversidad (**MDP**) consiste en **seleccionar** un **subconjunto** de *m* elementos de un conjunto más grande de *n* de forma que la **diversidad** entre los elementos escogidos se **maximice**.

Para calcular la diversidad consideraremos la variante del problema ***MaxSum***, en la que la diversidad se calcula como la suma de las distancias entre cada par de elementos seleccionados.

Por lo tanto el problema consistirá en elegir *m* elementos del conjunto inicial de *n* elementos de forma que la suma de las distancias entre ellos sea máxima, siendo la definición matemática:

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

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

$$x_i = \{0, 1\}, \quad i = 1, \ldots, n.$$

Xi contiene un 1 si el elemento "i" se ha seleccionado y un 0 si no

MDP es de la clase de problemas **NP-Duros** de optimización combinatoria, por lo que **no es factible** encontrar la solución óptima en un tiempo razonable. Por ello en esta práctica se desarrollan y analizan algoritmos eficientes para encontrar soluciones buenas aunque no sean las óptimas.

2. Aplicación de los algoritmos empleados al problema

La práctica consiste en aplicar 4 variantes de Algoritmos Genéticos y 3 variantes de Algoritmos Meméticos (uno de los algoritmos genético hibridado de distintas formas con la BL de la P1)

Primero vamos a ver varios aspectos comunes a la implementación de los distintos algoritmos al problema. Los operadores que son específicos de los algoritmos de la P1 se entienden explicados.

2.1.1 Esquema de representación de soluciones

Para la función objetivo y la Búsqueda Local se ha usado el esquema de representación de soluciones de la P1, que usa números enteros por ser más cómodo y eficiente que la representación binaria.

Siendo n = número de elementos en el conjunto inicial, m = número de elementos a seleccionar, la solución se representa como un vector de números enteros que:

- Contiene exactamente m elementos
- No contiene elementos repetidos,
- Cada elemento es un entero perteneciente al intervalo [0, n).

Sin embargo, para los operaciones necesarias en los algoritmos genéticos el esquema de representación de soluciones binario se adapta mejor.

Siendo n = número de elementos en el conjunto inicial, m = número de elementos a seleccionar, la solución se representa como un vector binario ‘x’ que:

- Tiene tamaño n
- Si Xi = 0 el elemento i no se ha seleccionado
- Si Xi = 1 el elemento i se ha seleccionado
- Contiene exactamente m componentes con valor 1.

2.1.2 Transformación entre las representaciones de soluciones

Es necesario poder pasar de una representación a otra para poder aplicar BL a un cromosoma de un algoritmo genético o para obtener la función objetivo de una solución en representación binaria. Esta transformación es rápida y sencilla.

Para pasar de la representación binaria a la decimal solo tendremos que crear un vector que contenga los indices de cada valor 1 en la solución binaria original. Si tenemos por ejemplo [0, 0, 1] en el que X_2 = 1 la representación decimal sería simplemente con el índice en el que se encuentra el 1 → [2]

Para pasar de la representación decimal a la binaria se presenta el siguiente pseudocódigo:

```
decimalABinaria(Entradas: ‘sol_d’ vector solución con representación decimal
                                Salida:‘sol_b’ vector solución con representación binaria):
  1. sol_b := vector con todas sus componentes a cero de tamaño N del problema.
  Para cada índice en sol_d:
     2. sol_b[índice] = 1
  devolver sol_b
```

2.2. Función objetivo

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

La función objetivo evalúa una solución para MDP siguiendo el modelo MaxSum, teniendo como entrada de la solución un vector con la representación decimal usada en la P1.

El objetivo de los algoritmos es maximizar esta función objetivo, que consiste en calcular la suma de distancias entre pares de elementos seleccionados.

La manera de implementar esta función ha sido cambiada con respecto a la práctica 1, ya que en esta práctica los algoritmos hacen un uso intensivo de esta función (con unas 100.000 ejecuciones en varios), por lo que su optimización es clave.

Primero se buscó la manera de realizar todas las sumas de manera vectorizada, todas a la vez, en vez de ir recorriendo una a una en un bucle. Si nos damos cuenta la fórmula descrita para la función objetivo con indices es equivalente a la suma de la matriz triangular superior, tomando como matriz una submatriz de la matriz de distancias que solo contenga las filas y columnas de los elementos seleccionados.

Tras probar varias formas de implementar esta operación con el lenguaje usado se encontró que el tiempo dedica al cálculo se reducía enormemente, pero ahora el cuello de botella se encontraba en los datos. Se redujo la cantidad de acceso a los datos y las copias realizadas, más algunas optimizaciones propias del lenguaje.

Finalmente el pseudocódigo de la función queda de la siguiente manera:

FuncionObjetivo(Entradas: distancias entre elementos ‘dist’ y vector solución “Sol”
Salida: Coste de la solución):
1. matrizDistanciasSeleccionadas := dist con las filas y columnas de los elementos en ‘Sol’
2. Coste := sumatoriaTriangularSuperior(matrizDistanciasSeleccionadas)
devolver Coste

Pseudocódigo de la implementación actual

FuncionObjetivo(Entradas: distancias entre elementos y vector solución “Sol”
Salida: Coste de la solución):
1. Coste = 0
Para i desde 0 hasta m - 1:
Para j desde i+1 hasta m:
2. Coste := Coste + distancia entre Sol_i y Sol_j
devolver Coste

Pseudocódigo de la implementación en P1

A continuación se muestra la diferencia entre los tiempos de ejecución evaluando la solución del caso MDG-a_32_n2000_m200

```
In [10]: timeit fObjetivo(dist, sel)
```

```
In [14]: timeit fObjetivo(1, dist, sel)
134 µs ± 1.22 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

Tiempo medio de la implementación actual

2.3.1. Generación de la Población

Seguiremos considerando n como el número de elementos seleccionables, m el número de elementos a seleccionar y ahora además la variable n_cromosomas como el número de cromosomas que tiene una población. Para los algoritmos genéticos la población inicial se genera de forma aleatoria como se indica en el siguiente pseudocódigo:

inicializarPoblacion():
1. Poblacion := matriz de ceros de tamaño n_cromosomas * n

Para cada Cromosoma en Poblacion (cada fila):
2. posiciones a seleccionar := elegir m números aleatorios diferentes de 0 a n
3. Cromosoma[posiciones a seleccionar] = 1

2.3.2. Operador de Selección

Para realizar la selección tanto en los algoritmos genéticos generacionales como en el estacionario, aunque usándose un número diferente de veces, se utiliza el torneo binario. Este consiste en seleccionar dos padres de manera aleatoria y elegir el mejor de ellos como hijo.

En vez de hacerlo uno a uno los números aleatorios necesarios se generan todos primero en un vector aleatorio, por cuestiones de eficiencia. Teniendo esto en cuenta el Pseudocódigo queda de la siguiente manera:

seleccionTorneoBinario(poblacion, n_hijos_a_seleccionar):
1. Participantes := Matriz de números aleatorios de 0 a n_cromosomas en la que cada columna contiene los participantes de un torneo binario.

Para cada Columna en Participantes:
2. Añadir_a_población_de_hijos(Mejor padre entre poblacion[Columna[0]] y poblacion[Columna[1]])

2.3.3. Operador de Mutación

Cada gen tiene una probabilidad de mutar muy pequeña, en vez de generar un número aleatorio para saber si cada gen muta, operación muy costosa sabiendo que la mayoría no muta. Consideraremos la aleatoriedad que ya introduce el operador de selección y calcularemos la esperanza matemática de

genes que van a mutar: n_mutaciones = probabilidadDeMutacion * n_cromosomas_que_pueden_mutar * n_genes, mutando siempre este número de genes.

Para seleccionar los genes a mutar escogeremos un cromosoma y un gen aleatorio. Siguiendo la misma lógica del ejercicio anterior generaremos todos los números aleatorios necesarios para mutar una población a la vez. Después intercambiaremos este gen por otro del cromosoma con valor contrario para seguir manteniendo una solución factible.

mutar(poblacion, p_mutacion):
1. n_mutaciones := p_mutacion * n_cromosomas * n_genes
2. index_crom = vector de tamaño n_mutaciones con números aleatorios entre 0 y n_cromosomas
3. index_gen_i = vector de tamaño n_mutaciones con números aleatorios entre 0 y n_genes

Para i desde 0 hasta n_mutaciones:
    4. index_gen_j = Escoger una posición aleatoria de aquellas de poblacion[index_crom[i]] que != poblacion[index_crom[i]][index_gen_i[i]]
    5. Intercambiar poblacion[index_crom[i]][Index_gen_i[i] por poblacion[index_crom[i]][index_gen_j]

2.3.4 Operadores de cruce

Al igual que en el operador anterior consideraremos la aleatoriedad del operador de selección y cruzaremos tantas parejas como indique n_cruces = probabilidadDeCruce * n_cromosomas, cruzando los n_cruces primeros elementos de la población. En los subapartados siguientes se define específicamente como se cruza cada pareja de padres, ahora se mostrará el procedimiento genérico para cruzar una población.

cruzarPoblacion(poblacion, p_cruce, forma_de_cruzar):
1. n_cruces := p_cruce * n_cromosomas / 2

Para i desde 0 hasta n_cruces*2 con i+=2 por iteración
    2. hijos[i], hijos[i+1] := cruzar poblacion[i] y poblacion[i+1] usando forma_de_cruzar

2.3.4.1 Operador de Cruce Uniforme

Siempre se usan los operadores de cruce para obtener dos hijos cruzando dos veces dos padres, se ha implementado este operador de esta forma en vez de usar dos veces el operador para cada hijo repitiendo cálculos.

El cruce uniforme consiste en conservar en los hijos aquellas posiciones que contengan el mismo valor en ambos padres (para preservar las selecciones prometedoras) y seleccionar las restantes aleatoriamente entre un padre u otro.

cruzarUniforme(padre_a, padre_b):

1. Copiar en Hijo_a e Hijo_b padre_a.
2. index_valores_diferentes = posiciones en las que padre_a[posicion] != padre_b[posicion]
3. selPadre$_1$:= generamos vector de tamaño igual a index_valores_diferentes con números aleatorios entre 0 y 1
4. selPadre$_2$:= generamos un vector igual que en el paso 3.

Para cada i en index_valores_diferentes

5. Hijo_a[i] = padre_a[i] si selPadre$_1$[i] == 0, padre_b[i] si no.
6. Hijo_b[i] = padre_a[i] si selPadre$_2$[i] == 0, padre_b[i] si no

7. Devolvemos Hijo_a e Hijo_b.

Al seleccionar un valor aleatorio entre los dos padres podemos obtener una solución no factible. Para solucionar esto es necesario aplicar el operador de reparación del siguiente apartado.

2.3.4.2 Operador de Reparación

El operador de reparación consiste en añadir los elementos que más contribuyan al valor de la solución si faltan elementos y eliminar los elementos que más contribuyan al valor de la solución si sobran.

En los foros de prado se comentó que el operador original está pensado de esta manera (eliminando las mejores selecciones) para mejorar o empeorar la solución y así conseguir un equilibrio entre exploración y explotación en el algoritmo genético.

También se ha explorado la opción que tiene más explotación, al eliminar los elementos que menos contribuyan al valor de la solución, simplemente cambiando el paso 3. del siguiente pseudocódigo por ‘i := posición con menor valor de z’.

repararCromosoma(cromosoma):

1. Sol = representacionBinariaADecimal(cromosoma)

Si tamaño(Sol) > m

2. z := Sumas de la distancia entre cada elemento seleccionado y todos los demas (contribución de cada elemento a la solución)

Hacer:

3. i = posición con mayor valor de z
4. cromosoma[i] = 0
5. z := z – distancias entre los elementos seleccionados y el elemento eliminado
6. Eliminar z[i] y Sol[i]

Mientras tamaño(Sol) > m

Si tamaño(Sol) < m

7. z := Sumas de la distancia de cada elemento NO seleccionado a todos los seleccionados

Hacer:

8. i = posición con mayor valor de z
9. cromosoma[i] = 0

9. z := z + distancias entre los elementos seleccionados y el elemento añadido
11. Eliminar z[i] y Sol[i]

Mientras tamaño(Sol) > m

2.3.4.3 Operador de Cruce Posición

El cruce por posición consiste en conservar en los hijos aquellas posiciones que contengan el mismo valor en ambos padres (para preservar las selecciones prometedoras) y seleccionar las restantes de un padre en un orden aleatorio. Al conservar menos información de los padres es un cruce más disruptivo y es más complicado que converja.

cruzarPosicion(padre_a, padre_b):

1. Copiar en Hijo_a e Hijo_b padre_a.
2. index_valores_diferentes = posiciones en las que padre_a[posicion] != padre_b[posicion]
3. ordenAleatorio_a := generamos una permutación de index_valores_diferentes
4. ordenAleatorio_b := generamos un vector igual que en el paso 3.

Para cada i en index_valores_diferentes

5. Hijo_a[index_valores_diferentes[i]] = padre_a[ordenAleatorio_a]
5. Hijo_a[index_valores_diferentes[i]] = padre_a[ordenAleatorio_b]

7. Devolvemos Hijo_a e Hijo_b.

3. Descripción de la implementación de los Algoritmos

3.1. Algoritmo Genético Generacional con elitismo, con cruce Uniforme y Posición

Utilizaremos un esquema de evolución generacional, y le aplicaremos el cruce uniforme en una versión y el cruce posición en otra. En este algoritmo se generarán tantos hijos como hay en la población inicial, y finalmente se remplazará a toda la población inicial asegurándose de que se conserva al menos el mejor cromosoma de esta. Haremos uso de los operadores definidos anteriormente para describir la estructura del algoritmo.

agg(evaluaciones_maximas):
1. inicializarPoblacion()
2. Para cada cromosoma de la poblacion calcular su valor con
   fObjetivo(dist, representacionBinariaADecimal(cromosoma))
3. evals = n_cromosomas

Mientras evals < evaluaciones_maximas
   4. seleccionarTorneoBinario(Poblacion, n_cromosomas)
   5. cruzarPoblacion(poblacion, p_cruce=0.7, forma_de_cruzar=Uniforme ó Posicion)
   6. mutar(poblacion, p_mutacion=0.001)

   7. Calcular como en paso 2. el valor para la población ‘hijos’
   8. mejor_cromosoma := Cromosoma con mayor valor en la población actual
   Si mejor_cromosoma no está en hijos
      9. Intercambiar mejor cromosoma por el peor cromosoma de hijos
   10. poblacion := hijos
11. Devolvemos el mejor cromosoma de la población

3.2 Algoritmo Genético Estacionario Uniforme y Posición

Utilizaremos un esquema de evolución estacionario, y le aplicaremos el cruce uniforme en una versión y el cruce posición en otra. En este algoritmo se generarán tan solo dos hijos, que sustituirán a los dos peores de la población actual si son mejores que ellos. Haremos uso de los operadores definidos anteriormente para describir la estructura del algoritmo.

age(evaluaciones_maximas):

1. inicializarPoblacion()
2. Para cada cromosoma de la poblacion calcular su valor con
   fObjetivo(dist, representacionBinariaADecimal(cromosoma))
3. evals = n_cromosomas

Mientras evals < evaluaciones_maximas

4. seleccionarTorneoBinario(Poblacion, n_a_seleccionar=2)
5. cruzarPoblacion(poblacion, p_cruce=1, forma_de_cruzar=Uniforme ó Posicion)
6. mutar(poblacion, p_mutacion=0.001)

7. Calcular como en paso 2. el valor para la población ‘hijos’

Pada cada hijo (solo hay 2)

Si el hijo es mejor solución que el peor de la población

8. El hijo sustituye en la población al peor de esta

9. Devolvemos el mejor cromosoma de la población

3.3 Algoritmos Meméticos

Para desarrollar los distintos algoritmos meméticos se ha tomado como base el AGG_Uniforme por ser el que mejor resultados proporcionaba, y se le aplica una búsqueda local a ciertos cromosomas según los parámetros indicados.

am(evaluaciones_maximas, p_bl, periodo, mejor=False):
1. inicializarPoblacion()
2. Para cada cromosoma de la poblacion calcular su valor con
   fObjetivo(dist, representacionBinariaADecimal(cromosoma))
3. evals = n_cromosomas

Mientras evals < evaluaciones_maximas
   4. seleccionarTorneoBinario(Poblacion, n_cromosomas)
   5. cruzarPoblacion(poblacion, p_cruce=0.7, forma_de_cruzar=Uniforme)
   6. mutar(poblacion, p_mutacion=0.001)

   7. Calcular como en paso 2. el valor para la población ‘hijos’
   8. mejor_cromosoma := Cromosoma con mayor valor en la población actual
   Si mejor_cromosoma no está en hijos
      9. Intercambiar mejor cromosoma por el peor cromosoma de hijos
   10. poblacion := hijos

   Cada periodo generaciones de poblacion
      11.n_busquedas_locales = n_cromosomas * p_bl
      12. Aplicamos una BL suave de 400 evaluaciones a tantos cromosomas como indica n_busquedas_locales, de forma aleatoria si mejor==False, escogiendo los n_busquedas_locales mejores si mejor==True
      13. Actualizamos evals sumandole las evaluaciones realizadas en las Bls

14. Devolvemos el mejor cromosoma de la población

3.4 BL sobre el resto de algoritmos

Se ha analizado también los datos de aplicar una búsqueda local con 100.000 evaluaciones máximas a las soluciones obtenidas también con todos los algoritmos. Esto es interesante porque nos permite asegurarnos de que la solución este en un óptimo local, nunca la va a empeorar, el tiempo de ejecución es despreciable y nos permite ver una aproximación de cuanto han convergido los algoritmos originales.

No se describe el pseudocódigo ya que no tiene más complejidad que aplicar la BL sobre las soluciones obtenidas.

4. Implementación de la práctica y manual de usuario

La implementación de la práctica se ha realizado en **Python**, usando como interprete para las pruebas Python 3.8.1.

Se han utilizado 4 módulos, **time** para medir los tiempos de ejecución, **os** para recorrer el directorio de datos, **numpy** para realizar la mayoría de operaciones y **pandas** para guardar los resultados y generar las hojas de cálculo siendo estos dos últimos los no incluidos en Python.

Se ha utilizado un '**profiler**' para detectar las partes del código que más tiempo de ejecución ocupan y optimizarlas. La conclusión obtenida es que las partes que más tiempo de computación requieren son la función objetivo por su enorme cantidad de llamadas y el operador de reparación, por necesitar calcular las contribuciones de los elementos.

Se han optimizado estas partes lo máximo posible intentando realizar **operaciones vectorizadas** con los arrays de Numpy, **evitando copias innecesarias** de estos vectores y optimizando el acceso a memoria.

También se consideraron otras dos posibles optimizaciones.

La primera era tener **ordenados** los cromosomas **en función de su fitness** o valor de la función objetivo, sin embargo el tiempo de búsqueda del mejor cromosoma era prácticamente despreciable en comparación a los verdaderos cuellos de botella del programa y mantener los cromosomas en una estructura que permitiera cálculos rápidos (arrays de numpy) y el orden de estos a la vez introducía una complejidad que al final se considero peor opción.

La segunda era usar una **tabla hash con clave vector solución y valor el fitness asociado**. De esta forma no sería necesario re-calcular la función objetivo si sobrevive un cromosoma o lo volvemos a obtener.

Sin embargo no se puede obtener un hash de un array, y había que transformarlo en otra estructura, la forma más eficiente que se encontró era obtener directamente los bytes del array frente a convertirlo en un string o una lista o cualquiera de estas opciones después de transformar la representación binaria a decimal. Finalmente el tiempo para obtener un hash de la clase era ⅓ aproximadamente del tiempo necesario para calcular la función objetivo. La necesidad de comprobar si cada cromosoma se encontraba en la tabla hash provocaba peores tiempos de ejecución que simplemente calcular siempre las funciones objetivo de cada generaciones, por lo que se descartó la idea.

La implementación se ha realizado en una clase de Algoritmos que hacen uso de Genéticos para el MDP, desde esta se pueden cargar los datos de un caso del problema y ejecutar los distintos algoritmos implementadas con sus parámetros.

Cada operador y cada algoritmo está definido como una función diferente de la clase, que tienen fácil acceso desde dentro de la clase a la población actual y la población de hijos que se esté generando.

Para usar el programa es necesario instalar NumPy y Pandas. Se incluye un archivo **requirements.txt** que se puede usar para instalar la versión usada, utilizando “pip” con ‘pip install -r requirements.txt’.

Todo está contenido en un único script **MDP.py** que cuelga del directorio **FUENTES.** La ejecución

5. Análisis de los resultados

Para todos los algoritmos genéticos y meméticos se ha usado la semilla 510 y 100.000 evaluaciones de la función objetivo como máximo. Se han medido todos los tiempos en segundos.

Las datos obtenidos ocupan una gran cantidad de espacio, en total con los algoritmos originales considerados en la práctica, las dos versiones del operador de reparación, las mejoras aplicando búsqueda local y los 3 algoritmos considerados en la práctica anterior hay 27 tablas con casos, resultados, tiempos y desviación. Teniendo en cuenta que estos resultados se pueden consultar en la hoja de cálculo adjunta en el zip de la práctica no voy a copiar estos resultados o hacer gráficas comparando todos esto resultados, simplemente pasaré a comentar las conclusiones obtenidas de estos.

Los algoritmos genéticos basados en el cruce de posición son los que dan peores resultados, su desviación media es mucho mayor que todos los algoritmos de la P1, aun necesitando sobre 15 segundos.

Al no compartir tanta información de los padres, con este cruce no se conservan las selecciones prometedoras, es muy disruptivo y por lo tanto tiene problemas para converger.

Esto también explica la diferencia de resultados entre el Generacional y el Estacionario. El Estacionario se comporta mejor con este cruce que el algoritmo generacional (1,4 de desv media frente a 2,7), porque al tener mayor elitismo compensa algo las dificultades para converger.

Si aplicamos una Búsqueda Local fuerte a la solución se obtiene una mejora significativa, pasando de desv 2.7 a 0.8 y de 1.4 a 1.08, lo que nos indica que las soluciones se encontraban lejos del óptimo local.

Con los algoritmos genéticos del cruce uniforme presentado a las diapositivas se obtienen unos resultados algo mejores, aunque aumentando la media de tiempo sobre los 25 segundos por el coste que conlleva el operador de reparación.

Son similares los resultados con este cruce con ambos esquemas de evolución, con desv 1,11 y 1,102. No hay una diferencia tan grande como con el cruce de posición al no tener tantos problemas para converger. Por lo menos tenemos soluciones que mejoran al Greedy, pero todavía no obtenemos soluciones mejores a la BL.

Al aplicar a la solución la Búsqueda Local obtenemos una mejora significativa otra vez, por lo que vemos que los Algoritmos Genéticos no han terminado de converger. En este caso el generacional con BL consigue una desviación media de 0.77 y el estacionario de 0.91. El estacionario se queda en óptimos locales menos buenos, tal vez por no explorar tanto como el generacional. ¡Aun así ambas consiguen superar los resultados de todas los algoritmos de la P1!

Partiendo de que el operador de reparación intuitivamente no parece muy buena idea que elimine las mejores soluciones, pudiendo ser algo disruptivo y eliminando información del padre que es bastante buena, se han hecho las pruebas de los algoritmos con cruce uniforme (incluidos los meméticos que analizaremos más adelante) eliminando primero las peores selecciones.

En ambos algoritmos genéticos se consigue mejora, especialmente significativa en el genético. Desv media 0,8 en el generacional y 1,01 en el estacionario. Ya conseguimos solo con el algoritmo genético mejores soluciones que con los algoritmos simples de la P1 y aplicando la BL bajamos la desviación a 0.73 y 0.9, las mejores soluciones hasta el momento.

Parece que el Algoritmo Genético Uniforme Generacional con el operador de reparación que elimina las peores selecciones y añade las mejores es el que mejor resultados proporciona, ahora analizaremos que pasa con los algoritmos meméticos al hibridarlo con la búsqueda local, ya que si tiene demasiada explotación podría dar peores resultados. Estudiaremos los AM con los dos operadores de reparación.

Una ventaja claramente aparente del algoritmo memético es que la BL es mucho más rápida, por lo que al gastar evaluaciones en la BL el algoritmo finaliza mucho antes, con medias sobre 1,5 segundos. Vemos primero con el operador de reparación original.

El AM con (10, 1) da peores resultados que solo el algoritmo (desv 1,24), y prácticamente no mejora al aplicar la búsqueda local(desv 1,20), con soluciones especialmente malas en los casos más grandes. La explotación es demasiado alta al aplicar la búsqueda local sobre todos los cromosomas, por lo que están todos cerca de converger, además se gastan las evaluaciones en esta búsqueda local sin dar casi ningún lugar a las generaciones. En los casos grandes nos quedamos en peores óptimos locales.

El AM con (10, 0.1) proporciona un importante cambio, con desv 0,78 al tener mejor equilibrio entre explotación y exploración y llegando a 0.66 tras aplicar la BL a la solución final. Por ahora la mejor solución hasta el momento y en un tiempo muy reducido

El AM con (10, 0.1mej) sin necesidad de mejorarlo con la BL consigue una desviación media de 0.68. Siendo este el algoritmo que proporciona mejores resultados de forma consistente, en un tiempo muy reducido.

Los AMs con el otro operador de reparación considerado siguen dando soluciones muy buenas, pero son ligeramentes peores que con el operador original. La explotación que proporciona la búsqueda local ya es suficientemente buena, por lo que introducir explotación en el operador de reparación no proporciona ninguna mejora.

Como conclusión final, los algoritmos de la P1 son muy simples y quedan fácilmente atrapados en óptimos locales, por lo que los algoritmos estudiados en esta práctica que exploran y explotan el espacio de búsqueda pueden proporcionar soluciones mejores. Es muy importante sin embargo mantener un buen equilibrio entre exploración y explotación, porque si no tal como hemos visto en los algoritmos basados en el cruce de posición podemos llegar a obtener resultados incluso peores. También aplicar una búsqueda local a las soluciones obtenidas nos asegura una solución en un óptimo local y no añade prácticamente tiempo de ejecución (0.3 segundos), por lo que es una buena mejora.