

Metaheurísticas

Guión de prácticas 2

Grupo 5 de prácticas

Francisco Javier Sánchez Olmo -
77689760S

fjso0004@red.ujaen.es

Álvaro Martín Bacas - 76737859S

amb00093@red.ujaen.es

-
- Algoritmo Generacional
 - Algoritmo Estacionario



Universidad de Jaén

| | |
|---------------------------------------|-----------|
| 1. Introducción | 3 |
| 1.1 Planteamiento del problema | 3 |
| 1.2 Algoritmos empleados | 3 |
| 1.3 Representación de la solución | 3 |
| 1.4 Restricciones y consideraciones | 3 |
| 1.5 Operadores comunes | 4 |
| 1.5.1 Configuración del problema | 4 |
| 1.5.2 Métodos comunes | 5 |
| 1.5.2 Librerías utilizadas | 5 |
| 3. Algoritmo Generacional. | 6 |
| 3.1 Descripción. | 6 |
| 3.2 Pseudocódigo. | 7 |
| 3.3 Análisis | 11 |
| 4. Algoritmo Estacionario | 13 |
| 4.1 Descripción. | 13 |
| 4.2 Pseudocódigo. | 14 |
| 4.3 Análisis | 16 |
| 5. Resultados globales | 16 |
| 5.1 Comparativa gráfica de resultados | 16 |
| 5.2 Análisis de resultados | 16 |
| 5.3 Comparación entre algoritmos | 17 |
| 6. Fichero log | 18 |
| 7. Bibliografía | 18 |

1. Introducción

1.1 Planteamiento del problema

El problema que se presenta para esta práctica es el del Viajante de Comercio (TSP). En este caso, tenemos una serie de archivos con las coordenadas de las distintas ciudades que tiene que recorrer el viajero y nos encargaremos de encontrar una solución siguiendo una serie de algoritmos.

1.2 Algoritmos empleados

Durante las distintas sesiones llevadas a cabo para completar la práctica, se trabaja sobre una serie de algoritmos evolutivos en los que el objetivo es encontrar una ruta lo más corta posible en la que el viajante debe recorrer todas las ciudades del problema, volviendo a la inicial.

En esta práctica se implementaron los algoritmos evolutivos “Generacional” y “Estacionario” con los operadores de cruces “OX2” y “MOC”, los cuales se explicarán más adelante en este informe.

1.3 Representación de la solución

La solución se verá representada en todos los algoritmos como una secuencia de valores enteros, en los que cada valor representa el índice de la ciudad visitada, junto con la distancia total recorrida por el viajante en la ruta obtenida en la solución.

1.4 Restricciones y consideraciones

En cuanto al desarrollo de las prácticas, hemos decidido realizarlas en Python, ya que es un lenguaje que está siendo utilizado de una forma muy extendida hoy en día.

En cuanto al problema, se han de tener en cuenta una serie de restricciones y consideraciones planteadas en la definición general de un problema del tipo TSP,:

- **La solución es un ciclo cerrado:** la solución terminará en la misma ciudad en la que se empezó.
- **Cada ciudad será visitada una única vez:** el viajante no puede pasar dos veces por el mismo punto.

- **Todas las ciudades están conectadas entre sí:** el viajante podrá dirigirse hacia cualquier ciudad desde cualquier otra en la que se encuentre.
- **Tamaño de la solución:** el tamaño de la solución **siempre** será igual al *número de ciudades* + 1. Esto es debido a que, como se ha comentado anteriormente, el viajante debe visitar todas las ciudades una única vez, y volver a la inicial.

1.5 Operadores comunes

1.5.1 Configuración del problema

Para la ejecución del problema, se requieren una serie de parámetros que afectan en la forma de comportarse del algoritmo. Con el fin de no tener que estar modificando dichos parámetros en todos los archivos en los que sean requeridos, se incluye en el directorio raíz del proyecto un fichero “*config.txt*” en el que se indicarán una serie de parámetros:

- **Archivos:** este apartado de configuración contendrá el nombre de los archivos con las coordenadas de las ciudades. En este caso serán 5 problemas los que se van a analizar: “a280.tsp”, “ch130.tsp”, “d18512.tsp”, “pr144.tsp” y “u1060.tsp”
- **Semillas:** al trabajar con algoritmos probabilísticos, la aleatoriedad puede influir en gran medida en los resultados obtenidos. Es por eso que se inicializan una serie de semillas para cada problema con el fin de generar siempre las mismas secuencias de números aleatorios que se requieren para la ejecución de los algoritmos.

En nuestro caso, las semillas utilizadas son 76737859, 76775839, 77637859, 56739877 y 79737856, inicializadas en la *Ejecución 1*, *Ejecución 2*, *Ejecución 3*, *Ejecución 4* y *Ejecución 5*, respectivamente.

- **Algoritmos:** los algoritmos implementados en la práctica: *generacional*, *estacionario*.
- **K_GREEDY:** número de candidatos para seleccionar mediante el greedy
- **individuos_iniciales:** tamaño de la población inicial para los algoritmos evolutivos.
- **numero_max_elites:** número de individuos que se mantendrán con cada iteración para conservar el elitismo.
- **k_best_gen y k_best_est:** número de individuos que serán seleccionados para el torneo de ganadores.

- **k_worst_gen y k_worst_est:** número de individuos que serán seleccionados para el torneo de perdedores.
- **Parada_iteraciones y parada_tiempo:** condiciones de parada. O bien el número máximo de evaluaciones, o bien por tiempo de ejecución.
- **Porcent_cruce_gen:** Probabilidad de usar el operador de cruce del algoritmo generacional.
- **Porcent_mutacion:** Probabilidad que tiene un hijo de mutar.
- **Individuos_iniciales_aleatorios:** Porcentaje de población completamente aleatoria.
- **Individuos_iniciales_greedy:** Porcentaje de población creada mediante el algoritmo greedy.
- **Tipo_cruce:** para seleccionar entre los operadores de cruce “OX2” y “MOC”.

1.5.2 Métodos comunes

- **generar_poblacion_inicial:** método que genera la población inicial con 80% generados completamente aleatorios y un 20% generados por el algoritmo greedy.
- **torneo_ganadores:** método que selecciona el mejor individuo entre k_best individuos.
- **torneo_perdedores:** método que selecciona el peor individuo entre k_worst individuos.
- **cruce_ox2:** método que implementa la lógica del operador de cruce “OX2”.
- **cruce_moc:** método que implementa la lógica del operador de cruce “MOC”.
- **mutacion_2opt:** método que intercambia dos posiciones en una ruta.


1.5.2 Librerías utilizadas

- **Numpy:** es una librería especializada en trabajar con arrays multidimensionales, realizar computaciones numéricas de forma eficiente, es de código abierto y nos ha sido bastante fácil de utilizar. Gracias a Numpy, el programa es capaz de ejecutar en mucho menos tiempo de lo que lo haría utilizando estructuras y bucles convencionales.
- **Scipy.spatial.distance:** en concreto el método cdist, que nos permite generar la matriz de distancias en una línea de código tardando apenas 1 segundo en el caso de la matriz más grande.
- **Logging:** para realizar el fichero con los logs
- **uuid:** para generar un identificador único con cada ejecución para distinguir en los logs y poder comparar ejecuciones.
- **time:** para medir tiempos de ejecución.

- **random:** para obtener aleatorios.
- **os:** para la lectura de directorios

1.6 Aspecto importante a mencionar

En este informe, las tablas se incluirán en formato imagen, ya que al haber tantas tablas, ocupaban un espacio excesivo de la documentación. Sin embargo, tanto mediante en el siguiente enlace

 Tablas_MH2024.xlsx como en la [bibliografía](#), se encuentra el enlace al archivo original de hojas de cálculo donde se puede comprobar la información más detalladamente.

3. Algoritmo Generacional.

3.1 Descripción.

Primero de los algoritmos evolutivos de esta práctica. Es un tipo de metaheurística basada en poblaciones (P-metaheurística), lo que significa que el algoritmo irá evolucionando y mejorando una población iterativamente.

Este tipo de algoritmos evolutivos, como su nombre indica, consisten en una evolución. Están basados en los principios de evolución natural establecidos por Darwin en 1859.

En esta práctica, cada individuo de la población está representado por:

- **Genotipo:** Una solución al problema del TSP, expresada como una ruta que recorre todas las ciudades y vuelve a la inicial.
- **Fenotipo:** La calidad de esa solución, medida como la distancia total recorrida en la ruta.

De este modo, cada individuo combina tanto la representación de la solución como su evaluación, permitiendo que el algoritmo pueda acceder a cada elemento de la población de una forma más eficiente y comparar y ordenar dicha población según la calidad de las soluciones.

Tanto este algoritmo como el siguiente, al ser evolutivos, siguen una estructura general compuesta por tres fases principales:

1. **Selección:** Se eligen los individuos que formarán la población padre, es decir, aquellos que participarán en el proceso de reproducción. Se reproducirá en este caso todos los padres, pero

siguiendo un sistema de cruce mediante torneo de ganadores, es decir, se reproducen entre sí aquellos que tengan una mejor calidad.

2. **Cruce:** Se aplican operadores genéticos y/o mutaciones con cierta probabilidad a los individuos seleccionados para generar una nueva población descendiente. En este caso, según la configuración para la ejecución aplicaremos el operador de cruce OX2 o el operador de cruce MOC. En caso de no darse la probabilidad de cruce, obtendrán los valores de sus padres.
3. **Reemplazamiento:** Se decide qué elementos de la población serán reemplazados por la nueva generación para conformar la nueva población. En este caso, toda la población es reemplazada excepto los élites, que reemplazan a los individuos resultantes del torneo de perdedores.

En el caso del algoritmo generacional, destaca un mecanismo llamado **elitismo**, que garantiza la supervivencia de los mejores individuos de cada generación. Según el parámetro definido, se preserva un conjunto de individuos élite que pasan automáticamente a la siguiente generación. Estos individuos seguirán en la población mientras mantengan su posición como los mejores, asegurando la conservación de soluciones prometedoras en la población.

Este algoritmo se ejecutará hasta que se hayan realizado una serie de evaluaciones de individuos, o se haya llegado a un tiempo establecido. Ambos indicados en el archivo de configuración de la práctica.

3.2 Pseudocódigo.

Algoritmo Generacional

```
poblacion ← generar_poblacion_inicial()
elites ← lista vacía
ordenar(poblacion, por distancia)
inicio ← tiempo actual

elites ← primeros n_elites elementos de población

evaluaciones ← tamaño(poblacion)
nueva_generacion ← lista vacía
MIENTRAS no se cumpla criterio de parada:
    evaluaciones ← evaluaciones + 1
    PARA _en rango(tamaño_poblacion):
        MIENTRAS padre_1 != padre_2:
            padre_1 ← torneo_ganadores()
            padre_2 ← torneo_ganadores()
        FIN MIENTRAS
    SI percent_random < percent_cruce:
        hijo_1, hijo_2 ← cruce(padre_1, padre_2)
```

```

SI NO:
    hijo_1, hijo_2 ← padres
FIN SI
SI percent_random < percent_mutacion:
    mutar(hijo_1)
FIN SI
SI percent_random < percent_mutacion:
    mutar(hijo_2)
FIN SI
nueva_generacion.anadir(hijo_1, hijo_2)
evaluaciones ← evaluaciones + 2
FIN PARA
Comprobar si están elites
MIENTRAS no estén todos:
    perdedor ← torneo_perdedores()
    nueva_generacion.eliminar(perdedor)
    nueva_generacion.anadir(elite_que_falta)
FIN MIENTRAS
limpiar(elites)
elites.anadir(primeros n_elites elementos en nueva_generacion ordenada)
poblacion ← nueva_generacion
nueva_generacion ← lista vacía
FIN MIENTRAS

mejor_ruta, mejor_dist ← primer elemento poblacion ordenada por dist
DEVOLVER mejor_ruta, mejor_dist

```

generar_población_inicial

```

padres_iniciales ← lista vacía

PARA _ en rango(individuos_iniciales_greedy * tamaño_población):
    ruta, dist ← ejecutar_greedy_aleatorio
    añadir (ruta, dist) a padres_iniciales
FIN PARA

PARA _ en rango(individuos_iniciales_greedy * tamaño_población):
    ruta, dist ← generar_ruta_aleatoria
    añadir (ruta, dist) a padres_iniciales
FIN PARA
DEVOLVER padres_iniciales

```

generar_ruta_aleatoria


```
n_ciudades ← longitud de ruta - 1
ruta ← lista de números aleatorios desde 0 hasta n_ciudades - 1 sin repetir
ruta.añadir(ruta[0])
dist ← calcular_distancia_total(ruta, matriz)
DEVOLVER ruta, dist
```

torneo_ganadores

```
posibles_padres ← random(k_best, poblacion)
ordenar(posibles_padres, distancia)
DEVOLVER posibles_padres[0]
```

torneo_perdedores

```
posibles_perdedores ← random(k_worst, poblacion)
ordenar(posibles_padres, distancia, orden descendiente)
DEVOLVER posibles_perdedores[0]
```

cruce_ox2

```
n_ciudades ← len[padre_1] - 1

hijo[0] ← copiar lista de ciudades de padre_1
hijo[1] ← 0

posiciones_seleccionadas ← lista vacía
PARA i en el rango n_ciudades:
    SI random() < 0.5:
        añadir i a posiciones_seleccionadas
    FIN SI
FIN PARA

ciudades_seleccionadas ← ciudades en padre_2 en las posiciones seleccionadas
PARA cada ciudad en ciudades_seleccionadas:
    índice ← índice de la ciudad en hijo[0]
    hijo[0][índice] ← -1
FIN PARA

índices_cambiados ← posiciones en hijo[0] que son -1
PARA i desde 0 hasta tamaño(índices_cambiados) - 1:
    hijo[0][índices_cambiados[i]] ← ciudades_seleccionadas[i]
FIN PARA

SI 0 en índices_cambiados:
```

```

    hijo[0][n_ciudades] ← hijo[0][0]
FIN SI
SI n_ciudades en indices_cambiados:
    hijo[0][0] ← hijo[0][n_ciudades]
FIN SI

hijo_1 ← (hijo[0], calcular_distancia_recorrida(hijo[0], matriz))

hijo ← copiar lista de ciudades de padre_2
hijo[1] ← 0
ciudades_seleccionadas ← ciudades en padre_1 en las posiciones seleccionadas
PARA cada ciudad en ciudades_seleccionadas:
    índice ← índice de la ciudad en hijo[0]
    hijo[0][índice] ← -1
FIN PARA

indices_cambiados ← posiciones en hijo[0] que son -1
PARA i desde 0 hasta tamaño(indices_cambiados) - 1:
    hijo[0][indices_cambiados[i]] ← ciudades_seleccionadas[i]
FIN PARA

SI 0 en indices_cambiados:
    hijo[0][n_ciudades] ← hijo[0][0]
FIN SI
SI n_ciudades en indices_cambiados:
    hijo[0][0] ← hijo[0][n_ciudades]
FIN SI

hijo_2 ← (hijo[0], calcular_distancia_recorrida(hijo[0], matriz))

DEVOLVER hijo_1, hijo_2

```

cruce_moc

```

n_ciudades ← len[padre_1] - 1
punto_cruce ← random entre 1 y n_ciudades - 1

hijo_1 ← primeras ciudades de padre_1 hasta punto_cruce
PARA cada ciudad en padre_2:
    SI ciudad no está en hijo_1:
        hijo_1.añadir(ciudad)
    FIN SI
FIN PARA

hijo_1 ← cortar hijo_1 a n_ciudades + cerrar ciclo con hijo_1[0]

hijo_2 ← primeras ciudades de padre_2 hasta punto_cruce
PARA cada ciudad en padre_1:

```

```

SI ciudad no está en hijo_2:
    añadir ciudad a hijo_2
FIN SI
FIN PARA
hijo_2 ← cortar hijo_2 a n_ciudades + cerrar ciclo con hijo_2[0]

distancia_hijo_1 ← calcular_distancia_recorrida
distancia_hijo_2 ← calcular_distancia_recorrida

hijo_1 ← (hijo_1, distancia_hijo_1)
hijo_2 ← (hijo_2, distancia_hijo_2)

DEVOLVER hijo_1, hijo_2

```

mutacion_2opt

```

nCiudades ← número de ciudades en individuo - 1
dist ← distancia actual de individuo

i, j ← seleccionar dos índices aleatorios en rango(nCiudades)
sort(i, j)

arc_elim, arco_anad ← calcular_arcos(individuo[0], i, j, matriz)
swapRuta(individuo[0], i, j)
dist ← dist - arc_elim + arco_anad
individuo ← (individuo[0], dist)
DEVOLVER individuo

```

3.3 Análisis

Se adjuntan las tablas de resultados para las 8 versiones propuestas para este algoritmo

| Generacional OX2 KBest = 2 E = 1 | A280 | | CH130 | | D18512 | | PR144 | | U1060 | |
|---|------------|-------|------------|-------|-------------------|--------|------------|-------|-------------------|---------|
| | Tamaño | 280 | Tamaño | 130 | Tamaño | 18512 | Tamaño | 144 | Tamaño | 1060 |
| | encontrado | 2579 | encontrado | 6110 | Mínimo encontrado | 645238 | encontrado | 58537 | Mínimo encontrado | 224094 |
| | Sol | Time | Sol | Time | Sol | Time | Sol | Time | Sol | Time |
| Ejecución 1 | 17848,45 | 6,42 | 17655,56 | 2,076 | 51008261,38 | 60,000 | 242769,75 | 2,344 | 4404099,65 | 60,000 |
| Ejecución 2 | 15862,81 | 6,430 | 16048,71 | 2,063 | 50939277,51 | 60,000 | 283098,72 | 2,357 | 4331897,00 | 60,000 |
| Ejecución 3 | 17049,97 | 6,435 | 15961,81 | 2,08 | 50741252,87 | 60,000 | 232648,87 | 2,351 | 4250163,00 | 60,000 |
| Media | 556,08% | 6,43 | 170,96% | 2,07 | 5840,99% | 60,00 | 248,95% | 2,35 | 1373,74% | 60,000 |
| Desv. típica | 0,39 | 0,01 | 0,86 | 0,01 | 0,21 | 0,0000 | 0,46 | 0,01 | 0,34 | 0,00000 |

| Generacional OX2 KBest = 3 E = 1 | A280 | | CH130 | | D18512 | | PR144 | | U1060 | |
|---|------------|-------|------------|-------|-------------------|--------|-------------------|-------|-------------------|----------|
| | Tamaño | 280 | Tamaño | 130 | Tamaño | 18512 | Tamaño | 144 | Tamaño | 1060 |
| | encontrado | 2579 | encontrado | 6110 | Mínimo encontrado | 645238 | Mínimo encontrado | 58537 | Mínimo encontrado | 224094 |
| | Sol | Time | Sol | Time | Sol | Time | Sol | Time | Sol | Time |
| Ejecución 1 | 12397,27 | 6,407 | 14897,03 | 2,089 | 51008261,38 | 60,000 | 207676,61 | 2,375 | 3692305,15 | 60,000 |
| Ejecución 2 | 14590,51 | 6,394 | 12415,04 | 2,063 | 50939277,51 | 60,000 | 203112,03 | 2,359 | 3801885,21 | 60,000 |
| Ejecución 3 | 11848,65 | 6,384 | 13074,38 | 2,11 | 50741252,87 | 60,000 | 229866,89 | 2,367 | 3609328,35 | 60,000 |
| Media | 401,96% | 6,40 | 120,33% | 2,09 | 7787,98% | 60,00 | 264,82% | 2,37 | 1551,62% | 60,00000 |
| Desv. típica | 0,56 | 0,01 | 0,21 | 0,02 | 0,21 | 0,0000 | 0,24 | 0,01 | 0,43 | 0,00000 |

| Generacional OX2 KBest = 2 E = 2 | A280 | | CH130 | | D18512 | | PR144 | | U1060 | |
|---|------------|-------|------------|-------|-------------------|---------|------------|-------|-------------------|----------|
| | Tamaño | 280 | Tamaño | 130 | Tamaño | 18512 | Tamaño | 144 | Tamaño | 1060 |
| | encontrado | 2579 | encontrado | 6110 | Mínimo encontrado | 645238 | encontrado | 58537 | Mínimo encontrado | 224094 |
| | Sol | Time | Sol | Time | Sol | Time | Sol | Time | Sol | Time |
| Ejecución 1 | 15730,09 | 6,46 | 16670,15 | 2,086 | 51008261,38 | 60,0000 | 229828,86 | 2,372 | 4504489,04 | 60,000 |
| Ejecución 2 | 18126,65 | 6,428 | 18647,23 | 2,08 | 50939277,51 | 60,0000 | 211003,85 | 2,354 | 4189063,17 | 60,000 |
| Ejecución 3 | 16810,45 | 6,439 | 15117,54 | 2,08 | 50741252,87 | 60,0000 | 218561,82 | 2,367 | 4405714,09 | 60,000 |
| Media | 554,87% | 6,44 | 175,15% | 2,08 | 7787,98% | 60,00 | 275,49% | 2,36 | 1848,48% | 60,00000 |
| Desv. típica | 0,47 | 0,02 | 0,29 | 0,01 | 0,21 | 0,0000 | 0,16 | 0,01 | 0,72 | 0,00000 |

| Generacional OX2 KBest = 3 E = 2 | A280 | | CH130 | | D18512 | | PR144 | | U1060 | |
|---|------------|-------|------------|-------|-------------------|--------|-------------------|-------|-------------------|----------|
| | Tamaño | 280 | Tamaño | 130 | Tamaño | 18512 | Tamaño | 144 | Tamaño | 1060 |
| | encontrado | 2579 | encontrado | 6110 | Mínimo encontrado | 645238 | Mínimo encontrado | 58537 | Mínimo encontrado | 224094 |
| | Sol | Time | Sol | Time | Sol | Time | Sol | Time | Sol | Time |
| Ejecución 1 | 12426,14 | 6,417 | 13636,78 | 2,122 | 51008261,38 | 60,000 | 238385,60 | 2,391 | 3845811,73 | 60,000 |
| Ejecución 2 | 13430,12 | 6,482 | 14297,16 | 2,11 | 50939277,51 | 60,000 | 184081,38 | 2,387 | 3267802,51 | 60,000 |
| Ejecución 3 | 12836,96 | 6,441 | 17488,08 | 2,11 | 50741252,87 | 60,000 | 225444,59 | 2,367 | 3569619,36 | 60,000 |
| Media | 400,11% | 6,45 | 147,80% | 2,11 | 7787,98% | 60,00 | 268,95% | 2,38 | 1489,10% | 60,00000 |
| Desv. típica | 0,20 | 0,03 | 0,34 | 0,01 | 0,21 | 0,0000 | 0,48 | 0,01 | 1,29 | 0,00000 |

| Generacional MOC KBest = 2 E = 1 | A280 | | CH130 | | D18512 | | PR144 | | U1060 | |
|---|------------|-------|------------|------|-------------------|--------|------------|-------|-------------------|----------|
| | Tamaño | 280 | Tamaño | 130 | Tamaño | 18512 | Tamaño | 144 | Tamaño | 1060 |
| | encontrado | 2579 | encontrado | 6110 | Mínimo encontrado | 645238 | encontrado | 58537 | Mínimo encontrado | 224094 |
| | Sol | Time | Sol | Time | Sol | Time | Sol | Time | Sol | Time |
| Ejecución 1 | 15226,00 | 9,544 | 17879,17 | 2,46 | 51008261,38 | 60,000 | 247894,63 | 2,894 | 4320563,81 | 60,000 |
| Ejecución 2 | 15608,44 | 9,578 | 18976,46 | 2,44 | 50939277,51 | 60,000 | 257398,78 | 2,851 | 4186993,08 | 60,000 |
| Ejecución 3 | 14884,74 | 9,711 | 17485,76 | 2,47 | 50741252,87 | 60,000 | 263294,65 | 2,863 | 4283591,30 | 60,000 |
| Media | 490,92% | 9,61 | 196,46% | 2,45 | 7787,98% | 60,00 | 337,67% | 2,87 | 1802,65% | 60,00000 |
| Desv. típica | 0,14 | 0,09 | 0,13 | 0,02 | 0,21 | 0,0000 | 0,13 | 0,02 | 0,31 | 0,00000 |

| Generacional MOC KBest = 2 E = 2 | A280 | | CH130 | | D18512 | | PR144 | | U1060 | |
|---|------------|-------|------------|------|-------------------|--------|------------|-------|-------------------|----------|
| | Tamaño | 280 | Tamaño | 130 | Tamaño | 18512 | Tamaño | 144 | Tamaño | 1060 |
| | encontrado | | encontrado | | Mínimo encontrado | | encontrado | | Mínimo encontrado | |
| | 2579 | | 6110 | | 645238 | | 58537 | | 224094 | |
| | Sol | Time | Sol | Time | Sol | Time | Sol | Time | Sol | Time |
| Ejecución 1 | 14265,89 | 9,632 | 18258,93 | 2,47 | 51008261,38 | 60,000 | 281586,19 | 2,901 | 4222476,44 | 60,000 |
| Ejecución 2 | 15138,14 | 9,613 | 20697,66 | 2,46 | 50939277,51 | 60,000 | 222791,33 | 2,873 | 4094011,81 | 60,000 |
| Ejecución 3 | 15527,60 | 9,657 | 17894,46 | 2,47 | 50741252,87 | 60,000 | 258817,40 | 2,874 | 4198343,12 | 60,000 |
| Media | 480,74% | 9,63 | 210,15% | 2,47 | 7787,98% | 60,00 | 334,59% | 2,88 | 1761,54% | 60,00000 |
| Desv. típica | 0,25 | 0,02 | 0,25 | 0,01 | 0,21 | 0,0000 | 0,51 | 0,02 | 0,30 | 0,00000 |

| Generacional MOC KBest = 3 E = 1 | A280 | | CH130 | | D18512 | | PR144 | | U1060 | |
|---|------------|-------|------------|------|-------------------|--------|------------|-------|-------------------|----------|
| | Tamaño | 280 | Tamaño | 130 | Tamaño | 18512 | Tamaño | 144 | Tamaño | 1060 |
| | encontrado | | encontrado | | Mínimo encontrado | | encontrado | | Mínimo encontrado | |
| | 2579 | | 6110 | | 645238 | | 58537 | | 224094 | |
| | Sol | Time | Sol | Time | Sol | Time | Sol | Time | Sol | Time |
| Ejecución 1 | 14225,68 | 9,571 | 18857,00 | 2,48 | 51008261,38 | 60,000 | 284250,45 | 2,901 | 4243437,57 | 60,000 |
| Ejecución 2 | 14392,98 | 9,487 | 17306,45 | 2,48 | 50939277,51 | 60,000 | 269285,61 | 2,884 | 3933710,86 | 60,000 |
| Ejecución 3 | 14459,02 | 9,491 | 16212,39 | 2,49 | 50741252,87 | 60,000 | 251035,55 | 2,894 | 3892612,48 | 60,000 |
| Media | 456,77% | 9,52 | 185,74% | 2,49 | 7787,98% | 60,00 | 358,16% | 2,89 | 1695,34% | 60,00000 |
| Desv. típica | 0,05 | 0,05 | 0,22 | 0,00 | 0,21 | 0,0000 | 0,28 | 0,01 | 0,86 | 0,00000 |

| Generacional MOC KBest = 3 E = 2 | A280 | | CH130 | | D18512 | | PR144 | | U1060 | |
|---|------------|-------|------------|------|-------------------|--------|------------|-------|-------------------|----------|
| | Tamaño | 280 | Tamaño | 130 | Tamaño | 18512 | Tamaño | 144 | Tamaño | 1060 |
| | encontrado | | encontrado | | Mínimo encontrado | | encontrado | | Mínimo encontrado | |
| | 2579 | | 6110 | | 645238 | | 58537 | | 224094 | |
| | Sol | Time | Sol | Time | Sol | Time | Sol | Time | Sol | Time |
| Ejecución 1 | 14018,83 | 9,692 | 16048,71 | 2,52 | 51008261,38 | 60,000 | 228113,99 | 2,906 | 4104756,58 | 60,000 |
| Ejecución 2 | 14480,27 | 9,542 | 16961,93 | 2,50 | 50939277,51 | 60,000 | 258840,69 | 2,935 | 3991826,98 | 60,000 |
| Ejecución 3 | 14351,90 | 9,615 | 15183,33 | 2,50 | 50741252,87 | 60,000 | 266959,13 | 2,911 | 3868792,65 | 60,000 |
| Media | 453,85% | 9,62 | 162,92% | 2,50 | 7787,98% | 60,00 | 329,31% | 2,92 | 1679,82% | 60,00000 |
| Desv. típica | 0,09 | 0,08 | 0,15 | 0,01 | 0,21 | 0,0000 | 0,35 | 0,02 | 0,53 | 0,00000 |

Cabe destacar como en todas las configuraciones, en el caso de la ciudad con mayor tamaño (d18512), se obtienen exactamente los **mismos resultados**. Esto se debe a que al ser una instancia de tanto tamaño, no permite al algoritmo a encontrar una nueva generación con un élite con una mayor calidad que el mejor de la población inicial, que siempre es la misma debido a la inicialización de la semilla previa a la ejecución. El algoritmo realiza un total de 215 evaluaciones en la ejecución que más realiza, lo que significa que no completa ni la generación de una nueva población completa. Esto se debe al tiempo que tardan los operadores de cruce, ya que debe iterar por todo el vector de ciudades y realizar cambios de posiciones para generar los descendientes.

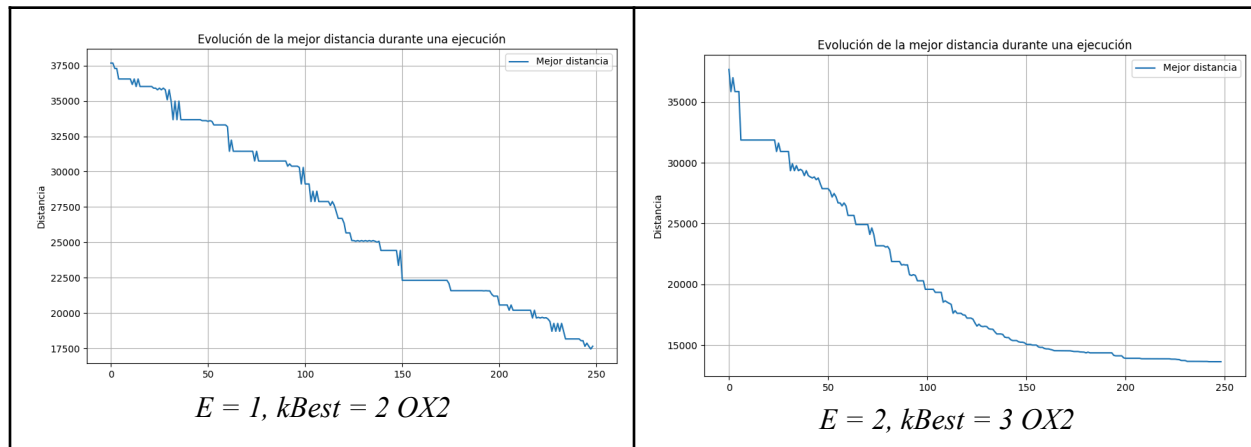
| Gen. OX2 | A280 | | CH130 | | D18512 | | PR144 | | U1060 | | MEDIA | |
|--------------|------|------|-------|------|--------|-------|-------|------|-------|-------|-------|-------|
| | Desv | Time | Desv | Time | Desv | Time | Desv | Time | Desv | Time | C | Time |
| E=1, KBEST=2 | 5,56 | 6,43 | 1,71 | 2,07 | 77,88 | 60,00 | 2,49 | 2,35 | 13,74 | 60,00 | 20,28 | 26,17 |
| E=1, KBEST=3 | 4,02 | 6,40 | 1,20 | 2,09 | 77,88 | 60,00 | 2,65 | 2,37 | 15,52 | 60,00 | 20,25 | 26,17 |
| E=2, KBEST=2 | 5,55 | 6,44 | 1,75 | 2,08 | 77,88 | 60,00 | 2,75 | 2,36 | 18,48 | 60,00 | 21,28 | 26,18 |
| E=2, KBEST=3 | 4,00 | 6,45 | 1,48 | 2,11 | 77,88 | 60,00 | 2,69 | 2,38 | 14,89 | 60,00 | 20,19 | 26,19 |

Imagen con la comparativa de las 4 versiones propuestas para el cruce OX2

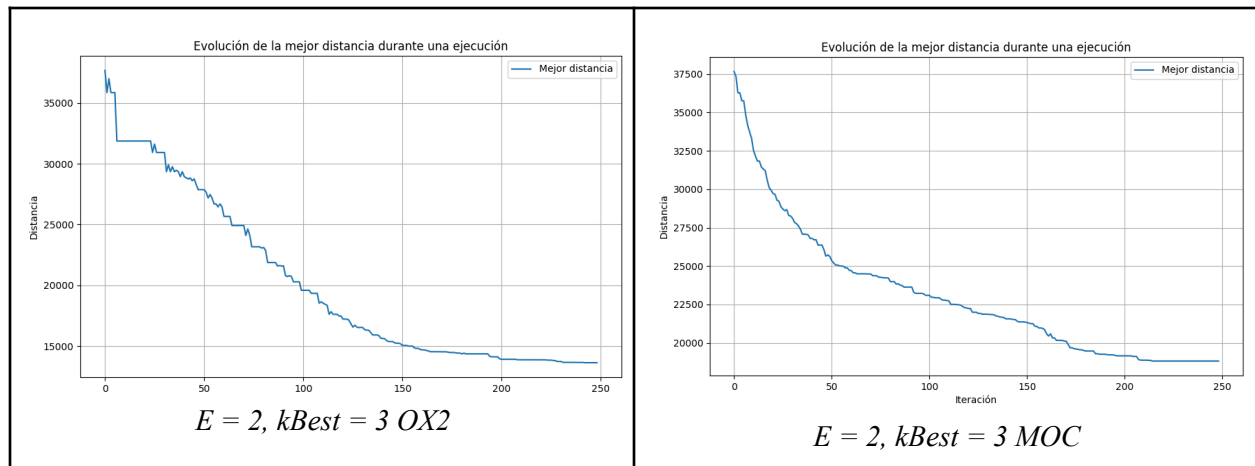
En la comparativa entre las 4 versiones del algoritmo generacional para el operador de cruce OX2, se puede ver como la **cuarta versión es la mejor**, ya que obtiene una media de desviación ligeramente inferior al resto de configuraciones, es decir, se acerca más en promedio al mejor resultado obtenido (best). Esto se debe a que al ser el número de élites y el kBest superior al resto, y por tanto, cada generación se queda con más individuos prometedores con respecto a las versiones que solo mantienen un individuo élite. Además, al seleccionar al ganador entre más individuos (3 en lugar de 2), también aumenta la probabilidad de que se elija a un mejor individuo para la reproducción.

| Gen. MOC | A280 | | CH130 | | D18512 | | PR144 | | U1060 | | MEDIA | |
|--------------|------|------|-------|------|--------|-------|-------|------|-------|-------|-------|-------|
| | Desv | Time | Desv | Time | Desv | Time | Desv | Time | Desv | Time | C | Time |
| E=1, KBEST=2 | 4,91 | 9,61 | 1,96 | 2,45 | 77,88 | 60,00 | 3,38 | 2,87 | 18,03 | 60,00 | 21,23 | 26,99 |
| E=1, KBEST=3 | 4,57 | 9,52 | 1,86 | 2,49 | 77,88 | 60,00 | 3,58 | 2,89 | 16,95 | 60,00 | 20,97 | 26,98 |
| E=2, KBEST=2 | 4,81 | 9,63 | 2,10 | 2,47 | 77,88 | 60,00 | 3,35 | 2,88 | 17,62 | 60,00 | 21,15 | 27,00 |
| E=2, KBEST=3 | 4,54 | 9,62 | 1,63 | 2,50 | 77,88 | 60,00 | 3,29 | 2,92 | 16,80 | 60,00 | 20,83 | 27,01 |

En el caso del operador de cruce MOC, obtenemos el mismo resultado, es decir, **la mejor versión es la cuarta** y la explicación es idéntica al caso anterior, ya que gestionan la población de la misma manera, solo que los descendientes se forman de una manera distinta.

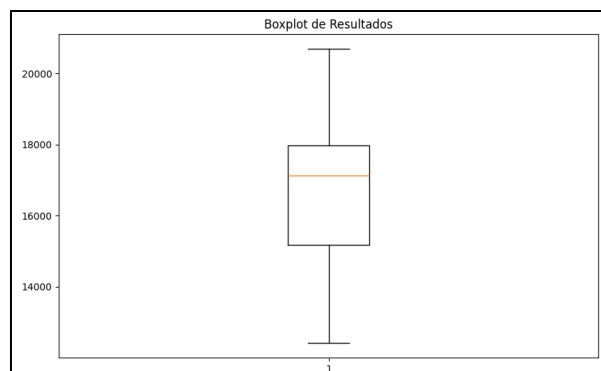


Entre ambas versiones, **la cuarta versión del OX2 es mejor que la cuarta versión del MOC**. A continuación se muestra una tabla comparando ambas.



Observamos como el cruce OX2 es más constante manteniendo ese ritmo hasta la generación 150, en el que empieza a reducir la pendiente de mejora. Sin embargo, en el MOC a partir de la generación 50 se observa como reduce su pendiente.

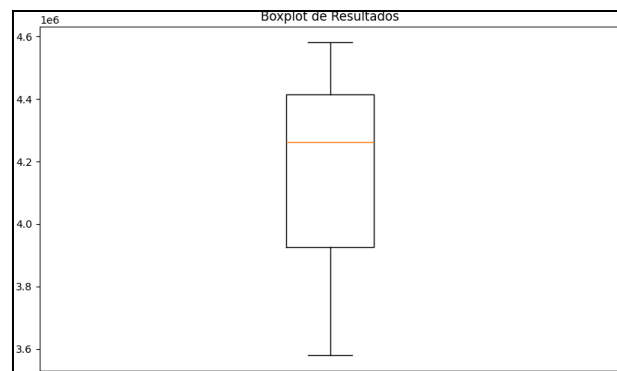
Aquí se demuestra la diferencia entre la peor y la mejor configuración. Se ve cómo la mejor configuración se acerca al mínimo en un menor número de generaciones



Boxplot Generacional ch130.tsp

En este boxplot obtenido tras ejecutar las 24 posibles ejecuciones para el archivo ch130.tsp (8 versiones x 3 semillas), se puede observar una consistencia del algoritmo, ya que la mayoría (entre Q1 y Q3) de soluciones encontradas se encuentran entre 15.500 y 18000. En los bigotes, vemos como el mejor resultado es de 12415, lo cual supone una desviación del doble del mejor resultado posible, y como peor

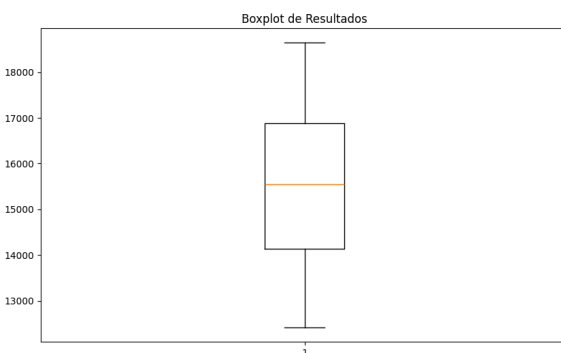
resultado se obtiene 20697. La distribución de los datos es asimétrica hacia valores de menor calidad como indica la mediana.



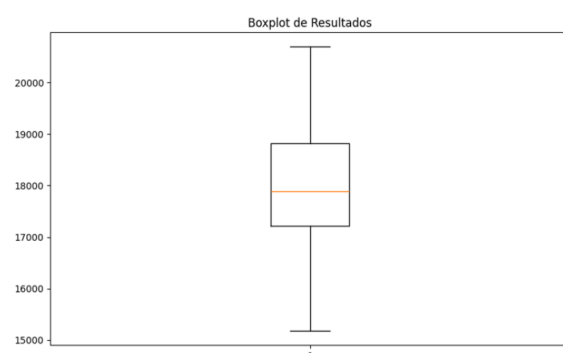
Boxplot Generacional u1060.tsp

Comparándolo con una instancia de mayor tamaño, como es la u1060.tsp (ya que con la d18512 no tiene sentido ya que se obtiene siempre el mismo resultado como se ha comentado antes), vemos como los resultados obtenidos son más dispersos por lo general (entre Q1 y Q3). Esto se puede deber a que el algoritmo, al parar por tiempo y en unas 15000 evaluaciones en promedio por debajo del caso anterior, no es capaz de encontrar un mejor individuo en generaciones posteriores y reducir la diferencia del intervalo intercuartil. En este caso también se observa una asimetría del mismo tipo.

Donde sí observamos una diferencia notable entre el total de datos obtenidos es cuando se utiliza el operador de cruce OX2 y el operador MOC.



Boxplot ch130 cruce OX2



Boxplot ch130 cruce MOC

Se puede observar como el operador de cruce MOC es mucho más consistente con los datos. Esto es debido a la aleatoriedad del operador de cruce OX2, que hace que los descendientes difieran en mayor medida de sus padres que en el caso de que se hiciera con el cruce MOC. Esto también implica que el cruce OX2 obtiene un mejor resultado individual y en promedio. En ambos casos, se ve cómo los datos

están distribuidos simétricamente, lo que justifica la asimetría mostrada en los boxplots anteriores a la hora de combinar ambos tipos de cruce.

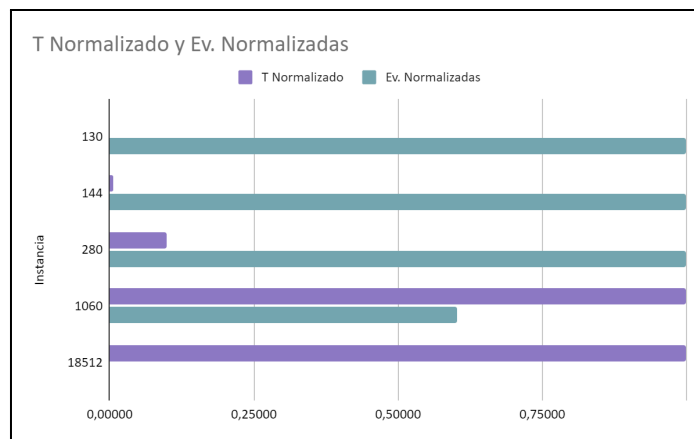


Gráfico que muestra la influencia del tamaño de la instancia en el comportamiento del algoritmo.

Aquí se puede ver cómo conforme va aumentando el tamaño de la instancia, el tiempo va creciendo a un ritmo que no es proporcional. En el caso de la de 280, el tamaño es en torno al doble de la instancia anterior, pero el promedio de tiempo que tarda en ejecutar es de casi 3.5 veces superior aunque al estar normalizados los datos parezca mucho mayor. También se observa cómo en los tres primeros casos se completan las 50.000 evaluaciones de la condición de parada, y en en las otras dos no, aunque en la de 1060 para por la condición de tiempo de parada, aún así alcanza unas 35.000 de promedio, dependiendo de la velocidad del ordenador en esa ejecución. En el caso de las 18512, como se ha comentado anteriormente, apenas alcanza las 200 evaluaciones de promedio.

4. Algoritmo Estacionario.

4.1 Descripción.

El algoritmo evolutivo estacionario es otro tipo de algoritmo evolutivo que también forma parte de las metaheurísticas basadas en poblaciones (P-metaheurísticas). En este caso, la evolución se realiza de manera más gradual, modificando únicamente una parte de la población en cada iteración, en lugar de reemplazando la generación totalmente. Esto permite un enfoque más local y menos agresivo en la evolución de la población.

Al igual que en el caso anterior, este algoritmo sigue los principios de la evolución natural de Darwin, donde los individuos compiten por reproducirse y transmitir sus características a la siguiente generación. En este caso, el genotipo y el fenotipo vienen representados por lo mismo que en el generacional.

La estructura del algoritmo estacionario también se divide en las mismas tres fases, pero con un enfoque diferente al generacional:

1. **Selección:** en este caso, se seleccionan únicamente dos padres por cada generación nueva, que se reproducen dando lugar a dos descendientes. Son elegidos mediante un torneo de ganadores también.
2. **Cruce:** no hay probabilidad de cruce, ya que si no tendríamos exactamente la misma generación a la anterior. En este caso, se aplican sí o sí los operadores de cruce OX2 o MOC y la mutación si corresponde.
3. **Reemplazamiento:** los dos nuevos descendientes únicamente reemplazarán a dos individuos, obtenidos mediante torneo de perdedores. Esto implica que en cada nueva generación de individuos, solo se diferencia de la anterior en dos individuos.

A diferencia del generacional, aquí no se tiene el mecanismo de elitismo y la condición de parada es la misma.

4.2 Pseudocódigo.

Solo se detalla el pseudocódigo del algoritmo estacionario, ya que tanto operadores de cruce, como mutación y torneos se usan los métodos detallados en la [sección 3.2](#).

Algoritmo Estacionario

```
poblacion ← generar_poblacion_inicial()
inicio ← tiempo actual
evaluaciones ← 0

MIENTRAS no se cumpla criterio de parada:
    evaluaciones ← evaluaciones + 1

    MIENTRAS padre_1 == padre_2:
        padre_1 ← torneo_ganadores(poblacion, k_best_est)
        padre_2 ← torneo_ganadores(poblacion, k_best_est)
    FIN MIENTRAS
```

```
SI tipo_cruce == "OX2":
    SI tiempo_expirado:
        PARAR
    hijo_1, hijo_2 ← cruce_ox2(padre_1, padre_2)
SI NO
    SI tiempo_expirado:
        PARAR
    hijo_1, hijo_2 ← cruce_moc(padre_1, padre_2)
FIN SI

SI percent_random < percent_mutacion:
    mutar(hijo_1)
FIN SI
SI percent_random < percent_mutacion:
    mutar(hijo_2)
FIN SI

SI tiempo_expirado:
    PARAR

perdedor ← torneo_perdedores(poblacion, k_worst_est)
poblacion.eliminar(perdedor)
perdedor ← torneo_perdedores(poblacion, k_worst_est)
poblacion.eliminar(perdedor)

poblacion.anadir(hijo_1)
poblacion.anadir(hijo_2)
evaluaciones ← evaluaciones + 2

mejor_solucion ← elemento con mínima distancia en poblacion
FIN MIENTRAS

final ← tiempo actual
DEVOLVER mejor_solucion.ruta, mejor_solucion.distancia
```

4.3 Análisis.

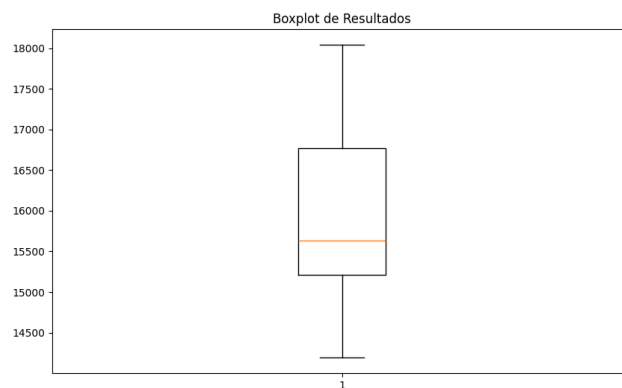
| | A280 | | CH130 | | D18512 | | PR144 | | U1060 | |
|------------------|------------|-------|------------|------|-------------------|--------|------------|-------|-------------------|----------|
| Estacionario OX2 | Tamaño | 280 | Tamaño | 130 | Tamaño | 18512 | Tamaño | 144 | Tamaño | 1060 |
| | encontrado | | encontrado | | Mínimo encontrado | | encontrado | | Mínimo encontrado | |
| | Sol | Time | Sol | Time | Sol | Time | Sol | Time | Sol | Time |
| Ejecución 1 | 10765,99 | 9,681 | 15783,08 | 3,94 | 51008261,38 | 60,000 | 195158,29 | 3,983 | 3187301,45 | 60,000 |
| Ejecución 2 | 10776,95 | 8,946 | 15121,07 | 3,88 | 50939277,51 | 60,000 | 218455,39 | 4,056 | 3150342,52 | 60,000 |
| Ejecución 3 | 12137,78 | 8,912 | 14193,37 | 4,62 | 50741252,87 | 60,000 | 183811,28 | 4,082 | 2989570,04 | 60,000 |
| Media | 335,32% | 9,18 | 146,03% | 4,15 | 7787,98% | 60,00 | 240,20% | 4,04 | 1287,40% | 60,00000 |
| Desv. típica | 617,42 | 0,43 | 0,13 | 0,41 | 0,21 | 0,0000 | 0,30 | 0,05 | 0,47 | 0,00000 |

| | A280 | | CH130 | | D18512 | | PR144 | | U1060 | |
|------------------|------------|--------|------------|------|-------------------|-------------|------------|-------|-------------------|----------|
| Estacionario MOC | Tamaño | 280 | Tamaño | 130 | Tamaño | 18512 | Tamaño | 144 | Tamaño | 1060 |
| | encontrado | | encontrado | | Mínimo encontrado | | encontrado | | Mínimo encontrado | |
| | Sol | Time | Sol | Time | Sol | Time | Sol | Time | Sol | Time |
| Ejecución 1 | 13755,38 | 12,904 | 18043,33 | 4,54 | 50999462,65 | 60,00000000 | 274244,71 | 4,724 | 3966318,45 | 60,000 |
| Ejecución 2 | 13643,85 | 12,646 | 17098,29 | 4,47 | 50851705,39 | 60,00000000 | 264518,31 | 4,724 | 3920417,24 | 60,000 |
| Ejecución 3 | 13293,18 | 12,712 | 15483,36 | 5,17 | 50715636,87 | 60,00000000 | 261796,12 | 4,742 | 3875628,09 | 60,000 |
| Media | 425,95% | 12,75 | 176,19% | 4,73 | 7781,68% | 60,00 | 355,87% | 4,73 | 1649,62% | 60,00000 |
| Desv. típica | 0,09 | 0,13 | 0,21 | 0,38 | 0,22 | 0,0000 | 0,11 | 0,01 | 0,20 | 0,00000 |

| | A280 | | CH130 | | D18512 | | PR144 | | U1060 | | MEDIA | |
|------------------|------|-------|-------|------|--------|-------|-------|------|-------|-------|-------|-------|
| | Desv | Time | Desv | Time | Desv | Time | Desv | Time | Desv | Time | C | Time |
| Estacionario OX2 | 3,35 | 9,18 | 1,46 | 4,15 | 77,88 | 60,00 | 2,40 | 4,04 | 12,87 | 60,00 | 19,59 | 27,47 |
| Estacionario MOC | 4,26 | 12,75 | 1,76 | 4,73 | 77,82 | 60,00 | 3,56 | 4,73 | 16,50 | 60,00 | 20,78 | 28,44 |

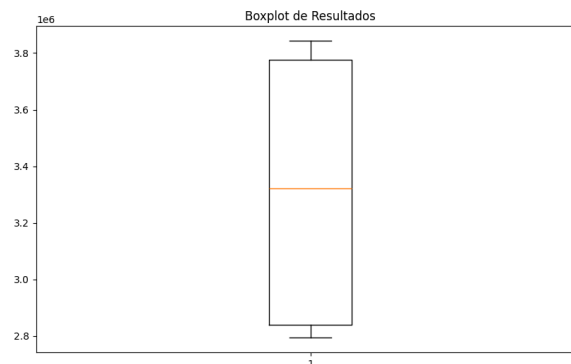
Para el algoritmo estacionario, con las ejecuciones realizadas obtenemos que la versión que utiliza el operador OX2 encuentra unos resultados promedio de mayor calidad que el MOC y en menor tiempo. Esto hace que la **primera versión sea la mejor** de este algoritmo evolutivo.

Cabe destacar que en este caso, para el operador de cruce MOC sí se encuentran resultados distintos en el caso del d18512, ya que aquí sí explora nuevas generaciones ya que solo necesita realizar 2 evaluaciones para generar una nueva evaluación.



Boxplot Estacionario ch130.tsp

Se puede observar una consistencia en los datos bastante elevada, ya que en el intervalo entre el Q1 y el Q3 las soluciones obtenidas difieren en un máximo de 1500 unidades aproximadamente. Además, la mediana está muy próxima al Q1, lo cual indica que la distribución de los datos es asimétrica hacia soluciones de mayor calidad. Sin embargo, los bigotes se encuentran alejados de estos cuartiles, lo que indica que hay un mayor número de valores atípicos, que podría deberse a la aleatoriedad de las semillas. Si se ejecutara con un mayor número de semillas seguramente estos bigotes reducirían su tamaño.



Boxplot Estacionario u1060.tsp

En el caso de ejecutar el algoritmo en una instancia de mayor tamaño, cabe destacar la ínfima presencia de valores atípicos, ya que se encuentran muy próximos al intervalo intercuartil entre Q1 y Q3. Además, siguen una distribución simétrica en gran medida. Esto se debe a la ligera variación que hay entre una generación y otra, ya que sólo cambian dos elementos y por tanto la variabilidad entre una generación y otra es mucho menor a la que puede haber en un algoritmo generacional en el que reemplazamos toda la generación por completo.

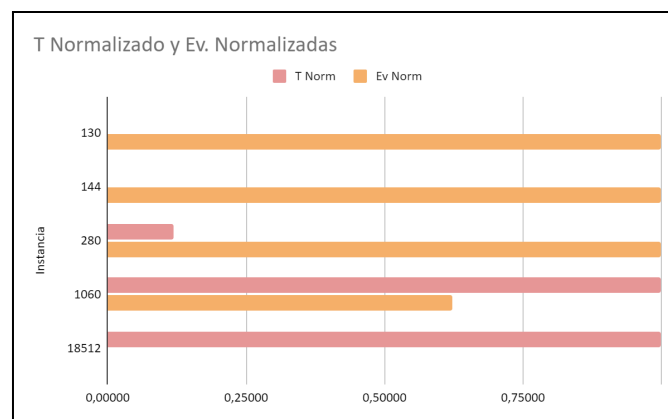
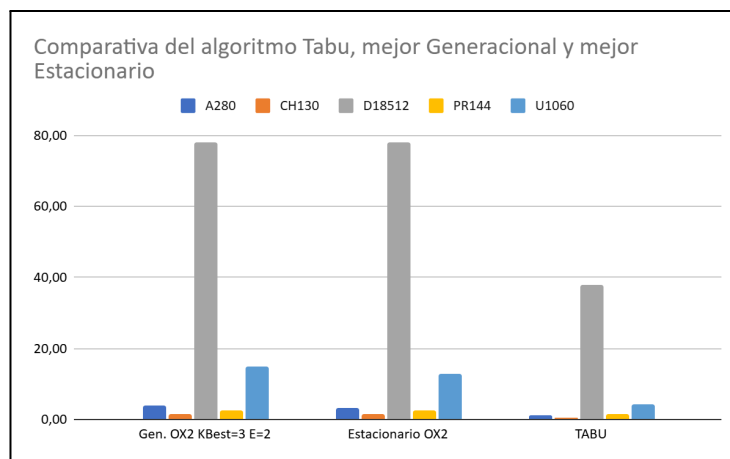


Gráfico que muestra la influencia del tamaño de la instancia en el comportamiento del algoritmo

Se observa cómo se comporta de una forma muy similar al [algoritmo generacional](#) en cuanto a tiempo de ejecución y número de evaluaciones realizadas, sin mayores aspectos a comentar a parte de los comentados en el algoritmo generacional.

5. Resultados globales.

5.1 Comparativa gráfica de resultados.

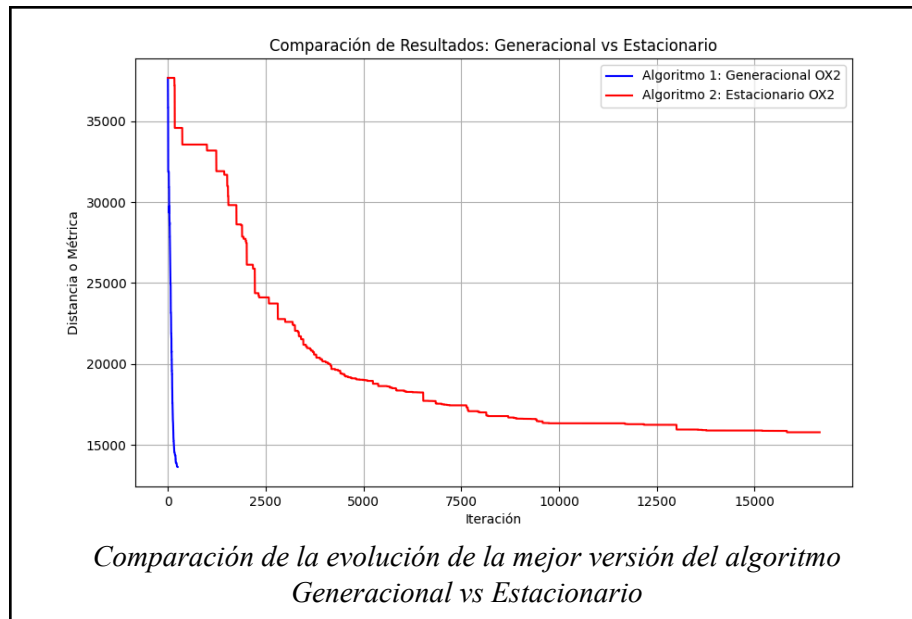


Comprobamos entonces que para nuestro enunciado y nuestros algoritmos, los algoritmos basados en trayectorias mejoran a los basados en poblaciones, porque el algoritmo implementado en la práctica 1 que mejor resultados daba era el algoritmo Tabú basado en la búsqueda local, y este genera unos resultados considerablemente más homogéneos que cualquiera de los evolutivos implementados para esta práctica. Esto se debe a que el tamaño de los problemas es lo suficientemente bajo como para que un algoritmo basado en trayectorias funcione mejor que un algoritmo evolutivo, incluso en el archivo más grande (D18512).

Al usar un archivo de una instancia muy superior a la máxima actual, el algoritmo que encontraría una solución mejor, o mejor dicho, la más consistente, sería el algoritmo evolutivo estacionario con operador de cruce OX2, ya que hemos podido observar en esta práctica que es el que mejores resultados proporciona.

5.2 Comparación entre algoritmos.

En la siguiente tabla, se observa la diferencia en la evolución de la mejor solución encontrada entre el algoritmo generacional y el estacionario.



En esta gráfica hay varios aspectos a comentar. Primero, se observa como la línea azul acaba antes, y esto es debido a que el algoritmo generacional necesita más evaluaciones por generación o iteración (101), y es por eso que el algoritmo estacionario finaliza en una generación más avanzada ya que realiza 3 evaluaciones por generación.


En esta ejecución en concreto, se observa cómo el algoritmo generacional obtiene un mejor resultado, aunque el algoritmo estacionario obtenga un promedio de resultados más próximo al óptimo. Se puede comprobar perfectamente lo comentado anteriormente: cómo el algoritmo estacionario progresa mucho más progresivamente que el generacional.

6. Fichero log.

Se adjunta enlace a continuación con el cual se puede acceder al fichero .log generado:

https://drive.google.com/file/d/1AuFjs3WT7gzJvAYgvE_mYz4j3SA_PNo3/view?usp=sharing

7. Bibliografía.

Fichero de datos de las tablas de ejecuciones para cualquier consulta:  Tablas_MH2024.xlsx .

https://matplotlib.org/stable/users/getting_started/