

|  |  |
| --- | --- |
| **Algoritmos metaheurísticos**  Implementación, resultados y análisis |  |
| **Moreno Cabello, César – cmc00150**  **Sánchez Nájera, Elena – esn00006**  Grado en ingeniería Informática  Universidad de Jaén  Curso 2025/2026 |  |

# Índice

[Índice 2](#_Toc211689646)

[Descripción del problema 3](#_Toc211689647)

[Parámetros de configuración 3](#_Toc211689648)

[Estructura del proyecto 3](#_Toc211689649)

[Carpeta principal 3](#_Toc211689650)

[Carpeta “datos” 4](#_Toc211689651)

[Carpeta “clases” 4](#_Toc211689652)

[Carpeta “logs” 4](#_Toc211689653)

[Carpeta “modulos” 4](#_Toc211689654)

# Descripción del problema

Para el análisis y comparación de los diferentes algoritmos metaheurísticos, se utilizará un problema común, del que se estudiarán cuatro instancias diferentes. La implementación de la práctica se hará con Python.

Dicho problema recibe el nombre de **“Cadena de montaje”**. Se trata de mejorar el flujo de piezas entre los diferentes departamentos dentro de la fábrica FORD de Valencia. En cada instancia, los datos de partida son dos matrices cuadradas de igual tamaño: F, definida como la matriz de flujo de piezas que pasan entre departamentos; y D: la matriz de distancias entre departamentos.

La solución vendrá representada como una permutación de números naturales, donde la posición indica el departamento y el contenido la localización. El objetivo del problema es minimizar el coste de asignación de los departamentos para las diferentes localizaciones.

# Parámetros de configuración

Con objetivo de agilizar las ejecuciones, se ha creado un fichero *config.txt*, donde se indicarán los parámetros que se han de tener en cuenta para una determinada ejecución. El uso de este fichero permitirá implementar varios algoritmos para diferentes datos y con múltiples parámetros en una sola ejecución. El formato que debe seguir el fichero de configuración debe ajustarse al siguiente:

* DATA = ./datos/*NombreArchivoDatos Ruta del archivo de datos*
* ALG = *NombreAlgoritmo Nombre del algoritmo*
* Seed = *Semilla Semilla a utilizar en algoritmos aleatorios (opcional)*
* K = *n Parámetro de aleatoriedad en greedy\_aleatorizado (opcional)*
* Max\_iteraciones = *n Número máximo de iteraciones para parar un algoritmo (opcional)*
* Tenencia = *n Tenencia tabú (opcional)*
* Oscilación = *n Oscilación tabú (opcional)*
* Estancamiento = *n Estancamiento tabú (opcional)*

Los parámetros de configuración opcionales podrán inicializarse a 0 si no se desean utilizar, pero debe **mantenerse la estructura** general del archivo para su correcto funcionamiento.

Además de cumplir con el formato del archivo de configuración, se ha de tener en cuenta que los archivos de datos con los que se trabajará deberán estar incluidos en la carpeta del proyecto, “**datos**”. Como es evidente, el programa solo funcionará si recibe como argumento el fichero de configuración, que como **mínimo** deberá contener la dirección de un **archivo de datos** y el nombre de un **algoritmo** para trabajar.

# Estructura del proyecto

El proyecto se divide en carpetas para facilitar la clasificación de los ficheros. Así pues, se encuentra el directorio dividido entre las siguientes carpetas:

## Carpeta principal

Contiene el fichero de configuración, así como el *main* y el *README*.

## Carpeta “datos”

En este directorio, el usuario deberá introducir los ficheros de datos con los que desea trabajar, ya que el programa los buscará ahí en las ejecuciones donde sea necesario.

## Carpeta “clases”

Aquí se definen clases auxiliares, como la clase *configurador*, *extractor*, o *logs*. Las dos primeras clases son útiles para importar la información del fichero de configuración en el programa. Mientras tanto, la clase *logs* servirá para extraer los datos obtenidos en cada ejecución de cada algoritmo a un fichero de texto, donde el usuario podrá consultar el comportamiento de las metaheurísticas de manera individualizada y sin perder los datos tras finalizar el programa.

## Carpeta “**logs**”

En “**logs**” se guardarán los ficheros que resumen el comportamiento de los diferentes algoritmos. Dentro de los archivos se incluirá la configuración aplicada, con objeto de tener claro qué pasó durante la ejecución, así como la asignación de la permutación, el costo que supone y el tiempo de ejecución del algoritmo. En algunos algoritmos, también se guardarán los pasos intermedios, indicando las permutaciones, así como los pares intercambiados en el 2-OPT. Cada documento será nombrado siguiendo la estructura “*NombreAlgoritmo*\_ *NombreArchivoDatos*\_ *Semilla*”, donde semilla será un parámetro opcional en función del algoritmo que se haya definido.

## Carpeta “**modulos**”

Por último, se encuentran archivos que contienen el funcionamiento general del programa, como *heurísticas, donde se define cómo se implementa cada algoritmo, func\_auxiliares*, donde se definen funciones de ayuda, como el cálculo del costo o la factorización; o *prints*, donde hay funciones que sirven de ayuda a la hora de imprimir resultados o errores por pantalla.

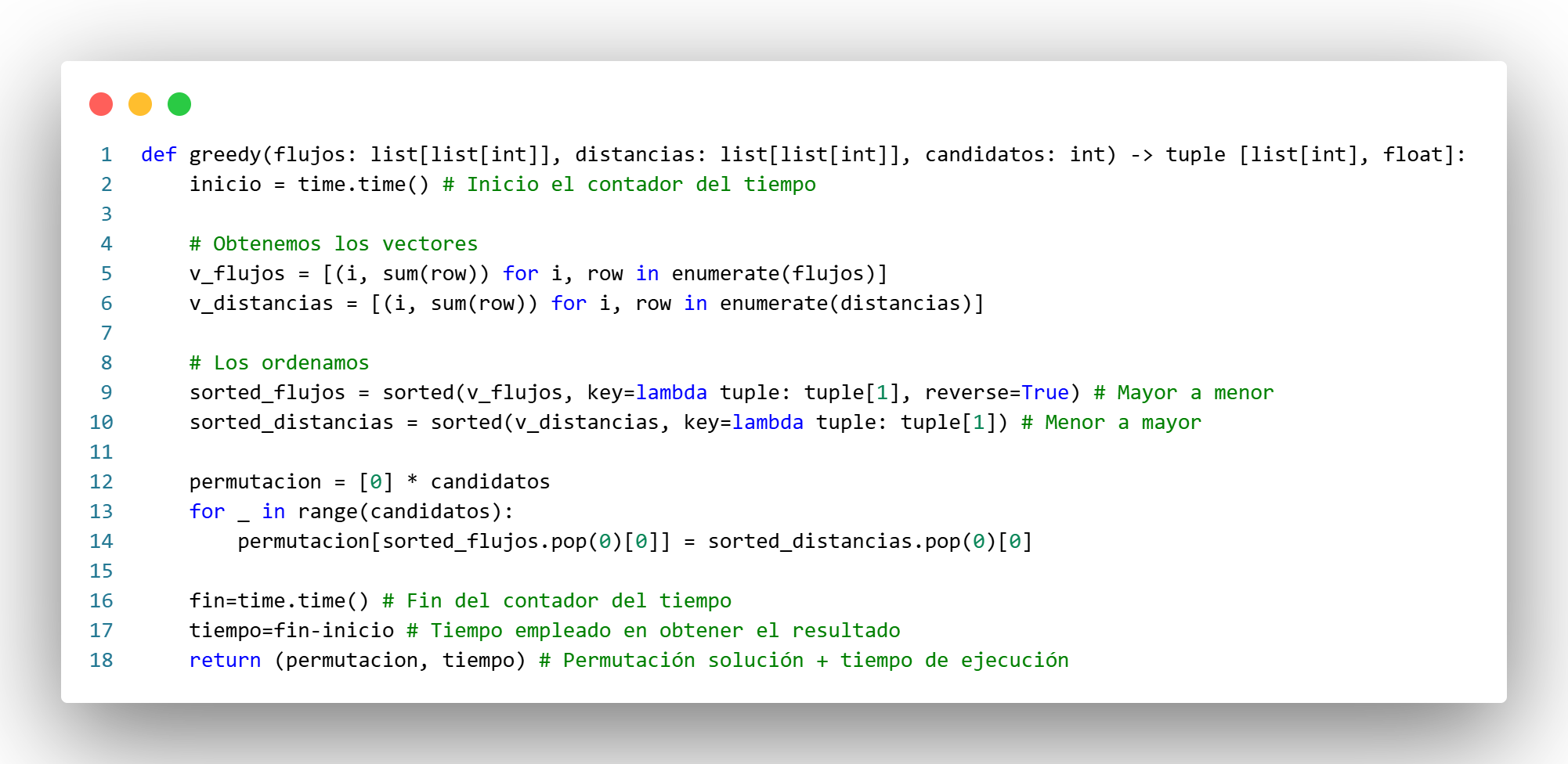
# Algoritmos utilizados

En este punto, se reflejarán los diferentes algoritmos que se han empleado para obtener diferentes soluciones para el problema. Para cada uno de ellos, se explicará su funcionamiento general, la implementación y los resultados obtenidos. Finalmente, se hará una comparación general con objetivo para poder analizar todos los algoritmos y llegar a conclusiones.

## Greedy

El algoritmo *greedy* tiene una lógica simple: buscar la mejor opción disponible, construyendo la solución de forma progresiva. Aplicado al problema, la solución se obtiene creando dos vectores: una con la sumatoria por filas de la matriz de flujos, y otro con la sumatoria por filas de la matriz de distancias. Una vez obtenidos, se ordenará el primer vector de mayor a menor, mientras que el segundo se ordenará a la inversa. De esta forma, se consigue que los departamentos con mayor flujo se asignen a las ubicaciones más cercanas, sacando de cada vector los elementos por el principio.

### Implementación



### Resultados obtenidos

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | FORD01 | | FORD02 | | FORD03 | | FORD04 | |
| GREEDY | **Tamaño** | 20 | **Tamaño** | 20 | **Tamaño** | 30 | **Tamaño** | 30 |
| **Mínimo global** | 3683 | **Mínimo global** | 27076 | **Mínimo global** | 13178 | **Mínimo global** | 151426 |
|
| ***Sol*** | **Time** | ***Sol*** | **Time** | ***Sol*** | **Time** | ***Sol*** | **Time** |
| Ejecución | 3876 | 0.0203 | 32748 | 0.0238 | 13786 | 0.0231 | 188018 | 0.0226 |
| Desviación | 5,24% |  | 20,95% |  | 4,61% |  | 24,16% |  |

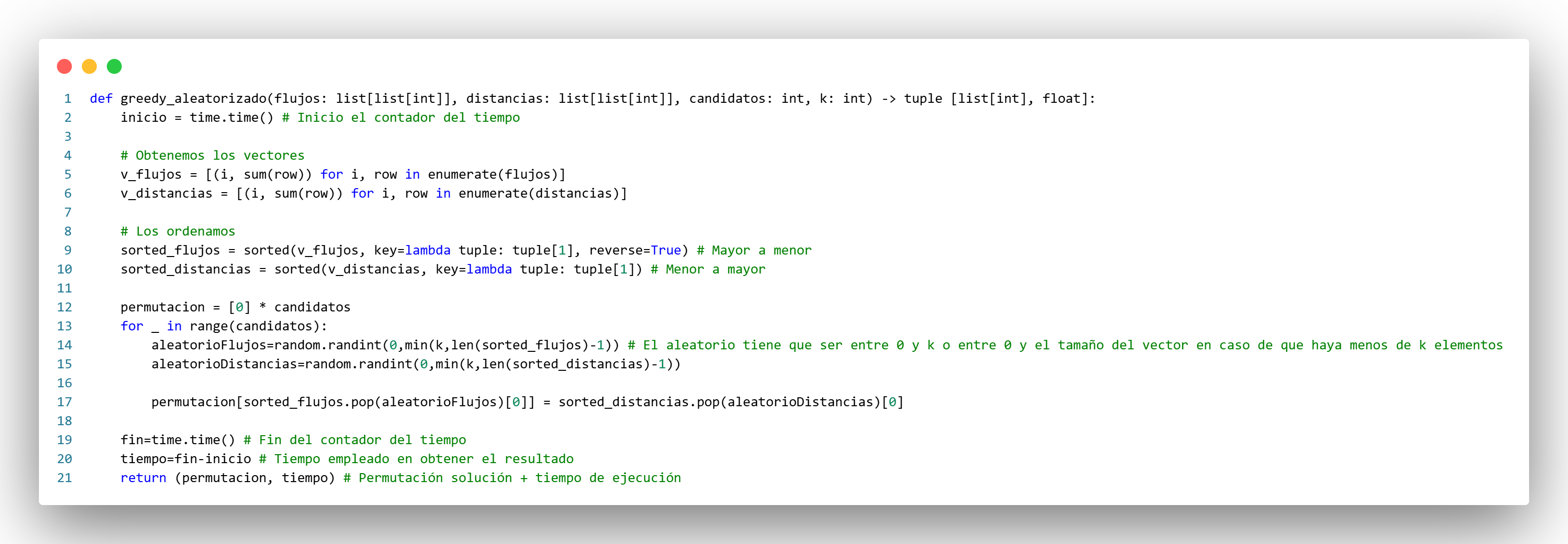
Como se puede apreciar según los resultados obtenidos, este algoritmo funciona muy rápido, pero no parece dar resultados muy óptimos, diferenciándose en 4.61% en el mejor de los casos del mínimo global para la instancia ford03, pero alejándose hasta en un 24.16% en el peor de los casos, en la ejecución de la instancia ford04.

## Greedy aleatorizado

Este algoritmo se basa en el anterior, con la diferencia de que, en este caso no se cogerá siempre la mejor opción (sacando los elementos de los vectores ordenados por el principio); sino que habrá un parámetro k que indicará un rango entre el que escoger para sacar los elementos de ambos vectores. Básicamente, se genera un aleatorio de 0 a k para elegir la posición a sacar del primer vector, repitiendo lo mismo para elegir la posición del segundo vector. Una vez obtenidas las posiciones, se asigna un elemento al otro.

### Implementación

Para la ejecución del algoritmo, es necesario que en el fichero *config.txt* se especifique al menos un parámetro k, así como una o varias semillas de inicialización para la generación de números aleatorios.



### Resultados obtenidos[[1]](#footnote-1)[[2]](#footnote-2)

El parámetro k se ha configurado con valor 5.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | FORD01 | | FORD02 | | FORD03 | | FORD04 | |
| GREEDY ALEATORIZADO | **Tamaño** | 20 | **Tamaño** | 20 | **Tamaño** | 30 | **Tamaño** | 30 |
| **Mínimo global** | 3683 | **Mínimo global** | 27076 | **Mínimo global** | 13178 | **Mínimo global** | 151426 |
|
| ***Sol*** | **Time** | ***Sol*** | **Time** | ***Sol*** | **Time** | ***Sol*** | **Time** |
| Ejecución 1 | 3878 | 0,0439 | 32748 | 0,0470 | 13786 | 0,0470 | 188018 | 0,0482 |
| Ejecución 2 | 3926 | 0,0408 | 34578 | 0,0358 | 13860 | 0,0594 | 191866 | 0,0761 |
| Ejecución 3 | 3914 | 0,0329 | 35722 | 0,0420 | 13788 | 0,0474 | 193280 | 0,0529 |
| Ejecución 4 | 3892 | 0,0391 | 35882 | 0,0355 | 13858 | 0,0463 | 191842 | 0,0775 |
| Ejecución 5 | 3902 | 0,0410 | 34194 | 0,0324 | 13704 | 0,0467 | 194866 | 0,0517 |
| Media | 3902 | 0,0395 | 35075 | 0,0385 | 13789 | 0,0494 | 193185 | 0,0613 |
| Desviación típica | 0,01 |  | 0,03 |  | 0,01 |  | 0,01 |  |
| Desviación | 5,96% |  | 29,54% |  | 4,64% |  | 27,58% |  |

Como se puede observar en la tabla, este algoritmo también cumple con una ejecución rápida, sin embargo, debido al factor aleatorio la calidad de las soluciones es muy dependiente de la semilla que se utilice para la generación de números. Esta diferencia de calidad se hace más notoria conforme se aumenta el tamaño y complejidad del fichero de datos inicial. Así, se obtienen soluciones con un costo de, aproximadamente, 188.000 y otras que superan 194.000 en la misma instancia, donde lo único que cambia es la semilla de generación. Como se puede apreciar en la desviación, los valores aún no terminan de acercarse al mínimo global que se busca conseguir; de hecho, utilizando la solución promedia no se ha conseguido reducir en ninguna de las instancias la desviación, respecto al algoritmo anterior. Posteriormente, se verá un análisis y comparación de algoritmos más detallado, donde se reincidirá sobre esto.

## Búsqueda local

Este algoritmo parte de una solución inicial, que será generada a través del algoritmo greedy\_aleatorizado en este caso. Se implementará con el operador de generación de vecinos 2-OPT, que consiste en intercambiar dos posiciones dentro de la permutación para obtener el vecindario. Además, se implementará el operador DLB (Don’t look bits), que permitirá a la búsqueda centrarse en zonas del espacio que puedan resultar más prometedoras. El operador se basa en inicializar un vector auxiliar con el mismo tamaño que la permutación original, inicializado a 0. Este vector se irá modificando, cambiando los 0 por 1, conforme se encuentren posiciones que no mejoran el coste con ninguna otra, de forma que se reduce el espacio de búsqueda.

La búsqueda local primero el mejor, que es el algoritmo que se va a implementar, consiste en, partiendo de una solución inicial, generar sus vecinos a través de un operador de vecindad (En este caso, 2-OPT). Una vez obtenidos los vecinos, se evaluarán hasta que se encuentre uno que mejore el coste actual, en cuyo caso habrá que moverse hacia él, actualizando la permutación y el coste. La técnica repite el proceso hasta que llega a un determinado número de iteraciones o, en su defecto, el vector DLB contiene todos sus valores a 1.

### Implementación

Para la ejecución del algoritmo, es necesario que en el fichero *config.txt* se especifiquen los parámetros necesarios para el algoritmo greedy\_aleatorizado (k, semillas); así como el número máximo de iteraciones para detener la ejecución del algoritmo en caso de que el algoritmo no pare por el operador DLB.

### 

### Resultados obtenidos[[3]](#footnote-3)[[4]](#footnote-4)

El parámetro k se ha configurado con valor 5 y max\_iteraciones con valor 5000.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | FORD01 | | FORD02 | | FORD03 | | FORD04 | |
| BÚSQUEDA LOCAL | **Tamaño** | 20 | **Tamaño** | 20 | **Tamaño** | 30 | **Tamaño** | 30 |
| **Mínimo global** | 3683 | **Mínimo global** | 27076 | **Mínimo global** | 13178 | **Mínimo global** | 151426 |
|
| ***Sol*** | **Time** | ***Sol*** | **Time** | ***Sol*** | **Time** | ***Sol*** | **Time** |
| Ejecución 1 | 3778 | 1,5390 | 32806 | 1,6232 | 13580 | 5,0101 | 185724 | 6,0971 |
| Ejecución 2 | 3782 | 1,5507 | 33796 | 1,6141 | 13568 | 5,3177 | 186198 | 5,7423 |
| Ejecución 3 | 3786 | 1,5738 | 33068 | 1,6346 | 13650 | 5,2619 | 185384 | 5,4643 |
| Ejecución 4 | 3734 | 1,5824 | 32642 | 1,7488 | 13568 | 5,0230 | 186194 | 5,5993 |
| Ejecución 5 | 3764 | 1,5495 | 33996 | 1,6348 | 13626 | 5,3170 | 188548 | 5,6772 |
| Media | 3769 | 1,5591 | 33262 | 1,6511 | 13598 | 5,1859 | 186410 | 5,7160 |
| Desviación típica | 0,01 |  | 0,02 |  | 0,00 |  | 0,01 |  |
| Desviación | 2,33% |  | 22,85% |  | 3,19% |  | 23,10% |  |

# Anexos

## Gráficas comparativas de algoritmo greedy\_aleatorizado

## Gráficas comparativas de algoritmo búsqueda local

1. Las semillas utilizadas *para cada ejecución son las siguientes: 15522133, 12255133, 55133122, 26959126, 95926126* [↑](#footnote-ref-1)
2. *En anexos se pueden ver gráficas del comportamiento del algoritmo con las diferentes semillas* [↑](#footnote-ref-2)
3. Las semillas utilizadas *para cada ejecución son las siguientes: 15522133, 12255133, 55133122, 26959126, 95926126* [↑](#footnote-ref-3)
4. *En anexos se pueden ver gráficas del comportamiento del algoritmo con las diferentes semillas* [↑](#footnote-ref-4)