INFORME PRÁCTICA 1

Metaheurísticas

Grupo 7:

David Rodríguez Muro, 26523873C
drm00035@red.ujaen.es
Juan Bautista Muñoz Ruiz, 26516720C
jbmr0001@eed.ujaen.es

Algoritmos implementados: Greedy, Primer Mejor con dlb iterativo, Primer Mejor con dlb aleatorio y Multiarranque

Grupo de prácticas 4, viernes (10:30-12:30)
Curso 2021/2022

ÍNDICE

DESCRIPCIÓN DEL PROBLEMA Y METAHEURÍSTICA UTILIZADA

PROBLEMA A RESOLVER

Los ingenieros de la multinacional estadounidense *FORD* necesitan realizar un estudio de su fábrica situada en Valencia para mejorar el flujo de piezas entre los distintos departamentos o unidades de fabricación.

El objetivo de este problema es encontrar la fórmula de distribución de los departamentos, conociendo la distancia entre los distintos departamentos y el flujo de piezas que existe.

Como vimos en el seminario se trata de un problema combinatorio de gran complejidad *NP-completo* de elevado coste computacional. Además la eficiencia de este problema se podría considerar cuadrática o incluso cúbica en algunos problemas similares. En cuanto a la estructura de los datos:

- Encontramos ***p*** departamentos donde **(p_i, i=1, ..., n)**
- Encontramos ***I*** localizaciones donde **(l_j , j=1, ... ,n)**
- Se definen dos matrices **F=(f_{ij})** y **D=(d_{ij})** con una dimensión cuadrada de n con la siguiente información:
    - **F** es la matriz de flujo de piezas, es decir, **f_{ij}** es el número de piezas que pasan del departamento i al departamento j.
    - **D** es la matriz de distancias, es decir, **d_{ij}** es la distancia entre el departamento i y el departamento j.
    - El coste de asignar p_i a l_k y p_j a l_l es:

      $$f_{ij} * d_{kl} + f_{ji} * d_{lk}$$

Matemáticamente el problema se centra en minimizar el coste de las asignaciones de los departamentos con las distintas localizaciones

$$min \quad \sum_{i,j=1}^{n} \sum_{k,p=1}^{n} f_{ij} d_{kp} x_{ij} x_{kp}$$

$$s.a. \quad \sum_{i=1}^{n} x_{ij} = 1 \quad 1 \leq j \leq n,$$

$$\sum_{j=1}^{n} x_{ij} = 1 \quad 1 \leq i \leq n,$$

$$x_{ij} \in \{0,1\} \quad 1 \leq i \leq n$$

REPRESENTACIÓN DE LA SOLUCIÓN

Representaremos la solución mediante un vector de enteros:

Descripción del vector:

- **N** se corresponde con el número total de unidades de fabricación o departamentos.
- Cada índice de la posición del vector se corresponde con una unidad de fabricación o departamento.
- En cada posición del vector almacenaremos un entero con la unidad de fabricación o departamento asignada en función de nuestros algoritmos.

Restricciones del vector:

- No debe haber elementos repetidos.
- Todas las unidades de fabricación índice deben tener un entero con una unidad de fabricación asignada en su posición.
- Los enteros asignados a cada índice deben encontrarse entre cero y el tamaño total (número de unidades de fabricación)

EVALUACIÓN DE LA SOLUCIÓN

En la evaluación de la solución utilizamos la siguiente función de factorización:

$$\Delta(\phi, r, s) = \begin{bmatrix} f_{rr} \cdot (d_{\pi_s\pi_s} - d_{\pi_r\pi_r}) + \\ f_{ss} \cdot (d_{\pi_r\pi_r} - d_{\pi_s\pi_s}) + \\ f_{rs} \cdot (d_{\pi_s\pi_r} - d_{\pi_r\pi_s}) + \\ f_{sr} \cdot (d_{\pi_r\pi_s} - d_{\pi_s\pi_r}) + \end{bmatrix} + \sum_{k=1, k\neq r,s}^{n} \begin{bmatrix} f_{kr} \cdot (d_{\pi_k\pi_s} - d_{\pi_k\pi_r}) + f_{ks} \cdot (d_{\pi_k\pi_r} - d_{\pi_k\pi_s}) + \\ f_{rk} \cdot (d_{\pi_s\pi_k} - d_{\pi_r\pi_k}) + f_{sk} \cdot (d_{\pi_r\pi_k} - d_{\pi_s\pi_k}) + \end{bmatrix}$$

Gracias a esta fórmula podemos observar si el movimiento de intercambio a realizar mejora respecto a la solución anterior, ya que en caso de que el valor de la función de factorización sea negativo nuestra nueva solución sería mejor que la anterior. Al encontrarnos con un problema de optimización la solución sería aceptada, en caso contrario se descarta.

```
Funcion factorizacion

COMIENZO

   valor += CalculaFuncionl()
   PARA k = 0 HASTA k < tamTotal HACER
      SI r ! k && s != k ENTONCES
         valor += Funcion(k, r, s)
       TERMINASI
    TERMINAPARA
FIN
```

EXPLORACIÓN DEL ENTORNO

Para realizar la exploración del entorno realizaremos movimientos de intercambio, encontrando así soluciones vecinas a nuestra solución inicial, la exploración varía en función del algoritmo utilizado.

Algoritmo Primer Mejor: En este caso, la exploración del entorno es igual para la versión iterativa y la versión random:

La exploración en este algoritmo consiste en realizar movimientos de intercambio Int(π,r,s) sobre nuestra solución inicial, pero tratamos de no buscar en el entorno completo ya que podría ser de una gran complejidad (Dependiendo del problema) y se centra en la búsqueda del espacio en el que realmente pueda haber mejores soluciones, para ello usaremos un vector DLB.

MOVIMIENTO POR EL ENTORNO

El movimiento por el entorno se efectúa aplicando permutaciones 2-opt al vector solución actual:

- En caso de encontrarnos en el Primer Mejor Iterativo DLB o Primer Mejor Iterativo Random DLB esta permutación será indicada por el *PrimerMejorVecino* (primer vecino con mejor coste).
- En caso de no encontrar vecino (solo en la búsqueda tabú del multiarranque) será indicada por el *mejorPeorVecino* o por el movimiento obtenido en la exploración o explotación.

OPERADORES

Encontramos un Operador 2-opt mediante el cual permutamos dos posiciones (i,j) indicadas.

De la forma que una permutación **(2,5)** en el vector solución:
[1, 7, 9, 6, 10, 11, 12, 5, 0, 3, 13, 17, 8, 19, 4, 18, 15, 14, 16, 2]

Daría resultado al vector solución:
[1, **10**, 9, 6, **7**, 11, 12, 5, 0, 3, 13, 17, 8, 19, 4, 18, 15, 14, 16, 2]

A continuación podemos ver su implementación en pseudocódigo:

```
Funcion permutacion

BEGIN

   a = vector[i]
   b = vector[j]

   vector[i] = b
   vector[j] = a

END
```

DESCRIPCIÓN DEL CÓDIGO

GREEDY

Partimos de un vector solución vacío y procederemos a su llenado ayudándonos de los siguientes vectores auxiliares, estos vectores estarán presentes en todos nuestros algoritmos:

- Un vector con la sumatoria de flujo de piezas por filas de cada unidad de fabricación o departamento, es decir, la suma de los flujos de su fila en la matriz de flujos de los datos proporcionados.
  $(f_i = \sum_{j=1}^{n} f_{ij})$
- Un vector con la sumatoria de distancias por filas de cada unidad de fabricación o departamento, es decir, la suma de los flujos de su fila en la matriz de flujos de los datos proporcionados.
  $(d_k = \sum_{l=1}^{n} d_{kl})$.

Durante cada iteración hasta completar el vector solución:

- Buscaremos en nuestro vector auxiliar de sumatorias de flujos la unidad con mayor sumatoria.
- Buscaremos en nuestro vector auxiliar de sumatorias de distancias la unidad de fabricación o departamento con menor sumatoria.
- En nuestro vector solución, en la posición de la unidad con mayor sumatoria de flujo asignaremos la unidad con peor sumatoria de distancias. Además estableceremos como inaccesibles, para no volver a asignarlas, estas dos unidades en sus respectivos vectores estableciendo su valor a -9999 (en el caso del vector de sumatoria de flujos) o 9999 (en el caso del vector de sumatoria de distancias)

Obtendremos un vector solución

El Pseudocódigo del algoritmo es el siguiente:

```
Algoritmo Greedy

COMIENZO

   i = 0;
   MIENTRAS i != tamTotal HACER
       flujo = EncuentraMayor()
       VF[flujo] = -∞
       distancia = EncuentraMenor()
       VD[distancia] = -∞
       VS[flujo] = distancia
    TERMINAMIENTRAS

FIN
```

PRIMER MEJOR ITERATIVO

Mediante un bucle externo hasta que igualemos el número de iteraciones máximas establecido o el espacio sea completo, todas las posiciones del dlb estén a 1, llamaremos a la función *busquedaVecino(num).*

Cuando una búsqueda de Vecino acabe su ejecución guardaremos la posición de la última unidad en que se han probado movimientos para que en la siguiente

iteración podamos seguir iterativamente desde la última unidad explorada si tras una búsqueda de vecino Primer Mejor aún queda espacio por explorar. Esta función está basada en la filosofía Don`t Look Bits.

Función de busquedaVecino(num):

- Por cada unidad existente comprobamos todos los movimientos de permutación posibles.
    - Si tras probar todos los movimientos asociados a esa unidad, ninguno provoca una mejora, se pone su bit a 1 para desactivarla en el futuro. Es decir, dlb[i]=1.
    - Si la unidad está implicada en un movimiento que genera una solución vecina que mejora, se pone su bit a 0 para reactivarla. Es decir, dlb[i]=0 y dlb[j]= 0.
- Esto lo conseguimos con un doble bucle en el que si la i o la j llegan al tamaño Total la inicializamos a 0 para no dejar las unidades anteriores a la posición de partida (num) sin explorar. Este bucle itera hasta llegar a la posición de partida (num) -1.

Función en *buscarVecino(num)* Pseudocodigo:

```
Funcion BusquedaVecino

BEGIN

   vecino = falso
   VectorAux = SolucionActual
   conti = 0
   contj = 0
   i = num
   MIENTRAS i != (num - 1) HACER
       SI dlb[i] = 0 ENTONCES
          bandera_mejora = falso
          j = i + 1

          SI j = TamTotal ENTONCES
             j = 0
          TERMINASI

          contj = 0
          MIENTRAS j != (num - 1) HACER

             SI j = (TamTotal - 1) ENTONCES
                j = 0
             TERMINASI

             SI check(i,j) = verdadero ENOTONCES
                permutacion(VectorAux, i, j)
                dlb[i] = dlb[j] = 0
                bandera_mejora = verdadero
                NuevoVectorSol = VectorAux
                UltimoIntercambio = i
             TERMINASI
             j = j + 1
             contj = contj + 1
           TERMINAMIENTRAS


          SI bandera_mejora = falso ENTONCES
             dlb[i] = 1
          TERMINASI
       TERMINASI

       SI i = (TamTotal - 1) ENTONCES
          i = 0
       TERMINASI

       i = i + 1
       conti = conti + 1

       SI conti = (TamTotal - 1) ENTONCES
          i = num - 1;
       TERMINASI
    TERMINAMIENTRAS
    DEVUELVE vecino

END
```

Pseudocódigo del algoritmo:

```
Algoritmo Primer Mejor con DLB it

COMIENZO

    vecino = verdadero
    MIENTRAS cont != numIteraciones HACER

       num = UltimoIntercambio;
       vecino = BusquedaVecino(num);

       SI vecino = verdadero ENTONCES
          solucionActual = nuevoVectorSolucion
       TERMINASI

       SI espacioCompleto = verdadero ENTONCES
          cont = numIteraciones
       TERMINASI

    TERMINAMIENTRAS

FIN
```

PRIMER MEJOR RANDOM

Mediante un bucle externo hasta que igualemos el número de iteraciones máximas establecido o el espacio sea completo, todas las posiciones del dlb estén a 1, llamaremos a la función *busquedaVecino(num).*

Cuando una búsqueda de Vecino acabe su ejecución generamos un aleatorio entre 0 y el número de unidades -1 Y en la siguiente iteración la función *busquedaVecino(num)* será llamada con esa posición aleatoria si tras una búsqueda de vecino Primer Mejor aún queda espacio por explorar. Esta función está basada en la filosofía Don`t Look Bits.

En cuanto a la implementación de la función *busquedaVecino(num)* es la misma que la vista anteriormente.

Pseudocódigo del algoritmo:

```
Algoritmo Primer Mejor con DLB aleatorio

COMIENZO

    vecino = verdadero
    num = aleatorio(0, TamTotal - 1)
    MIENTRAS cont != numIteraciones HACER

       vecino = BusquedaVecino(num);

       SI vecino = verdadero ENTONCES
          solucionActual = nuevoVectorSolucion
       TERMINASI

       num = UltimoIntercambio;

       SI espacioCompleto = verdadero ENTONCES
          cont = numIteraciones
       TERMINASI

    TERMINAMIENTRAS

FIN
```

MULTIARRANQUE

El Pseudocódigo del algoritmo es el siguiente:

```
Algoritmo Multiarranque

COMIENZO

    soluciones = solInicial
    greedyAleatorizadoLRC(soluciones)
    mejorGlobal = actual
    i = 0
    MIENTRAS i < TamLRC HACER
       solTabu = busquedaTabú(LRC[i])
       SI coste(mejorGlobalTabu) < coste(mejorGlobal) ENTONCES
          mejorGlobal = mejorGlobalTabu
       TERMINASI
       i = i + 1
    TERMINAMIENTRAS

FIN
```

Contamos con una función para generar una solución inicial greedy aleatorizada con la que llenar nuestra Lista Restringida de Candidatos para el multiarranque:

- Mediante la función *encuentraMayores()* guardamos las cinco unidades de fabricación con mejor sumatoria de flujo en un vector auxiliar.

  A continuación se muestra la función en Pseudocodigo:

```
Funcion EncuentraMayores

COMIENZO

    ValoresFlujoAux = vectorFlujos
    pos = 0
    PARA i = 0 HASTA i < CandidatosGreedy HACER
       mayor = 0
       PARA j = 0 HASTA j < TamTotal HACER
          SI ValoresFlujoAux[j] > mayor ENTONCES
             mayor = ValoresFlujoAux[j]
             pos = j
          TERMINASI
          vectorMayores[i] = pos
          ValoresFlujoAux[pos] = 0
       TERMINAPARA
    TERMINAPARA

FIN
```

- Mediante la función *encuentraConcentricos()* guardamos las cinco unidades de fabricación con peor sumatoria de distancia en un vector auxiliar.

  A continuación se muestra la función en Pseudocodigo:

```
Funcion encuentraConcentricos

COMIENZO

   ValoresDistanciaAux = VectorDistancia
   pos = 0
   PARA i = 0 HASTA i < CandidatosGreedy HACER
      menor = 0
      PARA j = 0 HASTA j < TamTotal HACER
         SI ValoresDistanciaAuxAux[j] < menor ENTONCES
            menor = ValoresDistanciaAuxAux[j]
            pos = j
         TERMINASI
         vectorConcentricas[i] = pos
         ValoresDistanciaAuxAux[pos] = 9999
      TERMINAPARA
   TERMINAPARA

FIN
```

- Guardamos una unidad de las cinco con mayor flujo elegida de forma aleatoria y guardamos una unidad de las cinco más concéntricas elegida aleatoriamente.

Cada candidato de la LRC lo generamos aplicando al vector solución del greedy una permutación entre las dos unidades elegidas aleatoriamente.

Pseudocódigo de la función greedy aleatorio:

```
Algoritmo Greedy aleatorizada

COMIENZO

    cont = 0
    solLRC = solucionInicial
    MIENTRAS contador < TamLRC HACER
        encuentraConcentricos()
        encuentraMayores()
        numF = aleatorioGreedyF()
        numD = aleatorioGreedyD()
        permutacion(solLRC, numF, numD)
        nuevaSol = solLRC
        LRC.add(nuevaSol)
        cont = cont + 1
    TERMINAMIENTRAS

FIN
```

Posteriormente, para cada candidato de la Lista Restringida de Candidatos generada, arrancamos una Búsqueda Primer Mejor tabú mediante una modificación de la función *busquedaVecino(num)*

Modificación en la función *buscarVecinos(num)*:

- Adaptamos la función de búsqueda de vecinos para poder guardar todos los mejores peores vecinos que no sean tabú en caso de no encontrar vecinos que mejoren en un vector llamado *mejoresPeoresVecinos*. Por temas de eficiencia guardamos solo el *mejorPeorVecino*.
- Además se ha implementado un método get para obtener el mejor de los peores vecinos.

Memoria a Corto Plazo:

- Memoria que contiene los últimos tres movimientos. Lo guardaremos en forma de Par. Se trata de una clase definida dentro cuyos parámetros son la posición i, la posición j, el vector generado con dicho movimiento de permutación y su tenencia tabú.
- Un movimiento es tabú si está incluido dentro de la Memoria a Corto Plazo. Si la permutación (i,j) o la permutación (j,i), que también será comprobada, son tabú este movimiento no será posible.
- Si un movimiento es tabú no será posible efectuarlo.
- Estos movimientos son generados con la función *encontrarVecinos()* y posteriormente detallaremos el criterio de elección de un movimiento que empeore o un movimiento que mejore.
- La tenencia tabú se ha implementado de forma que borramos un elemento de la Memoria a Corto Plazo cuando lleva *tenenciaTabú* iteraciones en la Memoria.
- Cuando insertamos un movimiento, si la Memoria a Corto Plazo está completa eliminamos el primero.

Memoria de Largo Plazo:

- Al tratarse de un problema asimétrico se ha implementado con una Matriz de *tamTotal* x *tamTotal.*
- Cuando realicemos un movimiento incrementaremos en uno la posición (i,j) de esta permutación. De esta manera tenemos guardada la frecuencia con la que se ha dado una determinada permutación.

Pseudocódigo del algoritmo multiarranque, más abajo explicación de su funcionamiento:

```
Algoritmo búsqueda tabú

COMIENZO

   actual = actualAleatorizado
   mejorGlobalTabu = actualAleatorizado
   cont = 0
   iteracionesSinMejora = 0
   num = aleatorio(0, TamTotal - 1)
   MIENTRAS cont != numIteraciones HACER
      vecino = busquedaVecino(num)
      SI vecino = verdadero ENTONCES
         iteracionesSinMejora = 0
         SI coste(actual) < coste(mejorGlobalTabu) ENTONCES
            mejorGlobalTabu = actual
         TERMINASI
      SINO
         actualizarMemCortoPlazo()
         iteracionesSinMejora = iteracionesSinMejora + 1
         reseteoDLB()
         SI iteracionesSinMejora = (PorcentajeOscilacion * Iteraciones) ENTONCES
            iteracionesSinMejora = 0
            numElementos = Math.round(1 / ProbOscilacion)
            rand = aleatorio(1, numElementos)
            SI rand < (numElementos * ProbOscilacion) ENTONCES
               p = exploracion()
               permutacion(actual, p[0], p[1])
            SINO
               p = explotacion()
               permutacion(actual, p[0], p[1])
            TERMINASI

          SINO

            SI mejoresPeoresVecinos.isEmpty() == false ENTONCES
               actual = obtenerMejorPeorVecino()
               memCortoPlazo.add(UltimaI, UltimaJ, actual)
               memLargoPlazo[UltimaI][UltimaJ] = memLargoPlazo[UltimaI][UltimaJ] + 1
               actualizarMemCortoPlazo
               SI coste(actual) < coste(mejorGlobalTabu) ENTONCES
                  mejorGlobalTabu = actual
               TERMINASI
            TERMINASI
          TERMINASI
        TERMINASI
        cont = cont + 1
    TERMINAMIENTRAS

FIN
```

Hasta llegar al número máximo de iteraciones llamaremos a la función *buscarVecinos():*

- En caso de encontrar vecino que mejore y que no sea tabú:
    - Actualizamos la *soluciónActual* permutando las posiciones indicadas por el movimiento.
    - Guardamos el movimiento en la Memoria a Corto Plazo y en la Memoria a Largo Plazo.
    - Llamamos a la función que actualiza las tenencias tabú de los elementos de la Memoria a Corto Plazo tras la iteración.

    - Comparando la solución actual con la solución tabú global. En caso de que la evaluación de coste de la solución actual sea menor que la evaluación de coste de la *soluciónTabúGlobal()* actualizamos la *soluciónTabúGlobal* con la *soluciónActual().*
- En caso de no encontrar vecino:
    - Si no hemos llegado al número máximo de iteraciones sin encontrar un Primer Mejor vecino y si hay al menos un *mejorPeorVecino* que no se tabú, es decir, la lista de Par mejoresPeoresVecinos no está vacía:
        - Actualizamos la *soluciónActual* permutando las posiciones indicadas en el *mejorPeorVecino*.
        - Guardamos el *mejorPeorVecino* en la Memoria a Corto Plazo y en la Memoria a Largo Plazo.
        - Llamamos a la función que actualiza las tenencias tabú de los elementos de la Memoria a Corto Plazo tras la iteración.
        - Comparamos la solución actual con la solución tabú global. En caso de que la evaluación de coste de la solución actual sea menor que la evaluación de coste de la *soluciónTabúGlobal()* actualizamos la *soluciónTabúGlobal* con la *soluciónActual()*
    - Si hemos llegado al número máximo de iteraciones sin mejorar la solución lanzamos la oscilación estratégica.
        - En función del porcentaje de oscilación estrategia calculamos un número de elementos (1/probOscilaciónEstratégica) y un punto de corte (1*probOscilaciónEstrategíca).
        - Mediante un random con rango máximo en el número de elementos calculado:
            - Si se encuentra por debajo del punto de corte lanzamos la exploración.
            - Si se encuentra por encima del punto de corte lanzamos la explotación.

Exploración:

- Generamos dos posiciones aleatorias (aleatorioI, aletorioJ) y en caso de que esa posición en la Memoria a Largo Plazo sea 0, es decir, ese movimiento no se ha dado nunca, lo devolvemos en forma de Par.
- En caso contrario volvemos a generar aleatoriamente en bucle hasta cumplir la condición.

Explotación:

- Recorremos toda la Memoria a Corto Plazo y elegimos el movimiento más usado.

ANÁLISIS DEL RESULTADO

Para poder comparar los resultados de cada algoritmo, aparte de almacenar su solución final hemos generado unos ficheros que contienen que ha ocurrido dentro de cada algoritmo, es decir los vecinos que se han ido encontrando a excepción del algoritmo greedy que solo almacenamos su solución, tiempo de ejecución y coste de la solución.

GREEDY

En primer lugar el algoritmo greedy no cuenta con una variable aleatoria por lo que su resultado siempre va a ser el mismo independientemente de la semilla empleada, en este algoritmo tras realizar una ejecución de cada uno de los resultados obtuvimos los siguientes resultados:

Greedy	Tamaño	20	Tamaño	20	Tamaño	30	Tamaño	30
	Mejor coste	3683	Mejor coste	27076	Mejor coste	13178	Mejor coste	151426
	ford01		ford02		ford03		ford04	
	C	Time	*C*	Time	*C*	Time	*C*	Time
Ejecución	3941,00	1,00	33954,00	1,00	13806,00	1,00	191745,00	1,00
Desviación	7,01 %	0,00	25,40 %	0,00	4,77 %	0,00	26,63 %	0,00

Como podemos observar este algoritmo destaca por su rápido tiempo de ejecución, ya que es muy bajo, pero como inconveniente tiene que su resultado

de la solución se encuentra en algunos casos, como con el fichero *ford04* o ford02, bastante alejada de su mejor solución. Sin embargo, para los ficheros *ford01* y *ford03* se trata de unas soluciones bastante buenas en relación a su tiempo de ejecución.

PRIMER MEJOR DLB ITERATIVO

Este algoritmo aporta unos resultados un poco extraños, ya que es capaz de encontrar el mejor coste en el fichero *ford04* en un tiempo de ejecución muy bajo, pero nosotros hemos optados por incluir el algoritmo random en el algoritmo multiarranque ya que dependiendo de la semilla obtendremos mejores resultados en cada fichero ford y su tiempo de ejecución es incluso mejor que el iterativo.

PMDLB Iterativo	Tamaño	20	Tamaño	20	Tamaño	30	Tamaño	30
	Mejor coste	3683	Mejor coste	27076	Mejor coste	13178	Mejor coste	151426
	ford01		ford02		ford03		ford04	
	C	Time	*C*	Time	*C*	Time	*C*	Time
Ejecución	3797,00	18,00	31459,00	7,00	13457,00	19,00	151426,00	19,00
Desviación	3,10 %	0,00	16,19 %	0,00	2,12 %	0,00	0,00 %	0,00

PRIMER MEJOR DLB RANDOM

En este algoritmo introducimos el concepto de semilla, la cual tiene una tarea importante, ya que según su semilla el resultado que obtendremos será uno u otro, las semillas empleadas se pueden observar en la siguiente imagen junto a sus resultados de ejecución:

PMDLB Random	Tamaño	20	Tamaño	20	Tamaño	30	Tamaño	30	
	Mejor coste	3683	Mejor coste	27076	Mejor coste	13178	Mejor coste	151426	
	ford01		ford02		ford03		ford04		Semillas
	C	Time	*C*	Time	*C*	Time	*C*	Time	
Ejecución 1	3817,00	6,00	29033,00	7,00	13525,00	16,00	178642,00	17,00	26523873
Ejecución 2	3799,00	7,00	32269,00	5,00	13530,00	14,00	178246,00	11,00	16720265
Ejecución 3	3810,00	5,00	32369,00	5,00	13483,00	11,00	179440,00	12,00	52387326
Ejecución 4	3812,00	7,00	32125,00	6,00	13535,00	12,00	176703,00	11,00	72026516
Ejecución 5	3806,00	5,00	32274,00	5,00	13491,00	12,00	175028,00	10,00	87326523
Media	3,42 %	6,00	16,76 %	5,60	2,54 %	13,00	17,29 %	12,20	
Desv. típica	0,18 %	1,00	5,34 %	0,89	0,18 %	2,00	1,16 %	2,77	

Como podemos observar que sus resultados mejoran respecto al algoritmo anterior y aunque sus tiempos de ejecución son algo más elevados vemos que se trata de un algoritmo bastante robusto ya que aunque cuente con una semilla su desviación es bastante pequeña siendo en el mejor de los casos de un 0.18% y en el peor de los casos de un 5,34%, por lo que no difieren demasiado.

MULTIARRANQUE

El último de los algoritmos mezcla los conceptos anteriores, ya que en su interior se ejecutan diferentes algoritmos tratando que sus resultados sean los mejores posibles, estos son los resultados obtenidos:

Trayectoria	Tamaño	20	Tamaño	20	Tamaño	30	Tamaño	30	
	Mejor coste	3683	Mejor coste	27076	Mejor coste	13178	Mejor coste	151426	
	ford01		ford02		ford03		ford04		Semillas
	C	Time	*C*	Time	*C*	Time	*C*	Time	
Ejecución 1	3759,00	11076,00	27076,00	12692,00	13410,00	64612,00	151426,00	70850,00	26523873
Ejecución 2	3770,00	11779,00	28171,00	12646,00	13385,00	63845,00	151426,00	74236,00	16720265
Ejecución 3	3770,00	12855,00	28638,00	13253,00	13402,00	90920,00	151426,00	68576,00	87326523
Ejecución 4	3763,00	12617,00	27076,00	13960,00	13407,00	76431,00	155185,00	85880,00	72026516
Ejecución 5	3769,00	12392,00	28638,00	11264,00	13420,00	68746,00	156779,00	69492,00	52387326
Media	2,26 %	12143,80	3,12 %	12763,00	1,72 %	72910,80	1,20 %	73806,80	
Desv. típica	0,13 %	718,44	2,93 %	992,30	0,10 %	11237,52	1,69 %	7082,86	

Podemos observar que en algunos casos como en los ficheros *ford04* y *ford02* con la semilla 26523873 hemos obtenido la mejor solución con el mejor coste, lo malo de este algoritmo es que el tiempo de ejecución es mucho más elevado que en el anterior, pero también se trata de un algoritmo más robusto ya que su

desviación es incluso más pequeña que en el anterior siendo en el mejor de los casos de 0.1% y en el peor de los casos de 2.93%.

Por lo que este algoritmo sin duda nos aporta las mejores soluciones a nuestro problema.

COMPARATIVA

	ford01		ford02		ford03		ford04		MEDIA	
	C	Time	*C*	Time	*C*	Time	*C*	Time	*C*	Time
Greedy	7,01 %	0,00	25,40 %	0,00	4,77 %	0,00	26,63 %	0,00	0,16	0,00
Búsqueda Local	3,42 %	600,00	16,76 %	560,00	2,54 %	1300,00	17,29 %	1220,00	10,00 %	920,00
Trayectoria	2,26 %	12143,80	3,12 %	12763,00	1,72 %	72910,80	1,20 %	73806,80	2,07 %	42906,10

En la tabla donde se recogen las medias de cada algoritmo podemos observar que sin duda el mejor de todos ellos es el algoritmo de multiarranque, ya que se trata de un algoritmo muy robusto ya que sus resultados no difieren demasiado en cada ejecución y son bastantes cercanos al mejor de los costes, si bien es cierto que este algoritmo tiene el problema de que sus tiempo de ejecución es el más elevado pensamos que sus resultados son lo suficientemente buenos como para esperar este tiempo de ejecución.

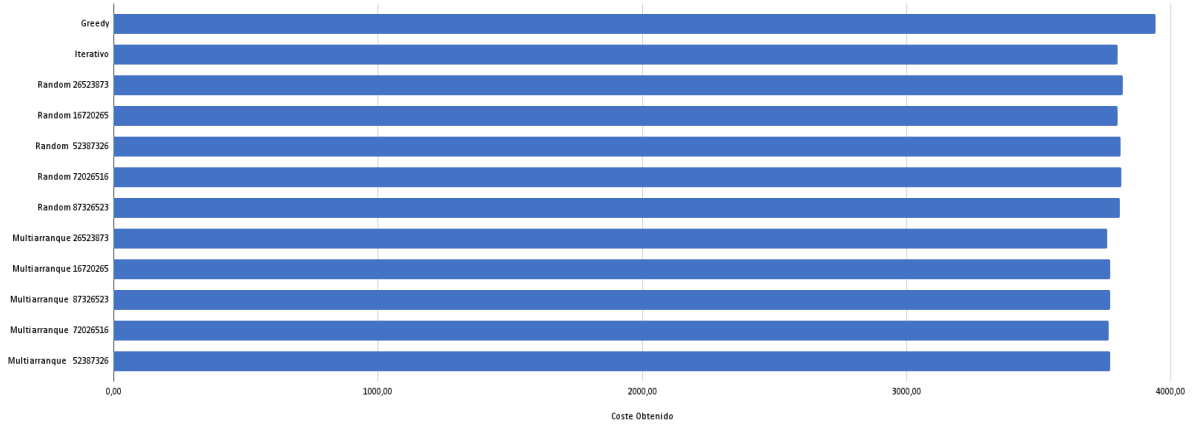
El algoritmo greedy es el más rápido de todos pero sus soluciones aunque son buenas tienen bastante margen de mejora, en conclusión este algoritmo es bueno para obtener una solución inicial y a partir de ella explorar sus soluciones vecinas para mejorarla.

En el algoritmo primer mejor al partir de la solución greedy exploramos el espacio de vecinos y en un tiempo muy bueno conseguimos mejorar la solución inicial dándonos mejores resultados que el algoritmo anterior.

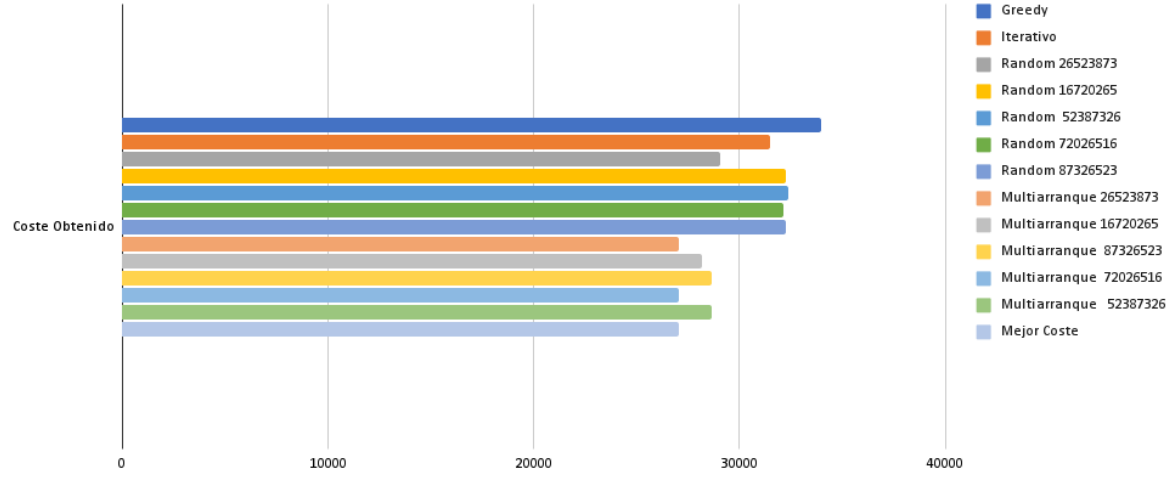
En comparación al algoritmo primer mejor iterativo y primer mejor random decidimos quedarnos con el segundo, ya que su tiempo de ejecución es bastante mejor que el iterativo y en función de la semilla empleada en algunos casos mejora la solución de nuestro algoritmo iterativo.

Finalmente mostramos en forma de gráficas las comparaciones de cada algoritmo en función a su semilla:

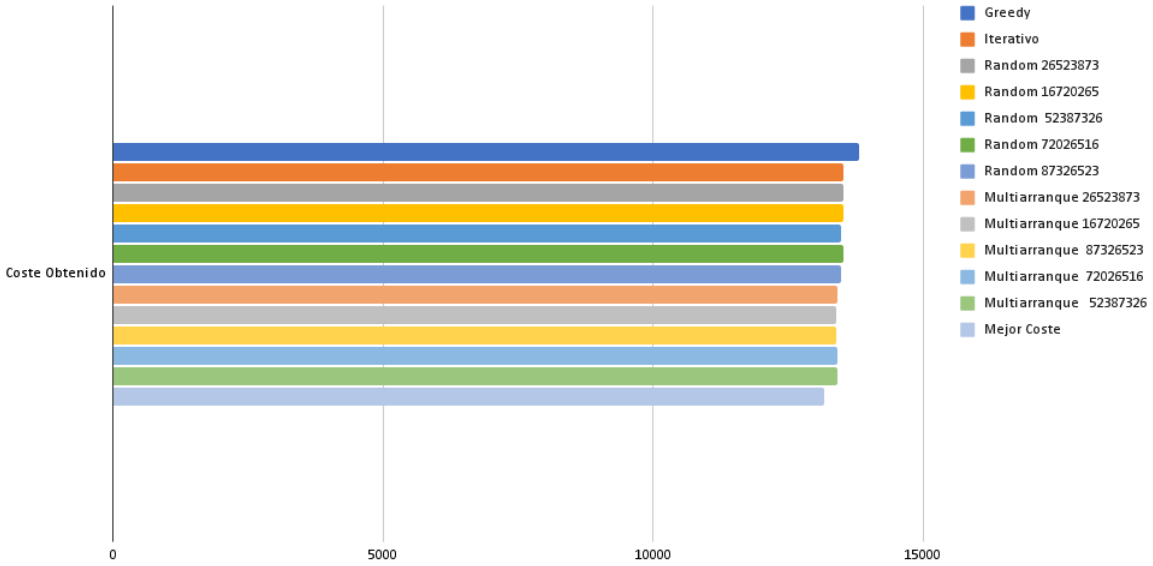
COMPARACIÓN DE COSTES FORD01



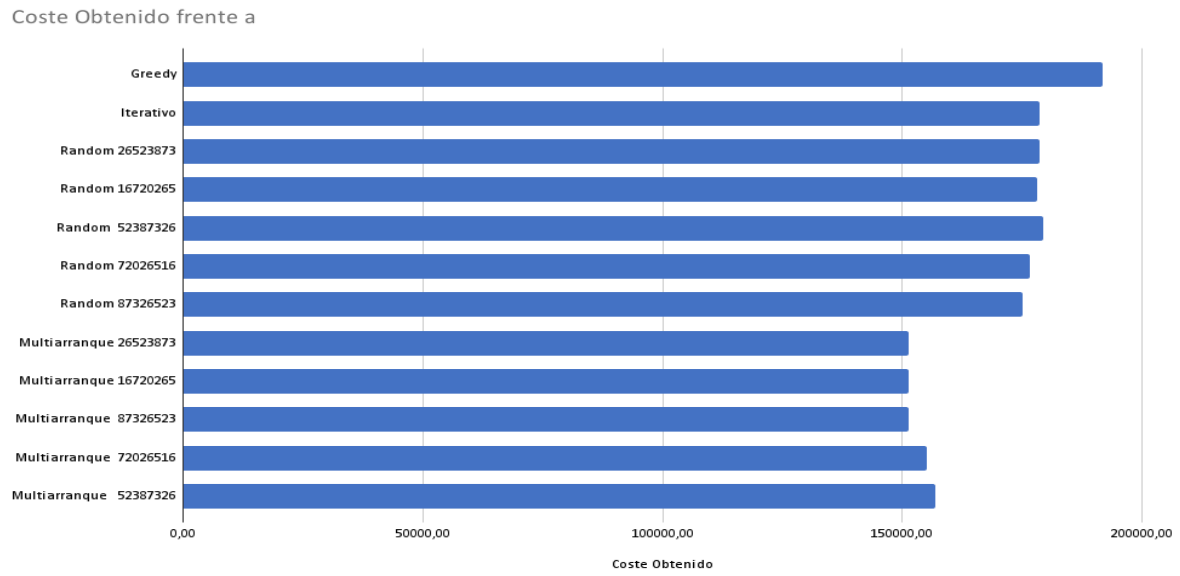
COMPARACIÓN DE COSTES FORD02



COMPARACIÓN DE COSTES FORD03



COMPARACIÓN DE COSTES FORD04



Claramente se observa que para cada problema se respeta lo descrito anteriormente, en cada algoritmo se obtiene un menor coste siendo el que mejor coste tiene el algoritmo multiarranque, su resultado también se ve ligeramente afectado por la semilla escogida, variando la mejor en cada fichero.