Grado en Ingeniería Informática

Metaheurística

Curso 2021/2022

Universidad de Jaén

Informe práctica 1

- Algoritmo Greedy
-Algoritmo Búsqueda Local
-Algoritmo Multiarranque

Adrián Arboledas Fernández
Grupo: 2
aaf00022@red.ujaen.es
26523518X

José Antonio Morales Velasco
Grupo: 2
jamv0007@red.ujaen.es
76593264K

Descarga Logs

ÍNDICE

INTRODUCCIÓN

Descripción del problema.

El problema consiste en “minimizar el coste” de distribución de piezas entre fábricas, asignando a los mayores flujos de piezas, las menores distancias entre departamentos. Para ello dado un conjunto ***p*** de ***P*** departamentos hemos de seleccionar un conjunto ***l*** de ***L*** localizaciones, de manera que la distribución de piezas entre departamentos sea lo más eficiente posible.

Definición del problema.

- $Sea\ p = \left(p_i,\ i=1,\ \ldots,\ n\right)$, departamentos.
- Sea $l = \left(l_j,\ j=1,\ \ldots,\ n\right)$, localizaciones.
- Se definen dos matrices $F = \left(f_{ij}\right)$ y $D = \left(d_{ij}\right)$, de dimsensión cuadrada $\mathbf{n}$.
- F: matriz de flujo de piezas, siendo f_{ij}, el número de piezas que pasan del departamento i al j.
- D: matriz de distancia, siendo d_{ij}, la distancia entre los departamentos i, j.
- El coste de asignar p_i a l_k y p_j a l_l es : $\left(f_{ij} \cdot d_{kl} + f_{ji} \cdot d_{lk}\right)$.

Función Objetivo.

La función objetivo del problema consiste en minimizar el coste de asignación de los departamentos con las distintas localizaciones.

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

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

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

Representación de la solución

Como representación de la solución se utilizará un vector de permutaciones del conjunto ***N,*** donde el índice de dicho vector corresponderá al flujo de piezas y el valor contenido será el departamento.

Por tanto para la implementación del vector de permutaciones, se usará un vector de enteros de tamaño ***N***, tal que: int [ ] vectorSolucion.

ALGORITMOS PARA LA RESOLUCIÓN DEL PROBLEMA

En este apartado se explicarán cómo se han implementado los distintos algoritmos para la resolución del problema:

ALGORITMO GREEDY

El algoritmo Greedy o también conocido como algoritmo voraz, es una estrategia de búsqueda basada en heurística que consiste en escoger la primera solución más óptima en el momento.

El algoritmo Greedy no garantiza siempre obtener un solución óptima, por tanto siempre hay que estudiar la corrección del algoritmo para demostrar si las soluciones obtenidas son óptimas o no.

Es un algoritmo que generalmente es usado para la resolución de problemas de optimización, toman la información que está disponible en cada momento y una vez tomada no vuelve a replantearse en el futuro.

Una de sus mayores ventajas es la rapidez y la facilidad de implementación del mismo.

Los distintos elementos que hay que tener en cuenta para el Greedy son los siguientes:

1. **Conjunto de candidatos:** para nuestro problema un conjunto de ***n*** elementos, tanto de flujo como de distancia.

2. **Solución:** representado por un vector de permutaciones de tamaño ***n***, cuyo contenido son los departamentos que están asignados al flujo correspondiente al índice.

3. **Función de selección:** informa de cuál es el elemento más prometedor para completar la solución. Como nosotros buscamos minimizar el coste, se le asignan a los mayores flujos, las menores distancias entre departamentos, ignorando los elementos ya seleccionados.

4. **Función de factibilidad:** Comprueba que no haya elementos repetidos. En nuestro caso como los elementos se escogen en orden nunca va a haber repetidos.

5. **Función solución:** Indica si la solución está completa, es decir, si ya tiene los ***N*** elementos.

6. **Función objetivo:** A los elementos de mayor flujo se les asigna la menor distancia entre departamentos, es decir, el departamento más cercano.

Implementación del algoritmo

Partimos de un vector de enteros de tamaño ***N*** que inicialmente se encuentra vacío. Se crean dos vectores adicionales donde se almacenan un par con la posición y el valor de la sumatoria de las filas de las matrices de flujo y distancia.

$$\text{Sumatoria de flujos de una unidad al resto} \left(f_i = \sum_{j=1}^{n} f_{ij} \right)$$

$$\text{Sumatoria de distancias desde una localización al resto} \left(d_k = \sum_{l=1}^{n} d_{kl} \right)$$

Una vez que están ambos vectores calculados se realiza una ordenación de menor a mayor de los valores de los mismos.

Posteriormente mediante el enfoque greedy, se selecciona el elemento con mayor flujo del vector (valor), el menor del vector de distancias y se añaden al vector solución en la posición del elemento del flujo (índice).
Dando como resultado el vector de permutaciones, en nuestro caso vectorSolución.

Función Principal

calculoGreedy(): int[ ] → Función que permite calcular el vector solución y lo devuelve una vez calculado. Dentro se realizan los cálculos antes descritos en el apartado de implementación del algoritmo.

Pseudocódigo

```
calculoGreedy()
        Begin
                vectorSolucion[nElementos];
                For i = 0 to n  do
                        sumFilaFlujo = 0;
                        sumFilaDistancia = 0;
                        For j = 0 to n do
                                sumFilaFlujo +=matrizFlujo[i][j];
                                sumFilaDistancia += matrizDistancia[i][j];
                        endFor

                        Pair parFlujo(i,sumFilaFlujo);
                        Pair parDist(i,sumFilaDistancia);
                        vectorFlujo.add(parFlujo);
                        vectorDist.add(parDist);
                endFor
                sort(vectorFlujo);
                sort(vectorDist);
                For i = 0 to n  do
                        vectoSolucion[vectorFlujo[key(tamVectorFlujo -1 -i)] =key(vectorDist[i])
                endFor

        Return vectorSolucion
        END
```

BÚSQUEDA LOCAL PRIMERO MEJOR ITERATIVO

La idea esencial de la búsqueda local iterativa radica en enfocar la búsqueda en un pequeño subespacio de soluciones definido por las soluciones que son óptimos locales. En primer lugar, se debe crear una solución inicial, la cuál además es la mejor solución hasta el momento. Esta solución puede generarse aleatoriamente, siguiendo alguna regla de despacho o mediante cualquier otro método que nos pueda proporcionar una solución inicial lo suficientemente buena, en nuestro caso la solución obtenida anteriormente por el algoritmo Greedy.

Uno de los grandes problemas que concierne a este algoritmo es el estancamiento en óptimos locales.

El algoritmo realiza el cambio de dos posiciones (2-opt) del vector solución que se le ha pasado, generando un vecino que puede o no ser mejor que la solución anterior.

Entorno:

El entorno del que se dispone con este algoritmo serían todas las posibles permutaciones del vector de soluciones. Dichas permutaciones serán válidas si el DLB está a 0.
El DLB se representa como un vector de ***N*** elementos de forma binaria, de forma que si una posición está a 0 indica que el intercambio es válido en esa posición. Por tanto el entorno de búsqueda se va acotando en función de cómo va cambiando. Si la posición está a 1 indica que ese intercambio no es válido, evitando así realizar la búsqueda sobre el espacio completo.

Factorización:

Como se intercambian dos localizaciones asignadas a las posiciones r y s de la permutación π, por tanto debemos hacer uso de la factorización de la función objetivo en cada caso.

La factorización permite el cálculo del coste de forma que solo se calcule el coste de la permutación realizada con respecto a la solución anterior.

Sea C(π) el coste de la solución actual, para generar π' el operador de vecino Int(π,r,s) escoge dos unidades r y s e intercambian sus localizaciones π(r) y π(s).

$$\pi' = \left[\pi(1), \ldots, \pi(s), \ldots, \pi(r), \ldots, \pi(n)\right]$$

De esta forma quedando afectados 2·n sumandos, los relacionados con las dos viejas y las dos nuevas localizaciones de las dos unidades alteradas, considerando un coste del movimiento igual a la diferencia de costes entre las dos soluciones que se puede factorizar como:

$$\Delta\, C(\pi, \mathrm{r}, \mathrm{s}) \;=\; C(\pi') \;-\; C(\pi)$$

$$\sum_{k=1, k!=r,s}^{n} \left[\begin{array}{l} f_{rk} \cdot \left(d_{\pi(s)\pi(\mathrm{k})} - d_{\pi(\mathrm{r})\pi(\mathrm{k})} \right) + f_{sk} \cdot \left(d_{\pi(\mathrm{r})\pi(\mathrm{k})} - d_{\pi(\mathrm{r})\pi(\mathrm{k})} \right) + \\ f_{kr} \cdot \left(d_{\pi(k)\pi(r)} - d_{\pi(k)\pi(r)} \right) + f_{ks} \cdot \left(d_{\pi(k)\pi(r)} - d_{\pi(k)\pi(s)} \right) \end{array} \right]$$

En el caso de que el valor sea negativo quiere decir que la solución es mejor que la anterior, al ser un problema de minimización se aceptaría.

Operador de intercambio

Para la implementación de la búsqueda local hemos usado un operador de intercambio ***2-opt***, en otras palabras realizamos un intercambio en dos posiciones del vector, es decir la posición “i” se intercambia por la posición “j”.

Implementación del algoritmo

Se parte de un vector de permutaciones de tamaño ***N*** como solución obtenida en el algoritmo Greedy, ejecutado anteriormente. Se tendrá un bucle while cuya condición de parada es que no se superen el número de iteraciones establecido y que el entorno DLB no se encuentre completamente a 1.

Se tienen dos bucles i, j, que se encargan de recorrer el entorno DLB en busca de posibles intercambios. El bucle i empieza en el último elemento donde se quedó en la última iteración del bucle while, que inicialmente se encuentra en la posición 0 y el bucle j empieza en la posición i + 1.
La condición de parada de ambos bucles, es que se haya recorrido todo el entorno o que encuentre una mejora en la solución.

Antes del comienzo del bucle j, se comprueba que la posición i, del dlb se encuentre a 0.
Una vez comprobado que se encuentre el dlb a 0, se accede al bucle j, donde se realiza una comprobación con la función *“checkMove(pos1,pos2,vectSol)”*, en caso de que el movimiento produzca una mejora en la solución, se llevará a cabo la reinicialización a 0 de la posición i,j del dlb y posteriormente se realiza el intercambio llamando a la función *“aplicarMovimiento(pos1, pos2, vectSol)”* .
En el caso de que el movimiento no produzca mejora, actualizamos la posición i del dlb, cambiando su valor de 0 → 1.

Función principal

PseudoCódigo

```
dlbIterativa(solucionActual)
        BEGIN

                While(dlbNoCompleto and noLimiteIteraciones)
                        For i = iPos to n do
                                If dlb[i] = 0 then
                                        mejora_solucion = false
                                        For j = i + 1 to n do
                                                CheckMove(i,j)
                                                If move improves then
                                                        aplicarMovimiento(i,j)
                                                        dlb[i] = dlb[j] = 0
                                                        mejora_solucion = true
                                                EndIf
                                        EndFor
                                        If  No mejoraSolucion then
                                                dlb[i] = 1
                                        EndIf
                                Endif
                        EndFor
                EndWhile


                Return solucionActual

        END
```

BÚSQUEDA LOCAL PRIMERO MEJOR ALEATORIZADA

La estructura y el funcionamiento de este algoritmo es similar al algoritmo de búsqueda local iterativa, la única diferencia es la utilización de un número generado aleatoriamente entre 0 y ***n.***
El número generado es usado como posición inicial del bucle *"i"*.
El resto de las funciones, parámetros son completamente idénticos a la búsqueda local iterativa.

Función principal

PseudoCódigo

```
dlbRandom(solucionActual)
        BEGIN
                iPos = numeroAleatorio entre 0 y n
                While(dlbNoCompleto and noLimiteIteraciones)
                        For i = iPos to n do
                                If dlb[i] = 0 then
                                        mejora_solucion = false
                                        For j = i + 1 to n do
                                                CheckMove(i,j)
                                                If move improves then
                                                        aplicarMovimiento(i,j)
                                                        dlb[i] = dlb[j] = 0
                                                        mejora_solucion = true
                                                EndIf
                                        EndFor
                                        If  No mejoraSolucion then
                                                dlb[i] = 1
                                        EndIf
                                Endif
                        EndFor
                EndWhile


                Return solucionActual

        END
```

ALGORITMO MULTIARRANQUE

La búsqueda local suele caer en óptimos locales que a veces están bastante alejados del óptimo global del problema.
Un algoritmo multiarranque consta de dos etapas, la *generación solución inicial* y *Búsqueda local.*
En la primera etapa se generan soluciones iniciales aleatorias en la región factible del problema mediante un greedy aleatorizado , y posteriormente en la segunda etapa se realiza una búsqueda local, en nuestro caso la búsqueda local aleatorizada.

A este algoritmo también se le implementa algunos rasgos del algoritmo de búsqueda tabú, como en la memoria a largo plazo y la memoria a corto plazo.

Lista Restringida de Candidatos - LRC

Se trata de una lista de tamaño específico, el cual genera una serie de posibles soluciones (poblaciones) que servirán como solución inicial del algoritmo Multiarranque. La generación de dicha lista se realiza con un algoritmo GRASP o greedy aleatorizado.

Greedy aleatorizado

Basándonos en la solución obtenida en nuestro algoritmo Greedy generamos diez soluciones que serán construidas de la siguiente manera:

1. Cogemos las 5 unidades con mayor flujo, y seleccionamos una de ellas mediante un aleatorio.

2. Cogemos las 5 distancias más concéntricas, y seleccionamos una de ellas mediante un aleatorio.

3. Se asigna la localización del paso 2 a la unidad escogida del paso 1

El procedimiento se repite de forma constructiva hasta generar todas las soluciones de la población LRC completa, evitando repeticiones.

Memorias adaptativas

- **Memoria a corto plazo**

  Se trata de una lista circular de tamaño 3 que almacena el último movimiento de mejora realizado, se almacenan los elementos durante 3 iteraciones. De esta forma se restringen los movimientos, y no puede volver a realizar una permutación de dos posiciones anteriores, lo que permite no volver a probar soluciones ya probadas al menos durante un tiempo.

- **Memoria a largo plazo**

  Se trata de una matriz de enteros de ***N x N*** elementos de forma que los índices son los distintos elementos a permutar y el valor de las celdas es el número de veces que se ha realizado la permutación. Este método permite realizar una oscilación estratégica, una **diversificación**(menos repetidos) o **intensificación**(más repetidos), de forma que se puedan escapar de una estancamiento de la solución en óptimos locales y haya más posibilidad de encontrar un óptimo global.

Oscilación estratégica

La oscilación estratégica es un método que permite escapar de los óptimos locales y empezar en un entorno de búsqueda distinto. Para ello hacemos uso de nuestra memoria a largo plazo y dependiendo del azar se realizan dos métodos, ambos dispondrán de una probabilidad del **50%** de utilización:

- **Diversificación**

  Permite usar la memoria a largo plazo para encontrar aquellas posiciones que se han intercambiado con menor frecuencia y realizar dicho intercambio para empezar en un espacio de búsqueda distinto. Esto permite por ejemplo ir a zonas del espacio de búsqueda inexploradas o poco exploradas y mejorar la exploración o directamente diversificar la búsqueda.

- **Intensificación**

  Permite usar la memoria a largo plazo para encontrar aquellas posiciones que se han intercambiado con mayor frecuencia y realizar dicho intercambio para empezar

en un espacio de búsqueda ya explorado. Esto permite encontrar mejores soluciones en ese espacio de búsqueda al seguir explorando.

En nuestro caso al realizar la oscilación estratégica, no reinicializamos la memoria a largo plazo, dado que es necesario llevar un control de las posiciones intercambiadas para no volver a explorar un entorno de búsqueda ya explorado.

Implementación del algoritmo.

Partimos de un un bucle do-while, mediante el cual se irán recorriendo las soluciones del LRC.

La solución actual se inicializa a la primera obtenida del LRC, guardamos la solución anterior y la mejor solución actual como una copia de la solución actual.
Por cada candidato del LRC se reinicia el dlb a su estado inicial, es decir todo a 0.

Posteriormente procedemos a la ejecución de la búsqueda local aleatorizada, pero con unos pequeños cambios internos en el código para incluir el uso de memorias adaptativas.

Lista de cambios búsqueda local aleatorizada:

1. Si en una iteración no mejora se cambia la solución actual, por la mejor de las peores, es necesario reiniciar el dlb al estado inicial.

2. Dentro del bucle *"j"*, se comprueba si el movimiento dado por el par (i,j) se encuentra dentro de la memoria a corto plazo.

3. Si el movimiento es de mejora, se actualiza la solución actual y la mejor de las soluciones actuales en el caso de ser mejor, si no es un movimiento de mejora, comprobamos si es mejor que la mejor de las peores y en caso afirmativo se actualiza. Si no se produce un movimiento de mejora dentro del porcentaje de oscilación se realiza la misma llamando a la función "*oscilacionEstrategica"* .

Al finalizar la búsqueda local aleatorizada, comprobamos si la mejor solución actual obtenida es mejor o peor que la mejor solución global, en caso afirmativo se actualiza la mejor solución global.

Pseudocodigo

función principal:

```
algoritmoMultiaArranque(solucionActual)
START

    Do
        iPos = numeroAleatorio entre 0 y n
        While(dlbNoCompleto and noLimiteIteraciones)
            For i = iPos to n do
                If dlb[i] = 0 then
                    mejora_solucion = false
                    For j = i + 1 to n do
                        estaListaTabu =  false
                        if estaListaTabu then
                            estaListaTabu = true
                        Endif
                        CheckMove(i,j, aolActual)
                        If move improves  and no estaListaTabu then
                            aplicarMovimiento(i,j, solActual)
                            dlb[i] = dlb[j] = 0
                            mejora_solucion = true

                            if mejorActual peor solucionActual then
                                mejorActual = solucionActual
                            Endif
                        else if no estalistaTabu then
                            solIntercambiada = aplicarMovimiento(i,j,solIntercambiadal)

                            if solIntercambiada mejor peorMejores then
                                mejorPeores = solIntercambiada

                            Endif

                            if cumple oscilacion then
                                oscilacionEstrategica()
                            Endif


                        EndIf
```

```
            EndFor
                        If  No mejoraSolucion then
                            dlb[i] = 1
                        EndIf
                    Endif
                EndFor
            EndWhile
            if MejorSolGlobal es mayor que mejorSolActual then
                mejorSolGlobal = mejorSolActual
            Endif
        While no se usen todos los LRC

END
```

Oscilación estratégica

```
oscilacionEstrategica()
START

        valor = Aleatorio
        if valor menor coeficienteOscilacion then

                For i  = 0 to N do

                        For j = 0 to N do
                                if i distinto j then
                                menor = menorValorMatriz
                                Endif

                        EndFor

                EndFor
                aplicarmovimiento(pos1, pos2, solActual)
                memoriaLargoplaxo[pos1][pos2]++
        Endif
        Else

                For i  = 0 to N do

                        For j = 0 to N do

                                if i distinto j then
                                        mayor = mayorValorMatriz
                                Endif

                        EndFor

                EndFor
                aplicarMovimiento(pos1 , pos2, solActual)
                memoriaLargoPlazo[pos1][pos2]++

END
```

CONCLUSIONES

ANÁLISIS COMPARATIVO

Algoritmo Greedy

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 (ms)	*C*	Time (ms)	*C*	Time (ms)	*C*	Time (ms)
Ejecución	3978,00	131	34587,00	0,00	13765,00	1,00	190904,00	0,00
Desviación	8,01 %	0,00	27,74 %	0,00	4,45 %	0,00	26,07 %	0,00

En esta tabla, están recogidos los datos de ejecución del algoritmo greedy para los distintos archivos a ejecutar.

Como se puede observar Greedy al ser un algoritmo voraz, devuelve una solución en un tiempo mínimo, lo cual lo convierte en un buen algoritmo para problemas muy grandes y de alto costo computacional.

El problemas es que las soluciones rara vez serán óptimos globales, ya que distan bastante de los mismos.

En nuestro caso, en el análisis de resultados de la tabla, se puede observar que algunos resultados se desvían del óptimo global en un porcentaje alto, como se ve en el ford02 y en ford04 que se desvían un 27.74% y un 26.07% respecto de la mejor solución. En el caso opuesto tenemos el archivo ford01 y ford03 que aunque todavía los resultados difieren del mejor resultado, en un porcentaje más bajo, un 8.01% y un 4.45% respectivamente, pero la diferencia es mucho más baja que con los otros dos archivos.

Con esta comparativa, se puede observar que este algoritmo obtiene óptimos locales más o menos cercanos al mejor resultado aunque también puede devolver valores más lejanos , con lo que se puede deducir que en funcion de tamaño del problema y de los datos que se tienen, el resultado puede alejarse más o menos de una solución mejor.

Algoritmo Búsqueda local Iterativa

Búsqueda Local Iterativa	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 (ms)	*C*	Time (ms)	*C*	Time (ms)	*C*	Time (ms)
Ejecución	3786,00	3,00	29898,00	1,00	13474,00	2,00	151426,00	1,00
Desviación	2,80 %	0,00	10,42 %	0,00	2,25 %	0,00	0,00 %	0,00

Respecto a los resultados obtenidos en este algoritmo se pueden observar resultados mucho más cercanos al óptimo global en comparación al algoritmo Greedy, aunque con un incremento poco significativo del tiempo empleado en la ejecución del algoritmo.

Los porcentajes de desviación con respecto a los mejores resultados se han disminuido considerablemente, siendo el más distante de 10.42% en ford02 mientras que el menos distante de 0% en ford04, al obtener un óptimo global.

Con estos resultados se puede observar que mediante un algoritmo de búsqueda local se puede reducir el espacio entre óptimos locales y óptimos globales, siendo posible la obtención de alguno, como es en nuestro caso, por lo que este algoritmo permite obtener resultados buenos o muy buenos con un coste computacional no muy alto

Algoritmo Búsqueda local Aleatorizada

Búsqueda Local 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	
	C	Time (ms)	*C*	Time (ms)	*C*	Time (ms)	*C*	Time (ms)
Ejecución 1	3791,00	2,00	31711,00	1,00	13436,00	1,00	171779,00	1,00
Ejecución 2	3788,00	5,00	***27076,00***	1,00	13463,00	1,00	***151426,00***	0,00
Ejecución 3	3791,00	2,00	30903,00	1,00	13457,00	1,00	175724,00	1,00
Ejecución 4	***3787,00***	3,00	31327,00	1,00	13441,00	2,00	174139,00	0,00
Ejecución 5	3793,00	2,00	31321,00	0,00	***13435,00***	0,00	***151426,00***	1,00
Media	2,91 %	2,80	12,53 %	0,80	***2,04 %***	1,00	8,90 %	0,60
Desv. típica	***0,07 %***	1,30	7,08 %	0,45	0,10 %	0,71	8,17 %	0,55

En los resultados obtenidos por este algoritmo, se puede observar que al añadir aleatoriedad a la búsqueda local se pueden obtener mejores soluciones respecto a una búsqueda local iterativa aunque el resultado de la misma estará condicionada por la semilla utilizada en cada ejecución del algoritmo.

En nuestro caso podemos apreciar cómo tras la implementación de un elemento de aleatoriedad en la búsqueda local, se puede observar una mejora con respecto al algoritmo anterior, pasando de obtener un óptimo global a obtener 3 entre las ejecuciones de ford02 y ford04.

Por otro lado, en el resto de archivos aunque no se obtienen óptimos globales, los resultados obtenidos son parecidos, en alguna ocasión un poco peor o un poco mejor dependiendo de los datos y de la semilla utilizada.
Además el coste computacional no se eleva sino que se reduce con respecto a la búsqueda local iterativa en la mayoría de iteraciones.

Algoritmo Multiarranque.

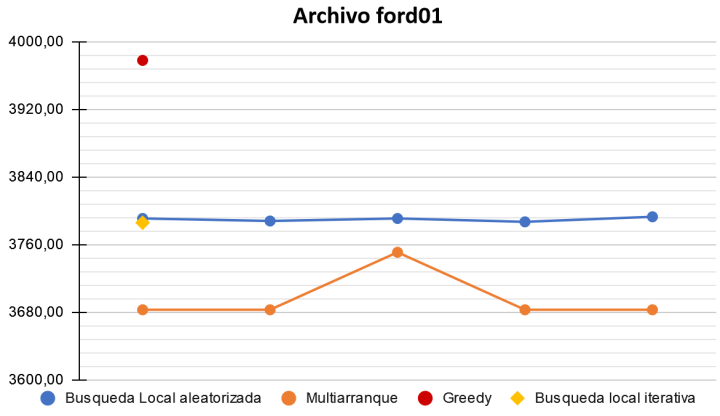
Multiarranque	Tamaño	20	Tamaño	20	Tamaño	30	Tamaño	30
	Mejor coste	3683	Mejor coste	27076	Mejor coste	13178	Mejor coste	151426
	ford01		ford02		ford03		ford04	
	C	Time (ms)	*C*	Time (ms)	*C*	Time (ms)	*C*	Time (ms)
Ejecución 1	***3683,00***	679,00	***27076, 00***	1452,00	13368,00	841,00	***151426,00***	3217,00
Ejecución 2	***3683,00***	702,00	***27076, 00***	1369,00	13396,00	839,00	***151426,00***	1869,00
Ejecución 3	3751,00	531,00	***27076, 00***	1480,00	13395,00	920,00	***151426,00***	3268,00
Ejecución 4	***3683,00***	2229,00	***27076, 00***	664,00	***13178,00***	897,00	***151426,00***	3332,00
Ejecución 5	***3683,00***	525,00	***27076, 00***	1394,00	***13178,00***	771,00	***151426,00***	3058,00
Media	0,37 %	933.20	**0,00 %**	1271,00	0,95 %	853,60	**0,00 %**	2948,80
Desv. típica	0,83 %	728,97	0,00 %	342,64	0,87 %	58,07	0,00 %	612,08

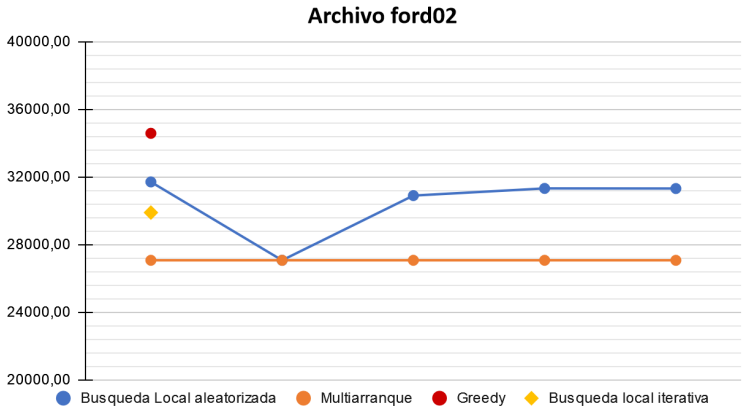
El algoritmo multiarranque, aunque tiene para cada ejecución un coste computacional mayor que el resto de algoritmos, llegando a tardar una ejecución hasta 3 segundos, esto lo suple con los resultados que devuelve.
Con respecto a los demás algoritmos para los ficheros, este obtiene en la mayoría de las ejecuciones óptimos globales, como es el caso de ford02 y ford04 donde se obtienen en las 5 ejecuciones el óptimo global.

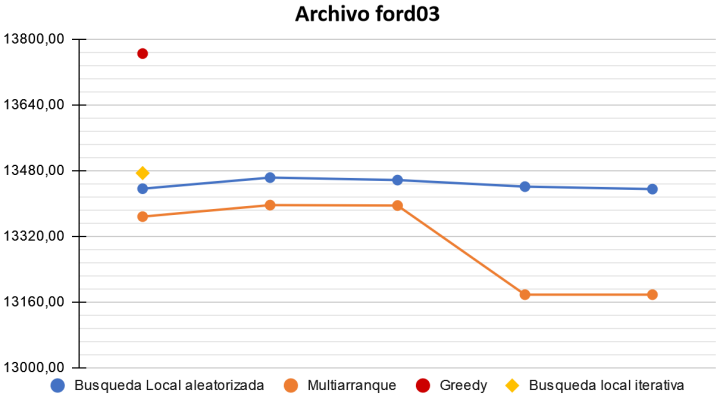
Aunque en el resto de archivos, se obtienen entre 2 y 4 óptimos globales. Todo esto también depende de las semillas usadas en cada ejecución del mismo.

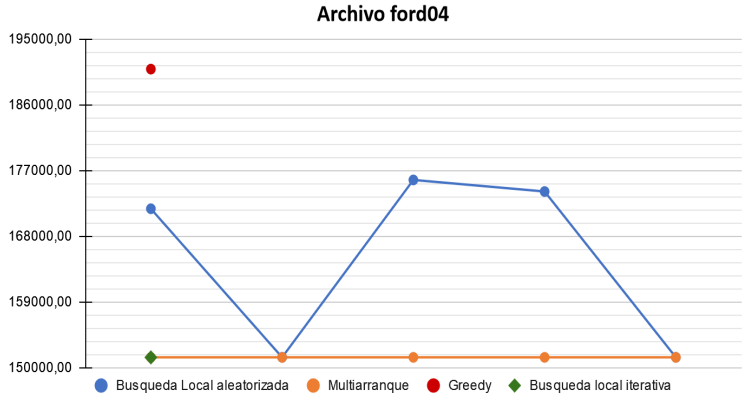
El uso de partir de un número de soluciones ***N*** generadas aleatoriamente a partir de la solución greedy aleatorio sumado al uso de movimientos de empeoramiento para explorar zonas de búsqueda inexploradas y el uso de la oscilación estratégica permiten que al intensificar o explorar nuevos entornos de búsqueda, se puedan encontrar óptimos globales con mayor facilidad aunque esto requiera un coste computacional más alto.

COMPARATIVA GRÁFICA









En las gráficas anteriores podemos observar como trabajan los distintos algoritmos en cada uno de los archivos.

Es importante recalcar que tanto el algoritmo greedy como la búsqueda local iterativa, se muestran en las gráficas como un único valor en el espacio, ya que solo cuentan con una ejecución, mientras que la búsqueda local aleatorizada y el multiarranque cuentan con 5 ejecuciones.

- **Archivo ford01:** La mayor diferencia la encontramos entre el costo obtenido del algoritmo greedy y el multiarranque, mientras que la búsqueda local iterativa y la búsqueda local aleatoria, se mantienen en el mismo rango de soluciones.

  Podemos apreciar como el conjunto de soluciones de la búsqueda local aleatorizada son robustas, ya que no hay mucha variación entre las soluciones obtenidas en cada una de las 5 ejecuciones.

El algoritmo multiarranque, obtiene soluciones robustas en casi todas sus ejecuciones, excepto en la tercera que como se ve, aumenta considerablemente el coste con respecto al resto de iteraciones.

- **Archivo ford02:** En este conjunto de datos podemos apreciar cómo el algoritmo multiarranque es más robusto que en conjunto de datos ford01, mientras que a la búsqueda local aleatorizada ocurre exactamente lo contrario, dado que en la segunda ejecución obtiene un óptimo global, mientras que en las otras 4 iteraciones obtiene un coste entre ***30903*** y ***31711*** .
  También en este caso la búsqueda local iterativa consigue mejor resultado que 4/5 de las ejecuciones de la búsqueda local aleatorizada.

- **Archivo ford03:** En este conjunto de datos el algoritmo de búsqueda local aleatorizada muestra más robustez en comparación con el conjunto de datos ford02. Respecto al multiarranque las tres primeras ejecuciones muestran unos valores poco distantes entre sí mientras que en las dos últimas ejecuciones, reduce su coste obteniendo 2 óptimos globales.
  En este caso la búsqueda local iterativa obtiene un mayor coste que la aleatorizada, mientras que el algoritmo greedy su solución al igual que en gráficas anteriores dista bastante de los valores de los otros 3 algoritmos.

- **Archivo ford04:** En el archivo ford04, podemos observar que el Greedy obtiene valores distantes con respecto a los demás algoritmos.

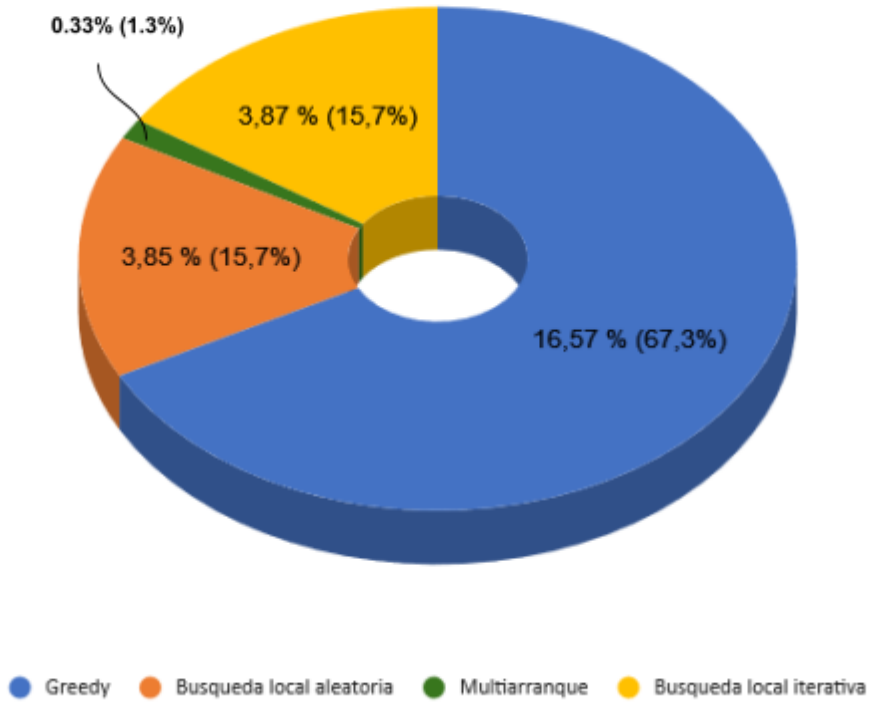
  La búsqueda local aleatorizada usada con este conjunto de datos, es menos robusta que con otros ficheros, ya que obtiene valores bastante dispersos. La diferencia más significativa es cuando encuentra óptimos globales, que pasa de valores más o menos parecidos sin mucha diferencia a un óptimo, reduciendo el coste obtenido considerablemente.

  El algoritmo multiarranque es robusto con este conjunto, devolviendo en todas sus iteraciones óptimos globales.

  El algoritmo de búsqueda local iterativa obtiene el óptimo global en su única iteración mejorando incluso a la búsqueda local aleatoria.

	ford01		ford02		ford03		ford04		MEDIA	
	C	Time	*C*	Time	*C*	Time	*C*	Time	*C*	Time
Greedy	8,01 %	0,00	27,74 %	0,00	4,45 %	0,00	26,07 %	0,00	16,57 %	0,00
Búsqueda Local	0,07 %	1,20	7,08 %	0,20	0,10 %	1,00	8,17 %	0,60	3,85 %	0,75
Trayectoria	0,37 %	933,20	0,00 %	1271,80	0,95 %	853,60	0,00 %	2948,80	0,33 %	1501,85
Búsqueda Local IT	2,80 %	3.0	10,42 %	1	2,25 %	2	0,00 %	1	3,87 %	1,333333333

Desviación Costes algoritmos



En esta gráfica se muestra la desviación de los costes generales del conjunto de cuatro archivos ford01, ford02, ford03 y ford04, respecto a sus óptimos locales.
Podemos por tanto apreciar cómo el algoritmo que más se desvía a la hora de calcular los costos es el algoritmo greedy, el cual se desvía un 67,3%, debido a que es un algoritmo constructivo donde la solución se va construyendo paso a paso.

Es curioso destacar como la desviación de la búsqueda local iterativa y la aleatorizada no distan mucho entre sí siendo tan solo en 0.02% de diferencia entre ambas, a favor de la aleatorizada. La diferencia está condicionada por la semilla utilizada para las ejecuciones del aleatorizado.

Como cabía esperar el algoritmo que obtiene menor desviación en sus resultados es el multiarranque, lo que se traduce en ser el que mejores costos de los 4 algoritmos empleados, debido a que intensifica y diversifica mucho más el entorno, además de poder moverse a peores movimiento evitando estancarse en óptimos locales, mientras que el entorno de los otros 3 algoritmos está más limitado.