TP2: Logística centralizada de primera milla

Bernardez Camila, Giménez Costa María, Oliva Micaela

2023-07-07

Introducción al problema y la decisión

En este trabajo nos centramos en intentar reducir costos operativos en terminos logisticos para entregar los paquetes hasta el punto de recepcion

En este trabajo nos centramos decidir como proveer servicios para reducir los costos operativos y logísticos a vendedores que son responsables de la entrega de paquetes hasta un punto de recepción de la red (llamada "la primera milla") para facilitar la operatoria. Para esto, debemos pensar formas para que cada vendedor elija eficientemente en que punto de consolidación (depósito) almacenar los ítems de su respectivo comercio. Buscamos encontrar un modelo de decisión que nos permita saber qué vendedor debe utilizar que deposito de forma que se utilice de manera eficaz la capacidad de almacenamiento de la red.

En principio, tomamos como métrica de éxito de una asignación, que cumpla con las capacidades máximas de cada depósito, la minimización de la distancia total recorrida por los vendedores.

Este problema lo vamos a modelar mediante el Problema de Asignación Generalizada (GAP) en el que contamos con N vendedores y M depósitos con capacidad máxima de recepción.

Para esto tenemos una matriz de costos, que son las distancias de cada vendedor a un depósito, una matriz de demandas de cada vendedor y otra matriz que posee las capacidades de cada depósito.

Aparte de esto, como cabe la posibilidad de que no todos los vendedores logren ser asignados a un depósito, consideramos un nuevo depósito cuya capacidad es infinita (INT_MAX) y el costo de cada vendedor para ser asignado a ese depósito será 3 veces su distancia máxima en concepto de penalización. Dado que la capacidad de este depósito es infinita, es irrelevante cuanto le demande cada vendedor, pero por simplicidad decidimos fijar todas estas demandas en cero.

Debido a que este problema es "difícil" en términos computacionales, vamos a implementar heurísticas constructivas, operadores de búsqueda local y metaheurísticas.

Descripción del modelo propuesto

Heurísticas

A continuación vamos a describir las heurísticas seleccionadas:

Inicialmente, planteamos una implementación *greedy* en la que le asignamos a un vendedor un depósito que tenga suficiente capacidad remanente para la cantidad de demanda de dicho vendedor para ese depósito. Notar que para cada vendedor no siempre va a haber un depósito en el que quepa. Debido a esto, tenemos un depósito extra al que asignamos a vendedores que no fueron asignados a ningún depósito. Cuando ocurra esto, se asumirá una penalización de 3xdmax (distancia máxima recorrida por un vendedor).

La segunda heurística, basada fuertemente en la penalización impuesta, es llamada "Worst fit". Si bien, lógicamente, tiene más sentido asignar vendedores a depósitos más cercanos, elegimos asignarle a un depósito

los vendedores más lejanos, el que más costo tenga, hasta agotar capacidad. Esta elección fue debido a la penalización, ya que ahora es de esperarse que los vendedores que queden sin asignar van a tener un dmax más chico y esto va a mejorar en nuestro cálculo de costo total, más que nada para instancias grandes.

Búsquedas locales

Para estas 2 heurística, implementamos 2 operadores de búsqueda local, swap y relocate.

Swap

Para el swap, dada una solución factible s, consideramos dos vendedores distintos j_1 y $j_2 \in s$ asignados a los depósitos $i_1 \neq i_2$.

- remover j_1 de i_1 e insertarlo en i_2;
- remover j_2 de i_2 e insertarlo en i_1;

Aplicar estos pasos para todos los vendedores j_1, $j_1 \in s$ asignados a depósitos distintos.

Definimos una solucion *s'* en el operador swap es una mejora de s si :

- $dist_{i_2 j_1} + dist_{i_1 j_2} < dist_{i_1 j_1} + dist_{i_2 j_2}$,
- $demanda_{i_2 j_1} \leq capacidad\ remanente_{i_2} + demanda_{i_2 j_2}$ y
- $demanda_{i_1 j_2} \leq capacidad\ remanente_{i_1} + demanda_{i_1 j_1}$

Relocate

Para el relocate, dada una solución factible s, consideramos un vendedor $j \in s$ asignado al depósito i.

Considerar un depósito $k \neq i$:

- remover j de su ubicacion actual;
- insertar j en el depósito k;

Aplicar estos pasos para todos los vendedores $j \in s$ y los depósitos distintos al actual.

Definimos una solucion *s'* en el operador relocate es una mejora de s si :

- $dist_{kj} < dist_{ij}$ y
- $demanda_{kj} \leq capacidad\ remanente_k$

Metaheurísticas

Como estamos buscando la minimización de la distancia total recorrida por los vendedores, nos gustaría escapar de óptimos locales, y llegar a óptimos globales, explorando mejor el espacio de soluciones y mejorando el costo total. Para esto, vamos a utilizar la metaheuristica GRASP (Greedy Randomized Adaptive Search Procedure) eligiendo como algoritmo goloso el *greedy* que implementamos y en lugar de elegir la mejor opción local, definimos una lista restringida de candidatos (RCL) que sea el 40% de los depósitos más cercano. El depósito que se incorpora en la construcción de la solución, se selecciona aleatoriamente dentro del RCL.

Luego de obtener una solución factible, aplicamos el algoritmo de búsqueda local con el best-improvement. Para considerar los dos vecindarios de los operadores, los exploramos de manera secuencial, primero aplicamos relocate hasta llegar a un mínimo local y luego sobre esa solución aplicamos swap.

Consideraciones generales respecto a la implementación del modelo, incluyendo dificultades que encontramos

Para implementar modelo decidimos guardar las distancias y demandas como `vector<vector<int»`, donde cada "fila" (subvector) es un depósito, y cada "columna" (elemento de los subvectores) es un vendedor. Para las capacidades consideramos un `vector<int>`.

A pesar de que en las instancias reales las demandas de un vendedor son las mismas para cada depósito, y las capacidades de las instancias provistas son siempre las mismas, esto no se asume en el código.

Como en las instancias reales los costos (o distancias) son `floats` con una cifra decimal, multiplicamos siempre los costos por 10, y luego en los cálculos los volvemos a dividir por 10.0, para luego convertirlos de vuelta en `int`. Decidimos manejar números enteros porque es el tipo de datos de la mayoría de los valores y además resultan más cómodos.

Queremos mencionar algunos aspectos del código que creemos pueden mejorarse, pero por falta de tiempo, ingenio o ambos no pudimos implementar.

- En primer lugar creemos que podríamos reducir un poco el tiempo de ejecución si cuando computamos el GRASP no buscamos en cada k iteración el RCL de depósitos más cercanos, ya que siempre van a ser los mismos (lo que cambia es cuál elegimos). *Anteriormente también habíamos mencionado que se podía evitar crear una nueva asignación temp en cada búsqueda local, pero para esta reentrega conseguimos implementarlo y mejoramos significativamente el tiempo de ejecución.*
- En segundo lugar somos conscientes de que podríamos deshacernos de ciertos ciclos, mejorando así la complejidad, en las clases de `worstFit` y `GRASP`. En el segundo no es necesario iterar por cada vendedor todos los depósitos *rcl_size* cantidad de veces, sino que podríamos ordenar el vector de distancias en orden creciente *a priori*, pero el problema es que debemos ordenar también el de demandas, y perdemos las posiciones originales de los vendedores. Algo similar ocurre con `worstFit`, podríamos ordenarlo en orden decreciente.

Resultados obtenidos en la experimentación

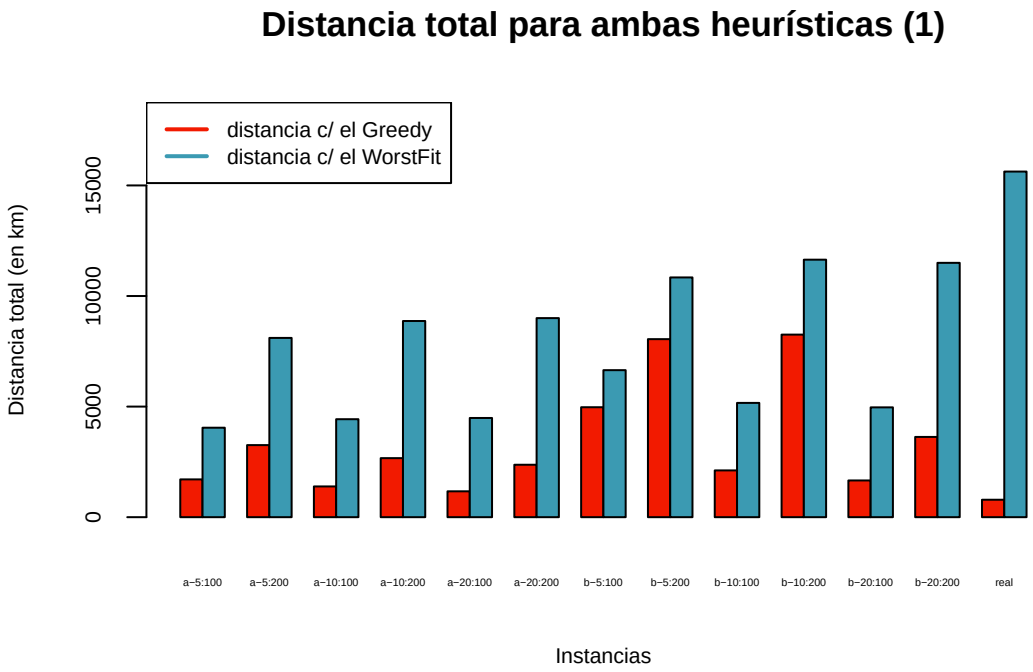
La expreimentación se realizó en base a todas las instancias de tipo a, b y la instancia real, y 5 instancias representativas de la instancia e. El análisis del `GRASP` se realizó sobre una instancia de cada tipo, incluyendo la real (originalmente esto fue por falta de tiempo, pero decidimos dejarlo así porque creemos que es suficiente para el análisis que realizamos más adelante).

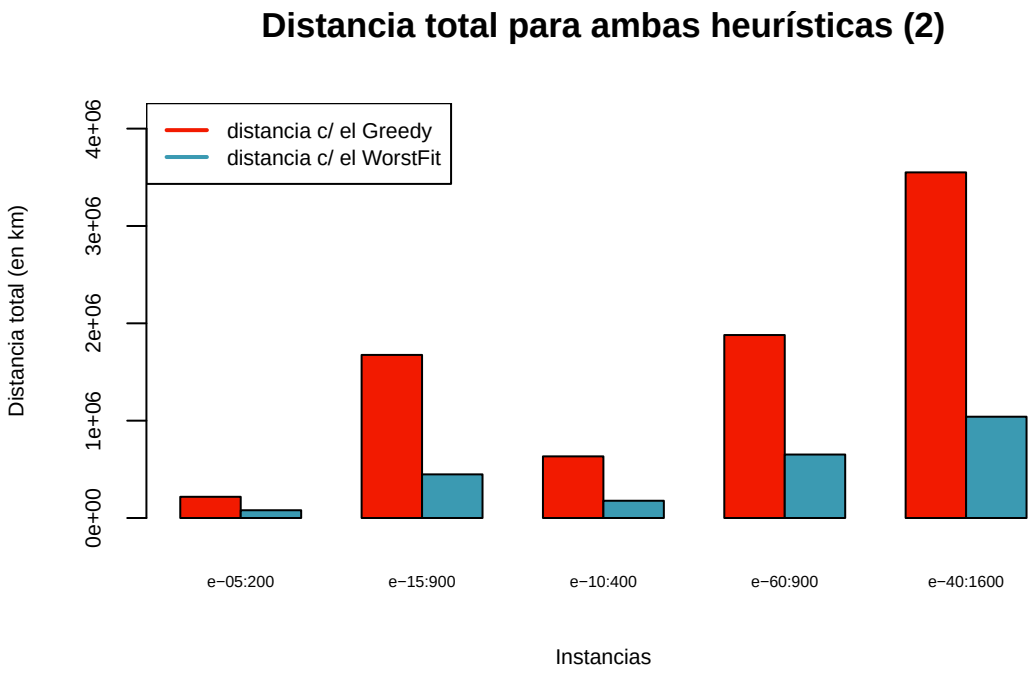
Diferencia en la distancia total computada por las heurísticas

Como podría intuirse, la heurística *greedy* es considerablemente mejor, ya que busca elegir a cada paso el depósito más cercano a cada vendedor, mientras que el otro elige los más lejanos de manera contraintuitiva (1).

Sin embargo, si observamos estos resultados para las instancias de tipo e, el `worstFit` es la mejor opción. Habiendo observado los archivos de estas instancias, notamos que hay mayor variación en los costos que en otros archivos. Por ejemplo, en el archivo *a05100* todas las distancias parecen rondar el 50, mientras que en el *e10400* hay distancias de menos de 10 y otras de casi 1000. Esto ocasiona que si elegimos los más cercanos, la penalización sobre los más lejanos sea mucho mayor, por ejemplo 3000 contra 30 en el ejemplo comentado (2). *Esto es teniendo en cuenta la penalización propuesta en el archivo original y con la que trabajamos a lo largo del TP.*

Aunque como vemos este método es eficiente para encontrar una mejor distancia total, es posible que la cantidad de vendedores que quedaron desasignados sea mayor. Computar ese dato quedará para un trabajo futuro.





Diferencia en la distancia total mejorada por las búsquedas locales

Para probar cuánto podemos mejorar la solución original encontrada por las heurísticas probamos ambos operadores por separado hasta llegar a un óptimo local.

Greedy

El operador relocate no logró ninguna mejora para esta heurística en ninguna de las instancias probadas. Sin embargo, como el greedy elige siempre el depósito más cercano a cada vendedor siempre que haya suficiente capacidad remanente, era de esperarse que no hubiera mejora, porque sino lo hubiera elegido en la solución inicial.

El operador swap, en cambio, sí logró mejoras más considerables. Para las instancias representativas mostradas a continuación las mejoras relativas son de 0.5%, 11.17%, 14.28% y 2.53%, respectivamente.

	dist_greedy	dist_relocate	dist_swap
a-05:100	1709	1709	1700
b-10:200	8255	8255	7333
e-10:400	633169	633169	542750
real	788	788	768

WorstFit

Ambos operadores resultan en mejoras significativas para las soluciones encontradas con el método de *worst fit*, aunque cuál es más efectiva depende de cada instancia, y en algunos casos el relocate sigue sin dar ninguna mejora. Por ejemplo, para las instancias representativas vemos que para *a-05100* y *b-10200* es preferible el swap, y para *e-10:400* y la instancia real el relocate, además de que para *b-10200* el relocate directamente no hace nada.

	dist_wf	dist_relocate	dist_swap
a-05:100	4046	1710	2001
b-10:200	11643	11643	5844
e-10:400	177231	93518	174715
real	15627	922	1633

Creemos que tiene sentido que en varios casos el relocate logre mejorar la solución encontrada con esta heurística, y que las mejoras con ambos sean más notorias que las mejoras sobre el *greedy*, porque ya habíamos mencionado antes que esta heurística puede encontrar soluciones muy malas (en especial a instancias pequeñas), por lo que queda mucho lugar para mejoras. Sin embargo, nos resulta extraño que en algunos casos el relocate no funcione, y no le encontramos una explicación como al relocate sobre el *greedy*.

Diferencia en el tiempo para ambas heurísticas y por los operadores de búsqueda local

En cuestiones de tiempo, el `worstFit` resulta peor en la teoría, pero en la práctica no hay mucha diferencia. En la tabla (4) del apéndice podemos observar que la instancia que más tardó en computar el `worstFit` tuvo un tiempo de 34071 microsegundos, o 0.034 segundos.

El operador relocate es el más rápido de los dos, lo cuál es esperable porque tiene que considerar nxm combinaciones de vendedores-depósitos, mientras que el swap considera toda combinación de vendedores nxn (y m *leq* n). Igualmente, si bien el relocate *es* más rápido que el swap, tarda más en encontrar un mínimo local a partir de una solución s encontrada mediante `worstFit`, y esto se debe a lo explicado anteriormente de que hay más lugar para mejoras (salvo en los casos donde el relocate no hace nada, y podemos ver también en el apéndice que en esos casos tarda muy poco).

	time_greedy	time_relocate	time_swap
a-05:100	23	15	1774
b-10:200	42	27	117083
e-10:400	77	49	704358
real	3078	11048	1297760

	time_wf	time_relocate	time_swap
a-05:100	163	3074	24461
b-10:200	491	64	243103
e-10:400	1893	12943	77456
real	17626	13475900	57784300

Diferencia en la distancia total greedy vs GRASP

Resulta relevante comparar la mejora que ofrecen una metaheurística a una heurística porque, como se mencionó anteriormente, el GRASP utiliza al *greedy* como heurística golosa de base.

Para obtener los mismos resultados en cada ejecución, seedeamos el randomizador con 1, y obtuvimos lo siguiente:

En líneas generales el GRASP logró encontrar mejores soluciones que la heurística original (o iguales en la primer fila), con excepción de la instancia real. Ambos resultados se explican con el funcionamiento de la metaheurística: al randomizar y obtener varios puntos de partida, luego aplicando búsqueda local podemos obtener varios óptimos locales (potencialmente distintos) y así elegir el mejor de ello, lo cuál nos da más probabilidades de encontrar una mejor solución. Como punto de comparación incluimos al swap, que era un operador decente para aplicar al *greedy*.

Sin embargo, la randomización también puede jugar en contra, y darnos soluciones malas que incluso habiendo aplicado búsqueda local, no llegan a buenos óptimos. Creemos que esta segunda situación es lo que ocurrió para la instancia real, y también por eso al aumentar la cantidad de iteraciones va mejorando un poco, porque quizás algunas randomizaciones no resultan tan malas. De cualquier manera no nos parece que la solución obtenida con el `GRASP` sea tan mala, ya que 788 km totales suponen aproximadamente 0.7 por vendedor en promedio, y 937 km (el mejor resultado obtenido con `GRASP`) son 0.85 km por cada uno.

	iteraciones	dist_greedy	dist_swap	dist_GRASP
a-05:100	1	1709	1700	1709
a-05:100	25	1709	1700	1698
a-05:100	50	1709	1700	1698
a-05:100	100	1709	1700	1698
b-10:200	1	8255	7333	5973
b-10:200	5	8255	7333	5025
b-10:200	10	8255	7333	4783
b-10:200	25	8255	7333	4571
e-10:400	1	633169	542750	335056
e-10:400	5	633169	542750	301227
e-10:400	10	633169	542750	268133
real	1	788	768	984
real	5	788	768	949
real	10	788	768	937

Como era de esperarse el tiempo del *greedy* es mucho mejor, ya que sólo realiza una ejecución de 2 ciclos, mientras que el `GRASP` debe encuentra k soluciones de manera *greedy*, en dónde además debe encontrar los *rcl_size* depósitos más cercanos, y luego aplicar búsqueda local hasta el óptimo para cada iteración.

En nuestro peor caso, la ejecución demoró 720 segundos, o 12 minutos (aunque como mencionamos en consideraciones, creemos que podría realizarse un poco más rápido), y además cuando la cantidad de vendedores es muy grande (en especial en comparación con la cantidad de depósitos), el tiempo aumenta exponencialmente. Además, aunque consigue mejores resultados que el swap casi siempre, lógicamente demora mucho más (el caso que `GRASP` tarda 12 minutos, swap tarda aproximadamente 1 segundo).

	iteraciones	time_greedy	time_swap	time_GRASP
a-05:100	1	23	1774	20562
a-05:100	25	23	1774	443059
a-05:100	50	23	1774	977843
a-05:100	100	23	1774	1816570
b-10:200	1	42	117083	218093
b-10:200	5	42	117083	1087340
b-10:200	10	42	117083	2173260
b-10:200	25	42	117083	5175090
e-10:400	1	77	704358	999191
e-10:400	5	77	704358	4827170
e-10:400	10	77	704358	10527700
real	1	3078	1297760	89024100
real	5	3078	1297760	410057000
real	10	3078	1297760	720833000

Análisis del caso real

Buscamos responder las siguientes preguntas:

- ¿Es suficiente la capacidad de la red de depósitos, o es necesario expanidr la misma para poder dar respuesta a todos los vendedores? No creemos que sea necesario expandir la red, ya que observamos que todos los vendedores fueron asignados a los depósitos originales (es decir, no al agregado de penalización) con la heurística *greedy*.
- ¿Es posible encontrar una asignación donde los vendedores recorrran una distancia razonable para entregar sus paquetes? Sí, de hecho la encontrada nos parece razonable, ya que como mencionamos anteriormente 788km implica en promedio menos de 1km por vendedor.
- ¿Es factible desarrollar una herramienta que nos permita experimentar con distintos escenarios y obtener soluciones de buena calidad en unos pocos minutos? Si bien como explicamos el `GRASP` no dio malos resultados, en nuestro caso no logró mejorar al *greedy* normal. En cuestiones de tiempo tampoco diríamos que es en tiempo real, pero podría ser útil si se está dispuesto a esperar un poco más (quizás un día). Igualmente, como mencionaos también debería poder afinarse el código para deshacernos de cosas innecesarias como computar los *rcl_size* depósitos más cercanos cada vez que hacemos el *greedy*, y eso debería reducir considerablemente el tiempo.

Conluciones

A modo de breve resumen, planteamos dos heurísticas muy distintas para resolver el problema GAP de modo ingenuo, y dieron resultados aceptables en distintos casos (el *greedy* para instancias con menor variación de costos, y el `WorstFit` para mayor variación)

Luego aplicamos dos operadores de búsqueda local, el relocate y el swap. El primero no dio resultados cuando lo aplicamos sobre soluciones encontradas con nuestra heurística *greedy*, pero era de esperarse porque sino esa solución debería haberse elegido primero, por como funciona la heurística. Tampoco sirvió para algunos casos

del `WorstFit`, pero para otros fue muy efectivo, otra vez esperable por el funcionamiento del `WorstFit`. El swap resultó el mejor operador para ambos, y por eso luego cuando realizamos el VND con el `GRASP` elegimos hacerlo en orden swap-relocate.

Finalmente, los resultados obtenidos para algunas instancias con GRASP fueron bastante buenos, dando mejores resultados que el *greedy* y que luego de aplicar el operador swap (que ya establecimos era el único útil para el *greedy*), pero no sirvió en nuestros casos analizados para la instancia real, y además demoró mucho más. Habría que analizar cuánto tiempo se está dispuesto a esperar y cuánto mejor puede ser el resultado obtenido para decidir si vale la pena utilizar el `GRASP`. En el contexto de ThunderPack, si el randomizado ayudara y diera una mejor solución, podemos llegar a ver la metaheurística como una opción viable pero no en tiempo real.

Apéndice

Tabla 1 - distancia total con el greedy, y luego de aplicar relocate y swap + mejora relativa del swap
Excluimos la mejora relativa del *greedy* porque en todos casos es 0.

	dist_greedy	dist_relocate	dist_swap	mejora_rel_swap
a-05:100	1709	1709	1700	0.53
a-05:200	3259	3259	3238	0.64
a-10:100	1390	1390	1364	1.87
a-10:200	2670	2670	2637	1.24
a-20:100	1170	1170	1168	0.17
a-20:200	2371	2371	2353	0.76
b-05:100	4972	4972	4174	16.05
b-05:200	8050	8050	6896	14.34
b-10:100	2115	2115	1912	9.60
b-10:200	8255	8255	7333	11.17
b-20:100	1660	1660	1506	9.28
b-20:200	3628	3628	3256	10.25
e-05:200	217701	217701	124213	42.94
e-15:900	1675159	1675159	1565485	6.55
e-10:400	633169	633169	542750	14.28
e-60:900	1879708	1879708	1858939	1.10
e-40:1600	3550531	3550531	3513624	1.04
real	788	788	768	2.54

Tabla 2 - distancia total con el worst fit, y luego de aplicar relocate y swap + mejora relativa del swap y el relocate

	dist_wf	dist_relocate	mejora_rel_relocate	dist_swap	mejora_rel_swap
a-05:100	4046	1710	57.74	2001	50.54
a-05:200	8105	3238	60.05	3403	58.01
a-10:100	4430	1362	69.26	1459	67.07
a-10:200	8870	2632	70.33	2866	67.69
a-20:100	4486	1176	73.79	1400	68.79
a-20:200	9001	2351	73.88	2541	71.77
b-05:100	6648	6648	0.00	4491	32.45
b-05:200	10840	10840	0.00	6106	43.67

	dist_wf	dist_relocate	mejora_rel_relocate	dist_swap	mejora_rel_swap
b-10:100	5167	5167	0.00	2332	54.87
b-10:200	11643	11643	0.00	5844	49.81
b-20:100	4965	4965	0.00	1675	66.26
b-20:200	11500	11500	0.00	4976	56.73
e-05:200	79684	40546	49.12	78660	1.29
e-15:900	448140	311797	30.42	443649	1.00
e-10:400	177231	93518	47.23	174715	1.42
e-60:900	652047	494058	24.23	646155	0.90
e-40:1600	1040451	815344	21.64	1034953	0.53
real	15627	922	94.10	1633	89.55

Tabla 3 - tiempo de ejecución del greedy, y luego de aplicarle relocate y swap

	time_greedy	time_relocate	time_swap
a-05:100	23	15	1774
a-05:200	26	20	12640
a-10:100	22	21	2555
a-10:200	39	37	16447
a-20:100	33	60	1357
a-20:200	58	74	9951
b-05:100	16	7	16737
b-05:200	28	14	94949
b-10:100	24	15	8627
b-10:200	42	27	117083
b-20:100	36	25	9454
b-20:200	64	56	77244
e-05:200	59	43	99578
e-15:900	207	168	8289920
e-10:400	77	49	704358
e-60:900	854	553	9420320
e-40:1600	905	794	47605800
real	3078	11048	1297760

Tabla 4 - tiempo de ejecución del worst fit, y luego de aplicarle relocate y swap

	time_wf	time_relocate	time_swap
a-05:100	163	3074	24461
a-05:200	418	9884	212548
a-10:100	271	4853	33258
a-10:200	470	14576	238257
a-20:100	180	7619	40471
a-20:200	489	27670	315592
b-05:100	146	7	23133
b-05:200	446	15	202234
b-10:100	174	13	36082
b-10:200	491	64	243103
b-20:100	172	24	40134
b-20:200	510	47	290572
e-05:200	512	2091	4345

	time_wf	time_relocate	time_swap
e-15:900	10292	72083	533930
e-10:400	1893	12943	77456
e-60:900	10034	304088	2794870
e-40:1600	34071	471282	11324600
real	17626	13475900	57784300