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 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, cumpliendo 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 una 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 a cada deposito 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, 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 para lograr computar una solucion en un tiempo polinomial.

Descripción del modelo propuesto

Heurísticas

A continuación vamos a describir las heurísticas seleccionadas:

Inicialmente, planteamos una implementación *greedy* en la que le asignamos a cada vendedor a un depósito que tenga suficiente capacidad remanente para la cantidad que demanda de dicho vendedor. 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 `3 x dmax`(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.

Busquedas Locales

Para de 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 $s\acute{}$ en el operador swap es una mejora de s si :

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

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 deposito 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 $s\acute{}$ en el operador relocate es una mejora de s si :

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

Metaheuristica

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 cercanos por vendedor. 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 el costo de ejecución puede reducirse significativamente. Por un lado si cuando realizamos búsquedas locales no creamos una nueva asignación `temp`, y por otro lado 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). De todas formas queremos enfatizar que los resultados varían mucho con el poder de cómputo de las computadoras: el código fue corrido en una laptop de 8GB de RAM y en otra de 32GB, y la diferencia observada fue significativa.
- 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

Los experimentación se realizó en base a todas las instancias de tipo *a* *b* y la instancia real, y 5 instancias representativas de la instancia *e*. Por motivos de tiempo, el análisis del `GRASP` solo se realizó sobre una instancia representativa de cada tipo, excluyendo la real (no llegó a computarse).

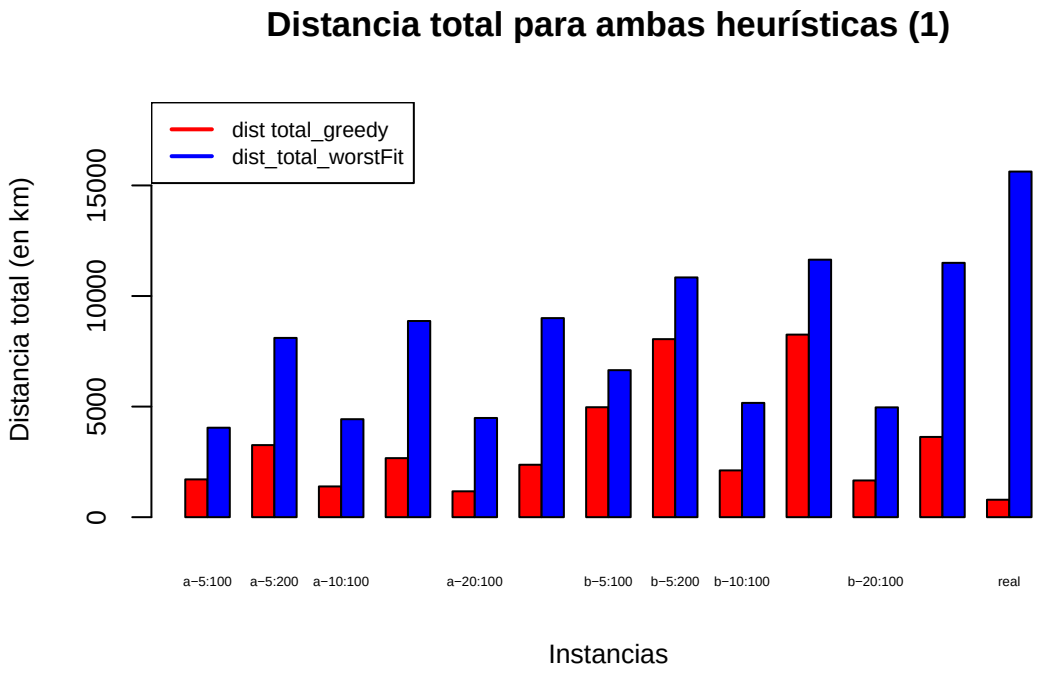
No llegamos a computar mejoras relativas ni absolutas, pero creemos que los gráficos (o en su defecto las tablas) son lo suficientemente visuales para decidir que método logra mejores resultados.

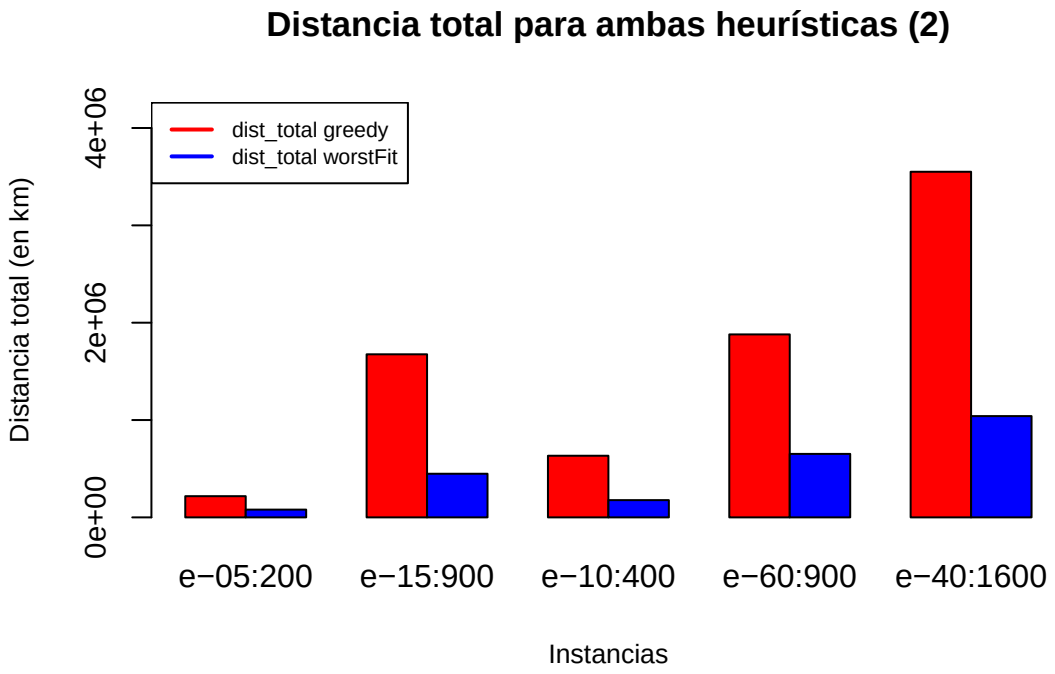
Diferencia en la distancia total computada por las heurísticas

Como podría intuirse, la heurística *greedy* es considerablemente mejor, ya que busca elegir a cada paso el déposito 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 ocasiano que si elegimos siempre los más cercanos, la penalización sobre los más lejanos sea mucho mayor (3000 contra 30, en el caso comentado).(2)

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 este dato quedará para un trabajo futuro.


Distancia total para ambas heurísticas (1)
dist total_greedy
dist_total_worstFit
Distancia total (en km)
0
5000
10000
15000
a−5:100
a−5:200
a−10:100
a−20:100
b−5:100
b−5:200
b−10:100
b−20:100
real
Instancias


Distancia total para ambas heurísticas (2)
dist_total greedy
dist_total worstFit
Distancia total (en km)
0e+00
2e+06
4e+06
e−05:200
e−15:900
e−10:400
e−60:900
e−40:1600
Instancias

Diferencia en la distancia total mejorada por las búsquedas locales

En general, no encontramos una mejora significativa para los operadores de búsqueda locales aplicados a una sola solución. En orden del listado las mejoras relativas para las instancias evaluadas son del 0.64%,. 1.6%, 0.3% y 0.3% para el relocate, y 0.35%, 0.67%, 0.38% y 0.3% para el swap aplicados sobre el *greedy*. Observamos valores similares para el *worst ft*, aunque el swap parece ser más útil, probablemente porque el relocate se dificulta dado que buscamos llenar los depósitos a su capacidad máxima (es decir, es más fácil quitar y agregar vendedores, que sólo agregar)

La primera tabla es del greedy y la segunda del worst fit

	dist_relocate	dist_swap
a-05:100	1702	1703
b-10:200	8115	8199
e-10:400	630179	630736
real	785	785

	dist_relocate	dist_swap
a-05:100	4006	3970
b-10:200	11515	11563
e-10:400	176237	175264
real	15604	15582

En cuestiones de tiempo, a pesar de que el *worst fit* resulta levemente peor en la teoría, no hay mucha diferencia para cuánto tardan sus operadores. Ambos crecen considerablemente a medida crece el tamaño y las restricciones de las instancias (sabemos que las *e* son más difíciles de asignar por motivos explicados con anterioridad), pero además el swap tarda más que el relocate, lo cual es lógico porque tiene que considerar toda combinación de vendedores nxn en vez de combinación vendedores-depósitos nxm ($m \leq n$).

	time_greedy	time_relocate	time_swap
a-05:100	37	3194	46799
b-10:200	56	17455	321713
e-10:400	143	53506	1272580
real	2688	42062400	153790000

	time_wf	time_relocate	time_swap
a-05:100	134	2438	43651
b-10:200	505	17506	326461
e-10:400	1888	45855	1468990
real	16629	44151500	153539000

Diferencia en la distancia total greedy vs GRASP

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

Observamos que en principio para instancias más pequeñas no hay mucha diferencia (una mejora relativa del 0.9%), pero ya en instancias medianas el GRASP cobra una utilidad mucho mayor, encontrando una mejora

relativa del 67.7%. Es cierto que esto es a cambio de una pérdida de eficiencia en tiempo, pero con nuestra computadora de 32 GB de RAM estuvo en el orden de los segundos y con la otra en las decenas de segundo (no es relevante a efectos prácticos).

Ya con las instancias *e* podemos notar una mejora de 97.9%, y otra vez la diferencia en tiempo no es demasiado significativa en la práctica. De cualquier manera, aunque este tiempo fuera peor consideramos que es un precio justo a pagar para *ThunderPack* si esta asignación va a servirles para al menos un trimestre.

	Greedy	GRASP
a-05:100	1709	1693
b-10:200	8255	2661
e-10:400	633169	13073

	time_greedy	time_grasp
a-05:100	37	21687600
b-10:200	56	200198000
e-10:400	143	216053000

Análisis del caso real

Por motivos de tiempo no pudimos computar la mejora del GRASP para el caso real, pero por lo charlado en clase no creemos que hubiera pudido mejorar demasiado (el óptimo rondaba los 700 y obtuvimos 788 con el *greedy* y 785 luego de las búsquedas locales (el *worst fit* fue significativamente peor). De cualqueir manera, buscamos responder brevemente las siguientes preguntas:

- ¿Es suficiente la capacidad de la red de depósitos, o es necesario expandir la misma para poder dar respuesta a todos los vendedores? No creemos necesario, ya que observamos que todos los vendedores fuero asignados a los depósitos originales (es decir, no al agregado de penalización) en el *greedy*
- ¿Es posible encontrar una asignación donde los vendedores recorran una distancia razonable para entregar sus paquetes? Creemos que sí, y que de hecho 788 o 785 km son distancias razonables, ya que en promedio es 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 consideramos que las soluciones que obtuvimos son buenas entonces sí, ya que el greedy lleva pocos milisegundos de ejecutarse con el tamaño provisto. Si quisieramos implementar el GRASP, sin embargo, por lo que pudimos llegar a computar no la consideramos una herramienta viable para cómputos de timepo real.

Conclusiones

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 locales que por sí mismos no dan muchos resultados, pero que cuando combinados en un VND con la randomización en el GRASP dieron resultados sorprendentes, logrando mejoras relativas de casi el 100% en el mejor de nuestros casos analizados. Sin embargo, hay que considerar que el tiempo de cómputo fue considerable para nuestras laptops de poca RAM, y por ende puede no ser una opción viable en tiempo real, sino algo a aplicar por año o incluso trimestre en una empresa. Este análiss es válido también para la instancia de caso real y el problema de ThunderPack.