Trabajo Práctico 2

Logística centralizada de primera milla

Introducción

En el siguiente trabajo buscamos evaluar distintas estrategias para optimizar las operaciones de una red de depósitos. Se nos contrató para asistir a la start-up ThunderPack por lo que tendremos que encontrar buenas soluciones para su problema de logística.

Implementación

Para la implementación del trabajo práctico se utilizaron tres estructuras de datos (tres clases):

- ReadInstance: Estructura que se encarga de obtener y almacenar los datos de los archivos.
- Solución: Estructura que se encarga de almacenar una solución para el problema de GAP. Sus métodos más importantes son el assign() y el reassign():
    - Assign se encarga principalmente de asignar a un vendedor a un determinado depósito, en esta misma se calculan todos los aspectos relacionados con esta asignación (Objective_value, Capacidades Restantes del depósito utilizado, cantidad de vendedores asignados). Como complemento, esta función se encarga de que en el caso de que como parámetro se pase un vendedor que ya está asignado, también se calculen primeramente los aspectos relacionados a la desasignación para luego reasignar.
    - Reasign hace esencialmente lo mismo que Assign salvo que se emplea en las búsquedas locales para corregir las penalizaciones cuando se interfiere con vendedores no asignados.

- MetaHeuristica: Estructura que se encarga de procesar la instancia para generar una solución. Principalmente contiene a las heurísticas, los operadores de búsqueda local y la metaheurística.

Para compilar los archivos y correr el main se debe ejecutar en la terminal:

```
make
.\tp_2
```

Heurística 0 - Los vendedores eligen

Se conoce que el modelo empleado actualmente por la empresa consta de que los vendedores elijan a qué depósito llevar sus paquetes. Decidimos simular esta estrategia en la Heurística 0 para poder comparar posteriormente si nuestras alternativas serán mejores o no.

Funcionamiento del algoritmo

Cada vendedor “en orden de llegada” (del primero al último *n*) irá seleccionando el depósito que le quede más cerca, fijándose si tiene capacidad disponible para abastecer su demanda. De tenerla se realizará la asignación y si no buscará el próximo más cercano que

logre cubrir su demanda. Si este no logra encontrar lugar en ningún depósito pasa a impactar en la penalización. ¿Cómo funciona la penalización? La suma de todos los vendedores que no logren asignarse será multiplicada por 3 y por el costo máximo que se halle en la instancia. De esta forma el costo se incrementará haciendo pagar por una mala asignación de vendedores a depósitos.

Complejidad

La implementación de la heurística se conforma básicamente de dos ciclos anidados: el primer ciclo itera exactamente *n* veces siendo *n* la cantidad de vendedores, mientras que segundo ciclo, anidado dentro del primero, itera *m* veces siendo *m* la cantidad de depósitos. Todo el código restante son simples evaluaciones booleanas, definiciones de parámetros y/o accesos a posiciones de listas los cuales son constantes. Por lo tanto, la complejidad algorítmica de la heurística será O(n*m).

Heurística 1 - Los depósitos eligen

Para nuestra primera heurística pensamos en revertir los roles de la estrategia actual. En esta oportunidad cada depósito elegirá a sus vendedores por cercanía hasta saturar la capacidad (o no poder cubrir alguna de las demandas restantes). Así cada depósito se irá llenando uno por vez hasta que todos los vendedores tengan un depósito asignado.

El objetivo de esta heurística es evaluar si se pueden minimizar los costos si dejamos que cada depósito elija a su clientela al estilo FCFS donde “el primero en llegar” es el primero en elegir qué demandas va a cubrir.

- Pros: los depósitos podrán saber de antemano con qué demanda estarán trabajando.
- Cons: los últimos depósitos probablemente tengan asignaciones menos eficientes al tener menos vendedores para elegir.

Funcionamiento del algoritmo

Cada depósito “en orden” (del primero al último *m*) irá encontrando a sus vendedores “más cercanos”, es decir, los que tengan menor costo. ¿Cómo lo hace? Va probando vendedor por vendedor hasta encontrar el que minimice el costo. Al hallar uno realiza la asignación y repite este procedimiento hasta que haya ocupado toda su capacidad o que su capacidad restante no pueda abastecer la demanda de ninguno de los vendedores disponibles. De esta forma cada depósito al llegar su turno tomará los vendedores más cercanos que tenga disponible. De quedar vendedores sin asignar se replica el sistema de penalización propuesta por las consignas (y explicado en Heurística 0).

Complejidad

De forma casi igual que la Heurística 0, la implementación de esta heurística se conforma básicamente de dos ciclos anidados: el primer ciclo itera exactamente *m* veces siendo m la

cantidad de depósitos, mientras que segundo ciclo, anidado dentro del primero, itera *n* veces siendo *n* la cantidad de vendedores. Todo el código restante son simples evaluaciones booleanas, definiciones de parámetros y/o accesos a posiciones de listas los cuales son constantes. Por lo tanto, la complejidad algorítmica de la heurística será O(m*n)

Heurística 2 - Capitanes de equipo

Para nuestra segunda heurística decidimos compensar el problema de la anterior y esta vez los depósitos irán eligiendo un vendedor cada uno. Creemos que de esta forma se realizarán asignaciones más justas al ir turnándose como lo hacen los capitanes de equipo al ir eligiendo sus integrantes.

El objetivo de esta heurística es evaluar si con esta forma más justa para asignar los vendedores a los depósitos se puede lograr minimizar los costos aún más.

- Pros: asignaciones más justas.
- Cons: tarda más en ejecutarse.

Funcionamiento del algoritmo

Para empezar establecemos un ciclo que itera tantas veces como vendedores haya en la instancia para asegurarnos que todos los vendedores tengan la chance de fijarse si pueden ser asignados a algún depósito. Dentro de este primer ciclo, tenemos otro que nuevamente itera sobre los vendedores, pero esta vez para intentar buscar el vendedor más cercano al depósito considerado actualmente. Los depósitos irán realizando una asignación cada uno y esta rotación fue implementada de la siguiente manera:

en *i* almacenamos al depósito actual y cada vez que el segundo ciclo termina (se haya encontrado o no un vendedor para el depósito actual) el *i* se incrementa para pasar al siguiente depósito. Cuando *i* llega a ser igual a la cantidad de depósitos es porque todos fueron considerados y entonces *i* debe reiniciarse a 0 para comenzar nuevamente la rotación.

Por último, si al terminar nos quedaron vendedores sin asignar se aplica la penalización.

Complejidad

Al igual que las anteriores heurísticas la implementación de la heurística se conforma básicamente de dos ciclos anidados, pero esta vez ambos ciclos iteran exactamente *n* veces siendo *n* la cantidad de vendedores. Todo el código restante son simplemente evaluaciones booleanas, definiciones de parámetros y/o accesos a posiciones de una lista los cuales son constantes. Por lo tanto, la complejidad algorítmica de la heurística será $O(n^2)$

Operador de Búsqueda Local 1 - Swap

Como primer operador elegimos Swap porque consideramos que como para nuestras dos heurísticas quienes eligen son los depósitos, Swap permitiría que dos vendedores intercambien destinos ahorrando distancias recorridas.

Funcionamiento del algoritmo

Para comenzar se toma la solución guardada en el objeto solución de la clase metaheurística. Esta solución proviene de haber aplicado anteriormente una heurística a la instancia siendo considerada.
Luego, hasta que no haya más mejoras posibles, es decir, hasta que no se logre disminuir más el costo, se realizarán intercambios entre dos vendedores chequeando que las nuevas asignaciones sean factibles y que provean una disminución en el costo.
En el caso de estar considerando un vendedor que previamente no había sido asignado se va a utilizar el método reassign() el cual se encarga particularmente de tener en cuenta estos casos de la siguiente manera:

En el caso de que ambas personas estén designadas simplemente se saltea esta iteración ya que que swapearlos no provocaría ningún cambio.
Si alguna de las personas está desasignada el método reassign se encarga de desasignar a la persona asignada y asigna a la que no está asignada. Se encarga de aumentar la penalización y modificar Objective_value, Capacidades Restantes del depósito utilizado, cantidad de vendedores asignados.

Complejidad

Tenemos un ciclo principal que itera hasta que no se logre mejorar la solución, cuya complejidad depende de la cantidad de iteraciones que se realicen. Dentro de esto tenemos dos ciclos anidados que recorren los vendedores para ver los swaps de todos contra todos. Por cómo está implementado (evitamos que un par de vendedores ya evaluados no se vuelva a considerar al ir iniciando los índices de forma tal que j=i+1) tenemos una complejidad O((n*(n-1))/2). Por lo tanto, siendo k la cantidad de iteraciones hasta que no haya mejora, tenemos una complejidad total de O(k*([n*(n-1)]/2)).

Operador de Búsqueda Local 2 - Relocate

Elegimos Relocate como segundo operador porque existe la posibilidad de que no hayan intercambios posibles (Swap no útil) pero que igual hubiese lugar para que un vendedor cambie de depósito.

Funcionamiento del algoritmo

Al igual que para swap, partimos de una solución que nos dejó una heurística. Iteramos sobre cada vendedor e intentamos encontrar el mejor cambio de depósito posible, es decir, que disminuya el costo. Esto se repite las veces que sea necesario hasta que no haya mejoras.
Este proceso contempla más fácilmente a los vendedores no asignados dado que sólo debe intentar ubicarlos en un depósito que tenga lugar.

Complejidad

Tenemos un ciclo principal que itera hasta que no se logre mejorar la solución, cuya complejidad depende de la cantidad de iteraciones que se realicen. Luego dentro de este tenemos dos ciclos anidados: el primer itera sobre la cantidad de vendedores (n), mientras que el segundo itera sobre la cantidad de depósitos (m). Por lo tanto, siendo k la cantidad de iteraciones hasta que no haya mejora, tenemos una complejidad total de O(k*n*m).

Metaheurística - Iterated Local Search

Dadas las heurísticas que tenemos implementadas y los operadores de búsqueda local propuestos elegimos ILS para la metaheurística. Esta nos permitirá realizar un ajuste de destinos parámetros que vamos a estar conversando.

Funcionamiento del algoritmo

Para la implementación elegimos seguir la estructura del pseudocódigo de clase:

IteratedLocalSearch(s_0, n_iters)

1. $s = s_0$, $s_{best} = s$
2. **for** $k = 1, \ldots, n\_iters$
3. $\quad s = \text{LocalSearch}(s)$
4. $\quad$ **if** $f(s) < f(s_{best})$
5. $\quad\quad s_{best} = s$
6. $\quad s = \text{GetNextSolution}(s, s_{best})$
7. **return** s_{best}

En nuestro caso ILS levanta la solución guardada en el objeto solución de la clase y toma por parámetros los siguientes 4 items: cantidad máxima de iteraciones a realizar, cantidad máxima de iteraciones para la búsqueda local, un bool que permite seleccionar el orden de los operadores para la búsqueda local y un porcentaje que nos servirá para determinar la cantidad de candidatos para perturbar la solución. Estos 4 parámetros deberan ser tuneados para ajustar nuestros resultados. Además la elección de la heurística para generar la solución inicial es otro de los parámetros que ajustaremos manualmente para encontrar la mejor estrategia.

- ***paso 3: LocalSearch(s)***
  Como búsqueda local decidimos implementar **Variable Neighborhood Descent (VND)** con el propósito de combinar la posibilidad de que vendedores puedan intercambiar depósitos entre ellos tanto como la posibilidad de que ya no hayan intercambios por realizar pero si capacidad restante para que vendedores sean reubicados. Es decir, utilizaremos los operadores Swap y Relocate alternadamente como vecindarios para ir refinando la solución.
  Nuestra implementación de VND sigue los mismos pasos que el pseudocódigo propuesto en clase:

VND(s, k_{max})

1. Definir $k = 1$
2. **do**
3. Explorar el k-ésimo vecindario
   $s' = \arg\min_{x \in N_k(s)} f(x)$
4. Determinar la nueva solución y el próximo vecindario a explorar
   $s, k = \text{NeighbourhoodChange}(s, s', k)$
5. **while** $k \neq k_{max}$
6. **return** s

NeighbourhoodChange(s, s', k)

1. **if** $f(s') < f(s)$ **then**
2. $s = s', k = 1$
3. **else**
4. $k = k + 1$
5. **return** s, k

donde *k* nos indica que operador de búsqueda local utilizar, reiniciandose si encuentra una mejora y aumentando para cambiar de vecindario si no la encuentra. El orden de la utilización de los operadores estará dado por el booleano pasado por parámetro donde 0 implica N1: Swap y N2: Relocate, y 1 implica el orden inverso. Además recordemos que pasamos por parámetro un máximo de iteraciones para acotar el tiempo que tarda.

- ***paso 6: GetNextSolution(s, s_{best})***
  En este paso perturbamos la solución buscando escapar de un óptimo local. Para esto implementamos el método **perturbacion** que levanta la solución que está siendo tratada y toma como parámetro el porcentaje que le habíamos pasado a ILS. Este porcentaje se lo aplicamos a la cantidad de vendedores de la instancia para luego generar una lista de ese tamaño, llena de vendedores elegidos al azar, que serán considerados para realizar swaps “destructivos” entre ellos siempre que sea factible. La idea es que estos swaps realicen intercambios que empeoren la solución para lograr escapar de óptimos locales. Su implementación es básicamente la presentada en swap() con la excepción de que no chequea que el intercambio presente una disminución en los costos, únicamente se fija que las capacidades de los depósitos considerados puedan abastecer las demandas intercambiadas.

Así ILS realizará mejoras y perturbaciones hasta que se alcance el número máximo de iteraciones ingresado por parámetro.

Complejidad

A la hora de ver la complejidad del código nos va a ser útil los parámetros de máxima cantidad de iteraciones, esto porque ILS realizará mejoras y perturbaciones exactamente la cantidad máxima pasada como parámetro. La complejidad de ILS pasa principalmente por la complejidad de VND y la de perturbación. VND tarda en el peor de los casos la cantidad máxima de iteraciones pasada como parámetro a esta función más la complejidad de hacer swap o relocate por cada iteración. Como la complejidad de relocate es mayor entonces la complejidad de VND será: O(max_iteraciones_vnd*(k*n*m)). La función perturbación consta de dos ciclos anidados que iteran hasta la cantidad que es generada por el porcentaje pasado por parámetro. Entonces la complejidad de esta funcion sera $O(n^2)$. Por lo tanto, la complejidad algorítmica de ILS será: O(cant_max_iter_ILS * (complejidadVND + complejidadPerturbacion))

Experimentación y discusión

Para realizar la experimentación construimos el archivo *experimentacion_gap.csv* donde almacenamos los resultados.

Tuning de parámetros

Tras correr reiteradas veces nuestra metaheurística llegamos a los siguientes ajustes:

- **Cantidad máxima de iteraciones a realizar:** observamos que el número adecuado es 15 dado que permite mejorar significativamente la solución sin tardar demasiado en las instancias más chicas y llegan a 2 iteraciones en las instancias más grandes del gap_e.
- **Elección de la heurística para computar la solución inicial:** en *experimentacion_gap.csv* vamos a testear cada heurística por separado para encontrar cual es la mejor estrategia.
- **Cantidad máxima de iteraciones a realizar en VND:** observamos que fue necesario diferenciar la cantidad de iteraciones para VND por el hecho de que se necesitaban unas cuantas más que en ILS para llegar a que los costos lleguen a un punto donde difícilmente sigan disminuyendo, mientras que siga corriendo en un tiempo razonable (menor a 10 minutos). Este número fue 300 para las instancias más chicas y esto fue decreciendo hasta llegar a 2 en las instancias más grandes de gap_e.
- **Orden de los operadores de búsqueda local:** encontramos que el orden que arroja mejores resultados en VND es (pasando un 1 por parámetro) N1: Relocate y N2: Swap, dado que tarda increíblemente menos y se lograron minimizar más los costos.
- **Porcentaje de perturbación:** como dijimos anteriormente este porcentaje se encargaba de ayudar a definir la cantidad de vendedores (aleatorios) a ser swapeados para perturbar un poco la solución. ¿Por qué usamos un porcentaje en vez de una cantidad máxima de iteraciones? Porque notamos que una cantidad fija no se ajustaba correctamente a las distintas instancias, por lo que consideramos que al tomar una porción de la cantidad de vendedores era algo más justo. Encontramos que el valor que genera un poco de ruido sin cambiarnos totalmente de dirección es 0.7 para instancias más chicas y 0.1 para instancias más grandes.

Ahora sí, pasamos a correr cada heurística para cada instancia. En nuestra tabla pintamos de verde al mejor resultado y en azul en caso de empate.

Input	H0	MetaH0	H1	MetaH1	H2	MetaH2
a05100	1719	1694	1932	1693	1712	1694
a05200	3259	3235	3800	3235	3289	3235
a10100	1370	1358	1732	1358	1416	1359
a10200	2657	2623	3290	2622	2698	2623
a20100	1177	1160	1589	1159	1275	1159
a20200	2369	2343	2804	2342	2458	2341
b05100	5669	1576	5336	1575	5152	1572
b05200	8703	3280	8927	3290	8794	3269
b10100	3461	1401	3445	1407	3150	1408

b10200	7551	2711	6331	2709	6416	2691
b20100	3157	1187	2781	1165	2850	1170
b20200	4945	2370	4724	2373	4995	2353
e05100	23799	4714	27003	4743	63516	4735
e05200	57228	10113	84420	10192	120105	10122
e10100	24261	3092	31405	3132	45344	3092
e10200	61811	6614	69502	6658	71516	6636
e10400	249218	13269	279669	13239	249207	13186
e15900	3791	2360	5895	2337	23951	2395
e20100	54165	4977	54624	4991	65519	4955
e20200	232167	9951	274241	10001	259264	9916
e20400	620956	25142	573739	25265	530662	24962
e30900	537376	19185	539815	19286	482144	18894
e40400	180792	7881	178702	7823	189913	7733
e60900	478813	15342	387076	15390	343636	14956
e201600	1.07E+06	39213	989652	39460	924564	38899
e401600	1.01E+06	30200	950492	30505	891993	30020
e801600	931184	24000	776391	24364	676425	24043

Con la ayuda de Excel creamos una segunda tabla para evaluar el porcentaje de lejanía al óptimo local al correr la metaheurística para cada una de las tres heurísticas. Este porcentaje se calculó con una regla de tres y restando 100 para hablar de lejanía:

((resultado_post-metaheuristica * 100) / optimo_local_instancia_actual) - 100

Primeras líneas de la tabla para dar una idea…

Optimo local	***%lejanía H0***	***%lejanía H1***	***%lejanía H2***
1693	*0.06*	*0*	*0.06*
3235	*0*	*0*	*0*
1358	*0*	*0*	*0.07*
2622	*0.04*	*0*	*0.04*

Y con los datos obtenidos calculamos el promedio de lejanía al óptimo local para los resultados de aplicar la metaheurística a cada heurística.

	%lejania H0	**%lejania H1**	**%lejania H2**
PROMEDIO	0.55	0.74	0.17

Observaciones

- Obtenemos mejores resultados post-Metaheurística cuando elegimos la Heurística 2. Creemos que esto se debe a que la construcción de la solución en H2 es más “aleatoria” que en H0 donde se van asignando uno por uno los vendedores y que en H1 donde van uno por uno los depósitos. Esta forma de asignar, al ir llenando las capacidades de los depósitos de forma medianamente pareja, deja más lugar a la MH para que realice los cambios que crea necesarios sin estancarse porque las capacidades de algunos pocos depósitos estén saturadas.

- Notamos que para las instancias en las que H2 no gana, solo pierde por poco y esto podría estar deberse a la aleatoriedad de las perturbaciones de ILS. Creemos que podría haber otros parámetros que hacen que H2 sea mejor para esas instancias, pero preferimos mantenerlos de la forma más general posible.

- Observando el promedio de lejanía al óptimo local vemos que no solo utilizar H2 da mejores resultados en promedio, si no que H0 y H1 son en comparación mucho peores.

Conclusión

Para las instancias de GAP queda demostrado por la experimentación que la mejor estrategia es utilizar la **Heurística 2 - Capitanes de equipo** al ser mejor en promedio y en la mayoría de las instancias**.**

Análisis del caso real

En esta segunda etapa de experimentación construimos el archivo *experimentacion_real.csv* donde además de los resultados incluimos el tiempo de ejecución para tener un segundo parámetro de elección.

Tuning de parámetros

Se ajustaron los siguientes parámetros para obtener mejores resultados en la experimentación con la instancia real:

- **Cantidad máxima de iteraciones a realizar:** 15.
- **Cantidad máxima de iteraciones a realizar en VND:** 200.
- **Orden de los operadores de búsqueda local:** el orden se mantuvo en 1 (N1: Relocate, N2: Swap)
- **Porcentaje de perturbación:** debido a la gran cantidad de vendedores en la instancia disminuimos el porcentaje a 1% para evitar tiempos de ejecución excesivamente extensos y que no mejoraban significativamente nuestros resultados.

Pintado en verde se encuentra el mejor resultado y en naranja el menor tiempo.

Input	H0	MetaH0	Tiempo 0	H1	MetaH1	Tiempo 1	H2	MetaH2	Tiempo 2
real_instance	788.2	676.9	719	1770.3	676.5	665	11242.6	677.9	708

Observaciones

- Los resultados obtenidos con las tres opciones de heurísticas son casi iguales, esto puede ser debido a que las perturbaciones de la metaheurística que nos permiten escapar de óptimos locales acercándonos cada vez más al óptimo global.

- El menor tiempo de ejecución y de valor objetivo provienen ambos de utilizar la Heurística 1. Esto nos llamó un poco la atención dado que por la experimentación para GAP esperábamos que H2 fuera mejor. Aunque tampoco nos sorprendió demasiado dado que si los tiempos de ejecución no fueran tan extensos podríamos aumentar las iteraciones para H2 y probablemente encontraríamos que esta es mejor.

- Es importante notar que por las perturbaciones aleatorias es posible que no se obtenga siempre el mismo resultado para una misma instancia con mismos parámetros. Estando los tres resultados tan cerca es altamente probable que corriéndolo varias veces se nos presenten óptimos distintos y hasta perdamos el mejor. Los resultados presentados en la tabla fueron justamente los mejores obtenidos tras varias rondas de experimentación.

Conclusión

Con el análisis de la instancia real si bien la **Heurística 1 - Los depósitos eligen** dio el mejor resultado, creemos por la experimentación en GAP y por la cercanía entre los resultados que la **Heurística 2 - Capitanes de equipo** es la mejor estrategia a tomar y le recomendamos a ThunderPack que lo haga dado que casi siempre proporciona mejores resultados.