Valentín Bonás, Lara Rodrigues La Moglie y Juan Ignacio Vila Bach

TP2: Logística centralizada de primera milla

Introducción al problema

La problemática a abordar se trata de encontrar la manera óptima, o en su defecto efectiva, de realizar una asignación. Trabajaremos sobre un caso particular en el cual buscamos asignar un conjunto N de negocios a un conjunto M de depósitos. Cada negocio posee una distancia a cada depósito, denotada por c_{ij}, con $i \in M$ y $j \in N$. También posee una demanda, d_{ij} que se refiere a la cantidad de stock que el negocio pedirá de dicho depósito. Por último, cada depósito posee una capacidad, denotada por c_i. En el este caso particular, los negocios no pueden pedir stock de distintos depósitos, por lo cual serán asignados tan solo a uno de estos. Asimismo, la demanda de cada negocio será la misma para todo depósito, por lo cual la variable d_{ij} no depende de i y podría escribirse como d_j. Consideramos una solución óptima al problema de asignación a aquella que minimiza la suma de las distancias entre cada negocio y cada depósito asignados, teniendo en cuenta no exceder la capacidad de estos últimos.

El tipo de problema con el cual trataremos es conocido como *GAP* (Generalized Assignment Problem), y su versión de optimización se encuentra en la categoría NP-Hard. Esto quiere decir que todo problema perteneciente a la clase NP puede ser reducido polinomialmente a *GAP*. Además, un problema NP-Hard no necesariamente pertenece a NP, es decir que no necesariamente existe un algoritmo polinomial para verificar un certificado positivo de una instancia del problema. Si encontráramos un algoritmo polinomial para *GAP* entonces encontraríamos una solución polinomial para todo problema en NP y demostraríamos que P = NP. De todos modos, eso aún no ha sucedido y es por eso que se desarrollan heurísticas para resolver el problema *GAP,* buscando un balance entre la complejidad temporal y la calidad de la solución.

En el siguiente trabajo buscamos modelar distintas alternativas de heurísticas y búsquedas locales en pos de hallar una solución de cierta calidad para la problemática GAP, en tiempo polinomial. Comenzaremos diseñando heurísticas constructivas, luego operadores de búsqueda local y por último generaremos una metaheurística que los englobe.

Diseño de heurísticas constructivas

Heurística 1: depósito más cercano

En primer lugar, diseñamos e implementamos una heurística cuyo funcionamiento es el siguiente: por cada negocio buscamos el depósito más cercano, es decir aquel cuyo c_{ij} sea el menor, y asignamos el negocio a dicho depósito. Asimismo, vamos disminuyendo la capacidad c_j del depósito en función a la demanda d_{ij} de la asignación. En caso de que el depósito de menor distancia a un negocio no tenga capacidad restante suficiente, este sería asignado al próximo depósito más cercano.

Esta heurística es posiblemente la más intuitiva, y tiene ciertas ventajas como no necesitar actualizar una lista de disponibilidad, ya que los depósitos pueden tener tantos negocios asignados como su capacidad permita. En otros enfoques, es necesario llevar la cuenta de qué negocios ya fueron asignados, pero en este caso al iterar por negocios y en cada iteración asignarlos a un depósito, no tenemos dicho problema.

La complejidad de este algoritmo goloso es $O(nm)$ siendo n la cantidad de negocios y m la cantidad de depósitos. Como vemos, es polinomial. Claramente no encontraremos la solución óptima, pero encontraremos una solución que creemos que tendrá cierto grado de efectividad.

Heurística 2: negocio más cercano

En segundo lugar, desarrollamos una heurística que funciona de manera inversa a la primera, es decir que itera por cada depósito y le asigna aquellos negocios que se encuentren cerca, mientras la capacidad del depósito lo permita. Es importante asegurarse que no se evalúen aquellos negocios que ya han sido asignados, por lo cual debemos llevar la cuenta de estos. Además, es necesario encontrar un modo de definir qué quiere decir que un negocio está cerca de un depósito. Nuestro criterio fue el siguiente: por cada depósito, ordenamos los negocios en función a cuán lejos están de este, de menor a mayor. Así, iteramos por los depósitos asignándoles en orden los negocios mientras estos aún no estén asignados y mientras la capacidad del depósito sea mayor a la demanda del negocio. A grandes rasgos, asignamos a cada depósito los negocios más cercanos a él.

La complejidad de este algoritmo goloso es de $O(n^2m)$, polinomial pero no tan eficiente como la heurística anterior.

Analizando el funcionamiento de nuestra heurística, llegamos a la conclusión de que el orden en que evaluamos los depósitos alterará el resultado. Luego, una mejora posible sería definir un orden de manera que ayude a reducir aún más el valor de la función objetivo. El primer depósito tenido en cuenta tendrá prioridad sobre los otros y será probablemente el único que tenga todos los negocios más cercanos a él asignados. Imaginemos la situación en la cual existe un depósito considerablemente más alejado de los negocios que el resto. Si este depósito es considerado antes que el resto, se le asignarán múltiples negocios que a pesar de estar más cerca del depósito que otros negocios, claramente estarán más cerca de los otros depósitos que de aquel en cuestión. Así, estaríamos dando mucho peso a un depósito al cual deberíamos tratar de asignarle la menor cantidad de negocios posibles. Luego, podría ser útil ordenar los depósitos según distancia promedio a los negocios y comenzar por aquellos con menor promedio. Así, evitaríamos situaciones como la descripta anteriormente. Aquí partimos de la base de que la suma de las capacidades de los depósitos sea mayor a la suma de las demandas de los negocios, por lo cual preferimos que los depósitos “más vacíos” sean aquellos más lejanos. Si la situación fuera diferente entonces priorizar por cercanía los depósitos quizás no sea necesario.

Consideraciones

Se pueden diseñar heurísticas que tomen un enfoque relacionado a las demandas que tiene cada negocio en relación a cada depósito, buscando quizás asignar negocios a depósitos de los cuales ocupen menor capacidad si es que dichos depósitos son cercanos a otros negocios y buscamos asignarlos a la mayor cantidad posible de negocios. Pueden surgir muchos matices considerando las demandas y capacidades. De todos modos, a pesar de que en el problema general de GAP las demandas de los negocios pueden variar dependiendo del depósito, como en nuestra instancia real todo negocio demanda lo mismo de cualquier depósito, no profundizaremos en estas alternativas. A pesar de estar diseñando una heurística para GAP en general, nuestro objetivo final se refiere en específico a la instancia particular de *ThunderPack*.

Operadores de búsqueda local

¿Cómo representamos una solución?

Por una cuestión de comodidad al momento de implementar los distintos métodos, diseñamos dos maneras distintas de representar la solución. Por un lado, generamos un vector de vectores de enteros llamado asignaciones, donde por cada depósito guardamos los índices de los negocios asignados a él. Así, `asignaciones[1][3] = 45` quiere decir que el negocio 45 se encuentra asignado al segundo depósito. Por otro lado, generamos un vector de enteros llamado correspondencia en el cual cada índice representa un negocio y su valor representa el depósito al cual fue asignado. Así, `correspondencia[68] = 0` quiere decir que el negocio 67 (ya que incluimos al 0) fue asignado depósito 1. Actualizamos ambos paralelamente, de manera de que siempre representen la misma asignación.

Operador 1: relocate

El primer operador que definimos es un relocate. Este itera por los negocios evaluando si hay algún depósito con menor distancia al negocio que aquel al cual se encuentra asignado actualmente, y en caso afirmativo, si las capacidades lo permiten, asigna el negocio a su nuevo depósito. Definimos como vecindario a todas las soluciones alcanzables desde la solución actual haciendo un único relocate. Utilizamos como criterio de aceptación el *first improvement*, y por lo tanto apenas hallamos una solución mejor a la actual redefinimos nuestra solución como la nueva y continuamos explorando su nuevo vecindario.

Para aceptar una nueva solución debemos evaluar 2 condiciones, con i_1 el depósito anterior, i_2 el nuevo depósito y j el negocio:

- $c_{i_2 j} < c_{i_1 j}$
- $d_{i_2 j} \leq \bar{c}_{i_2}$, siendo $\bar{c}_{i_2}$ la capacidad restante del depósito i_2

Este operador no aportará una mejora con toda heurística posible como base para la solución inicial. Por ejemplo, si tomamos nuestra primera heurística, como cada negocio está asignado al depósito más cercano, nunca encontraremos otro depósito que sea más

preferible y tenga capacidad disponible, sin realizar antes alguna otra modificación sobre la solución como un swap, por ejemplo. En cambio, en nuestra segunda heurística sí ofrecería mejoras desde el inicio, ya que que cada depósito tenga asignado los negocios más cercanos no asegura que cada negocio este asignado a su depósito más cercano. Consecuentemente, en dicho caso el operador relocate sería útil.

El operador relocate nos permite asignar los negocios que no era posible asignar a ningún depósito dadas las capacidades y las asignaciones ya realizadas. Se sufría una gran penalización sobre el valor objetivo por cada negocio no asignado, por lo cual el hecho de poder incluirlos en la solución mejora considerablemente el valor objetivo. Por supuesto, no se podrán ubicar dichos negocios sin antes haberse realizados otros relocate o swap que generen suficiente espacio en algún depósito.

La complejidad del operador relocate es de $O(nm)$ siendo n la cantidad de negocios y m la cantidad de depósitos. Notemos que la búsqueda corre hasta haber analizado todas las posibles combinaciones de negocios y depósitos, realizando las alteraciones de la solución simultáneamente.

Operador 2: swap

El segundo operador que diseñamos es un swap. Este itera por los negocios, en busca de dos negocios que al intercambiar el depósito en el que se encuentran el valor objetivo disminuya, respetando las capacidades. Utilizamos de nuevo el criterio de *first improvement*. Este operador no servirá para añadir los negocios no asignados a la solución, ya que swapearlos simplemente añadiría un negocio para quitar otro.

Para aceptar una nueva solución debemos evaluar 2 condiciones, con i_1 el depósito A, i_2 el depósito B, j_1 el negocio A (asignado al depósito A) y j_2 el negocio B (asignado al depósito B):

- $c_{i_1 j_1} + c_{i_2 j_2} > c_{i_1 j_2} + c_{i_2 j_1}$
- $d_{i_1 j_2} \leq \bar{c}_{i_1} + d_{i_1 j_1}$
- $d_{i_2 j_1} \leq \bar{c}_{i_2} + d_{i_2 j_2}$

Con $\bar{c}_k$ la capacidad restante del depósito k.

Pareciera que este operador funcionará con ambas heurísticas diseñadas ya que siempre puede suceder que la suma de las nuevas distancias sea menor que la suma de las anteriores.

La complejidad del operador swap es de $O(n^2)$ siendo n la cantidad de negocios, ya que la búsqueda evalúa cada par de negocios posible en busca de una mejora al valor objetivo.

¿Qué podría hacer a una solución infactible? En primer lugar, que la suma de las demandas de los negocios asignados a cierto depósito supere su capacidad. En segundo lugar, que un negocio sea asignado a más de un depósito. Ambos operadores evalúan que las capacidades

de los depósitos no sean excedidas. Asimismo, no permiten que un negocio pertenezca a dos depósitos: el operador relocate quita al negocio del depósito anterior al relocalizarlo y el operador swap quita a ambos negocios de sus anteriores depósitos. Luego, ambos son correctos.

Metaheurística

Como metaheurística diseñamos una variación de VND (Variable Neighbourhood Descent) utilizando ambos vecindarios, swap y operate. Tiene ciertas diferencias con el concepto de VND, ya que itera dentro de cada operador hasta alcanzar un mínimo local.

Definimos a N_1 como el operador relocate y a N_2 como el operador swap. Comenzamos investigando el vecindario N_1, buscando una solución mejor y pasando al vecindario de dicha solución hasta llegar a un mínimo local. Luego, comenzamos a investigar el vecindario N_2 del mismo modo. Cuando llegamos a un mínimo local, retornamos a analizar el vecindario N_1. Alteramos entre ambos vecindarios de este modo hasta que ninguno ofrezca una mejora a la solución, es decir, nos encontramos en una solución que es un mínimo local para ambos. En dicho caso, retornamos la solución como el mínimo obtenido por la metaheurística.

¿Cuál es la calidad de la solución? Nada nos asegura que la solución hallada por nuestra metaheurística sea efectivamente la solución óptima. Es muy probable que no lo sea. De todos modos, a partir de experimentación que explicaremos a continuación, notamos que cuánto realmente mejora la solución inicial depende en un alto grado de la calidad de la heurística llevada a cabo y de cuanto *room for improvement* tenga.

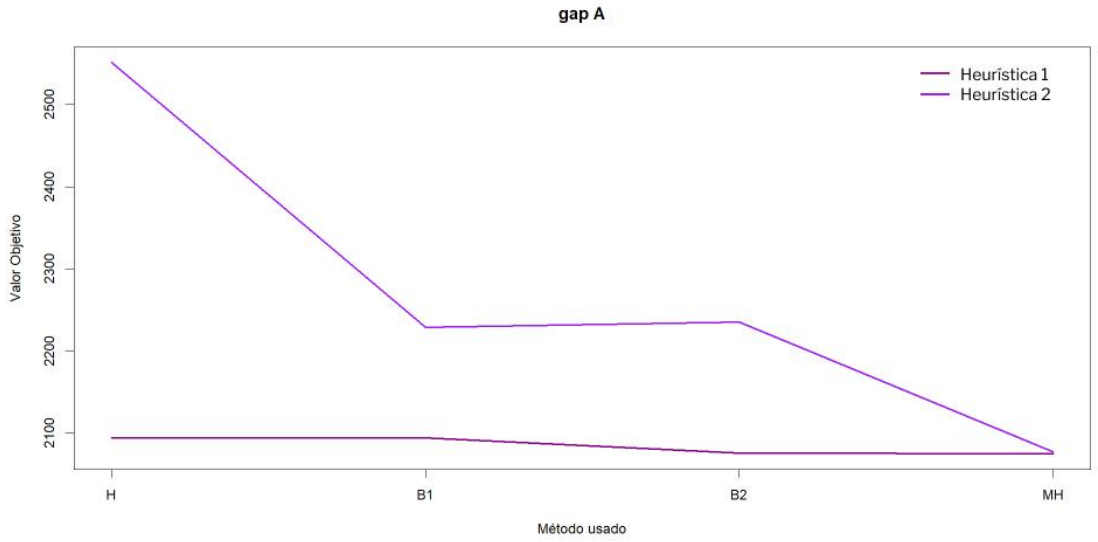
Idealmente, podríamos resolver una vez el problema con un algoritmo exacto, obteniendo el óptimo, y así podríamos determinar con mayor seguridad la calidad de nuestras soluciones aproximadas. Por otro lado, la solución inicial, y en mayor escala la heurística, genera un gran impacto en la metaheurística. Puede suceder que una heurística que genera una "peor" solución inicial nos dé un mayor espacio a mejoras. Puede suceder que nuestra heurística sea muy buena y por lo tanto no podamos mejorar. Las posibilidades son múltiples, y dependen de variados factores, como también de la naturaleza de la instancia que evaluamos.

La complejidad de la metaheurística depende de cuantas iteraciones se realicen de esta, lo cual no es posible definir de antemano. Los operadores relocate y swap que utiliza son de complejidad $O(nm)$ y $O(n^2)$ respectivamente por lo cual por supuesto la complejidad de la metaheurística es al menos de $O\big(max(nm, n^2)\big)$. La complejidad depende de la cantidad de iteraciones, es decir de la cantidad de soluciones mejores hacia las cuales nos movemos, por lo cual podríamos decir que depende de cuán cerca está nuestra solución inicial de un mínimo local para ambos operadores, lo cual depende de nuevo de la heurística utilizada para obtener dicha solución inicial.

Experimentación y discusión

Para evaluar el funcionamiento y la calidad de nuestras heurísticas, búsquedas locales y por último metaheurística, utilizamos tres conjuntos de bases de datos.

Los siguientes gráficos fueron generados con el promedio de los valores objetivo obtenidos por cada método aplicado a los diversos conjuntos de datos (de izquierda a derecha: heurística i, heurística i + búsqueda local 1, heurística i + búsqueda local 2, metaheurística i):



Este primer gráfico muestra como mejora el valor objetivo (la sumatoria de las distancias entre los negocios y el depósito al que fueron asignados) promedio para los distintos datasets del conjunto *gap_a*. Observamos que el valor objetivo de la solución aplicando la heurística 2 (la nombramos s_{h2}) es significativamente más alto que el de la heurística 1 (la nombramos s_{h1}) si las aplicamos por sí solas.

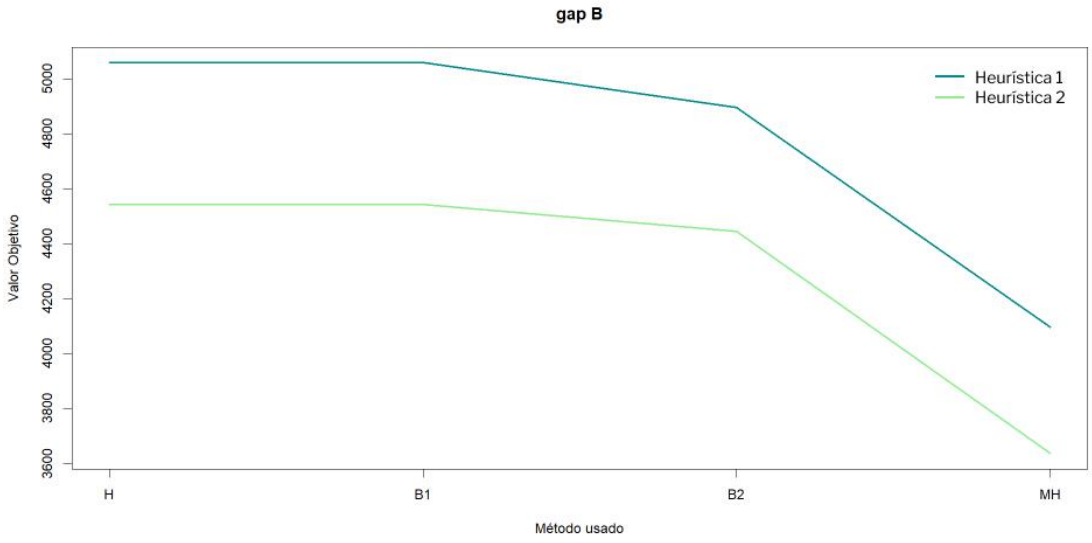
Al realizar la búsqueda local 1, si partimos de s_{h2} obtenemos una solución mucho menor (en promedio) mientras que si partimos de s_{h1} no vemos una mejora.
Esto se debe a que la heurística 2 se basa en asignar negocios a depositos, los cuales cuentan con una considerable capacidad y esto resulta en que un primer deposito puede recibir a varios negocios que en realidad estaban más cerca de otro depósito que examinaremos posteriormente, y a este le van a quedar disponibles negocios que tal vez se encuentran mucho más lejos, por lo que el valor objetivo se ve incrementado de sobremanera. Esto se solucionará con el operador *relocate* de la búsqueda local 1 que reasigna negocios a otros depósitos que tal vez fueron asignados golosamente al primer deposito que se examinó. La heurística 1 en cambio asigna depósitos a negocios, por lo que utilizar el operador *relocate* no debería a priori resultar en una solución mejor dado que todos los negocios ya se encuentran asignados al depósito que minimiza su distancia, dadas las capacidades y demandas.

En el caso de la búsqueda local 2, estamos utilizando el operador *swap* el cual podría encontrar una solución mejor intercambiando negocios entre depósitos y realizando

diferentes asignaciones, una vez obtenida la foto completa de la instancia final. Básicamente queremos remediar la visión tan a corto plazo que tiene una heurística. No parece haber gran diferencia entre el valor objetivo de la solución encontrada través del operador relocate y aquella a través del operador swap.

La metaheurística resulta naturalmente en la mejor solución que podremos obtener dado que combina todas las herramientas a nuestra disposición: podemos partir ya sea desde s_{h1} como desde s_{h2} y obtener una solución considerablemente superior dado que realizaremos la búsqueda 1 y 2 en sincronía, siempre mejorando siquiera levemente la solución. En el caso de la heurística 2, la metaheurística mejora significativamente el valor objetivo en comparación a las otras herramientas, mientras que en la heurística 1 no hay un gran cambio.

Como podemos observar, dadas las características de los conjuntos de datos pertenecientes al grupo *gap_a*, la heurística 1 funciona mejora que la heurística 2, pero ambas llegan prácticamente al mismo mínimo local. Una particularidad que notamos de este conjunto de datos es que las capacidades de los depósitos son muy amplias, por lo cual no quedan negocios sin asignar y asimismo son las mismas para todos los depósitos. Luego, las demandas y capacidades no generan impedimentos al momento de asignar los negocios al depósito más cercano, por lo cual creemos que funciona tan efectivamente la heurística 1.



Este segundo gráfico nos muestra dos progresiones similares entre s_{h1} y s_{h2} en el caso de los conjuntos de datos en *gap_b*. Ambas soluciones se ven afectadas de manera similar al aplicarles las diversas búsquedas y al realizar la metaheurística.

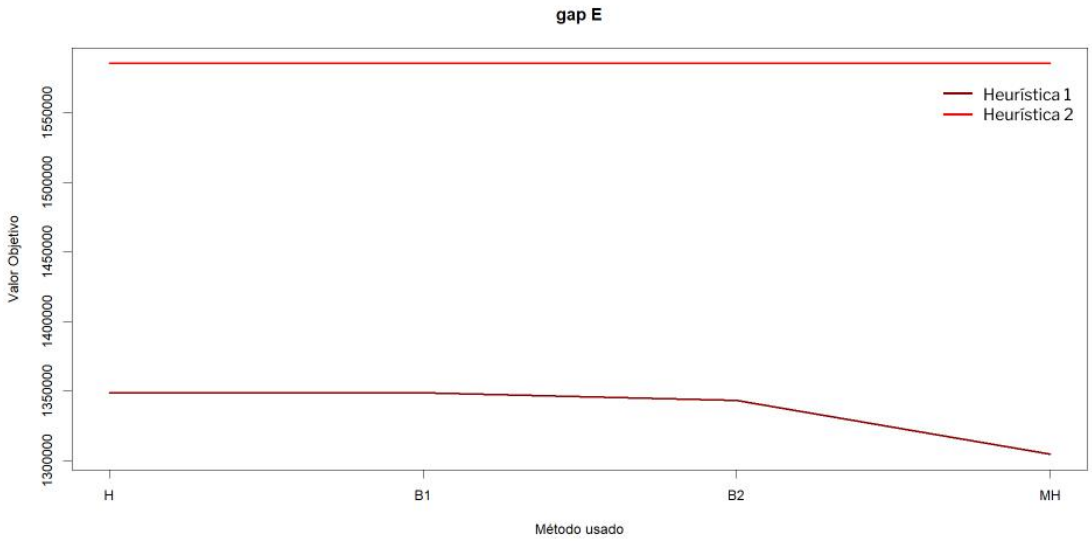
Los conjuntos de datos pertenecientes a *gap_b*, a diferencia de *gap_a* nos presentan depósitos con mucha menos capacidad, pero demandas similares, por lo que no nos resulta extraño observar una inversión en desempeño por parte de las heurísticas ni un incremento en el valor objetivo medio, dado que esperamos terminar con muchos más negocios sin asignar, lo que inflará el valor de nuestra solución. La capacidad de los depósitos con la heurística 1 se agotará significativamente más rápido que con la 2 puesto

que los primeros negocios se asignan golosamente a sus depósitos más cercanos, sin importar los que vienen detrás, lo que resulta en muchos negocios sin asignar.

En el caso de la heurística 2, las distancias se "cuidan" solas puesto que el depósito determina qué negocio recibe, por lo que si bien se agotan más rápido lo hacen más eficientemente. Se llena el depósito al máximo posible antes de avanzar al siguiente.

La búsqueda 1 resulta inefectiva puesto que no contamos con tanto espacio en los depósitos de sobra como para realizar muchos *relocate*. Con la búsqueda 2 en cambio, la cual realiza el operador *swap*, podremos asignar más eficientemente a los negocios que tenemos ya asignados. También sufre bastante de la poca capacidad de los depósitos.

La metaheurística, como de costumbre, nos proporciona un valor objetivo medio mucho menor dada la utilización sincrónica de las búsquedas, como en *gap_a*.



En este último gráfico vemos como todos los métodos son completamente inefectivos al buscar reducir s_{h2}. La media de los valores objetivo obtenidos sin importar el método se mantiene constante. Esto se debe a que el conjunto de datos *gap_e* nos presenta depósitos con capacidades distintas, y asimismo pequeñas en relación a las demandas por lo cual de nuevo tendremos una gran cantidad de negocios sin asignar, generando un gran impacto en el valor objetivo.

Las búsquedas locales no proporcionan mejoras significativas para s_{h1}, si bien la búsqueda 2 mejora la solución levemente.

La metaheurística claramente resulta en una solución mejor dado que podremos realizar *relocate* y *swap*, tomando a los que quedaron sin asignar y óptimamente asignándolos, reduciendo así nuestro excedente por no asignación.

Podemos concluir que qué heurística, búsqueda local o combinación de ambas es mejor depende plenamente de las características de la instancia, tales como la capacidad de los depósitos en relación a la demanda y otras particularidades que pueden darse. Por lo tanto,

es primordial pensar qué características definen a nuestra instancia al momento de abordar la solución de este. Nada es absoluto.

Análisis del caso real

Nuestro caso real se trata de la problemática de la empresa *ThunderPack*, la cual posee 1100 negocios, 310 depósitos y busca encontrar una asignación que sea lo más efectiva posible. Asimismo, tiene la particularidad de que la demanda de los negocios es la misma para cada depósito, es decir que el parámetro d_{ij} no depende del depósito i y por lo tanto se puede escribir como d_j.

A continuación, mostramos en una tabla los valores objetivo y tiempos de ejecución obtenidos a través de distintas combinaciones de las herramientas desarrolladas en este proyecto:

	H1	**H2**	**H1+BL1**	**H1+BL2**	**H2+BL1**	**H2+BL2**	**MH1**	**MH2**
V.O.	788.2	1781.1	788.2	740.7	1354.4	1361.9	733.6	722.4
T.E.	6 ms	1589 ms	16 ms	144 ms	1579 ms	1730 ms	1033 ms	2810 ms

Referencias:

H1 = Heurística 1 (negocio más cercano)

H2 = Heurística 2 (depósito más cercano)

BL1 = Búsqueda local 1 (relocate)

BL2 = Búsqueda local 2 (swap)

MH1 = Metaheurística con solución inicial obtenida a través de la heurística 1

MH2 = Metaheurística con solución inicial obtenida a través de la heurística 2

V.O. = valor objetivo

T.E. = tiempo de ejecución en milisegundos

Observando la tabla podemos ver que en el caso real la heurística 1 es ampliamente más efectiva que la heurística 2, siendo 2.26 veces más grande el valor objetivo de esta última. Recordemos que nuestro objetivo es la minimización. El operador relocate no genera ningún cambio sobre la solución de la heurística 1, como fue explicado previamente, y mejora en un 24% a la solución de la heurística 2. Por otro lado, el operador swap genera una mejora de 6% sobre la heurística 1 y una mejora de 23.5% sobre la heurística 2. Por último, la metaheurística devuelve una solución un 7% mejor que la heurística 1 y un 60% mejor que la heurística 2.

Podemos observar que la solución de la heurística 2 tiene mucho espacio para ser mejorada, mientras que la heurística 1 varía en un grado muy pequeño. Esto claramente no quiere decir que la heurística 2 sea mejor, ya que la solución ofrecida por esta tiene un valor objetivo significativamente peor que la solución de la heurística 1.

Una observación curiosa es que la metaheurística aplicada partiendo de una solución inicial generada con la heurística 2 obtiene un valor objetivo menor que cuando es aplicada sobre la heurística 1, a pesar de que la solución inicial de esta última sea ampliamente menor. Como hemos mencionado antes, existen tantos factores en juego al utilizar estas herramientas que las posibilidades varían ampliamente y no siempre encontraremos una regla fija que establezca la superioridad de un método frente a otros.

En relación al tiempo de ejecución, la primera heurística es significativamente más rápida que la segunda, y consecuentemente su combinación con ambas búsquedas locales es más rápida que la de la heurística 2. Esto tiene sentido dadas sus complejidades previamente calculadas. En relación a la metaheurística, aplicarla sobre una solución inicial proveniente de la heurística 1 tiene un tiempo de ejecución aproximadamente 3 veces menor a aplicarla partiendo de la heurística 2, pero no olvidemos que el valor objetivo de esta última es mejor que el de la anterior.

Podemos concluir entonces que en el caso particular de *ThunderPack*, pareciera ser conveniente resolver la problemática utilizando la metaheurística sobre una solución inicial obtenida a partir de la heurística 2. El tiempo de ejecución es muy bajo, por lo cual podemos sacrificar la velocidad en pos de encontrar una mejor solución. Como fue mencionado antes, a pesar de parecer contraintuitivo dado que la heurística 2 arroja una solución precaria, la metaheurística logra realizar muchas mejoras así superando aún la solución de gran calidad generada por la primera heurística. Como vemos, todo caso es diferente y no es posible generalizar.

Analizando nuestra solución final de ambas metaheurísticas, en ningún caso quedan negocios sin asignar a un depósito, por lo cual podemos asegurar a *ThunderPack* que la capacidad de la red de depósitos es suficiente.

La distancia total mínima que alcanzamos es de 722.4, por lo cual dado que hay 1100 negocios la distancia promedio de cada uno es de 0.65. Este valor es bajo en comparación a las otras distancias del conjunto de datos, por lo cual diríamos que es definitivamente razonable para los negocios recorrer las distancias asignadas.

Es posible que según el escenario que tomemos como instancia los resultados no siempre tengan buena calidad. Asimismo, no es certero qué método arrojará una mejor solución. Luego, es importante evaluar las herramientas en cada caso particular.

Conclusiones

A partir del desarrollo de este proyecto llegamos a la conclusión de que, para un problema tan amplio y variado como GAP, es difícil definir una jerarquía de heurísticas y operadores. Entran muchos factores en juego y a veces es necesario decidir cual priorizar. Por ejemplo,

en nuestro caso priorizamos minimizar la distancia sin prestar atención a las demandas ya que en el caso de ThunderPack estas no variaban según depósito. Si el caso fuera distinto, podría ser conveniente evaluar las demandas a modo de ubicar los negocios en el depósito del cual menos demanden, así minimizamos el espacio ocupado y permitimos potencialmente que más negocios sean asignados a un depósito cercano. Por otro lado, dado que la penalización no depende de la distancia del negocio no asignado, es conveniente que, si es necesario que algún negocio quede fuera de la solución, sea aquel que posee mayor distancia con los depósitos, a modo de que no agregue más peso al valor objetivo. Como vemos, las decisiones dependen del caso particular con el cual trabajaremos, por lo que no podemos asegurar la calidad de nuestras herramientas para todo problema del tipo GAP, pero podemos asegurarle una buena y rápida solución a la empresa ThunderPack.

Anexo instrucciones de compilación y ejecución

Para realizar este trabajo determinamos que lo más adecuado sería establecer una clase con todos los atributos que nos serían útiles durante la realización de este proyecto. En el archivo `instance.h` se encuentra la definición de esta como `AssignmentInstance`, que utilizaremos para inicializar una instancia dado un conjunto de datos. Los atributos y funciones de la clase `AssignmentInstance` son:

Atributos:

- `int n`: cantidad de negocios
- `int m`: cantidad de depósitos
- `vector<int> correspondencia`: vector de enteros de tamaño n, donde `correspondencia[i]` representa a qué depósito está asignado el $i-esimo$ negocio, $\forall i: int, 1 \leq i \leq n$ (indexación por negocios)
- `vector<vector<int>> asignaciones`: vector de vectores de enteros donde `asignaciones[i][j] = k` representa que el k $-$ é$simo$ negocio fue asignado al $i-$ é$simo$ depósito, con $1 \leq i \leq m$ y $1 \leq$ k $\leq n$
- `vector<vector<double>> distancias`: vector de vectores de valores tipo double, todos de tamaño n, donde `distancias[i][j]` representa el costo c_{ij}, $\forall i, j, 1 \leq i \leq m$ y $1 \leq j \leq n$
- `vector<vector<int>> demandas`: vector de vectores de enteros, todos de tamaño n, donde `demandas[i][j]` representa la demanda d_{ij} $\forall i, j, 1 \leq i \leq m$ y $1 \leq j \leq n$
- `vector<int> capacidades`: vector de enteros que almacena la capacidad de los depósitos
- `vector<int> tiempo`: vector de enteros que almacena los tiempos de ejecución de todos los métodos:

  - `tiempo[0]` almacena el tiempo de ejecución al realizar la primera heurística
  - `tiempo[1]` almacena el tiempo de ejecución al realizar la segunda heurística
  - `tiempo[2]` almacena el tiempo de ejecución al realizar la búsqueda local 1
  - `tiempo[3]` almacena el tiempo de ejecución al realizar la búsqueda local 2
  - `tiempo[4]` almacena el tiempo de ejecución al realizar la metaheurística
- `double valor_objetivo`: la sumatoria de todas las distancias resultantes luego de las asignaciones de negocios a depósitos
- `double dist_max`: la mayor distancia de un negocio a un depósito que existe en el conjunto de datos. Este valor es pertinente dado que lo usaremos para penalizar a los métodos que dejen negocios sin asignar.

Métodos

- `AssignmentInstance(std::string filename)`: método constructor.
- recorremos la sección del dataset que corresponde a las distancias y las almacenamos en `distancias`
- recorremos la sección del dataset que corresponde a las demandas y las almacenamos en `demandas`
- recorremos la sección del dataset que corresponde a las capacidades y las almacenamos en `capacidades`
- recorremos `distancias` para obtener el valor máximo para guardarlo en `dist_max`
- inicializamos `asignaciones` con m vectores vacíos
- inicializamos el vector de correspondencias con n posiciones, todas con valor -1
- inicializamos `tiempo` con 5 posiciones, todas en 0
- inicializamos `valor_objetivo` en 0
- `void heuristica1()`: aplica la heurística 1 cuyo funcionamiento ya fue explicado
- `void heuristica2()`: aplica la heurística 2 cuyo funcionamiento ya fue explicado
- `void busqueda1()`: aplica la búsqueda local 1 cuyo funcionamiento ya fue explicado
- `void busqueda2()`: aplica la búsqueda local 2 cuyo funcionamiento ya fue explicado
- `void metaheuristica(bool i)`: aplica la metaheurística (si i es verdadero se realiza la metaheurística con la solución inicial siendo la de la heurística 1 y si no se realiza la metaheurística con la solución inicial siendo la de la heurística 2) cuyo funcionamiento ya fue explicado

- `vector<int> ordenamiento(vector<double> distancias)`: función auxiliar que utilizamos para ordenar las distancias dentro del vector de distancias en orden creciente.
- `void crear_archivo(string nombre)`: función que escribe a un archivo el vector de asignaciones
- `bool condiciones_swap(int i, int j)`: función auxiliar que evalúa las condiciones para realizar el operador `swap`.

Compilación

Para compilar el proyecto:

1. Abrir una terminal en el directorio del TP. Ej: `C:\Users\juani\OneDrive\Escritorio\TD5\TDV_TP2\src`
2. Correr el comando *make*, el cual prepara los ejecutables para correr el comando */gap_simulator*
3. Deberían entonces generarse los archivos .csv con el output de los distintos métodos
4. Aclaración: alteramos el archivo MakeFile para agregar en SRC a *instance.cpp*, la clase que generamos