Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Rutas de recuperación de productos en almacenes con selección de múltiples ubicaciones: Una solución heurística

Tesis de Licenciatura en Ciencias de la Computación

Leila Yasmin Abdala

Director: Javier Leonardo Marenco

Buenos Aires, 2022

RUTAS DE RECUPERACIÓN DE PRODUCTOS EN ALMACENES CON SELECCIÓN DE MÚLTIPLES UBICACIONES: UNA SOLUCIÓN HEURÍSTICA

El almacenamiento de estantes mixtos es una estrategia de guardado de productos que se encuentra a menudo en el comercio electrónico. Los productos se dividen en ítems individuales que se distribuyen por todo el almacén, lo que genera múltiples posiciones de almacenamiento por producto.
Esta tesis investiga el problema de enrutamiento del recolector en un almacén con estantes mixtos, que se diferencia de los problemas clásicos de enrutamiento del recolector por ser un problema combinado de selección y enrutamiento, en nuestro caso en particular, con una limitación por tiempo pseudo-online.
Se presenta un modelo de programación lineal entera y se evalúan los límites de tamaño de instancias que puede resolver. También se proporcionan diferentes heurísticas eficientes para resolverlo, las cuales utilizan múltiples y variados criterios parametrizables. Se realiza una búsqueda exhaustiva de la mejor combinación de criterios para solucionar el problema y como ésta se ve afectada por cambios en la distribución del almacén. Para la mejor combinación presentamos un análisis de la complejidad temporal teórico y práctico.
Además, se analiza el impacto del grado de dispersión en el tiempo de ejecución de los algoritmos y se evalúa el rendimiento de la mejor solución en instancias reales.

Palabras clave: Almacenamiento, Estanterías mixtas, Recuperación de pedidos, Rutas de recuperación, Tiempo pseudo-online

PICKER ROUTING IN MIXED-SHELVES WAREHOUSES: A HEURISTIC SOLUTION.

The mixed-shelves warehouses strategy is a goods storage strategy often found in e-commerce warehousing operations. Items are catalogued by assigning them a stock keeping unit (SKU), representing a single product and its origin. Afterwards, items are scattered through different storage positions along the warehouse, generating possibly many storage positions for the same SKU.
This thesis investigates the Picker Routing Problem in a warehouse with mixed shelves, which differs from the classic Picker Routing Problems by adding storage position selection, besides the usual picker assignment and route sequencing involved in the process. We consider the case where we also have a pseudo-online running time bound constraint.
We present an Integer Programming formulation and find the instance size limits for it to fulfill the running time constraint. Alternatively, we provide different efficient heuristics, having multiple parametrizable criteria. We perform an exhaustive search on the parameters to get the optimal combination for provided problem instances. We also study how sensible they are to changes in warehouse distribution. We conclude this part with a theoretical and practical analysis of time complexity.
We then investigate how several degrees of dispersion of goods along the warehouse impact the execution time of the algorithms. Afterwards, we evaluate the performance of the best found solution, on real instances.

Keywords: Warehousing, Mixed shelves, Order picking, Picker routing, Pseudo-online time

// 0. AGRADECIMIENTOS

A Javi Marenco, por ser la mejor persona del mundo. Por su apoyo durante el desarrollo de esta tesis y durante el desarrollo de mi carrera, como mi tutor de la beca Sadosky.

A Ale “La Mole” Quadrini, por presentarme este problema y por enseñarme a convertir mis ideas en realidad.

A Diego Picco, por ser mi tutor en la beca, por su paciencia, por su guía, por enseñarme a avanzar.

A Hugo Battellini, por hacer que ame los algoritmos, aún antes de entender que eran.

A Nico Rosner, también por ser mi tutor de la beca, y por enseñarme a preparar finales.

A mis hermanas, por que desde algo tan chiquito como traerme una taza de algo o pagarme un boleto cuando no tenía para viajar, contribuyeron día tras día a que pudiera estudiar.

A mis amigos, compañeros, docentes, por ayudarme a disfrutar este viaje. Porque me trajeron hasta acá, aunque fuera a rastras.

Índice general

1. INTRODUCCIÓN

Mi interés se tornó bien pronto analítico. Cansado de maravillarme quise saber; he ahí el invariable y funesto fin de toda aventura.
–Estación de la mano. Julio Cortázar

En las últimas dos décadas el comercio electrónico ha pasado de ser una fuente de escepticismo a ser una de las vías más usadas de compra, además de proveer una herramienta útil para conseguir información de mercado. Este crecimiento se vio acompañado por los últimos dos años de pandemia, durante los cuales en muchos casos se hizo prácticamente imposible adquirir algunos productos de manera presencial. Todo esto genera que la cantidad de productos crezca a un ritmo cada vez mas rápido, lo que nos lleva a pensar nuevas formas de procesamiento para poder acompañar al mercado.

Para establecer una buena posición entre las múltiples opciones de compra electrónica, es vital que la empresa cree una relación de confianza con los usuarios. Esta confianza se basa en la experiencia del usuario, entre otras cosas, por lo que factores como un corto tiempo de envío o mantener la promesa de entrega se vuelven vitales. Una de las vías más usadas por las empresas de comercio electrónico para asegurar estos requerimientos consiste en desarrollar sus propios depósitos y red logística, ofreciendo a sus vendedores el servicio de almacenamiento y entrega de sus productos con menos demoras y a un menor costo que el que podría conseguir un vendedor particular.

Para lograr buenos tiempos de procesamiento se tienen que atacar en conjunto múltiples problemas de optimización, entre ellos podemos mencionar la recolección de los productos desde los domicilios de vendedores, la optimización del lugar de almacenamiento, la distribución de los productos en el depósito, la recuperación de dichos productos desde las estanterías para su envío a los compradores, el despacho en vehículos desde los depósitos al domicilio de los compradores, etc. Algunos de estos problemas pueden ser atacados en conjunto mostrando una mayor mejora con relación a su optimización por separado; ver por ejemplo el trabajo de Van Gils et al. [**?**]. En el marco de esta tesis contamos con almacenes operativos, con los productos ya almacenados en múltiples estanterías por lo que nos vamos a enfocar en el problema de recuperación de productos. Este proceso consiste en, dado un conjunto de órdenes, definir de cuál ubicación vamos a tomar cada producto, y una vez seleccionadas, definir los conjuntos de ubicaciones y el orden en que serán visitadas por distintos operarios. En general, este proceso es el que conlleva más trabajo de todos los del depósito, pudiendo consumir hasta el 55 % de todas las actividades en el mismo, según se presenta en Tompkins et al. [**?**].

El contenido de esta tesis se organiza de la siguiente manera: en el Capítulo 2 presentaremos el problema y discutiremos algunas variantes del mismo y el estado del arte. En el Capítulo 3 mostramos un modelo lineal entero y en el Capítulo 4 abordaremos este problema con diferentes enfoques heurísticos. En el Capítulo 5 analizaremos los resultados de las soluciones discutidas en los Capítulos 3 y 4. Finalmente en el Capítulo 6 discutiremos sobre las conclusiones logradas en este trabajo y los siguientes pasos posibles.

2. EL PROBLEMA

Nada es más doloroso para la mente humana que un cambio grande y repentino.
–Frankenstein o el moderno Prometeo. Mary Shelley

Este estudio se desarrolla en el contexto del más grande e-commerce de América Latina, el manejo de cuyos depósitos buscamos agilizar. En ellos, los productos se almacenan en distintos tipos de áreas, y un mismo producto puede tener ítems en más de una ubicación. Cada área agrupa productos con un conjunto de características similares, pudiendo de esta forma definir modos de operación diferentes por área. Es decir, podemos definir un área donde se almacenen los productos de tamaño pequeño, y otra área donde los productos sean de gran tamaño. Se hace esta distinción ya que por ejemplo, si se tiene una estantería pequeña donde se almacenen teléfonos, probablemente no pueda soportar el peso de una heladera. De la misma manera, para llenar un estante perteneciente a una estructura que soporte el peso de heladeras probablemente necesitemos decenas o cientos de cajas de maquillaje, lo que podría complicar la búsqueda de un tono particular de maquillaje.

Dado el gran volumen de ventas que maneja este e-commerce, la entrada de nuevas órdenes de compra, y por lo tanto la cantidad de nuevos productos a ser despachados, es frenética. Esto implica que recibamos constantemente nuevas solicitudes para generar rutas para despacho de productos, provocando que el lapso de tiempo disponible para generar esta ruta sea acotado. Entonces tenemos como restricción adicional del problema que el tiempo de generación de rutas sea menor a 5 minutos, y a su vez, debemos ser capaces de procesar hasta 10K ítems en ese lapso, estos requerimientos son impuestos por la necesidad de procesamiento del e-commerce.

Denominaremos de ahora en más *waves* a las solicitudes de generación de rutas. Las waves constan de un conjunto de órdenes, productos a ser recuperados, y una foto del stock disponible para esos productos. En base a estos datos, necesitamos devolver un conjunto de rutas que múltiples operarios, habitualmente llamados *pickers*, deberán seguir para recuperar los productos. Las rutas se formarán como una lista ordenada de tuplas $(item, cantidad, ubicacion)$ dejando a criterio del picker el modo de desplazarse entre dos ubicaciones. Hacemos esto porque se considera que los pickers, al ser más experimentados en su lugar de trabajo específico, serán los más indicados para evaluar el camino mínimo entre dos ubicaciones, tomando en consideración incluso obstáculos temporales indistinguibles para el algoritmo tales como un operario realizando una tarea en el pasillo a visitar. Las rutas generadas son asignadas por orden de llegada a diferentes operarios cuando terminan con su ruta anterior. Es esperable tener en simultáneo un conjunto de operarios realizando rutas de una wave anterior, mientras otro conjunto de operarios realiza rutas de la última wave generada.

Para transportar los productos cada picker cuenta con un carro que tiene una capacidad de transporte de peso y volumen limitada, algunos ejemplos de los distintos carros se muestran en las Figuras **??** y **??**. Otra cosa a destacar sobre el uso de carros es la maniobrabilidad. Debemos pensar si el costo de caminar en línea recta, o dando vueltas, es el mismo. En general, no lo es. Incluso los operarios más experimentados deben reducir la velocidad y utilizar varias maniobras para cambiar de calle, añadiendo la dificultad adicional de que los pasillos son en general estrechos para poner a disposición más espacio de

Fig. 2.1: Carro estándar para recolección de productos pequeños.

Fig. 2.2: Grúa utilizada para el transporte de electrodomésticos voluminosos.

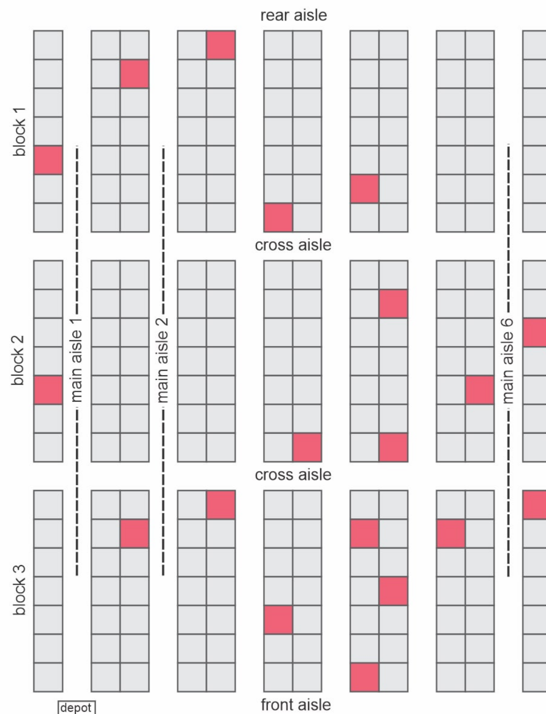

Fig. 2.3: Diagrama explicativo de una área del depósito.

almacenamiento. También se debe considerar que a medida que la ruta transcurre, el carro lleva más productos y por lo tanto es más pesado, lo que complica la maniobrabilidad del mismo; en general es una buena idea minimizar las maniobras y distancia recorridas con el carro lleno. Algunos enfoques de resolución de este problema se llevan mejor que otras con el modelado de las complejidades generadas por el carro; uno de los más básicos es la estimación del tiempo de resolución de la ruta. En este caso se puede agregar algún costo estadístico al cambio de rutas y una velocidad del carro en función del peso. Si bien estos aspectos son relevantes para resolución del problema, no contamos con los datos necesarios para realizar dichas estadísticas por lo que en esta versión de nuestro problema los dejaremos de lado al modelar.

Los productos se caracterizan por un identificador universal, llamado generalmente *stock keeping unit* o simplemente *SKU*. También tienen asociados sus dimensiones tales como la altura y ancho, disponibles para calcular el volumen del producto, y además su peso. Cada orden a su vez tiene una cantidad de ítems del mismo SKU que debe recuperar. El stock asocia a cada SKU un conjunto de ubicaciones en el depósito donde se almacenó dicho SKU, y la cantidad de ítems del mismo disponible para cada ubicación. Estas ubicaciones nos permiten posicionar los ítems en el espacio del depósito.

Los depósitos constan de varios *blocks*, o bloques, cada uno de los cuales consta de varias *aisles*, o calles, paralelas. Los ítems se almacenan a ambos lados de las calles. Con *main aisle* nos referimos a una calle entre el extremo delantero y trasero del depósito, atravesando todos los bloques. La calle delantera, *front aisle*, y la calle trasera, *rear aisle*, están ubicados completamente en la parte delantera y trasera del depósito, respectivamente. Estas dos calles no contienen ítems, pero se pueden usar para cambiar de calle. Entre

SKU	Cantidad
A	1
B	10
C	1
D	5
E	3
F	2

Tab. 2.1: Ejemplo: Cantidad de ítems requeridos por SKU.

cada par de bloques, hay una *cross aisle*, o calle transversal, que se puede utilizar para pasar de una calle a otra o de un bloque a otro, esta calle tampoco contiene ítems. Los pickers pueden atravesar las calles en ambas direcciones y cambiar de dirección dentro de las calles, aunque para algunas áreas este supuesto puede ser demasiado permisivo ya que por ejemplo en algunas áreas se utilizan grúas. Las calles son lo suficientemente estrechas como para permitir recoger desde ambos lados de la calle sin cambiar de posición. Cada wave consta de una serie de ítems que normalmente se distribuyen en varias calles. Los cambios de calle son posibles en la parte delantera, trasera y en cualquiera de las calles transversales. Los ítems recolectados deben llevarse al punto de fin, señalizado como *depot*, donde el picker también recibe las instrucciones para la siguiente ruta. El punto de fin está ubicado en la calle delantera en la cabecera de la primer calle principal. La Figura **??** da un ejemplo de un depósito de este tipo.

Las calles están numeradas de manera creciente desde 1 y contienen estanterías que también siguen esta numeración iniciando en la parte delantera del depósito. Las estanterías se numeran secuencialmente de manera enfrentada, por lo que de un lado de la calle se encuentran todas las estanterías pares y del lado de enfrente todas las estanterías impares. Luego las ubicaciones están representadas por un par indicador del área y el piso, y otro par que fija la calle y la estantería dentro de ese área y ese piso. Cada estantería a su vez tiene subdivisiones internas, que facilitan la búsqueda de los productos en la misma, sin embargo dado que la longitud de las estanterías es al menos tres órdenes de magnitud menor que la longitud de la calle, consideraremos esta granularidad despreciable y nos enfocaremos únicamente en las calles y estanterías.

Para obtener las rutas debemos resolver en simultáneo qué cantidad de rutas vamos a realizar, qué ítems pertenecerán a cada ruta, de qué ubicación será levantado cada ítem en cada ruta y el orden en que estas ubicaciones serán visitadas. A continuación, analizaremos un ejemplo artificial construido únicamente con fines ilustrativos. En la Tabla **??**.1 tenemos la cantidad total de ítems necesarios para cada SKU. Cada orden se considera mono ítem, por lo que en esta tabla resumimos 22 órdenes individuales, entre las cuales se encuentran, por ejemplo 10 órdenes de un ítem para el SKU B. Este punto es importante porque trabajar con órdenes mono ítem nos permite dividir libremente la cantidad de ítems a recolectar entre varias rutas. Las órdenes multi ítem no tienen esta propiedad y requieren una multitud de consideraciones diferentes. Este tipo de orden queda fuera del alcance de esta tesis.

En la Tabla **??**.2 tenemos asociado cada SKU con su respectivo peso y volumen por unidad. Debemos considerar estos datos a la hora de asignar los ítems a las rutas, ya que cada ruta debe completarse con un único carro, el cual puede transportar un volumen y un

SKU	Volumen	Peso
A	0.3	2
B	0.35	1
C	0.24	1
D	0.2	0.4
E	0.3	0.75
F	1.4	3

Tab. 2.2: Ejemplo: Volumen y peso por SKU.

peso máximo limitado de productos. El volumen es un tema que requiere una aclaración especial. Ya que los productos que transportamos son formas rígidas, no todo el volumen del carro es utilizable; es decir, si nuestro carro es un ortoedro y debemos transportar en él un producto de forma esférica, no hay forma de posicionarlo de modo tal que no se desperdicie volumen. El problema de organizar productos irregulares dentro de un ortoedro es un problema NP-Hard y escapa al alcance de esta tesis. Por este motivo, optamos por seguir los ejemplos de la literatura y tratar el problema del volumen como si se tratase de un volumen líquido, simplemente limitando la suma de los volúmenes. Como esto no modela correctamente la realidad, dejamos un porcentaje de volumen libre para que el picker pueda tomar decisiones de apilamiento de productos sub óptimas y lograr aún así transportar todos los ítems de la ruta. Heurísticamente se ha comprobado que un 20 % de volumen libre es una buena aproximación que no genera desperdicio innecesario. Para nuestro ejemplo, usaremos un carro con una limitación máxima de peso de 15 kilogramos y una limitación máxima de volumen de 6 metros cúbicos a los cuales ya se les ha aplicado la limitación del 20 %.

En la Tabla **??**.3 tenemos las ubicaciones disponibles para SKU, junto con las cantidades de ítems de dicho stock almacenadas por ubicación. Esta información conforma el stock. Debemos elegir entre estas ubicaciones cuál usaremos para satisfacer cada orden; recordemos que las órdenes son mono ítem por lo que con cualquier ubicación podemos satisfacer una orden. En la Figura **??** se muestran en colores las ubicaciones de cada SKU. La letra en la ubicación indica cuál es el SKU del producto almacenado en dicha posición.

Luego debemos dar como resultado un conjunto de rutas que satisfagan que todos los ítems solicitados son recuperados, y no se tomen ítems de más, y que ninguna ruta viole la restricción de volumen y peso máximo. Dadas estas restricciones, es deseable que las rutas sean de la mayor calidad posible, donde la calidad se define por alguna métrica elegida. En nuestro caso, dicha métrica será la distancia que buscaremos minimizar; pero se podrían usar otras como el tiempo de ejecución de la ruta, la cantidad de unidades recuperadas por metro cuadrado, entre otras.

En la Tabla **??**.4 vemos una posible solución para nuestro ejemplo, y la misma se encuentra representada en la Figura **??**. Nótese que para ser una solución válida no necesita ser óptima; cualquier conjunto de rutas que recupere los ítems solicitados y no supere la capacidad de carro en ninguna ruta se considera válida.

Si bien el depósito consta de múltiples áreas y una wave puede tener stock en varias áreas diferentes, es requisito mandatorio de la operación no generar rutas que atraviesen diferentes áreas. Analicemos el motivo. Vamos a presentar para este ejemplo tres tipos de áreas diferentes: el área más usual consta de productos medianos a chicos, en general

Area	Piso	Calle	Estantería	SKU	Cantidad
Regular	0	6	4	A	10
Regular	0	1	16	B	3
Regular	0	2	5	B	1
Regular	0	2	24	B	3
Regular	0	2	35	B	4
Regular	0	4	31	B	2
Regular	0	6	21	B	1
Regular	0	1	18	C	5
Regular	0	2	6	C	2
Regular	0	4	19	C	7
Regular	0	1	40	D	4
Regular	0	4	14	D	10
Regular	0	5	18	D	3
Regular	0	5	37	D	3
Regular	0	3	7	E	4
Regular	0	3	37	E	5
Regular	0	4	41	F	2
Regular	0	6	20	F	2
Regular	0	6	60	F	2

Tab. 2.3: Ejemplo: Stock.

Ruta	Area	Piso	Calle	Estantería	SKU	Cantidad
1	Regular	0	1	16	B	3
1	Regular	0	1	18	C	1
1	Regular	0	1	40	D	4
1	Regular	0	2	35	B	4
1	Regular	0	2	24	B	3
1	Regular	0	3	7	E	3
2	Regular	0	5	18	D	1
2	Regular	0	6	20	F	2
2	Regular	0	6	4	A	1

Tab. 2.4: Ejemplo: Rutas.

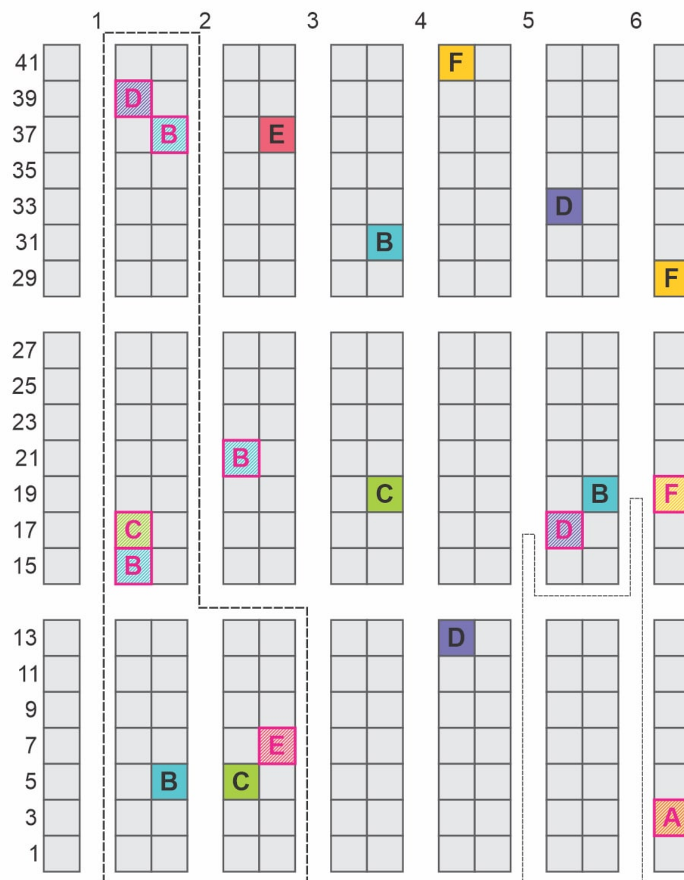

Fig. 2.4: Ejemplo: Rutas y distribución del stock.

livianos, con estanterías de menos de un metro de ancho y múltiples estantes y calles angostos, esta área podría tener varios pisos. En esta área vamos a tener que ir a buscar un producto barato y pequeño, por ejemplo una funda para teléfono. La siguiente área que vamos a analizar se encarga de almacenar productos voluminosos y pesados, que en general deben ser transportados con grúas. De esta área vamos a necesitar recuperar un lavarropas. Y finalmente, visitaremos un área especial donde se almacenan los productos pequeños y costosos en la que debemos obtener un teléfono. ¿Cómo podemos hacer para obtener esos tres elementos en una sola ruta? Debemos decidir cómo transportaremos los productos de un área a otra. Asumamos por ejemplo que nuestra funda de teléfono está en el piso cuatro, y nuestro operario esta parado en el piso cero. Si consideramos el tiempo que pasaría el picker subiendo escaleras, entonces este producto parece el más lejano, por lo que quizás podríamos ir a buscarlo al final. Pero entonces debemos resolver el problema de cómo subir un lavarropas hasta el cuarto piso. Si tenemos que llevar el lavarropas en una grúa por su peso, ¿cómo haríamos para lograr que la grúa entre en las calles pequeñas de las áreas donde están el teléfono y su funda? Quizás el lavarropas debería ser el último producto que vayamos a buscar. Entonces, vamos a buscar la funda y nos dirigimos a buscar el teléfono, para luego ir a buscar el lavarropas. Pero los productos pequeños y costosos podrían ser fácilmente extraviables y generar así una pérdida económica muy alta, por lo que la empresa impone una restricción de que no se puede sacar el teléfono del área hasta su despacho. Por lo tanto, no podemos dejar el lavarropas para el final, porque lo último debe ser el teléfono. Pero no podemos ingresar con una grúa al área del teléfono. La solución radica en no llevar productos de un área a otra, y esta es la restricción que se impone para la generación de rutas. Luego, tenemos un conjunto de órdenes que debe dividirse en rutas asociadas a diferentes áreas para completarse. Dada la naturaleza de las áreas, recordemos que se generan en base a las características de los productos, podemos asumir que un producto específico no va a tener ítems en más de un área. Tomando todo esto en cuenta, simplificaremos sin pérdida de generalidad los ejemplos al considerar una sola área, sabiendo que resolver el problema para múltiples áreas consta simplemente de resolver cada área por separado.

2.1. Estado del arte

Muchos trabajos en la literatura se han enfocado en los problemas orientados al manejo de almacenes y recuperación de pedidos. Hay múltiples enfoques entre los que destacan dos posturas claramente distinguibles: en la primera se busca optimizar los problemas de diseño del almacén, de almacenamiento de productos y recuperación de los mismos como un todo [**?**], mientras que la otra busca optimizar una parte del proceso. Como la recuperación de ítems se estima que consume más de la mitad de los recursos humanos del almacén [**?**], este problema ha recibido especial atención en los últimos 35 años, desde que se popularizaron los grandes almacenes de productos.

En general, la búsqueda de optimizar la recuperación de los pedidos puede centrarse en dos enfoques diferentes, o combinarlos. El primero busca optimizar el problema de asignación a conjuntos [**?**] para separar las órdenes en distintos grupos que luego serán recolectados con alguna política de recorrido del almacén. El segundo enfoque busca optimizar las políticas [**?**] de recorrido del almacén en sí, una vez resuelta la asignación de conjuntos.

No es hasta 1998 [**?**] que se plantea la asignación del mismo producto a múltiples

ubicaciones del depósito. Weidinger [?] en 2018 presenta una definición formal del problema y presenta la prueba de la complejidad NP-Hard de este problema. En Weidinger et al. [?] se discute sobre la utilidad de usar este enfoque y en qué condiciones sería conveniente implementar esta distribución en un depósito. Sin embargo, a nuestro entender, no existen aún estudios publicados sobre la optimización de políticas de recolección de productos en un almacén que tiene efectivamente múltiples ubicaciones por SKU junto con la asignación a conjuntos, en un tiempo acotado. Es este enfoque el que distingue este estudio de otros tipos de análisis de este problema.

2.2. Tipos de rutas

Una vez seleccionados los conjuntos en los que los ítems serán recuperados y las ubicaciones de donde se recuperaran esos ítems, debemos analizar cuál es el orden conveniente para visitar esas ubicaciones; este orden terminará de definir las rutas. Si bien es intuitivo pensar que una ruta más corta será más productiva en general, la realidad de los depósitos a veces dista mucho de esta idea. Para el desarrollo de esta tesis experimentamos de primera mano el proceso de recolección de ítems, tomando un carrito en un depósito y yendo a buscar ítems siguiendo las instrucciones dadas por nuestros propios algoritmos. Descubrimos así, por ejemplo, que la maniobrabilidad de un carro, incluso un carro vacío, es muy poca, agregando un tiempo para nada despreciable invertido únicamente en cambiar de una calle a otra. Además, conversando con supervisores y personal altamente experimentado en el proceso, fuimos informados de que las rutas que siguen un patrón estándar suelen ser las que tienen mejor productividad. Esto se debe a que el proceso de recolección de pedidos es llevado a cabo por humanos, los cuales tienden a buscar una interpretación a las decisiones de movimiento de las rutas. De esta forma una ruta anti-intuitiva, si bien podría ser óptima, genera en el usuario frustración y demoras, pues el operario no tiene a su disposición las herramientas para entender el motivo de cada decisión, por ejemplo la distancia milimétrica entre dos ubicaciones, y aún si las tuviera, no tendría el tiempo de calcular el recorrido cada vez que debe ir a buscar un ítem. Esto es apoyado por varios autores Gademann and Velde [?], Elbert et al. [?], Glock et al. [?]. Es por esto que en general se prefieren patrones de rutas repetitivos que si bien no son el resultado óptimo, mejoran la experiencia de los trabajadores.

Para esta sección nos basamos en un resumen de Roodbergen, que se puede encontrar en su sitio web [?], de las rutas presentadas en la literatura, para poder analizar los distintos tipos de rutas más usuales. En general, el diseño básico del depósito es uno con calles paralelas de donde se pueden recolectar ítems, también llamados *aisles*, un único punto de inicio y fin de las rutas que en general es el mismo, y dos posibilidades para cambiar de calle, en la parte delantera y trasera del depósito, que son las que llamamos previamente *cross aisles*. Para este tipo de almacenes, se conocen varias heurísticas de rutas (ver, por ejemplo, Hall [?]). Ratliff y Rosenthal [?] han desarrollado un algoritmo eficaz para encontrar las rutas de preparación de pedidos más cortas.

En nuestro caso nos encontramos con depósitos que no cumplen con este diseño, sino que contienen múltiples cross aisles intermedias por las que se puede cambiar de calle. Para este tipo de depósitos necesitamos otro tipo de heurísticas de ruteo.

En esta sección veremos cinco tipos diferentes de ordenamiento. Dos tipos se basan en heurísticas conocidas para el diseño básico: forma de S y brecha más grande. Además, se introduce una estrategia híbrida que combina aspectos tanto de la forma de S como de

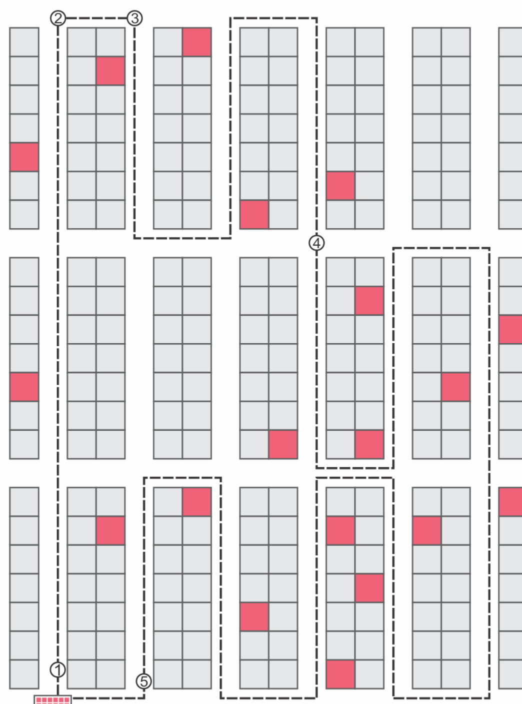

Fig. 2.5: Heurística S-Shape

la brecha más grande. La cuarta heurística es una simplificación de la tercera. El quinto método de enrutamiento consiste en encontrar la ruta más corta. Cada método se describe a continuación y se ilustra en una figura. Para los ejemplos vamos a utilizar un depósito con la distribución mostrada en la Figura **??**.

2.2.1. Heurística S-Shape

Recordemos que un bloque es el espacio del conjunto de calles comprendido entre dos calles de cruce. Esta política de ruteo avanza por bloques, recuperando todos los productos en el bloque actual antes de avanzar al siguiente.

Comienza atravesando el depósito hasta el bloque más alejado del punto de fin. Los bloques se recorren desde la calle menor o mayor que tenga ítems, tomando como inicio la más cercana a la posición actual del picker. Las calles con ítems se atraviesan en toda su longitud del bloque, alternando la dirección en forma de S para ir desde la calle superior del bloque a la inferior hasta terminar de juntar todos los ítems de ese bloque. Se saltan las calles en las que no hay que recoger nada. Luego se continúa con el bloque inferior siguiendo el mismo procedimiento. Finalmente el picker se dirige al punto de fin. En la siguiente descripción más elaborada de la heurística, los números entre paréntesis corresponden a los números de la Figura **??**.

La ruta de preparación de pedidos comienza en el punto de inicio. Va al frente de la calle principal más cercana al depósito, que contiene al menos un artículo (1). Esta calle se atraviesa hasta e incluyendo el bloque más alejado del depósito, que contiene al menos un artículo (2).

Si el bloque actual contiene al menos un artículo: va a la calle más a la izquierda que

contenga artículos o va a la calle más a la derecha que contenga artículos, la que sea más cercana (3); ir de una calle a la siguiente y atravesar cualquier calle que contenga artículos por completo; después de elegir el último artículo de este bloque, regrese al frente del mismo (4). Si este bloque no contiene elementos: Atraviese el bloque actual por la calle que este más cerca de la posición actual. Repita este procedimiento para todos los bloques hasta el que se haya considerado el bloque más cercano al punto de fin (5). Finalmente, regrese al punto de fin. En el Algoritmo **??** mostramos el pseudo código de esta heurística de ruteo.

Algorithm 1 Ruteador: S-Shape

Entrada: Lista de ubicaciones, Layout
Salida: Ruta de ubicaciones

1: **route**($locationList, layout$):
2: $route \leftarrow \emptyset$
3: agregar a $route$ el $punto_de_inicio$ de $layout$
4: ir al bloque de $layout$ mas alejado del punto de fin
5: $bloques \leftarrow$ ordenar los bloques de $layout$ desde el punto actual hacia el $punto_de_fin$ de $layout$
6: **for** $bloque \in bloques$ **do**
7:     **if** $bloque$ tiene al menos una $ubicacion$ a visitar **then**
8:         ir a la $calle$ más cercana que contenga ubicaciones y sea la mayor o menor de todas
9:         $calles \leftarrow$ todas las calles del bloque con ubicaciones ordenadas de manera ascendente si estamos en la menor o descendente si estamos en la mayor
10:         **while** $\exists calle \in calles$ con al menos una $ubicacion$ **do**
11:             atravesar la $calle$ agregando las $ubicaciones$ a $route$ en el orden visitado
12:         **if** estamos en el $cross_aisle$ contiguo al $bloque$ mas lejano al $punto_de_fin$ **then**
13:             atravesar la $calle$ actual en dirección al $punto_de_fin$
14:     **else**
15:         atravesar la $calle$ actual en dirección al $punto_de_fin$
16: agregar a $route$ el $punto_de_fin$ de $layout$
17: **return** $route$

2.2.2. Heurística Largest Gap

En la Figura **??** se representa una ruta resultante de esta heurística. Los números de esta figura se explican a continuación.

Similar a la heurística de la forma de S, la ruta de preparación de pedidos comienza en el punto de inicio; va al frente de la calle principal más cercana, a dicho punto, que contiene al menos un artículo; atraviesa esta calle hasta e incluyendo el bloque más alejado del depósito que contiene al menos un artículo (1).

Al atravesar la calle transversal (que en realidad es la calle trasera en el ejemplo de la Figura 3), se ingresa a cada calle hasta el “espacio mas grande” y se deja por el mismo lado por el que se ingresó (2). Un espacio representa la distancia entre dos artículos adyacentes o entre una calle transversal y el artículo más cercano. Por lo tanto, el espacio más grande es la parte de la calle que no se atraviesa.

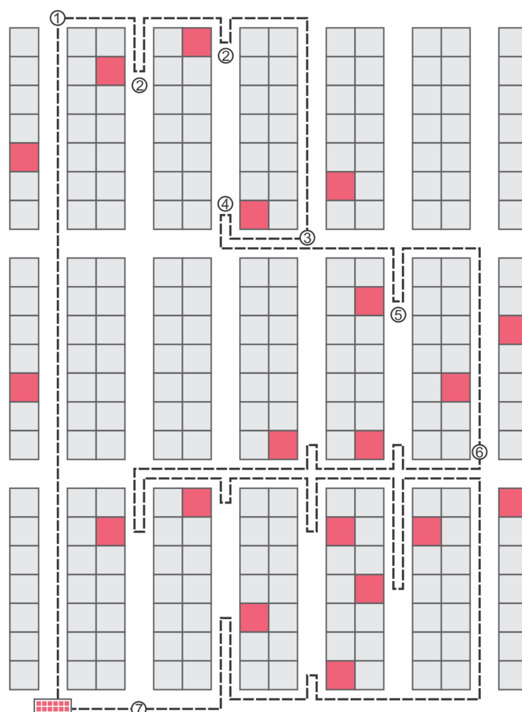

Fig. 2.6: Heurística Largest Gap

Se atraviesa en su totalidad la última calle del bloque, por el que llegamos a la siguiente calle transversal (3). Esta calle transversal se atraviesa recorriendo las calles de los bloques a ambos lados de la calle transversal hasta el espacio más grande. Primero se visitan las calles de un lado de la calle transversal (4) y posteriormente las calles del otro lado (5). Se vuelve a atravesar una calle por completo para llegar a la siguiente calle transversal (6). Esta puede ser la calle más a la izquierda o a la derecha que contiene artículos, dependiendo de cuál de los dos ofrece la distancia de viaje más corta dentro de la calle transversal.

Este proceso se repite para todos los bloques que contengan ítems. Si un bloque no contiene artículos, entonces la calle de este bloque más cercano a la posición actual se atraviesa por completo. Después de considerar el último bloque, se debe dirigir al punto de fin (7).

2.2.3. Heurística combinada

Esta heurística crea rutas de recolección de ítems que visitan cada calle que contiene artículos exactamente una vez. Las calles de cada bloque se visitan de forma secuencial, ya sea de izquierda a derecha o de derecha a izquierda.

De forma similar a las heurísticas S-shape y largest gap, la ruta de preparación de pedidos comienza en el punto de inicio; va al frente de la calle principal más cercana al punto de inicio, que contiene al menos un ítem; luego atraviesa esta calle hasta e incluyendo el bloque más alejado del depósito que contiene al menos un artículo.

Para cada bloque realizamos un pequeño algoritmo de programación dinámica. Para utilizar el concepto de programación dinámica, tenemos que definir los estados potenciales,

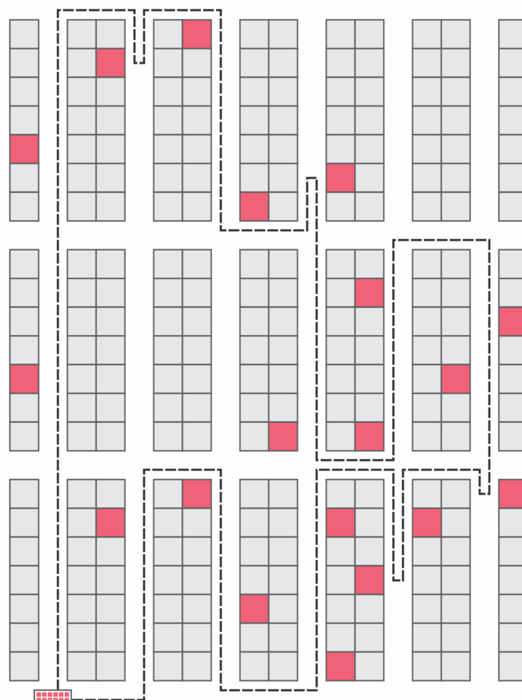

Fig. 2.7: Heurística combinada

las posibles transiciones entre estados y los costos involucrados en tal transición. Definimos dos estados:

- el picker está al frente del bloque
- el picker está detrás del bloque.

Definimos seis transiciones:

- Ir de la calle actual a la siguiente calle a lo largo del frente del bloque y atravesar esta calle por completo, terminando en la parte trasera del bloque,
- Ir de la calle actual a la siguiente calle a lo largo de la parte trasera del bloque y atravesar esta calle por completo, terminando en la parte delantera del bloque,
- Ir de la calle actual a la siguiente a lo largo del frente de la cuadra y no entrar a esta calle,
- Ir de la calle actual a la siguiente a lo largo de la parte trasera del bloque y no entrar a esta calle,
- Ir de la calle actual a la siguiente calle a lo largo del frente del bloque y atravesar esta calle hasta el ítem más alejado del frente y regresar al frente,
- Ir de la calle actual a la siguiente calle a lo largo de la parte trasera del bloque y atravesar esta calle hasta el ítem más alejado de la parte trasera y regrese a la parte trasera.

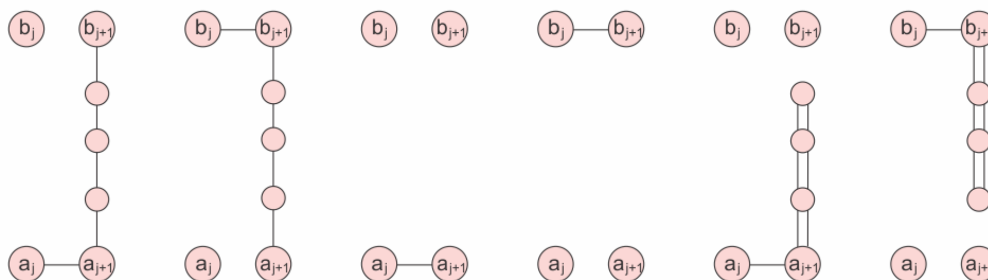

Fig. 2.8: Seis transiciones utilizadas para la heurística combinada.

Claramente, las transiciones (3) y (4) solo se permiten si la calle no contiene ningún item necesario. El costo de cada transición es igual al tiempo de viaje necesario para la distancia en la transición.

La Figura **??** muestra las seis transiciones. La calle actual es la calle j , la siguiente calle es la calle $j + 1$. El extremo trasero de la calle j se indica con una a_j y el extremo delantero con b_j .

Aplicamos el algoritmo de programación dinámica a cada bloque por separado. La conexión entre los bloques se realiza de tal forma que se minimiza la distancia recorrida en los calles transversales. Si se ha evaluado el bloque más cercano al punto de fin, el picker regresa. Una ruta resultante de este algoritmo se muestra en la Figura **??**. Para una descripción más elaborada de esta heurística, se puede consultar Roodbergen y De Koster [**?**].

2.2.4. Heurística Aisle by Aisle

Esta heurística se describe en Vaughan y Petersen [**?**]. Básicamente, cada calle principal se visita una vez. Los pickers comienzan en el depósito y van a la calle más a la izquierda o más a la derecha que contiene ítems. Todos los ítems de esta calle principal se recogen y se elige una calle transversal para pasar a la siguiente calle. Una vez más, se recogen todos los ítems de esta calle y luego los pickers pasan a la siguiente calle. La heurística aisle by aisle determina qué calles transversales utilizar para ir de una calle a la siguiente de tal manera que se minimicen las distancias recorridas.

En la Figura **??** se ejemplifica una ruta formada por esta heurística. Esta heurística no toma en cuenta los bloques para su resolución, teniendo solo en cuenta a qué distancias se encuentran los cruces de su posición actual.

2.2.5. Algoritmo óptimo

Para el diseño básico del depósito podemos encontrar rutas de preparación de pedidos más cortas con el algoritmo de Ratliff y Rosenthal [**?**]. Sin embargo, agregar solo una calle transversal a este diseño básico, complica significativamente el procedimiento de solución. En Roodbergen y De Koster [**?**] se describe un algoritmo para encontrar rutas de preparación de pedidos más cortas en un depósito con tres posibilidades para cambiar de calle: en la parte delantera, en la parte trasera y en algún punto intermedio. Siguiendo las líneas de Ratliff y Rosenthal [**?**] este algoritmo utiliza programación dinámica para

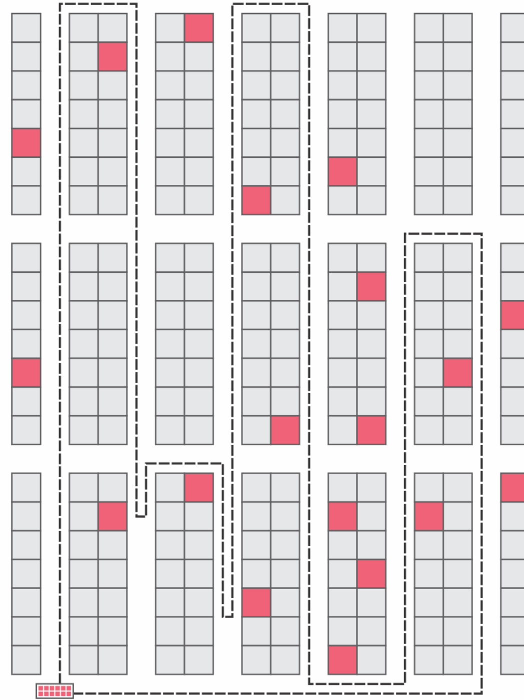

Fig. 2.9: Heurística Aisle by Aisle

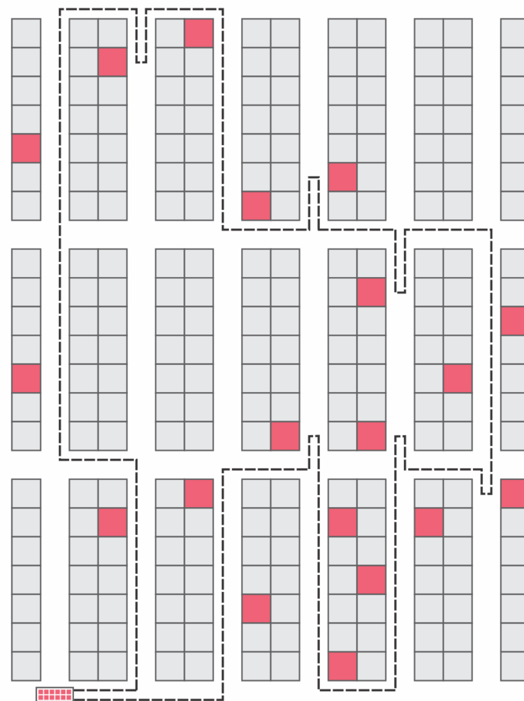

Fig. 2.10: Algoritmo óptimo

resolver el problema. Del documento se desprende que cualquier extensión a más calles transversales no es trivial y el número de clases de equivalencia y posibles transiciones aumenta rápidamente.

Por supuesto, en teoría es posible calcular rutas óptimas para almacenes con cualquier número de bloques mediante el uso de un algoritmo exhaustivo. La ruta en la Figura **??** se encuentra de esta manera (ver también Roodbergen y De Koster, [**?**]). Sin embargo, para este caso de uso, dicho algoritmo es demasiado complejo.

3. MODELO DE PROGRAMACIÓN LINEAL ENTERA

No es bueno dejarse arrastrar por los sueños y olvidarse de vivir.
–Harry Potter y la piedra filosofal. J. K. Rowling

En este capítulo plantearemos un modelo de programación lineal entera que busca aproximar la solución óptima del problema estudiado en esta tesis. Decimos que busca aproximar pues no modela ninguno de los tipos de rutas introducidos en la sección anterior, ya que la cantidad de variables requeridas para modelarlo sería mucho más grande. También, podríamos pensar en pre-computar las distancias entre todos los pares de ítems, pero esto resulta prohibitivo dado que tenemos que manejar instancias de hasta 10K ítems. Manejar estos números en cinco minutos no parece razonable. Por este motivo buscaremos aproximar la distancia de la heurística aisle by aisle de manera simplificada, restringiendo que los cambios de calle solo se puedan realizar en las calle transversales delantera y trasera.

Llamaremos *batch* al conjunto de ítems que agruparemos para construir luego una única ruta, sin embargo el batch queda desligado del tipo de ruta que construiremos. Es decir, un mismo batch podría ser recuperado con diferentes tipos de ruta. La cantidad de batches posibles está acotada por la cantidad de ítems que debemos recuperar, ya que en el peor de los casos necesitaríamos un batch por orden.

Tenemos como datos de entrada un conjunto de ítems a recuperar y un stock. A los ítems que debemos recuperar los llamaremos $Orders_{SKU}$, este parámetro indica para cada SKU la cantidad de ítems requeridos. El $Stock_{SKU,a}$ indica para cada calle la cantidad de ítems del SKU disponible; este valor puede ser cero ya que no todos los ítems están disponibles en todas las calles.

3.1. Formulación

Para este modelo, utilizaremos las siguientes variables de decisión:

- Para cada SKU perteneciente al conjunto de ítems a recuperar, para cada calle a del depósito, para cada posible batch b, usaremos una variable numérica entera $X_{b,SKU,a} \in \mathbb{N}_0$, de modo tal que $X_{b,SKU,a}$ represente la cantidad de ítems del producto SKU, levantados de la calle a en el batch b.

- Para cada posible batch b, usaremos la variable de decisión $Y_b \in \{0, 1\}$ de modo tal que $Y_b = 1$ si se levantó al menos un producto en el batch b.

- Para cada posible batch b, para cada posible calle a, usaremos la variable de decisión $V_{b,a} \in \{0, 1\}$ de modo tal que $V_{b,a} = 1$ si se levantó al menos un producto en la calle a para el batch b.

- Para cada posible batch b, para cada posible calle a, usaremos la variable de decisión $L_{b,a} \in \{0, 1\}$ de modo tal que $L_{b,a} = 1$ si existe una calle menor o igual que a en la que se levantó algún producto para el batch b.

- Para cada posible batch b, para cada posible calle a, usaremos la variable de decisión $R_{b,a} \in \{0,1\}$ de modo tal que $R_{b,a} = 1$ si existe una calle mayor o igual que a en la que se levantó algún producto para el batch b.

Sea M la cantidad de ítems a recuperar, $B = \{1...M\}$ el conjunto de los posibles batches, $A = \{1...LastAisle\}$ el conjunto de todas las calles que conforman el almacén, y $SKUS$ el conjunto de todos los identificadores de productos que debemos recuperar; nuestro modelo esta dado por:

$$\min G * \sum_{b \in B} Y_b + \sum_{b \in B, a \in A} V_{b,a} * AisleLength + \sum_{b \in B, a \in A} (L_{b,a} + R_{b,a} - 1) * AisleWidth \tag{3.1}$$

$$\sum_{SKU \in SKUS, a \in A} X_{b,SKU,a} * V_{SKU} \leq MaxVol \qquad \forall b \in B \tag{3.2}$$

$$\sum_{SKU \in SKUS, a \in A} X_{b,SKU,a} * W_{SKU} \leq MaxWei \qquad \forall b \in B \tag{3.3}$$

$$\sum_{b \in B} X_{b,SKU,a} \leq Stock_{SKU,a} \qquad \forall a \in A, SKU \in SKUS \tag{3.4}$$

$$\sum_{b \in B, a \in A} X_{b,SKU,a} = Orders_{SKU} \qquad \forall SKU \in SKUS \tag{3.5}$$

$$\frac{\sum_{SKU \in SKUS, a \in A} X_{b,SKU,a}}{M} \leq Y_b \qquad \forall b \in B \tag{3.6}$$

$$\frac{\sum_{SKU \in SKUS} X_{b,SKU,a}}{M} \leq V_{b,a} \qquad \forall b \in B, a \in A \tag{3.7}$$

$$\frac{\sum_{SKU \in SKUS} \sum_{i=1}^{a} X_{b,SKU,i}}{M} \leq L_{b,a} \qquad \forall b \in B, a \in A \tag{3.8}$$

$$\frac{\sum_{SKU \in SKUS} \sum_{i=a}^{LastAisle} X_{b,SKU,i}}{M} \leq R_{b,a} \qquad \forall b \in B, a \in A \tag{3.9}$$

$$X_{b,SKU,a} \in \mathbb{Z}^+ \qquad \forall b \in B, SKU \in SKUS, a \in A \tag{3.10}$$

$$Y_b \in \{0,1\} \qquad \forall b \in B \tag{3.11}$$

$$V_{b,a} \in \{0,1\} \qquad \forall b \in B, a \in A \tag{3.12}$$

$$R_{b,a} \in \{0,1\} \qquad \forall b \in B, a \in A \tag{3.13}$$

$$L_{b,a} \in \{0,1\} \qquad \forall b \in B, a \in A \tag{3.14}$$

Podemos ver que la función (**??**) minimiza la estimación de las distancias y la cantidad de rutas generadas, donde G es el costo asociado a la creación de una ruta, $AisleLength$ es el largo total de una calle y $AisleWidth$ el ancho. El segundo y tercer término de la función calculan la distancia de una heurística en forma de S de un sólo bloque, es decir sin considerar las calles transversales. Con las restricciones (**??**) aseguramos que cada batch no supere el máximo volumen soportado por el carro, el cual está representado con $MaxVol$, donde V_{SKU} representa el volumen asociado a cada SKU. La restricción (**??**) actúa del mismo modo para el peso máximo que se representa con $MaxWei$ siendo W_{SKU} el peso asociado a cada SKU. La restricción (**??**) asegura que no tomemos más

stock del disponible, siendo $Stock_{SKU,a}$ el stock de cada SKU asociado a cada calle a; a fines de este modelo las estanterías en las que está ubicado cada ítem es indiferente, pues se considera que una vez que el picker ingresa a una calle, ésta será recorrida en su totalidad. La restricción (**??**) se encarga de que el conjunto de todos los batches recupere exactamente la cantidad de $SKUs$ necesaria.

Las siguientes restricciones se encargan de asociar las variables entre sí. La restricción (**??**) establece que si al menos un ítem es recuperado para un batch b entonces ese batch debe considerarse como usado. La restricción (**??**) identifica las calles que deberemos atravesar para realizar nuestra ruta, por tener algún ítem recolectado de la calle a para el batch b. La restricción (**??**) identifica por batch, todas las calles que son menores o iguales a la última calle donde se recupera un ítem, a su vez, la restricción (**??**) identifica todas las calles que son mayores o iguales a la primer calle donde se recupera un ítem.

Juntas las variables L y R nos permiten obtener todas las calles comprendidas entre la primera y la última calle con ítems. Con esto obtenemos el desplazamiento vertical de la ruta. Analicemos un ejemplo, para una ruta. Sabemos que L y R son variables boolenas, por lo tanto solo pueden tomar los valores 0 o 1. Tomemos un depósito de seis calles, la primera calle que contiene ítems de nuestra ruta es la calle 3 y la última es la calle 5. Por la restricción (**??**) sabemos que vale $L_a = 1$ para todas las calles tales que $a \leq 5$. Por la restricción (**??**) sabemos que vale $R_a = 1$ para todas las calles tales que $a \geq 3$. Ahora, eso no nos garantiza nada sobre los demás valores de a. Ahora analicemos el término de la función objetivo que contiene a L y R:

$$\sum_{b \in B, a \in A} (L_{b,a} + R_{b,a} - 1) * AisleWidth \tag{3.15}$$

Como los coeficientes que acompañan a L y a R son positivos, el solver va a asignar cero a todos los valores de L y a R que no estén limitados por una restricción, ya que de esa forma se alcanza el mínimo de la función objetivo. Ahora sabemos cuánto valen L y R en todos sus puntos, falta ver que efectivamente estamos sumando a la función objetivo solo aquellas calles que están comprendidas entre la primera y la última calle con ítems. Si L y R valen 1 en simultáneo, quiere decir que estamos en una calle intermedia entre 3 y 5, inclusive. Estas son las calles que queremos sumar, y podemos ver que el valor de la expresión $(L_{b,a} + R_{b,a} - 1) = 1$ por lo que el término total en esas calles vale $AisleWidth$. Si L o R vale 0, entonces, $(L_{b,a} + R_{b,a} - 1) = 0$ por lo que el termino total vale 0. Nótese que por construcción L y R no pueden valer 0 simultáneamente. Luego, la cantidad de veces que sumamos $AisleWidth$ a la función objetivo es exactamente la cantidad de calles entre la primera y la última calle que contiene ítems a recuperar en nuestra ruta.

En este modelo dejamos de lado la distancia necesaria para ir desde el punto de inicio a la primera calle y desde la última calle al punto de fin, pues consideramos que esos valores se pueden promediar estadísticamente y sumarse al costo G de crear una ruta, simplificando de esta forma el modelo.

Como mencionamos antes, M será la cantidad de ítems diferentes a recuperar y A la cantidad de calles en el depósito. Este modelo se plantea con M^2A variables numéricas enteras positivas, $M(1 + 3A)$ variables binarias y $4M(1 + A)$ restricciones.

4. HEURÍSTICAS

La simplicidad es la clave de la verdadera elegancia.
–Coco Chanel

En el capítulo anterior planteamos un modelo para representar nuestro problema a resolver. Si bien logra computar una solución en casos acotados, no alcanza para cubrir los requerimientos de tiempo y cantidad de órdenes a recuperar. En este capítulo analizaremos un conjunto de heurísticas desarrolladas a fin de obtener soluciones de calidad aceptable en el tiempo estimado.

4.1. Estructura general

Recordemos los datos de entrada de nuestro modelo. Tenemos una lista de órdenes de compra, o ítems identificados con SKUs, que debemos recuperar del depósito. Cada ítem se considera independiente de los demás y no hay reglas sobre qué ítems deban recuperarse en conjunto con otros. También tenemos un stock, que se representa mediante la asociación de locaciones a un determinado SKU y una cantidad de ítems disponibles en esa ubicación para ese SKU. En una ubicación puede haber más de un SKU disponible, y un SKU puede tener múltiples ubicaciones. Las ubicaciones son una tupla conformada por el área, piso, calle y estantería; sin embargo definimos tratar el problema como mono área, mono piso, por lo que los primeros dos elementos de la tupla serán ignorados. Además, tenemos disponible un *layout*. El layout representa la distribución física del depósito donde se almacenan los ítems, almacenando por ejemplo el tamaño de las estanterías que tiene cada área, o la longitud de las calles.

Algorithm 2 Estructura general:

Entrada: Lista de órdenes, Stock, Layout
Salida: Lista de rutas de recuperación de pedidos

1: **Runner**(ordenes, stock, layout):
2: $result = genera_particion_inicial(orders, stock, layout)$
3: $result = aplicar_refinadores(result, stock, layout)$
4: $generar_rutas(result)$
5: **return** $result$

La estructura general de nuestra implementación consiste en generar una partición en batches inicial a la que luego se le aplican heurísticas de refinamiento. Como queremos procesar 10K órdenes en 5 minutos, o menos, y teniendo en cuenta que se esperan en promedio 10 ubicaciones por SKU, debemos asegurarnos que el algoritmo sea lo mas rápido posible. Para esto, priorizamos realizar algoritmos simples y rápidos, usando prácticas de programación orientadas a la performance. Algunas de estas son la reutilización de estructuras de datos, ciclos sobre índices en lugar de iteradores, y agrupaciones de datos por los atributos más pesados, por ejemplo, agrupar el stock por SKU en lugar de ubicación, ya que las ubicaciones tienen muy pocas colisiones. Tanto la partición inicial como los refinamientos se realizan sobre batches, usando módulos para saber la distancia que una

ruta definida sobre un batch para una política de ruta específica tendría pero sin computar las rutas reales hasta el final de la ejecución. Esto se realiza de este modo para optimizar la performance, ya que no es necesario considerar todos los puntos de una calle, por ejemplo, para saber la distancia de la ruta, basta con conocer los extremos. La partición inicial tiene mucho impacto en la calidad de la solución final, así que se evalúan muchas opciones de generación parametrizables con diferentes criterios. Analizaremos estos criterios en la siguiente sección.

La primera decisión que se parametriza es en qué orden se van a evaluar las órdenes. La primera orden vista obtiene prioridad sobre sus iguales en el objetivo; por ejemplo, dos ítems que tengan ubicaciones a la misma distancia promedio de la ruta serían ponderadas con el mismo valor por lo que se vuelve relevante decidir cómo desempatar en estos casos. Tomar una mala decisión sobre qué orden ubicar primero puede llevarnos a una baja muy fuerte de la productividad de esa ruta. Podemos analizar como ejemplo qué pasaría si al evaluar agregar una orden con ubicación alejada, nos decantaremos por usar una ruta abierta simplemente para completar el volumen del carro, aun teniendo órdenes más cercanas simplemente porque evaluamos esa orden primero. Cambiar el orden de evaluación en este caso podría resultar en que una orden con una ubicación cercana complete la ruta, y se genere una nueva ruta para la orden alejada, minimizando la distancia total recorrida en las dos rutas.

Siguiendo el mismo razonamiento, se parametriza de manera explícita e individual la selección de la semilla para la creación de un batch. A veces esta elección queda determinada por el flujo de la solución inicial; por ejemplo, cuando una orden no entra en ningún batch abierto, por lo que debemos crear un nuevo batch para almacenarla.

También hay múltiples algoritmos para elegir la partición inicial y refinadores, y todos son parametrizables. Es decir, podemos elegir un algoritmo para la partición inicial y luego seleccionar una serie de refinadores específicos para ese algoritmo de solución inicial; incluso podríamos elegir no refinar la solución inicial.

Como discutimos en la Sección 1.2 existen diferentes heurísticas de rutas que podemos usar, por lo que la elección de cuál usar queda como un parámetro más de la solución.

Además existe un modulo llamado *scorer*, que es el encargado de asignar 'puntaje' a las rutas que se forman, decidiendo así cuál combinación "de orden en tal ruta" es mejor en la iteración actual del algoritmo. De momento el único scorer implementado es el de distancia, buscando siempre minimizar la distancia final, sin embargo sería fácilmente extendible a distintos tipos de métricas. Un ejemplo de métrica posible sería

$$tiempo = dist/velocidad + \#ubicaciones_visitadas * tiempo_de_servicio + constantes \tag{4.1}$$

pudiendo evaluarse otras a solicitud de la operación.

Para finalizar, tanto la partición inicial como los refinamientos utilizan un manager del stock que es el encargado de seleccionar la mejor ubicación para ir a buscar el ítem solicitado; esto se hace en base a las ubicaciones disponibles para el ítem y las ubicaciones ya seleccionadas para los demás elementos de esa ruta. El criterio por el cual el manager define si una ubicación es mejor que otra, también es parametrizable.

4.2. Criterios parametrizables

A continuación detallaremos los criterios que elegimos para cada parte del algoritmo, y los motivos de dicha elección. También agregaremos nombres para cada criterio, a fin de

referenciarlos fácilmente en la siguiente sección.

4.2.1. Particiones iniciales

En la práctica hemos notado que las particiones iniciales tienen un impacto alto en la calidad de las rutas generadas, por lo tanto presentamos varias heurísticas para conseguir una partición válida inicial.

La primera es SEQUENTIAL_INSERTION. Una inserción secuencial, explicada en el Algoritmo **??**, que evalúa cada batch hasta ocupar toda la capacidad disponible y luego crea otro. Para esto, itera sobre todas las órdenes cada vez que tiene que decidir cuál agregar al batch en curso.

Algorithm 3 Partición inicial: Inserción secuencial

Entrada: Lista de órdenes, Stock, Layout
Salida: Lista de rutas de recuperación de pedidos

```
1: generateBatch(ordersList, stock, layout):
2: batches ← Ø
3: current ← Ø
4: while ordersList ≠ Ø do
5:     elegir una semilla y asignarla en current
6:     for order ∈ ordersList do
7:         if order entra en current then
8:             elegir la mejor location para order en current
9:             si la distancia agregando order en location a current
10:            es la menor distancia encontrada hasta ahora, guardarlas
11:    if hay una order almacenada como la de menor distancia then
12:        agregar order en location a current
13:        eliminar order de ordersList
14:    else
15:        agregar current a batches
16:        current ← Ø
17: if current ≠ Ø then
18:     agregar current a batches
19: return batches
```

La siguiente es PARALLEL_INSERTION. Una inserción paralela, explicada en el Algoritmo **??**, que trabaja de manera similar, sólo que no descarta que alguna orden se pueda agregar a un batch anterior. La diferencia es que para cada orden, evalúa todas los batches creados o decide si es más conveniente crear otro.

La última es GLOBAL_ROUTE. Está basada en la idea de Theys et al. [?] de atacar este problema como TSP y se explica en el Algoritmo **??**. Consiste en hacer una ruta con todos los ítems a recuperar y después dividirla en sub rutas más pequeñas que entren en la capacidad de transporte de un carro. Esto se hace tratando de que las ubicaciones elegidas para la ruta global minimicen la distancia de la misma. La ruta global se genera usando el mismo criterio que se utiliza para las rutas finales, es decir que este algoritmo va a generar particiones diferentes para la misma instancia si se inicializa con una política de ruta diferente. Usamos como criterio de corte la capacidad del carro; sin embargo se

Algorithm 4 Partición inicial: Inserción paralela

Entrada: Lista de órdenes, Stock, Layout
Salida: Lista de rutas de recuperación de pedidos

```
 1: generateBatch(ordersList, stock, layout):
 2: batches ← Ø
 3: initialBatch ← Ø
 4: elegir una semilla y asignarla en initialBatch
 5: agregar initialBatch en batches
 6: while ordersList ≠ Ø do
 7:     order ← la primer orden en ordersList
 8:     for batch ∈ batches do
 9:         if order entra en current then
10:             elegir la mejor location para order en current
11:             si la distancia total agregando order en location a current
12:             es la menor distancia total encontrada hasta ahora
13:             guardar order, location y batch
14:     if hay un batch guardado como el mejor then
15:         agregar order en location a current
16:         eliminar order de ordersList
17:     else
18:         batch ← Ø
19:         elegir una semilla y asignarla en batch
20:         agregar batch en batches
21: return batches
```

Algorithm 5 Partición inicial: Ruta global

Entrada: Lista de órdenes, Stock, Layout
Salida: Lista de rutas de recuperación de pedidos

```
 1: generateBatch(ordersList, stock, layout):
 2: locations ← Ø
 3: elegir una semilla y asignarla en locations
 4: for order ∈ ordersList do
 5:     elegir la mejor location para order teniendo en cuenta las ubicaciones en locations
 6:     agregar location a locations
 7: route ← generar una ruta que pase por locations
 8: batches ← Ø
 9: current ← Ø
10: for location ∈ route do
11:     order ← la orden asignada a location
12:     if order no entra en current then
13:         agregar current en batches
14:         batch ← Ø
15:     agregar order en location a batch
16: agregar current a batches
17: return batches
```

podrían implementar otros criterios, como una búsqueda local en mejora de distancias totales.

4.2.2. Ordenamiento inicial de órdenes y selección de semillas

Usamos el nombre *semilla* para indicar aquella orden con la que se iniciará un nuevo batch. Esta orden es muy importante, así como la elección de su ubicación, porque funcionará como un punto indicador de la zona donde se realizará la ruta. Recordemos que lo que buscamos minimizar es la distancia, entonces las órdenes que se considerarán mejores para este batch son aquellas que tengan ubicaciones disponibles cerca de la orden semilla. Es por esto que cada vez que se crea un nuevo batch debemos elegir cuál es la mejor semilla.

El ordenamiento inicial de órdenes indica el orden en que las órdenes van a ser evaluadas para asignarse a los batches. Ante dos órdenes que tengan el mismo nivel de preferencia, la que sea vista primera tiene prioridad en la asignación. Es especialmente relevante en el Algoritmo **??**, ruta global, ya que este algoritmo no usa semillas para crear sus batches, sino que evalúa todas las órdenes siguiendo este orden y luego separa en batches cuando se sobrepasa la capacidad del carro.

Para el ordenamiento inicial de las órdenes y la elección de semillas usaremos los mismos criterios, porque al trabajar ambos sobre órdenes sin ubicar aún, los criterios relevantes son los mismos en ambos puntos. La idea de diferenciarlos se basa en poder usar un criterio para inicializar el batch, la semilla, y otro criterio para completarlo, el ordenamiento general. Estos criterios buscan algún patrón ya sea sobre la cantidad de ubicaciones disponibles, para dar mayor flexibilidad a la optimización al elegir, o la cantidad de stock disponible, independientemente de la ubicación del mismo. Tenemos los siguientes criterios:

- FEWER_LOCATIONS: Menor cantidad de ubicaciones distintas disponibles. Elige ubicar primero los ítems que tienen menos ubicaciones disponibles. Busca dar más flexibilidad a la optimización ya que los ítems con más ubicaciones pueden ser seleccionados de manera tal que las rutas se adapten a los ítems que tienen menos, reduciendo así la posibilidad de rutas con un único ítem muy alejado.

- FEWER_AISLES: Menor cantidad de calles distintas disponibles. Este criterio busca lo mismo que el anterior, pero agrupando las ubicaciones por calle bajo la premisa de que el costo de entrar a una calle es más significativo que la distancia entre los ítems. Esto se considera así por el costo de maniobrabilidad de los carros al ingresar a la calle.

- MORE_LOCATIONS: Mayor cantidad de ubicaciones distintas disponibles. Elige ubicar primero los ítems que tienen más ubicaciones disponibles. Busca influir positivamente en la performance del algoritmo al despachar primero aquellos para los que tenemos más ubicaciones para elegir.

- MORE_AISLES: Mayor cantidad de calles distintas disponibles. Este criterio busca lo mismo que el anterior, de manera análoga a FEWER_AISLES.

- MINIMUM_STOCK: Menor cantidad de stock disponible. Con este criterio se busca evaluar una posible correlación entre la cantidad de stock disponible y la frecuencia de compra, pero sin tener relación con la cantidad de ubicaciones. Es decir, si un SKU

tiene mucho stock, quizás sea esperable que ese ítem salga mucho en lo sucesivo. Un ejemplo de este caso se puede dar en un evento como el Hot Sale, donde se reciben pallets repletos de productos que van a tener grandes descuentos y previendo el evento los mismos se almacenan todos juntos cerca de la zona de despacho para agilizar el proceso.

- MAXIMUM_STOCK: Mayor cantidad de stock disponible. Con este criterio se busca evaluar una posible correlación entre la cantidad de stock disponible y la frecuencia de compra, de manera análoga a MINIMUM_STOCK.
- NEAREST_TO_EXIT: Con la ubicación más cercana al punto de finalización de la ruta. Elegimos primero el que esté más cerca del punto donde la ruta tiene que terminar. No hay uno que sea equivalente con el punto de inicio, porque para simplificar el problema definimos que el punto de inicio y de fin es el mismo.
- MINIMUM_VOLUME: Menor volumen de ítem. Elegimos priorizar de acuerdo al volumen ya que agregar primero los ítems de menor tamaño permite aumentar la cantidad de ítems en las primeras rutas.
- MAXIMUM_VOLUME: Mayor volumen por ítem. Con este criterio agregamos primero los ítems de mayor tamaño en busca de poder aprovechar mejor la capacidad del carro ubicando al final los ítems más pequeños en los espacios dejados por los ítems de gran volumen.
- RANDOM: Como punto de comparación.

4.2.3. Selección de stock

Para los selectores de stock implementamos tres tipos básicos diferentes. El primero es RANDOM, una selección aleatoria, su finalidad es servir de punto de comparación. El segundo es FIRST_BEST, busca la mejor ubicación dado algún criterio de calidad, su pseudo código se encuentra explicado en el Algoritmo **??**. El tercero es SECOND_SCORER, es similar a FIRST_BEST pero usa dos criterios diferentes para decidir, desempatando con el segundo criterio si se presenta un empate con el primer criterio. Su pseudo código se muestra en el Algoritmo **??**. Este último selector se vuelve relevante en algunos de los criterios de calidad que tienen muchas colisiones.

Algorithm 6 Selector de stock: Primer mejor

Entrada: Lista de ubicaciones de ruta, Lista de ubicaciones candidatas
Salida: Ubicación

1: **selectBest**($currentLocations, candidatesLocations$):
2: $bestLocation \leftarrow null$
3: $bestScore \leftarrow \infty$
4: **for** $candidate \in candidatesLocations$ **do**
5: $\quad score \leftarrow$ asignar puntaje a $candidate$ en $currentLocations$
6: $\quad$ **if** $score < bestScore$ **then**
7: $\qquad bestLocation \leftarrow candidate$
8: $\qquad bestScore \leftarrow score$
9: **return** $bestLocation$

Algorithm 7 Selector de stock: Segundo mejor

Entrada: Lista de ubicaciones de ruta, Lista de ubicaciones candidatas
Salida: Ubicación

1: **selectBest**($currentLocations, candidatesLocations$):
2: $bestLocation \leftarrow null$
3: $bestScore \leftarrow \infty$
4: **for** $candidate \in candidatesLocations$ **do**
5: $score \leftarrow$ asignar primer puntaje a $candidate$ en $currentLocations$
6: **if** $score < bestScore$ **then**
7: $bestLocation \leftarrow candidate$
8: $bestScore \leftarrow score$
9: **else if** $score = bestScore$ **then**
10: $secondScore \leftarrow$ asignar segundo puntaje a $candidate$ en $currentLocations$
11: $secondBest \leftarrow$ asignar segundo puntaje a $best$ en $currentLocations$
12: **if** $secondScore < secondBest$ **then**
13: $bestLocation \leftarrow candidate$
14: **return** $bestLocation$

Para definir la selección de stock se usan los criterios detallados a continuación, que le asignan un puntaje a cada ubicación, basándose tanto en la nueva ubicación como las ya asignadas en el batch. Esto se realiza para cada nueva ubicación a evaluar. La ubicación que es elegida es aquella que minimiza el puntaje. Mostramos el pseudo código de uno de estos criterios de puntaje en el Algoritmo **??** de manera ilustrativa, los demás criterios se implementan de manera análoga muy fácilmente.

Algorithm 8 Puntuador de ubicaciones: AverageDistanceScorer

Entrada: Lista de ubicaciones de ruta, Ubicación candidata
Salida: Racional

1: **selectBest**($currentLocations, candidate$):
2: $accum \leftarrow 0$
3: **for** $location \in currentLocations$ **do**
4: $accum+ =$ distancia entre $candidate$ y $location$
5: **return** $\frac{accum}{|currentLocations|}$

- AVERAGE_BLOCK_AISLE: Promedio de la diferencia entre bloques. Busca minimizar el promedio de la diferencia entre bloques. Los bloques se numeran de 0 en adelante, contando como corte de bloque cuando hay un pasillo trasversal por la que se podría cambiar de calle de pickeo. Busca minimizar el promedio de la expansión de los bloques de las rutas.

- SHORTEST_BLOCK_AISLE: Diferencia más corta entre bloques. Busca minimizar la menor diferencia entre bloques, de manera análoga a AVERAGE_BLOCK_AISLE.

- LARGEST_BLOCK_AISLE: Diferencia más grande entre bloques. Busca minimizar la diferencia más grande entre bloques, de manera análoga a AVERAGE_BLOCK_AISLE.

- AVERAGE_SPAN_AISLE: Promedio de la diferencia entre span de calles. Siendo

$$span = mayor_calle_de_pickeo - menor_calle_de_pickeo \tag{4.2}$$

De manera análoga a las anteriores, busca minimizar el promedio de expansión de calles de las rutas.

- SHORTEST_SPAN_AISLE: Diferencia más corta entre span de calles. Busca minimizar la menor diferencia entre calles, de manera análoga a AVERAGE_SPAN_AISLE.
- LARGEST_SPAN_AISLE: Diferencia más grande entre span de calles. Busca minimizar la mayor diferencia entre calles, de manera análoga a AVERAGE_SPAN_AISLE.
- AVERAGE_DISTANCE: Promedio de la distancia entre pares de ubicaciones. Busca minimizar el promedio entre las distancias de todos los puntos ya fijados hacia la nueva ubicación candidata.
- SHORTEST_DISTANCE: Distancia más corta entre pares de ubicaciones. Busca minimizar la distancia más corta entre las distancias de todos los puntos ya fijados hacia la nueva ubicación candidata, de manera análoga a AVERAGE_DISTANCE.
- LARGEST_DISTANCE: Distancia más larga entre pares de ubicaciones. Busca minimizar la distancia más larga entre las distancias de todos los puntos ya fijados hacia la nueva ubicación candidata, de manera análoga a AVERAGE_DISTANCE.
- RANDOM: Puntaje aleatorio como punto de referencia.

4.2.4. Refinamientos

Hay dos refinamientos implementados para esta solución. El primer refinamiento es MERGER y se basa en reducir batches, buscando dos batches que sea posible combinar dada la capacidad del carro. Su pseudocodigo está en el Algoritmo **??**.

El segundo lo llamamos BIG_ROUTE y busca disminuir el solapamiento entre rutas aplicando una heurística también basada en TSP. La misma colapsa todos los baches en una única ruta y luego los separa nuevamente basándose en la capacidad del carro. Este refinamiento difiere de la solución inicial en el hecho de que las ubicaciones ya están decididas en esta etapa y no se vuelven a cambiar. El pseudocodigo está en el Algoritmo **??**.

Algorithm 9 Refinador: Merger

Entrada: Lista de rutas de recuperación de pedidos, Stock, Layout
Salida: Lista de rutas de recuperación de pedidos

1: **refineBatchs**($batchList, stock, layout$):
2: $batches \leftarrow \emptyset$
3: **do**
4: $merge \leftarrow \emptyset$
5: $used \leftarrow \emptyset$
6: **for** $batch_1 \in batchList$ **do**
7: $canMerge \leftarrow false$
8: $best \leftarrow \emptyset$
9: **for** $batch_2 \in batchList$ **do**
10: **if** $batch_1 \neq batch_2 \wedge batch_2$ entra en $batch_1 \wedge batch_1 \notin used \wedge batch_2 \notin used$ **then**
11: $canMerge \leftarrow true$
12: **if** la distancia total añadiendo $batch_2$ a $batch_1$ es más corta que añadir $best$ a $batch_1$ **then**
13: $best \leftarrow batch_2$
14: **if** $canMerge$ **then**
15: agregar $batch_1$ y $best$ a $used$
16: agregar el par $(batch_1, best)$ a $merge$
17: **for** $(b_1, b_2) \in merge$ **do**
18: borrar b_1 y b_2 de $batchList$
19: agregar b_2 en b_1
20: agregar b_1 en $batches$
21: **while** $merge \neq \emptyset$
22: **return** $batches$

Algorithm 10 Refinador: Ruta global

Entrada: Lista de rutas de recuperación de pedidos, Stock, Layout

Salida: Lista de rutas de recuperación de pedidos

```
1: refineBatchs(batchList, stock, layout):
2: checkpoints ← todas las orders con sus locations en batchList
3: route ← generar una ruta que pase por checkpoints
4: batches ← Ø
5: batch ← Ø
6: for location ∈ route do
7:     order ← la orden asignada a location
8:     if order entra en batch then
9:         agregar order a batch
10:    else
11:        if batch ≠ Ø then
12:            agregar batch a batches
13:        batch ← Ø
14:        agregar order a batch
15:    agregar order en location a batch
16: if batch ≠ Ø then
17:    agregar batch a batches
18: return batches
```

5. EXPERIMENTOS COMPUTACIONALES

Cierto que casi siempre se encuentra algo, si se mira, pero no siempre es lo que uno busca.
–El Hobbit. J. R. R. Tolkien

En este capítulo analizaremos el modelo de programación lineal entera que presentamos en el Capítulo 2. Analizaremos la capacidad de resolución del modelo y cómo se comportan sus tiempos de ejecución. También veremos las distintas heurísticas presentadas en el Capítulo 3. Comenzaremos partiendo de instancias pequeñas para realizar una búsqueda exhaustiva de la mejor combinación de criterios, avanzando en un análisis de la performance hasta evaluar finalmente instancias reales. En el camino discutiremos aquellos criterios que parecen tener más impacto en mejorar la solución y marcaremos claramente cuáles no contribuyen.

5.1. Instancias de prueba iniciales

Las waves usadas en casos reales son instancias muy grandes, por lo que luego de un experimento inicial con estas waves donde estimamos que el tiempo de análisis necesario para correr los experimentos era demasiado alto, decidimos usar para las pruebas iniciales instancias de menor tamaño generadas automáticamente. Para esto se desarrolló un script que aleatoriza la toma de decisiones básandose en un layout parametrizado. Los datos, tales como distancias o cantidad de calles, de los layouts utilizados no se adjuntan en la tesis por considerarse información sensible de la empresa con la que colaboramos.

El script de generación de instancias recibe la cantidad de ítems a recolectar, máxima y mínima, junto con un entero que determina qué variación de ítems se debe considerar entre una y otra instancia, pudiendo de este modo generar múltiples casos de prueba en una sola llamada. Además, parametriza un número que indica la cantidad de veces que un mismo SKU puede repetirse dentro de la instancia en cuestión, a fin de modelar de manera más realista el problema. Este parámetro es el límite máximo para la generación de números aleatorios que se usan para generar la cantidad de ítems requeridos por SKU.

Para la distribución del stock recibimos un número que indica la cantidad de ubicaciones distintas máxima que puede tomar cada SKU. Este parámetro es importante para el análisis de la complejidad del algoritmo, así como la cantidad de órdenes. Para cada SKU, generamos un número aleatorio k que indica la cantidad de ubicaciones tomando como máximo nuestro parámetro anterior. Luego debemos definir cuánto stock debemos almacenar en cada ubicación. Multiplicamos el stock mínimo necesario por un número aleatorio entre uno y dos, para generar redundancia en las ubicaciones, de esta forma conseguimos s la cantidad de ítems a distribuir en las k ubicaciones. Para esto elegimos k números aleatorios entre cero y uno. Luego dividimos cada uno por el total de la suma entre ellos, obteniendo así k números que representan el porcentaje que cada ubicación contendrá. Finalmente, multiplicamos cada uno de estos valores por s, la cantidad de ítems total a distribuir; a esta multiplicación le aplicamos la función *ceil* para garantizar que siempre estaremos en una cantidad igual o mayor a la necesaria.

Ahora conocemos la cantidad de ubicaciones y la cantidad de ítems por ubicación para cada SKU, pero nos falta generar las ubicaciones en sí. Como mencionamos antes, simplifi-

camos sin pérdida de generalidad este problema limitando los casos estudiados a una única área y piso, por lo que estos dos valores son elegidos a necesidad del usuario. Para definir la calle y la estantería de cada ubicación se usan números generados aleatoriamente dentro de los rangos descriptos en el layout; esto puede causar que para algún SKU se genere más de una vez la misma ubicación. En este caso se suman los ítems de ambas asignaciones aleatorias y se colocan juntos. En el caso del modelo de programación lineal entera, nos vimos en la necesidad de controlar el número exacto de ubicaciones por ítem para poder analizar el rendimiento del modelo con más precisión, por esto nuestro generador de casos tiene una versión alternativa que permite especificar si la cantidad de ubicaciones por SKU parametrizada es una cota superior o la cantidad exacta requerida.

Se describe el procedimiento para la generación de múltiples waves en el Algoritmo **??** que usa a su vez el Algoritmo **??** para generar cada wave. En el Algoritmo **??** se muestra cómo se genera cada ubicación.

Algorithm 11 Generador de waves

Entrada:
Salida: Lista de waves

1: **variables globales:**
2: $minOrdersQuantity \leftarrow 500$
3: $maxOrdersQuantity \leftarrow 10000$
4: $stepGrowOrders \leftarrow 500$
5: $minLocsPerSKUQuantity \leftarrow 10$
6: $maxLocsPerSKUQuantity \leftarrow 20$
7: $stepGrowLocs \leftarrow 5$
8: $layout \leftarrow$ datos del layout
9: **generateWaves**():
10: $waves \leftarrow \emptyset$
11: **for** $qo = minOrdersQuantity; qo \leq maxOrdersQuantity; qo+ = stepGrowOrders$ **do**
12: **for** $ql = minLocsPerSKUQuantity; ql \leq maxLocsPerSKUQuantity; ql+ = stepGrowLocs$ **do**
13: $wave \leftarrow generar_wave(qo, ql, layout)$
14: agregar $wave$ a $waves$
15: **return** $waves$

5.2. Metodología de experimentación

Para los experimentos descritos en este capítulo, usamos una computadora estándar de escritorio con un procesador AMD Rizen 7 de 8 núcleos con 16 threads, y 16 GB de memoria RAM. Todas las heurísticas y el modelo de programación lineal entera fueron implementados en Java 11 utilizando a nuestro entender la máxima optimización de cómputo mediante estructuras y funciones temporal y espacialmente óptimas.

Para la evaluación de los casos se crearon scripts en Java que leen las instancias de un archivo de texto, el layout y los parámetros de configuración, realizan el pre-procesamiento acorde, y ejecutan la solución, ya sea el modelo entero o alguna heurística, de acuerdo a la configuración. Para todos los casos usamos como métricas el tiempo de ejecución y la

Algorithm 12 Generador de wave

Entrada: Cantidad de órdenes, cantidad de ubicaciones, datos del depósito
Salida: Wave (listas de ordenes, stock)

1: **variables globales:**
2: $maxSkuRepeat \leftarrow 20$
3: $mediumDepth \leftarrow 20$
4: $mediumWidth \leftarrow 20$
5: $mediumHeight \leftarrow 20$
6: $gapVolume \leftarrow 5$
7: $mediumWeight \leftarrow 300$
8: $gapWeight \leftarrow 200$
9: **generateWave**$(quantityOrders, quantityLocations, layout)$:
10: $acum \leftarrow 0$
11: $orders \leftarrow \emptyset$
12: $skuQuantity \leftarrow MapaVacio$
13: **while** $acum \leq quantityOrders$ **do**
14: $id \leftarrow$ generar identificador aleatorio
15: $quantityitems \leftarrow crear_un_numero_aleatorio_entre(1, maxSkuRepeat)$
16: $depth \leftarrow crear_un_numero_aleatorio_entre(mediumDepth - gapVolume, mediumDepth + gapVolume)$
17: $width \leftarrow crear_un_numero_aleatorio_entre(mediumWidth - gapVolume, mediumWidth + gapVolume)$
18: $height \leftarrow crear_un_numero_aleatorio_entre(mediumHeight - gapVolume, mediumHeight + gapVolume)$
19: $weight \leftarrow crear_un_numero_aleatorio_entre(mediumWeight - gapWeight, mediumWeight + gapWeight)$
20: **for** $i \leftarrow 1; i \leq quantityItems; i \leftarrow i + 1$ **do**
21: agregar a $orders$ una nueva orden dada por $(id, depth, width, height, weight)$
22: agregar a $skuQuantity$ el par clave id, valor $quantityItems$
23: $acum \leftarrow acum + quantity$
24: $inventory \leftarrow MapaVacio$
25: **for** $(sku, quantityItems) \in quantityItems$ **do**
26: $quantityItems \leftarrow ceil(quantityItems * crear_un_numero_aleatorio_entre(1, 2))$
27: $quantityLocationsSKU \leftarrow crear_un_numero_aleatorio_entre(1, quantityLocations)$
28: $sum \leftarrow 0$
29: $percentages \leftarrow \emptyset$
30: **for** $i = 1; i \leq quantityLocationsSKU; i{+}{+}$ **do**
31: $p \leftarrow crear_un_numero_aleatorio_entre(0, 1)$
32: agregar p a $percentages$
33: $sum \leftarrow sum + p$
34: **for** $p \in percentages$ **do**
35: $p \leftarrow p/sum$
36: $location \leftarrow generar_ubicacion_aleatoria_en(layout)$
37: $auxMap \leftarrow$ el mapa que contiene $inventory$ para la clave sku o un $MapaVacio$ si no contiene ningún mapa para esa clave
38: agregar a $auxMap$ la cantidad de stock $ceil(p*quantityItems)$ para la ubicación $location$
39: agregar $auxMap$ a $inventory$ para la clave sku
40: **return** wave dada por $(orders, inventory)$

Algorithm 13 Generador de ubicaciones

Entrada: Datos del depósito
Salida: Ubicación

1: **variables globales:**
2: $area \leftarrow regular$
3: $floor \leftarrow 0$
4: **generateRandomLocation**($layout$):
5: $aisle \leftarrow crear_un_numero_aleatorio_entre(1, cantidad_de_calles(layout))$
6: $bay \leftarrow crear_un_numero_aleatorio_entre(1, cantidad_de_estanterias(layout))$
7: **return** *location* dada por $(area, floor, aisle, bay)$

distancia total de la solución. La distancia total se mide como la suma de las distancias de todas las rutas, las cuales se calculan mediante las especificaciones del layout, tomando como partida el punto *start*, luego siguiendo la ruta indicada por el algoritmo y finalizando en el punto $finish$. Entre cualesquiera dos puntos de una ruta tomamos la norma uno del depósito, definida como la suma de las distancias necesarias para llegar desde la estantería actual al cross aisle más conveniente, caminar desde el aisle actual al aisle destino, y llegar desde ese cross aisle a la estantería destino. Al considerar también los puntos de inicio y fin como parte de la ruta, estamos asignando un peso a la creación de rutas nuevas, ya que se debe pagar el costo de ir a buscar incluso un solo ítem como la ida a la ubicación del ítem y luego el regreso al punto final. De esta forma también logramos priorizar las ubicaciones que se encuentren más cerca del inicio y del fin por sobre las demás.

Es importante notar que el tiempo presentado en los experimentos es solamente el tiempo de ejecución algorítmico, y no incluye el tiempo de lectura, pre-procesamiento y escritura de las instancias.

5.3. Modelo de programación lineal entera

En esta sección reportamos los experimentos realizados para evaluar la performance del modelo de programación lineal entera presentado en la Sección 3.1. El objetivo es determinar si este modelo es una plataforma adecuada para intentar la resolución del problema, o si por el contrario debemos focalizarnos en alguna variante del procedimiento heurístico presentado en el Capítulo 4.

El modelo fue implementado en Java 11 con la librería ilog de CPLEX versión 12.10. El modelo se construye a partir de los datos de cada wave y del depósito almacenados en formato JSON. La solución del modelo se interpreta y convierte en rutas antes de almacenarse.

Cabe destacar que el modelo de programación lineal entera solo elige la calle de la cual se seleccionará cada SKU, no la ubicación final, pudiendo tener dos o más ubicaciones con el mismo SKU en la misma calle. Para la elección de la ubicación final simplemente elegimos una ubicación de dicha calle con un criterio aleatorio.

Para la evaluación del modelo generamos instancias desde cinco a diez órdenes, con variación en la cantidad de ubicaciones de una a quince ubicaciones por SKU. Para estas instancias forzamos que las ubicaciones fueran diferentes. A diferencia de las heurísticas donde cada ítem se trata como una orden independiente sin importar su SKU, en el caso del modelo se agrupan las órdenes que tienen el mismo SKU en una sola variable; es

order_quantity locations_quantity	5	6	7	8	9	10
1	91	30	520	414	37	394
2	317	35	336	103	103	250
3	274	97	415	647	2626	396
4	111	269	265	258	631	2685
5	1815	2522	3028	4301	Out of memory	Out of memory
6	34	7189	3426	4930	Out of memory	Out of memory
7	2815	801731	46126	3398	Out of memory	Out of memory
8	57	6073	36360	26339	Out of memory	Out of memory
9	39348	13251	Out of memory	Out of memory	Out of memory	Out of memory
10	74	3442	Out of memory	Out of memory	Out of memory	Out of memory
11	3076	41294	Out of memory	Out of memory	Out of memory	Out of memory
12	108	16099	Out of memory	Out of memory	Out of memory	Out of memory
13	71	Out of memory	Out of memory	Out of memory	Out of memory	Out of memory
14	265	Out of memory	Out of memory	Out of memory	Out of memory	Out of memory
15	148	Out of memory	Out of memory	Out of memory	Out of memory	Out of memory

Tab. 5.1: Tiempos de resolución en milisegundos del modelo lineal entero.

por esto que para poder analizar correctamente la performance durante el crecimiento forzamos que solo se pueda requerir un ítem por instancia para cada SKU. Si bien esto no es totalmente realista, nos permite analizar más claramente el impacto del tamaño de la instancia en la performance.

En la Tabla **??** tenemos los tiempos de ejecución en milisegundos y aquellos casos para los cuales el modelo no encontró ninguna solución por quedarse sin memoria durante la resolución de la instancia. Cada columna muestra la cantidad de órdenes de la instancia, iniciando en 5 órdenes mono ítem y terminando en 10 órdenes mono ítem, aumentando de a una orden. Las filas representan la cantidad exacta de ubicaciones diferentes por SKU, comenzando en 1 y terminando en 15 ubicaciones diferentes, aumentando de a una ubicación. Si bien forzamos a que las ubicaciones sean necesariamente distintas, dos o más ubicaciones podrían pertenecer a la misma calle en cuyo caso se agruparían, ya que el modelo no tiene granularidad por estantería.

Si bien los tiempos obtenidos están dentro de nuestra cota, 5 minutos, el problema de este modelo se hace evidente al quedarse rápidamente sin memoria en la construcción del modelo para instancias demasiado chicas. Basándonos en los análisis estadísticos realizados, podemos afirmar que una cantidad de ubicaciones promedio por SKU es realista, sin embargo tenemos instancias que deben ser resueltas donde la cantidad de ubicaciones puede llegar a ser más de un centenar. Tomando en cuenta la cantidad de órdenes máxima a resolver, 10K órdenes, queda rápidamente descartado el modelo lineal entero como solución posible.

5.4. Heurísticas

En la Sección 2.2 definimos los tipos de ruteo más usuales en la literatura, falta ver cuáles de éstos son aplicables a nuestro caso de uso. El primero en ser descartado es el algoritmo óptimo, ya que prácticamente todos los depósitos usados para el análisis contienen al menos un área con más de tres bloques. Además, como ya hemos mencionado, la previsibilidad de la ruta juega un papel importante en la productividad del proceso. Por este motivo, luego de algunas pruebas básicas, decidimos también descartar la opción

de usar la heurística combinada.

Para la evaluación de las heurísticas tenemos la combinación de todos los criterios descriptos en la Sección 3.2, incluyendo los casos de incluir o no los refinadores, el orden de los mismos o la cantidad variable de scorers usados para la selección de stock. La combinatoria total de criterios realizada, sin repeticiones semánticas, genera 360K combinaciones posibles.

Nuestro primer experimento consistió en generar una wave aleatoria siguiendo las alineaciones máximas de nuestro problema: 10K órdenes, con un máximo de diez ubicaciones distintas por SKU. Tomamos la decisión de que ningún SKU se repitiera en más de 20 órdenes de manera estadística analizando casos reales. Luego ejecutamos nuestra wave para la lista de combinaciones, la cual previamente había pasado por un proceso de shuffle para evitar que el orden con el que fue generada influyera en la prueba. Luego de las primeras doce horas de ejecución, la implementación había ejecutado solo 115 combinaciones de parámetros. Haciendo una estimación rápida, vimos que cada combinación tardaba en promedio alrededor de seis minutos y medio en resolverse, y extrapolando llegamos a la conclusión de que necesitaríamos alrededor de cuatro años y tres meses para evaluar todas las posibles combinaciones.

5.4.1. Caso inicial: Reduciendo combinaciones

Para manejar la situación descripta arriba, decidimos generar tres waves de test, con 100, 250 y 500 órdenes, con un máximo de 10 ubicaciones diferentes por SKU, con a lo sumo 20 órdenes con el mismo SKU. Luego evaluamos los 360K casos para las tres waves, tardando en el caso de 500 órdenes aproximadamente 14 horas en completar todas las combinaciones. Una vez obtenidos las distancias y los tiempos de ejecución, buscamos agruparlos de manera tal de determinar qué combinaciones de criterios proporcionaron las mejores soluciones.

Para analizar estos resultados tomamos las 5000 combinaciones de parámetros que generaron las soluciones con menor distancia total, luego realizamos una serie de agrupaciones diferentes y decidimos cuál de esa nos parecía más declarativa. Finalmente analizamos cuál era el porcentaje de presencia de cada combinación en los 5000 mejores resultados y observamos la dispersión de las soluciones.

Para la agrupación de datos realizamos agrupaciones por todos los criterios variables. Asignamos como más prioritario el criterio cuya agrupación (dentro de los 5000 primeros) resultaba en una menor cantidad de conjuntos. Luego volvimos a agrupar los restantes criterios dentro de cada conjunto y tomamos como prioritario el que, en promedio, resultaba en una menor cantidad de conjuntos. Continuamos de manera iterativa hasta terminar todos los criterios. Con esta visualización disponible nos dimos cuenta de que algunos criterios, la selección de semilla por ejemplo, no cambiaban la solución. Finalmente seleccionamos como los más representativos a los criterios route_type, initial_batcher_type y stock_selector. Para estos tres, realizamos una agrupación por la tupla contando la cantidad de veces que se repetía la misma distancia total en las 5000 mejores combinaciones. El resultado de esta agrupación junto con las distancias totales y la cantidad de repeticiones por distancia se muestra en la Tabla **??**. Las Tablas **??** y **??** se definieron de manera análoga.

En la Tabla **??** podemos observar que la combinación de criterios $route_type = LARGEST_GAP$, $initial_batcher_type = GLOBAL_ROUTE$ y $stock_selector = SECOND_SCORER$ tienen la menor distancia, en la celda destacada.

route_type	initial_batcher_type	stock_selector	total_distance	quantity
LARGEST_GAP	GLOBAL_ROUTE	FIRST_BEST	15096	80
LARGEST_GAP	GLOBAL_ROUTE	FIRST_BEST	15195	240
LARGEST_GAP	GLOBAL_ROUTE	FIRST_BEST	15509	40
LARGEST_GAP	GLOBAL_ROUTE	FIRST_BEST	15659	23
LARGEST_GAP	GLOBAL_ROUTE	SECOND_SCORER	14248	40
LARGEST_GAP	GLOBAL_ROUTE	SECOND_SCORER	15096	1000
LARGEST_GAP	GLOBAL_ROUTE	SECOND_SCORER	15195	2640
LARGEST_GAP	GLOBAL_ROUTE	SECOND_SCORER	15441	40
LARGEST_GAP	GLOBAL_ROUTE	SECOND_SCORER	15445	240
LARGEST_GAP	GLOBAL_ROUTE	SECOND_SCORER	15496	40
LARGEST_GAP	GLOBAL_ROUTE	SECOND_SCORER	15509	440
LARGEST_GAP	GLOBAL_ROUTE	SECOND_SCORER	15659	147
LARGEST_GAP	PARALLEL_INSERTION	FIRST_BEST	15272	2
LARGEST_GAP	PARALLEL_INSERTION	SECOND_SCORER	15272	22
LARGEST_GAP	SEQUENTIAL_INSERTION	SECOND_SCORER	15539	6

Tab. 5.2: Caso de 100 órdenes.

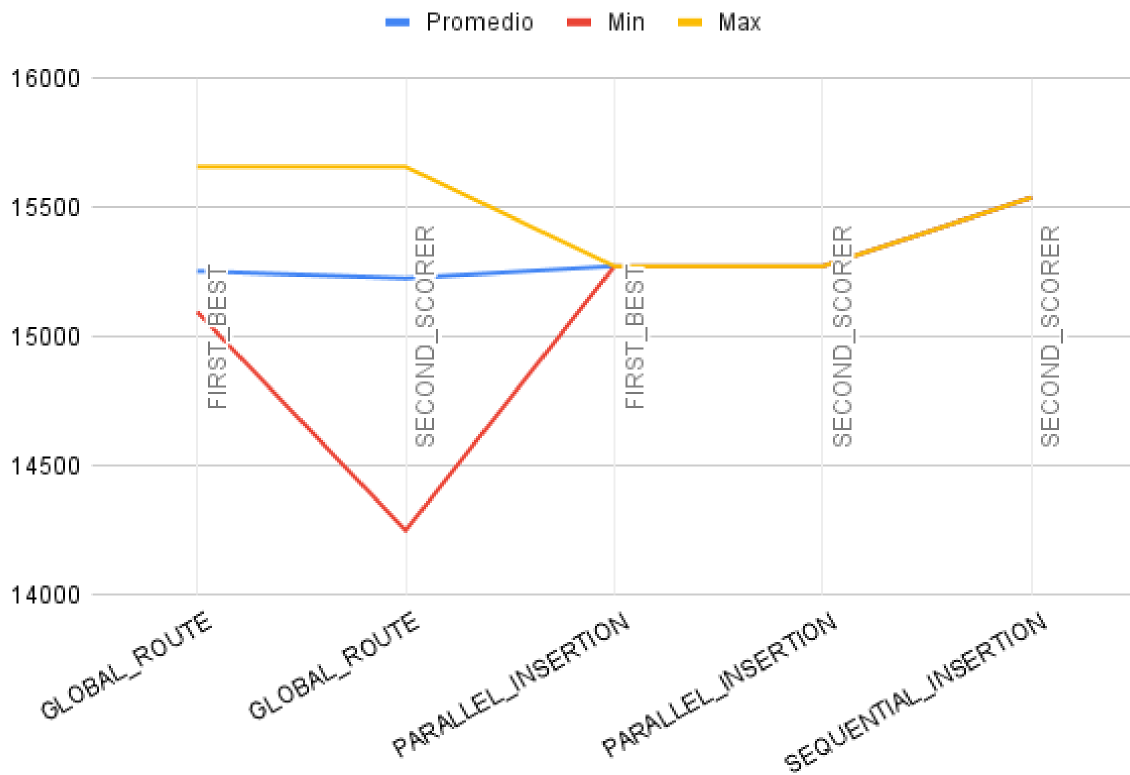

Fig. 5.1: Comparación de soluciones para la wave de 100 órdenes.

route_type	initial_batcher_type	stock_selector	percentage
LARGEST_GAP	GLOBAL_ROUTE	SECOND_SCORER	91.4 %
LARGEST_GAP	GLOBAL_ROUTE	FIRST_BEST	8 %
LARGEST_GAP	PARALLEL_INSERTION	SECOND_SCORER	0.44 %
LARGEST_GAP	SEQUENTIAL_INSERTION	SECOND_SCORER	0.12 %
LARGEST_GAP	PARALLEL_INSERTION	FIRST_BEST	0.04 %

Tab. 5.3: Porcentaje de soluciones para la wave de 100 órdenes.

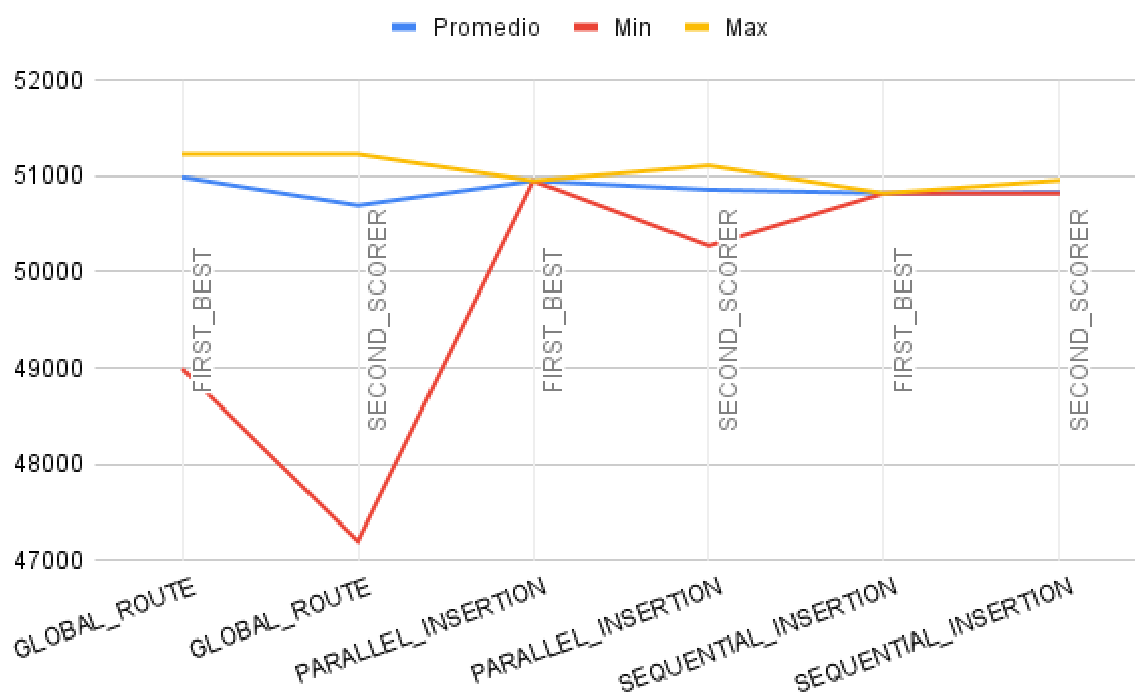

Fig. 5.2: Comparación de soluciones para la wave de 250 órdenes.

Siguiendo con la agrupación aplicada a la tabla, en la Tabla **??** realizamos una comparación que busca mostrar fácilmente el peso que tiene cada una de las combinaciones de criterios mostradas en la Tabla **??** en los primeros 5000 casos. Este peso se muestra en porcentajes. Así, la agrupación de criterios mencionada en el párrafo anterior, que agrupa 4570 combinaciones de las 5000 mejores, tiene un porcentaje de 91.4 % mientras que la combinación $route_type = LARGEST_GAP$, $initial_batcher_type = PARALLEL_INSERTION$ y $stock_selector = FIRST_BEST$ solo tiene 2 combinaciones, por lo que su porcentaje de presencia es de 0.04. Esta agrupación es independiente de las distancias. Se puede ver claramente que la combinación que obtuvo la menor distancia es por amplia diferencia la que tiene más presencia en los 5000 mejores, lo que respalda la elección como realmente buena en lugar de ser una coincidencia para esta wave particular.

En la Figura **??** tomamos el promedio de las soluciones encontradas agrupando de acuerdo al mismo criterio que en el caso anterior. Vemos que incluso en el caso promedio, la combinación más frecuente es mejor que las demás combinaciones.

route_type	initial_batcher_type	stock_selector	total_distance	quantity
LARGEST_GAP	GLOBAL_ROUTE	FIRST_BEST	48991	40
LARGEST_GAP	GLOBAL_ROUTE	FIRST_BEST	51063	280
LARGEST_GAP	GLOBAL_ROUTE	FIRST_BEST	51130	20
LARGEST_GAP	GLOBAL_ROUTE	FIRST_BEST	51229	125
LARGEST_GAP	GLOBAL_ROUTE	SECOND_SCORER	47196	40
LARGEST_GAP	GLOBAL_ROUTE	SECOND_SCORER	48991	440
LARGEST_GAP	GLOBAL_ROUTE	SECOND_SCORER	49260	40
LARGEST_GAP	GLOBAL_ROUTE	SECOND_SCORER	49493	40
LARGEST_GAP	GLOBAL_ROUTE	SECOND_SCORER	49744	120
LARGEST_GAP	GLOBAL_ROUTE	SECOND_SCORER	50026	20
LARGEST_GAP	GLOBAL_ROUTE	SECOND_SCORER	50030	120
LARGEST_GAP	GLOBAL_ROUTE	SECOND_SCORER	50031	240
LARGEST_GAP	GLOBAL_ROUTE	SECOND_SCORER	50312	20
LARGEST_GAP	GLOBAL_ROUTE	SECOND_SCORER	50631	40
LARGEST_GAP	GLOBAL_ROUTE	SECOND_SCORER	51063	1120
LARGEST_GAP	GLOBAL_ROUTE	SECOND_SCORER	51102	240
LARGEST_GAP	GLOBAL_ROUTE	SECOND_SCORER	51104	240
LARGEST_GAP	GLOBAL_ROUTE	SECOND_SCORER	51130	220
LARGEST_GAP	GLOBAL_ROUTE	SECOND_SCORER	51229	1405
LARGEST_GAP	PARALLEL_INSERTION	FIRST_BEST	50953	2
LARGEST_GAP	PARALLEL_INSERTION	SECOND_SCORER	50277	4
LARGEST_GAP	PARALLEL_INSERTION	SECOND_SCORER	50425	2
LARGEST_GAP	PARALLEL_INSERTION	SECOND_SCORER	50953	20
LARGEST_GAP	PARALLEL_INSERTION	SECOND_SCORER	51090	4
LARGEST_GAP	PARALLEL_INSERTION	SECOND_SCORER	51111	2
LARGEST_GAP	SEQUENTIAL_INSERTION	FIRST_BEST	50825	12
LARGEST_GAP	SEQUENTIAL_INSERTION	SECOND_SCORER	50825	132
LARGEST_GAP	SEQUENTIAL_INSERTION	SECOND_SCORER	50960	12

Tab. 5.4: Caso de 250 órdenes.

route_type	initial_batcher_type	stock_selector	percentage
LARGEST_GAP	GLOBAL_ROUTE	SECOND_SCORER	84.6 %
LARGEST_GAP	GLOBAL_ROUTE	FIRST_BEST	11.6 %
LARGEST_GAP	SEQUENTIAL_INSERTION	SECOND_SCORER	2.88 %
LARGEST_GAP	PARALLEL_INSERTION	SECOND_SCORER	0.64 %
LARGEST_GAP	SEQUENTIAL_INSERTION	FIRST_BEST	0.24 %
LARGEST_GAP	PARALLEL_INSERTION	FIRST_BEST	0.04 %

Tab. 5.5: Porcentaje de soluciones para la wave de 250 órdenes.

route_type	initial_batcher_type	stock_selector	total_distance	quantity
AISLE_BY_AISLE	GLOBAL_ROUTE	SECOND_SCORER	97698	20
AISLE_BY_AISLE	GLOBAL_ROUTE	SECOND_SCORER	97998	20
AISLE_BY_AISLE	GLOBAL_ROUTE	SECOND_SCORER	102002	40
AISLE_BY_AISLE	GLOBAL_ROUTE	SECOND_SCORER	102067	40
AISLE_BY_AISLE	GLOBAL_ROUTE	SECOND_SCORER	102840	40
AISLE_BY_AISLE	GLOBAL_ROUTE	SECOND_SCORER	102867	40
AISLE_BY_AISLE	GLOBAL_ROUTE	SECOND_SCORER	102895	240
AISLE_BY_AISLE	GLOBAL_ROUTE	SECOND_SCORER	103191	20
AISLE_BY_AISLE	GLOBAL_ROUTE	SECOND_SCORER	103240	240
AISLE_BY_AISLE	GLOBAL_ROUTE	SECOND_SCORER	103475	20
AISLE_BY_AISLE	GLOBAL_ROUTE	SECOND_SCORER	103498	200
LARGEST_GAP	GLOBAL_ROUTE	FIRST_BEST	102081	240
LARGEST_GAP	GLOBAL_ROUTE	FIRST_BEST	102514	40
LARGEST_GAP	GLOBAL_ROUTE	FIRST_BEST	102532	40
LARGEST_GAP	GLOBAL_ROUTE	FIRST_BEST	103195	40
LARGEST_GAP	GLOBAL_ROUTE	RANDOM	103277	20
LARGEST_GAP	GLOBAL_ROUTE	RANDOM	103488	20
LARGEST_GAP	GLOBAL_ROUTE	SECOND_SCORER	94182	40
LARGEST_GAP	GLOBAL_ROUTE	SECOND_SCORER	96514	120
LARGEST_GAP	GLOBAL_ROUTE	SECOND_SCORER	96716	120
LARGEST_GAP	GLOBAL_ROUTE	SECOND_SCORER	97470	20
LARGEST_GAP	GLOBAL_ROUTE	SECOND_SCORER	97623	20
LARGEST_GAP	GLOBAL_ROUTE	SECOND_SCORER	98400	240
LARGEST_GAP	GLOBAL_ROUTE	SECOND_SCORER	99565	240
LARGEST_GAP	GLOBAL_ROUTE	SECOND_SCORER	99583	80
LARGEST_GAP	GLOBAL_ROUTE	SECOND_SCORER	100172	40
LARGEST_GAP	GLOBAL_ROUTE	SECOND_SCORER	100764	240
LARGEST_GAP	GLOBAL_ROUTE	SECOND_SCORER	101080	40
LARGEST_GAP	GLOBAL_ROUTE	SECOND_SCORER	101682	40
LARGEST_GAP	GLOBAL_ROUTE	SECOND_SCORER	101802	40
LARGEST_GAP	GLOBAL_ROUTE	SECOND_SCORER	101893	40
LARGEST_GAP	GLOBAL_ROUTE	SECOND_SCORER	101900	40
LARGEST_GAP	GLOBAL_ROUTE	SECOND_SCORER	102081	960
LARGEST_GAP	GLOBAL_ROUTE	SECOND_SCORER	102099	240
LARGEST_GAP	GLOBAL_ROUTE	SECOND_SCORER	102514	160
LARGEST_GAP	GLOBAL_ROUTE	SECOND_SCORER	102532	440
LARGEST_GAP	GLOBAL_ROUTE	SECOND_SCORER	102782	20
LARGEST_GAP	GLOBAL_ROUTE	SECOND_SCORER	102792	20
LARGEST_GAP	GLOBAL_ROUTE	SECOND_SCORER	102863	40
LARGEST_GAP	GLOBAL_ROUTE	SECOND_SCORER	102940	240
LARGEST_GAP	GLOBAL_ROUTE	SECOND_SCORER	103195	160
LARGEST_GAP	GLOBAL_ROUTE	SECOND_SCORER	103376	20
LARGEST_GAP	GLOBAL_ROUTE	SECOND_SCORER	103386	20

Tab. 5.6: Caso de 500 órdenes.

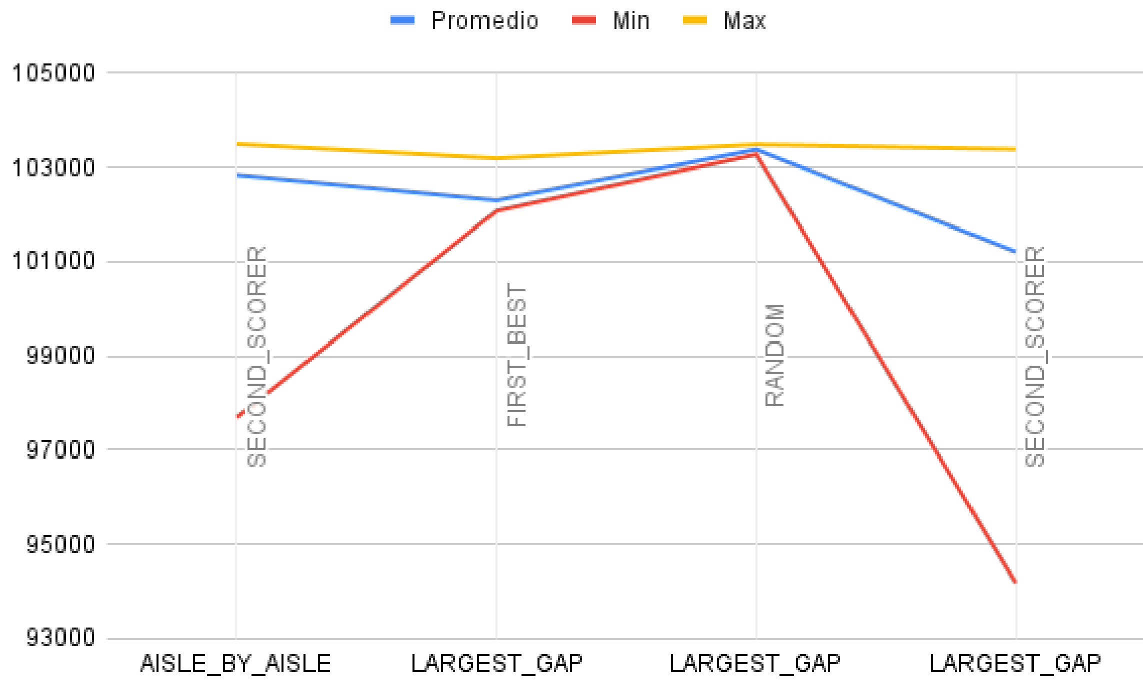

Fig. 5.3: Comparación de soluciones para la wave de 500 órdenes.

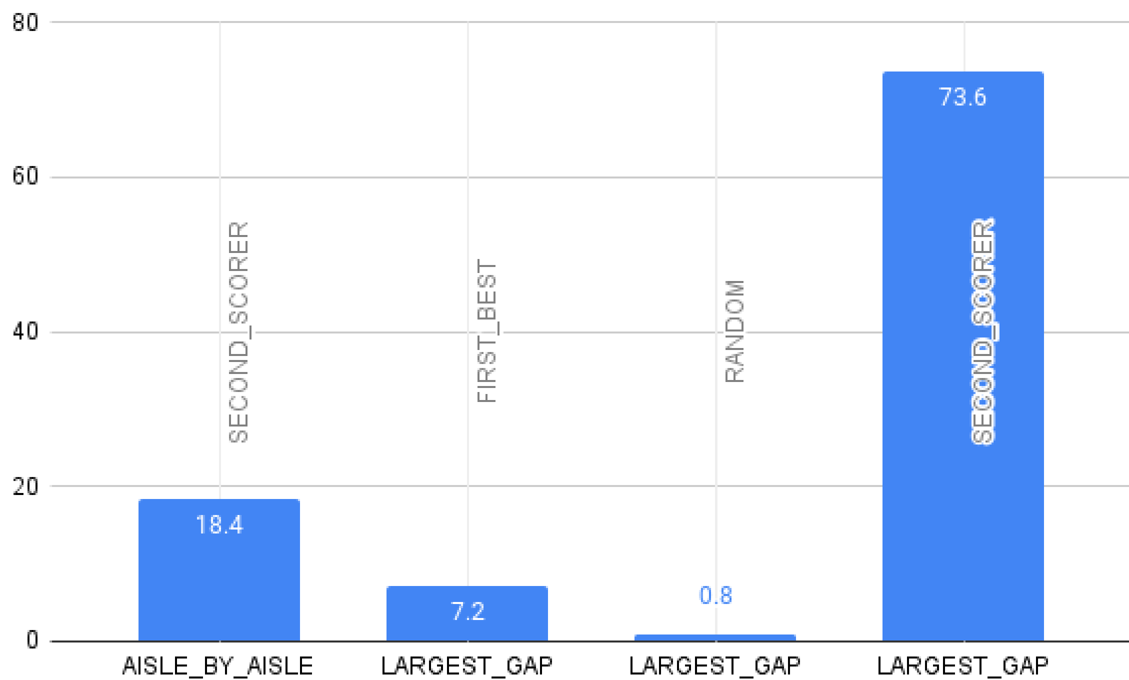

Fig. 5.4: Porcentaje de soluciones para la wave de 500 órdenes.

route_type	initial_batcher_type	stock_selector	percentage
LARGEST_GAP	GLOBAL_ROUTE	SECOND_SCORER	73.6 %
AISLE_BY_AISLE	GLOBAL_ROUTE	SECOND_SCORER	18.4 %
LARGEST_GAP	GLOBAL_ROUTE	FIRST_BEST	7.2 %
LARGEST_GAP	GLOBAL_ROUTE	RANDOM	0.8 %

Tab. 5.7: Porcentaje de soluciones para la wave de 500 órdenes.

Al analizar los demás casos vemos que esta tendencia se replica para waves mayores. En un análisis más detallado de los casos observamos que la mejor solución siempre se da con la misma combinación de criterios excepto uno: el primer criterio de selección de órdenes del selector de stock. Podemos ver cómo la tendencia de la Figura **??** y la Tabla **??** se replica para los casos de 250 órdenes en la Figura **??** y la Tabla **??** y 500 órdenes en la Figura **??** y la Tabla **??**, aunque en este último con menos peso para la combinación de criterios de stock, solución inicial y política de enrutamiento preferidos.

En los casos de 250 órdenes, en la Figura **??** y la Tabla **??**, y 500 órdenes, en la Figura **??** y la Tabla **??**, el primer criterio de selección de stock usado es LARGEST_DISTANCE, aquel que minimiza la distancia máxima de la nueva ubicación candidata a todas las demás ubicaciones ya pertenecientes a la ruta, y para el caso de 100 órdenes el criterio de scoring ganador es AVERAGE_DISTANCE, que minimiza el promedio de esas distancias. En ambos casos se tiene como segundo criterio la selección aleatoria para variar la muestra. Esta diferencia se puede deber a algún overfitting para el caso de pocas órdenes. También observamos que variaciones de refinadores y selección de semillas no generaban soluciones distintas para esta combinación de criterios, motivo por el cual hay 40 combinaciones que tienen la mejor distancia encontrada.

Cabe analizar en este punto la elección de la mejor ruta disponible. En el caso de la empresa de e-commerce considerada en esta tesis, rutas como Largest Gap no suelen ser deseables ya que el costo de cambiar de un cross aisle a un aisle, y viceversa, es alto. Sin embargo, como en este estudio usamos como resolución la distancia de las rutas, no nos es posible asignar un costo específico a las maniobras necesarias para realizar un cambio de calle. Podríamos lograrlo si usáramos como métrica, por ejemplo, el tiempo que lleva realizar una ruta; aunque esto plantea nuevos interrogantes: ¿cómo modelamos el tiempo de desplazamiento de una ubicación a otra? ¿deberíamos considerar el tiempo de búsqueda de un ítem en la ubicación como despreciable? ¿y si fueran muchos ítems? ¿El tiempo de desplazamiento depende del área, del piso, de la carga que lleva el carrito, de la experiencia que tenga el operario en el depósito? Todas estas, y muchas otras semejantes, son preguntas válidas para hacernos a la hora de modelar el problema y escapan al alcance de esta tesis.

Para poder verificar la elección del tipo de ruta, corrimos este mismo experimento en depósitos reales con distintas configuraciones físicas, y comprobamos que la mejor ruta es dependiente de la configuración del depósito. También con esto pudimos comprobar que para nuestro problema, los algoritmos que dan la mejor solución no varían, solo varía la política de ruta, siendo siempre los mejores la solución inicial: $GLOBAL_ROUTE$ y el selector de stock $SECOND_SCORER$, usando como primer criterio de selección de ubicaciones el que minimiza el promedio de esas distancias y después la selección random.

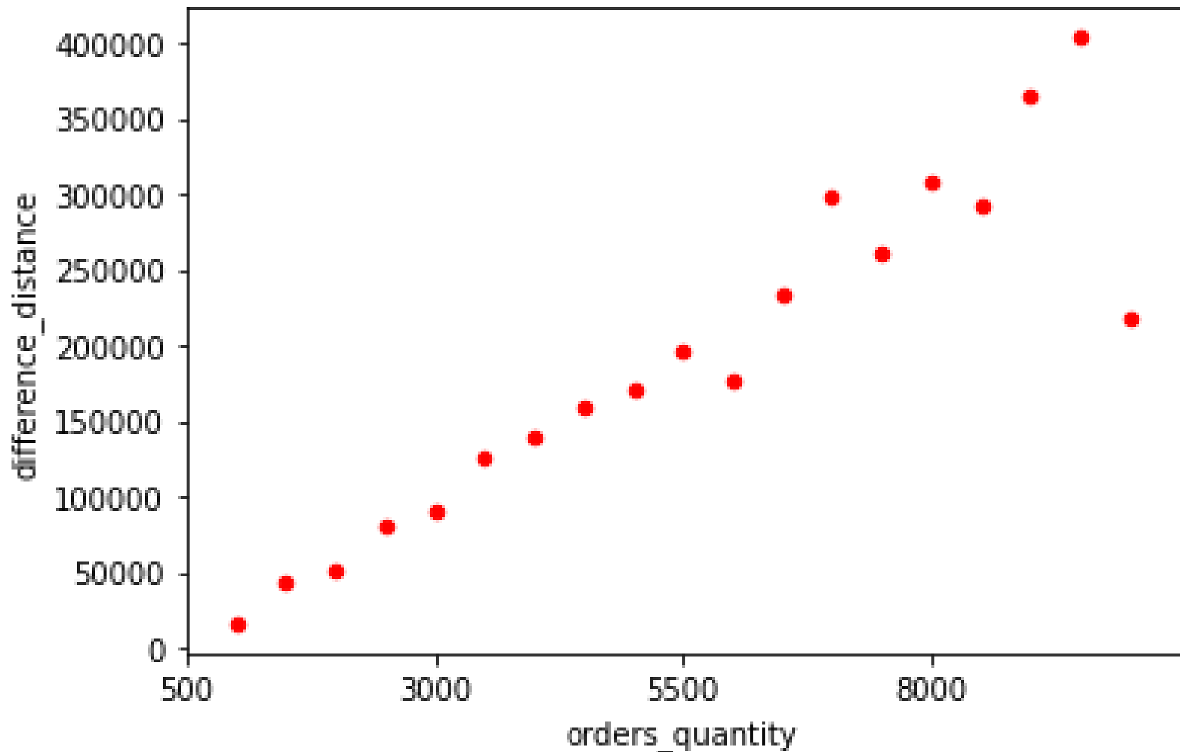

Fig. 5.5: Comparación entre mejores soluciones.

5.4.2. Caso inicial: Mejor combinación

Dado que encontramos dos conjuntos de criterios, muy similares pero con una pequeña diferencia, que alcanzaron la mejor solución encontrada en el experimento anterior, decidimos compararlos entre sí usando instancias más grandes para determinar cuál es la mejor opción. Por simplicidad, vamos a llamar al conjunto de parámetros que obtuvo la mejor solución para la wave de 100 órdenes como criterio A. Al conjunto de parámetros que obtuvo la mejor solución para en las waves de 250 y 500 órdenes lo llamaremos criterio B. Evaluamos ambos criterios para un conjunto de waves que van desde las 500 órdenes, a las 10 K órdenes, aumentando el tamaño entre las instancias en 500 órdenes.

En la Figura **??** se muestra la comparación entre las soluciones obtenidas por el criterio A y el criterio B. En este gráfico lo que se representa es la diferencia entre la distancia total de la solución generada por el criterio A, menos la solución generada por el criterio B. Como vemos en el gráfico, la diferencia siempre es positiva, es decir que la rutas generadas por el criterio B en todos los casos son más cortas que las generadas por el criterio A. Por este motivo definimos el criterio B, conjunto de parámetros que obtuvo la mejor solución para en las waves de 250 y 500 órdenes, como el mejor conjunto de parámetros global y vamos a continuar nuestro análisis sobre ese conjunto.

5.4.3. Caso general: Análisis del tiempo de ejecución

La mejor combinación encontrada en el análisis hasta ahora es, detalladamente, aquella que utiliza los criterios

- *route_type* = $LARGEST_GAP$: una política de ruteo por bloques, donde ingresa a cada calle hasta el "espacio mas grande", donde definimos un "espacio" como la distancia entre dos artículos adyacentes o entre una calle transversal y el artículo más cercano.
- *initial_batcher_type* = $GLOBAL_ROUTE$: una solución inicial basada en crear una ruta con todos los ítems a recuperar y después dividirla en sub rutas más pequeñas.
- *stock_selector* = $SECOND_SCORER$: un selector de stock que busca la mejor ubicación dado dos criterio de calidad, el segundo desempatando si hay colisión en el primer criterio.

Ademas, observamos que las combinaciones de refinadores no aportaron a la solución, probablemente debido a que los criterios de los refinadores son análogos a la solución inicial $GLOBAL_ROUTE$, por lo tanto dejaremos el algoritmo sin refinadores.

La selección de semilla no aporta en el caso de $GLOBAL_ROUTE$ ya que las rutas nuevas se inician por falta de capacidad en el carrito y las ubicaciones ya están determinadas. El orden en que se evalúan las órdenes para agregarlas a la solución es

- *sorter_type* = $NEAREST_TO_EXIT$: que prioriza las órdenes por cercanía al punto de fin.

Además debemos mencionar los criterios que usa el selector de stock para priorizar las órdenes. Estos son

- *scorer* = $LARGEST_DISTANCE$: que se evalúa primero, minimizando la mayor de las distancias desde la nueva ubicación candidata a todas las demás ubicaciones ya pertenecientes a la ruta.
- *scorer* = $RANDOM$: como segundo criterio usa para desempatar una selección aleatoria.

El hecho de usar como segundo selector el criterio aleatorio difiere del selector de stock mono criterio establecido en $LARGEST_DISTANCE$ en el hecho de que ante un empate el selector mono criterio devuelve siempre el primero, mientras que al tener el segundo criterio establecido aleatoriamente, nos permite mayor variabilidad, y por lo tanto mayor ahorro de distancia, en la búsqueda del mejor candidato a agregar.

Definida nuestra mejor combinación de criterios, ahora buscamos analizar cómo se comporta en la práctica. Definimos dos experimentos diferentes para este análisis. En el primero creamos 20 instancias diferentes, fijando la cantidad máxima de ubicaciones por SKU en 10, variando la cantidad de órdenes de a 500, empezando con 500 órdenes y finalizando con 10K. En el segundo creamos 200 instancias diferentes, fijando la cantidad de órdenes en 1000, y variando la cantidad de ubicaciones por SKU de a 5, iniciando en 5 y finalizando en 1000 ubicaciones por SKU. Para ambos casos un mismo SKU no puede repetirse más de 20 veces en las órdenes, y las dimensiones y peso asignados a cada SKU oscilan aleatoriamente en los mismos rangos; cada lado de los paquetes se tomó como $20 \pm 5cm$ y el peso como $300 \pm 200gr$. Tanto la cantidad máxima de ubicaciones diferentes por SKU como la cantidad máxima de órdenes analizadas se definió como cota del problema, por lo que vamos a analizar el crecimiento del problema hasta las mismas. Cabe destacar que lo que se espera en la práctica es un caso promedio, donde no todas las

Fig. 5.6: Crecimiento del tiempo de ejecución respecto a la cantidad de órdenes.

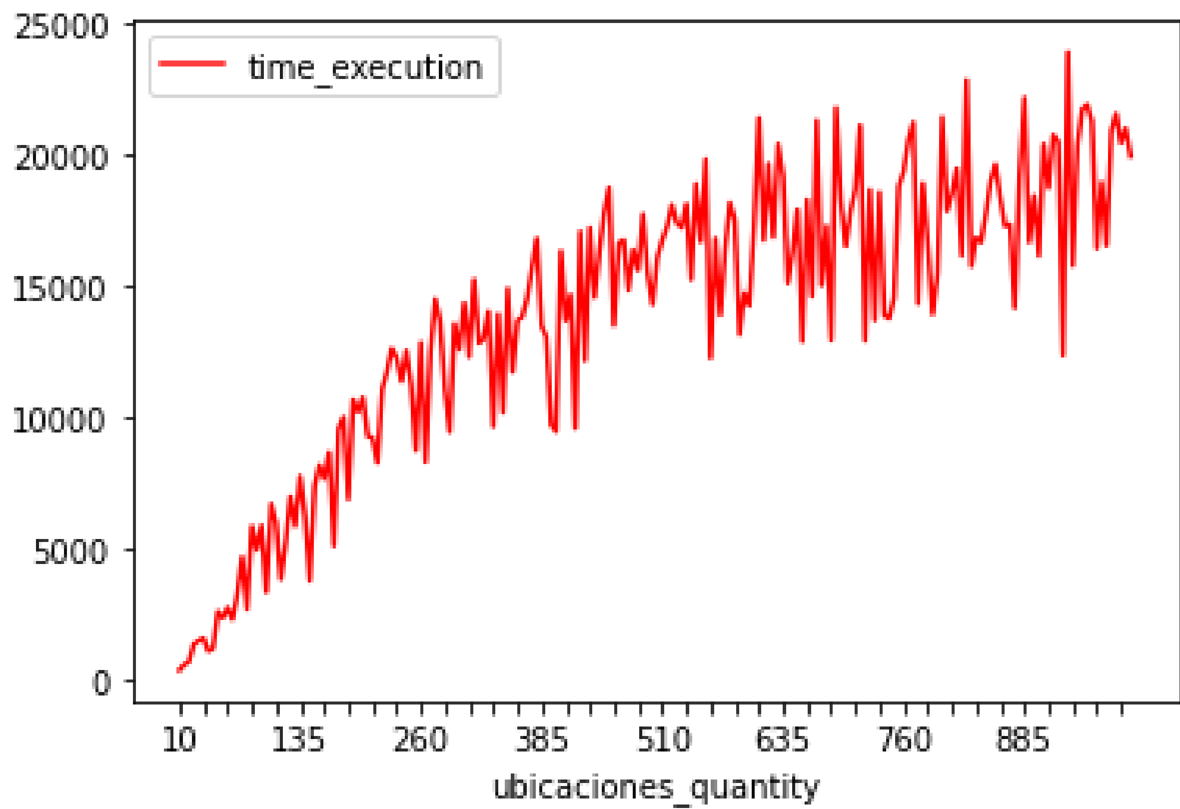

Fig. 5.7: Crecimiento del tiempo de ejecución respecto a la cantidad de ubicaciones.

waves que vamos a resolver van a tener 10K órdenes diferentes, ni todos los SKU dentro de la misma van a tener 1000 ubicaciones distintas.

En la Figura **??** y en la Figura **??** vemos el tiempo de ejecución del algoritmo para las instancias descritas en el párrafo anterior. La Figura **??** alcanza su máximo en 11.7 minutos aproximadamente, mientras que el valor máximo de la Figura **??** es de 25 segundos. Ambas figuras comienzan desde el mismo caso inicial, 1000 órdenes con 10 ubicaciones máximas por SKU tardando poco menos de 0.5 segundos. La siguiente intersección se da en 1500 órdenes con 10 ubicaciones máximas, y 1000 órdenes con 30 ubicaciones máximas. La siguiente se da en 2000 órdenes con 10 ubicaciones y 1000 órdenes con aproximadamente 100 ubicaciones máximas.

Realizamos un análisis de complejidad de nuestra mejor combinación, es decir, la solución inicial $GLOBAL_ROUTE$ descripta en el Algoritmo **??**. El ciclo planteado en la línea 5 es el predominante de la complejidad, ya que se repite N veces, con N la cantidad de órdenes, y el costo de las líneas internas en $O(NK)$, con K la cantidad de ubicaciones por SKU. Esto se debe a que para seleccionar la nueva ubicación se evalúa el scorer para todas las ubicaciones disponibles del SKU, el cual usa además todas las ubicaciones que están ya en la ruta, que son N. Para más detalles ver el Algoritmo **??**. Luego se genera la ruta global, cuya complejidad se mide como la cantidad de calles ocupadas por bloque, pero puede ser acotada superiormente por N. Finalmente se generan las rutas individuales para lo cual se itera sobre todos los puntos de la ruta global, que también son N. Finalmente, la complejidad total del algoritmo heurístico con la mejor combinación de parámetros es $O(N^2K)$. Esto se corresponde con la Figura **??** que tiene forma de parábola. Por otro lado en la Figura **??** no vemos un comportamiento tan lineal, sin embargo cabe notar que en este caso N es más grande que K, salvo en la ultima instancia que son iguales, lo que complica evaluar la tendencia pero aun así es posible notar la tendencia creciente de los datos.

En Weidinger et al. [**?**] se discute sobre el impacto de la diversidad de ubicaciones en la distancia de las rutas, por lo que no analizaremos este caso puntualmente.

5.5. Instancias de prueba dadas por casos reales

Para las pruebas en casos reales tomamos 18 waves de (aprox.) 10K órdenes cada una. En la Tabla **??** tenemos los datos de cada instancia. Cada fila se corresponde con una instancia diferente. La primera muestra un identificador para distinguirlas. La siguiente columna muestra la cantidad de ítems requeridos, incluyendo las repeticiones del mismo SKU. La tercera columna muestra la cantidad de SKUs diferentes en la wave. La siguiente, ips o ítems per SKU, muestra el promedio de ítems requeridos por SKU. La quinta columna tiene el percentil 75 de la cantidad de ítems requeridos por SKU, y la siguiente, el máximo de ítems requeridos por SKU. Las siguientes columnas muestran datos de las ubicaciones. En la columna locs se muestra la cantidad total de ubicaciones con repetidos pues se suman todas las ubicaciones disponibles para cada SKU. En la columna lnr por el contrario se eliminan los repetidos. En las siguientes columnas tenemos el mínimo de ubicaciones por SKU, el percentil 25, el percentil 75 y la mayor cantidad de ubicaciones que aparecen para un mismo SKU, respectivamente.

En la Tabla **??** se ven algunos datos de las soluciones de estas waves. En las primeras dos columnas repetimos el identificador, y agregamos nuevamente la cantidad de ítems de manera ilustrativa. La tercera columna tiene la cantidad de rutas usadas para recuperar

id	ítems	skus	ips	75	max	locs	lnr	lps	min	25	75	max
reg-0_0	10168	8510	1.195	1	18	20170	1782	2.342	1	2.342	2.342	22
reg-1_1	10163	8601	1.182	1	43	28796	1935	2.710	1	2.710	2.710	25
reg-1_2	10156	8496	1.195	1	43	29908	1936	2.824	1	2.824	2.824	25
reg-0_3	10744	9105	1.180	1	18	21868	1793	2.375	1	2.375	2.375	22
reg-3_4	10002	7524	1.329	1	38	44934	1482	2.466	1	2.466	2.466	23
reg-0_6	10455	8242	1.269	1	18	21311	1731	2.539	1	2.539	2.539	22
reg-2_7	10030	9456	1.061	1	15	41090	1498	2.170	1	2.170	2.170	26
reg-1_9	10058	8449	1.190	1	48	31187	1938	2.987	1	2.987	2.987	25
reg-0_10	10148	8693	1.167	1	18	21499	1782	2.441	1	2.441	2.441	22
reg-1_12	10723	9421	1.138	1	23	34726	1940	2.950	1	2.950	2.950	25
reg-0_13	10064	8881	1.133	1	18	23180	1783	2.566	1	2.566	2.566	22
reg-3_14	10098	8253	1.224	1	29	55622	1480	2.632	1	2.632	2.632	23
reg-1_15	10330	8835	1.169	1	44	33723	1941	3.045	1	3.045	3.045	25
reg-0_17	10272	8575	1.198	1	19	21556	1773	2.485	1	2.485	2.485	22
reg-0_19	10059	7761	1.296	1	13	19901	1738	2.518	1	2.518	2.518	22
reg-1_20	10032	8854	1.133	1	21	32585	1937	2.908	1	2.908	2.908	25
reg-2_21	10102	9555	1.057	1	20	41700	1500	2.205	1	2.205	2.205	26
reg-0_22	10228	8498	1.204	1	19	21038	1796	2.444	1	2.444	2.444	22

Tab. 5.8: Datos estadísticos de stock sobre de las instancias de prueba.

todos los ítems necesarios. En la cuarta ruta tenemos la suma de las distancias de todas las rutas que conforman la solución de cada wave. Finalmente, en la quinta columna tenemos la cantidad promedio de *checkpoints* que tiene la ruta, donde un *checkpoint* se corresponde a un par $(calle, estanteria)$. Se ve a simple vista que en nuestra solución, las rutas aprovechan para tomar múltiples ítems de la misma estantería.

En la Figura **??** podemos ver en azul el tiempo de ejecución de cada caso mientras que en rojo se marca el promedio de los tiempos, el cual se encuentra en 5.209 minutos. Si bien tenemos casos que superan la cota de 5 minutos de tiempo de ejecución, notamos al analizar los casos que estas waves son las que tienen un stock de mayor tamaño, superando el promedio de 10 ubicaciones por SKU.

Se puede evaluar a futuro alternativas para mejorar este tiempo de cómputo en dicho caso. Una de las opciones es tomar ubicaciones cercanas y usar un representante general para todas esas ubicaciones en el algoritmo inicial. Luego las ubicaciones finales se pueden decidir en una etapa posterior. Hablaremos más de estas ideas en el Capítulo 6.

id	ítems	quantity_routes	total_distance	checkpoints
reg-0_0	10168	576	115471	3.34
reg-1_1	10163	986	1192263	2.52
reg-1_2	10156	1310	1578118	2.12
reg-0_3	10744	695	142571	2.92
reg-3_4	10002	742	751369	2.63
reg-0_6	10455	804	148204	2.52
reg-2_7	10030	1171	1183914	2.08
reg-1_9	10058	2122	2552951	1.60
reg-0_10	10148	798	162271	4.69
reg-1_12	10723	1723	2069233	3.86
reg-0_13	10064	791	154344	4.71
reg-3_14	10098	1398	1412610	3.82
reg-1_15	10330	1775	2137228	3.77
reg-0_17	10272	759	155520	4.75
reg-0_19	10165	640	111384	4.06
reg-1_20	10059	1706	2053502	5.04
reg-2_21	10032	1276	1289692	3.86
reg-0_22	10102	807	158670	3.98

Tab. 5.9: Datos estadísticos de la solución sobre de las instancias de prueba.

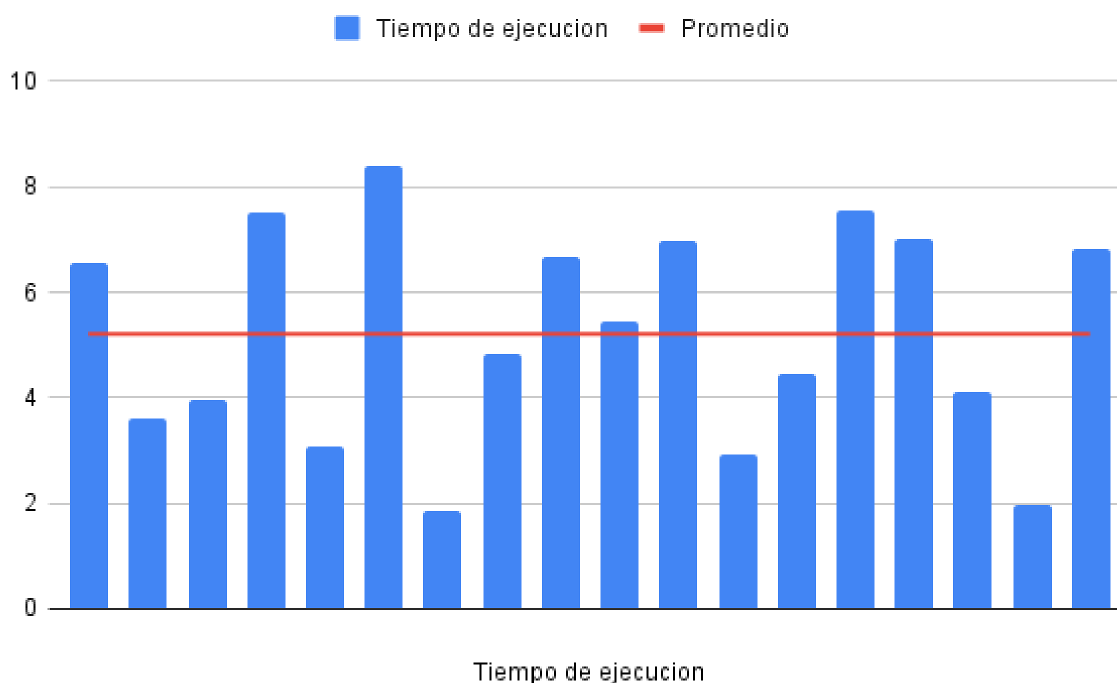

Fig. 5.8: Tiempo de ejecución de waves reales en minutos.

6. CONCLUSIONES Y TRABAJO FUTURO

Es muy peligroso, Frodo, cruzar la puerta. Vas hacia el camino y si no cuidas tus pasos, no sabes hacia dónde te arrastrarán.
–El señor de los anillos. J. R. R. Tolkien

La recuperación de órdenes en los depósitos es una tarea de planificación importante en todos los sistemas manuales de almacenamiento de ítems de los grandes e-commerce. Caminar por el almacén para recuperar los ítems solicitados de las ubicaciones de almacenamiento consume una cantidad significativa del tiempo de trabajo de un operario en el depósito. Para reducir el tiempo de viaje, durante las últimas décadas se han propuesto en la literatura diversos algoritmos de asignación a conjuntos de ítems y varias políticas de enrutamiento para los preparadores de pedidos. Agregamos a este modelo la condición de múltiples ubicaciones por ítems y presentamos diversas cotas de tiempo y recursos necesarias para operar en depósitos reales en la industria.

Observamos cómo el problema puede ser adaptado a un modelo de programación lineal entera. Realizamos una exploración en el tipo de instancias que nuestro modelo podría resolver pero nos vimos rápidamente limitados por la memoria consumida en la resolución del modelo.

Presentamos soluciones heurísticas capaces de resolver instancias de tamaño real en unos pocos minutos. En casos de pocas órdenes donde el tiempo de ejecución se reduce a pocos milisegundos se vuelve más atractiva la idea de utilizar el tiempo extra en refinamientos para mejorar la distancia global del algoritmo. Por el contrario, en casos de mayor cantidad de órdenes la optimización se vuelve simple porque tenemos muchas ubicaciones disponibles para ir a buscar ítems, por el simple hecho de que tenemos que buscar una gran cantidad de ítems. Esto hace que siempre sea posible elegir una ubicación próxima para continuar la ruta, llegando en los casos de 10K a completar la capacidad del carro en una o dos calles como máximo. En estos últimos casos, la dificultad se encuentra en lograr manejar un gran volumen de datos de manera óptima para cumplir con la cota temporal de 5 minutos.

Detectamos que las métricas usadas para medir la calidad de las soluciones tiene un alto impacto en la política de ruteo ganadora, así como la configuración física del depósito en el que se realicen las pruebas.

Los algoritmos utilizados pese a ser heurísticas golosas dan excelentes soluciones aprovechando al máximo la capacidad de los carros en tramos muy cortos del depósito, causando así que los refinadores no aporten ya que las rutas son muy eficientes desde la solución inicial. Así, podemos evitar iteraciones sobre las soluciones mejorando aun más la performance general del algoritmo para procesar un gran volumen de datos y cumplir con las estrictas cotas temporales, lo cual sería muy difícil de lograr con algoritmos del tipo exhaustivo.

Finalmente mostramos cómo la cantidad de órdenes impacta a la performance seguida de la cantidad de ubicaciones por ítem, lo que nos sugiere la pauta de que para resolver mas rápidamente instancias de gran cantidad de órdenes debemos enfocarnos en limitar la cantidad de ubicaciones evaluadas en el algoritmo.

Si bien este problema ha tenido una creciente atención en las últimas tres décadas, aún queda mucho por investigar en esta y otras variantes del mismo. Investigaciones futuras podrían tener en cuenta diversos aspectos.

Sobre el problema En esta tesis nos hemos limitado, por una cuestión de tiempo, a analizar uno de los casos de uso más comunes de los depósitos, el de las compras mono ítem, que en general son el 80 % de las compras realizadas en e-commerce. Sin embargo, el 60 % de los productos vendidos se despacha en órdenes multi ítem. Por esto vale la pena analizar cómo deberían expandirse estos algoritmos a órdenes con más de un ítem. Debe tenerse en cuenta que con la aparición de órdenes multi ítem aparece la necesidad de tener rutas que atraviesen más de un área, si se busca consolidar las órdenes en la ruta, o la necesidad de implementar una nueva etapa de optimización donde se consoliden las órdenes, para la cual se debe tener en cuenta la posibilidad de tener productos de la misma orden en distintas rutas, la diferencia de los tiempos de finalización de esas rutas, la necesidad de dedicar operarios a la tarea específica de consolidación, entre otras.

Sobre el modelo Otro elemento a tener en cuenta es la uniformidad del depósito asumida en este problema. Si bien puede pensarse que lo mas lógico es diseñar almacenes con bloques iguales o a lo sumo similares entre sí, éste no es siempre el caso. Muchas veces los directivos se ven forzados a sacrificar esta disposición por razones operativas. Por ejemplo, los depósitos en general no se construyen completamente desde el principio, por lo que puede ser que se vean forzados a tener distintos tipos de estanterías por cuestiones de distribuidores. Otro motivo es que si bien en general los almacenes suelen ser grandes rectángulos vacíos al principio, luego debemos destinar parte de ese espacio a cuestiones operativas como oficinas, un comedor de empleados, otros tipos de áreas tales como áreas de alta seguridad, área de objetos pesados, etc. Vale la pena dedicarle unos momentos a pensar cómo se podrían adaptar estos algoritmos a zonas de bloques, o incluso calles, no uniformes y la complejidad que eso acarrearía al modelado del depósito y la adaptación de las políticas de ruteo.

Sobre las heurísticas Hay muchas heurísticas que nos hubiera gustado probar pero que por lo acotado de este trabajo debimos excluir, aunque también tuvimos en cuenta la cota temporal de 5 minutos. Sin embargo, creemos que existen en la literatura algunas heurísticas que valdría la pena implementar de manera eficiente y evaluar si cumplen con este requerimiento temporal. Algunas de estas son savings [**?**], algoritmos genéticos [**?**] y tabú search [**?**]. Y también para los refinamientos valdría la pena probar, aunque sea acotando por un plazo de tiempo, un VNS [**?**] orientado con vecindades de cambio de ubicación por producto.

Sobre la optimización Se podría evaluar cómo aprovechar mejor las capacidades de los recursos que tenemos disponibles. Si bien en este caso encontramos claramente marcada una tendencia sobre cuál es la mejor combinación de parámetros para la heurística, la implementación de alguna de las ideas propuestas en el párrafo anterior, o incluso alguna completamente nueva, puede causar que encontremos un conjunto de heurísticas intrínsecamente diferentes pero que todas garanticen una buena calidad de la solución. En este caso se podría aprovechar el hecho de que tenemos múltiples cores en idle; ya que las

heurísticas planteadas son secuenciales (porque dependen de elecciones de ubicaciones anteriores) y no parece directo cómo paralelizarlas, podríamos hacer uso de esos cores para evaluar múltiples heurísticas en simultáneo y luego elegir la mejor solución entre todas las halladas. Otro elemento a tener en cuenta es que si bien las instancias pequeñas se resuelven en poco tiempo, el tiempo restante podría usarse para explorar otras soluciones, por ejemplo una búsqueda local guiada por vecindades. Es importante notar que esto se vuelve más necesario en las instancias pequeñas ya que al tener pocas órdenes que recuperar, las ubicaciones de sus SKUs pueden estar muy distantes entre sí por lo que una optimización más exhaustiva tiene oportunidad de generar una alta mejora. En cambio, con las waves más grandes, tenemos muchos SKUs y por lo tanto mucho muestreo de ubicaciones a visitar, por lo que es sensato pensar que cualquier heurística no trivial va a lograr una solución buena, porque tiene muchas opciones disponibles.

Sobre la elección de las mejores combinaciones de parámetros Finalmente, queremos destacar que aunque la metodología empleada en esta tesis resultó suficiente, puede haber casos donde la definición del mejor caso no es tan directo. Para estos casos sugerimos utilizar iRace, una herramienta que puede ser útil para utilizar en instancias pequeñas donde no quede del todo claro cuál es la mejor elección o no sea viable ejecutar todos los casos por la cantidad de combinaciones a elegir.

Bibliografía

[1] Ardjmand, E., Sanei Bajgiran, O. and Youssef, E. (2019) Using list-based simulated annealing and genetic algorithm for order batching and picker routing in put wall based picking systems. Applied Soft Computing Journal, 75, 106–119.

[2] Daniels, R. L., Rummel, J. L. and Schantz, R. (1998), A model for warehouse order picking. European Journal of Operational Research, 105, 1-17. doi:10.1016/S0377-2217(97)00043-X

[3] Elbert, R.M., Franzke, T., Glock, C.H. and Grosse, E.H. (2017), The effects of human behavior on the efficiency of routing policies in order picking: the case of route deviations. Comput. Ind. Eng. 111, 537–551.

[4] Gademann, N. and Velde, S. (2005), Order batching to minimize total travel time in a parallel-aisle warehouse. IIE Trans. 37, 63–75.

[5] Glock, C.H., Grosse, E.H., Elbert, R.M. and Franzke, T. (2017), Maverick picking: the impact of modifications in work schedules on manual order picking processes. International Journal of Production Research. 55, 6344–6360.

[6] Hall, R.W. (1993), Distance approximations for routing manual pickers in a warehouse. IIE Transactions 25, 76-87.

[7] Henn, S. and Wäscher, G. (2011), Tabu search heuristics for the order batching problem in manual order picking systems. European Journal of Operational Research, 222, 484–494

[8] De Koster, M. B. M., Van der Poort, E.S. and Wolters, M. (1999), Efficient order batching methods in warehouses. International Journal of Production Research, 37, 1479-1504, DOI: 10.1080/002075499191094

[9] Makusee M., Glock, C.H. and Grosse E.H. (2019), Order picker routing in warehouses: A systematic literature review. International Journal of Production Economics. 224, 107564

[10] Menéndez B, G. Pardo, E., Alonso-Ayuso, A., Molina, E. and Duarte, A. (2016), Variable Neighborhood Search strategies for the Order Batching Problem. Computers and Operations Research, http://dx.doi.org/10.1016/j.cor.2016.01.020i

[11] Ratliff, H.D., and Rosenthal, A.S. (1983), Order-picking in a rectangular warehouse: a solvable case of the traveling salesman problem. Operations Research 31, 507-521.

[12] Roodbergen, K.J. Routing order pickers in a warehouse. https://www.roodbergen.com/warehouse/background.php

[13] Roodbergen, K.J. and De Koster, R. (2001), Routing methods for warehouses with multiple cross aisles. International Journal of Production Research 39, 1865-1883.

[14] Roodbergen, K.J. and De Koster, R. (2001), Routing order pickers in a warehouse with a middle aisle. European Journal of Operational Research 133, 32-43.

[15] Theys, C., Bräysy, O., Dullaert, W. and Raa B. (2010), Using a TSP heuristic for routing order pickers in warehouses. European Journal of Operational Research, 200, 755–763.

[16] Tompkins, J.A., White, J.A., Bozer, Y.A. and Tanchoco, J.M.A. (2010), Facilities Planning. John Wiley & Sons.

[17] Van Gils, T., Ramaekers, K., Caris, A., and De Koster, R.B. (2018), Designing efficient order picking systems by combining planning problems: state-of-the-art classification and review. Eur. J. Oper. Res. 267 (1), 1–15.

[18] Vaughan, T.S. and Petersen, C.G. (1999), The effect of warehouse cross aisles on order picking efficiency. International Journal of Production Research 37(4), 881-897.

[19] Weidinger, F. (2018), Picker routing in rectangular mixed shelves warehouses. Computers & Operations Research, 95, 139–150.

[20] Weidinger, F., Boysen, N. and Schneider, M. (2019), Picker routing in the mixed-shelves warehouses of e-commerce retailers. European Journal of Operational Research, 274, 501–515.