



Universidad de Castilla-La Mancha
Escuela Superior de Ingeniería Informática

Trabajo Fin de Grado
Grado en Ingeniería Informática
Computación

**Algoritmos heurísticos para problemas de
recogida de pedidos en tienda**

Alejandro Fernández Arjona

Junio, 2021



TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Computación

Algoritmos heurísticos para problemas de recogida de pedidos en tienda

Autor: Alejandro Fernández Arjona

Tutor: Francisco Parreño Torres

Co-Tutor: María Teresa Alonso Martínez

Junio, 2021

Dedicado a mi familia y a mis amigos.

Declaración de autoría

Yo, Alejandro Fernández Arjona, con DNI 48259292W, declaro que soy el único autor del trabajo fin de grado titulado “*Algoritmos heurísticos para problemas de recogida de pedidos en tienda*”, que el citado trabajo no infringe las leyes en vigor sobre propiedad intelectual, y que todo el material no original contenido en dicho trabajo está apropiadamente atribuido a sus legítimos autores.

Albacete, a 16 de Junio de 2021

Fdo.: Alejandro Fernández Arjona

Resumen

Cada año aumentan las ventas online en nuestro país, en sectores como el ocio y el textil, pero sobre todo el de la alimentación. Este pasado 2020, debido a la pandemia del coronavirus, incrementaron tanto las compras online, que muchas empresas no estaban preparadas para ello y supuso incluso el fallo de grandes distribuidores.

Los supermercados y grandes almacenes quieren reinventarse y adaptarse a este nuevo estilo de mercado mejorando sus infraestructuras y los procesos que conlleva un envío, desde la recogida de los productos en el almacén, hasta la entrega al cliente.

En este proyecto vamos a estudiar distintas opciones para mejorar el proceso de recogida de los productos en grandes almacenes de empresas de alimentación. Los procesos a mejorar consisten en el *batching* de los pedidos, el espacio en las cajas de los pedidos y las rutas que los *pickers* deben seguir. Nuestro objetivo final es minimizar el número de horas empleadas en completar una serie de pedidos, lo que supone menos horas de trabajo y por tanto un beneficio económico para la empresa.

Agradecimientos

A mi familia, por ayudarme en la elección que hice hace cuatro años para acabar estudiando informática, y por apoyarme a pesar de no entender qué es lo que hacemos en prácticamente ninguna asignatura.

A mis amigos, por haber crecido juntos.

A todos los amigos que he hecho durante estos cuatro años, que serán mis amigos para toda la vida. Nos hemos apoyado tanto durante la carrera, que sin ellos no estaría donde estoy.

A mis tutores, por guiarme correctamente y ayudarme con el proyecto siempre que lo he necesitado.

Por último, a todos los profesores de la carrera. Aunque algunos nos caigan mejor que otros, al final todos nos ayudan a aprender unas cosas u otras.

Índice general

1	Introducción	1
1.1	Introducción	1
1.2	Motivación	4
1.3	Objetivos	5
1.4	Estructura del documento	6
2	Estado del arte y descripción del problema	7
2.1	Algoritmos de búsqueda	7
2.1.1	¿Qué es un algoritmo? (6)	7
2.1.2	Algoritmos voraces	8
2.1.3	Programación dinámica	9
2.1.4	Vuelta atrás (backtracking)	10
2.1.5	Algoritmos de búsqueda no informada	10
2.1.6	Algoritmos de búsqueda informada	10
2.1.7	Búsqueda con adversario	12
2.1.8	Agentes que aprenden	12
2.1.9	Optimización combinatoria	12
2.2	Definición de nuestro problema	14
2.3	Instancias	16
2.4	Subproblemas a resolver	18
2.4.1	Optimización de los paquetes en las cajas	18
2.4.2	El problema del camino más corto	21
2.4.3	Encontrar la ruta más rápida para cada picker	25
2.4.4	Asignación de los pedidos a los pickers	36
2.4.5	Colocación de los productos en las cajas	37

2.5	Situación actual	37
2.5.1	<i>Order picking</i>	37
2.5.2	<i>Wave picking (2)</i>	39
2.5.3	<i>Zone picking (3)</i>	40
2.5.4	<i>Batch picking (4)</i>	41
2.5.5	<i>Comparación de los distintos métodos</i>	42
3	Implementación	43
3.1	Interfaz gráfica	43
3.2	<i>Batching</i> de los pedidos.	44
3.2.1	<i>Algoritmo Genético</i>	45
3.2.2	<i>Búsqueda Tabu</i>	53
3.3	Optimización del número de cajas usadas para los pedidos	59
3.4	Cálculo de rutas	59
3.4.1	<i>Programación dinámica</i>	59
4	Experimentos.	63
4.1	Algoritmo genético.	63
4.2	Búsqueda tabú	69
4.3	Experimento final.	71
5	Conclusiones.	75
5.1	Conclusiones	75
5.2	Trabajo futuro	76
5.3	Competencias adquiridas	77
5.4	Probar código	79

Índice de figuras

1.1	Compras online tras el confinamiento de 2020	2
1.2	Europalets	4
1.3	Caja tamaño estándar	5
2.1	Problema del cambio de monedas	9
2.2	Problema de la mochila sin fraccionar	10
2.3	Problemas <i>backtracking</i>	11
2.4	Ejemplo de heurística	11
2.5	Almacén	15
2.6	2D-BPP	20
2.7	3D-BPP	21
2.8	Ejemplo de grafo	22
2.9	Solución con Dijkstra	22
2.10	Grafo con valores negativos	23
2.11	Solución con Bellman's Ford	23
2.12	Solución con Johnson	24
2.13	Esquema del algoritmo <i>Ant Colony</i>	27
2.14	Configuración de almacén	29
2.15	Solución óptima	30
2.16	Resultado mediante Programación Dinámica	31
2.17	Resultado mediante <i>S-shape</i>	32
2.18	Resultado mediante <i>Largest Gap</i>	34
2.19	Resultado mediante <i>Aisle by Aisle</i>	35
2.20	Ejemplo de almacén	38
2.21	Ejemplo de almacén con productos	38
2.22	Método <i>order picking</i>	39
2.23	Ejemplo de almacén con productos	39
2.24	Método <i>wave picking</i>	40
2.25	Ejemplo de almacén con productos	41
2.26	Método <i>zone picking</i>	41

2.27	Ejemplo de almacén con productos	42
2.28	Método <i>batch picking</i>	42
3.1	Interfaz Roodbergen	44
3.2	Pedido Roodbergen	45
3.3	Solución Roodbergen	46
3.4	Interfaz implementada	47
3.5	Solución en la interfaz implementada	48
3.6	Ejemplo de individuo	49
3.7	Padres operador de cruce	50
3.8	Primer hijo operador de cruce	51
3.9	Segundo hijo operador de cruce	51
3.10	Primer operador de mutación	52
3.11	Segundo operador de mutación	52
3.12	Estructura tabú inicial	55
3.13	Estructura tabú primera iteración	56
3.14	Estructura tabú segunda iteración	56
3.15	Estructura tabú tercera iteración	57
4.1	Fitness según distintos parámetros algoritmo genético	64
4.2	Fitness según distintos parámetros algoritmo genético	65
4.3	Fitness según distintas probabilidades de mutación algoritmo genético	66
4.4	Fitness medio según distintas probabilidades de mutación algoritmo genético	67
4.5	Comparación algoritmo genético con y sin hash	68
4.6	Comparación tiempos de ejecución medios algoritmo genético con y sin hash	68
4.7	Comparación búsqueda tabú con y sin hash	70
4.8	Comparación búsqueda tabú con y sin hash	70
4.9	Comparación búsqueda tabú con y sin hash mitad de vecinos	71
4.10	Comparación tiempos de ejecución medios búsqueda tabú con y sin hash mitad de vecinos	72
4.11	Comparación tiempos de ejecución entre las tres versiones	73
4.12	Comparación tiempos de las soluciones entre las tres versiones	73
4.13	Comparación media tiempos de las soluciones entre las tres versiones	74

1. Introducción

1.1. Introducción

Hoy en día los grandes almacenes (*warehouses*) tienen que ser competitivos si quieren un hueco en el mercado. Para ello intentan optimizar cualquier aspecto posible de su ámbito, desde el empaquetado hasta el envío de los productos. Cada vez es más necesario este tipo de prácticas, ya que el número de pedidos online crece cada año, debido a la comodidad de pedir productos desde casa, la posibilidad de realizar compras a cualquier hora del día y cualquier día de la semana, evitar largas colas en la tienda física, factores como la pandemia de este 2020...

Algunos datos estadísticos recogidos en un *blog* [34] sobre las compras online:

- En 2017 hubo 1.66 mil millones de compradores digitales en todo el mundo (21.8% de la población), y se espera que para el año 2021 se superen los 2.14 mil millones.
- En 2018 las ventas online supusieron casi el 12% de todas las ventas al por menor del planeta.
- Hay más de 254 millones de cuentas de PayPal activas.
- El 30% de los consumidores prefieren comprar en páginas que han usado otras veces. Por este mismo motivo, las empresas deben preparar y enviar sus pedidos lo más rápidamente posible, para no perder a este tipo de clientes frente a empresas de la competencia.
- Un dato específico de supermercados [36] : En marzo del 2020, por culpa de la pandemia del coronavirus (o gracias a ella), la compra online en supermercados aumentó casi en un 50%. Tanto es así que algunas empresas vivieron situaciones bastante problemáticas por servicios online colapsados y sin *stock* en algunos establecimientos.

En un artículo de principios de este mismo 2021 [3], Felipe Alonso explica con más detalle qué tipo de productos son los que más aumentaron el pasado año durante el

confinamiento, en cuanto a ventas online. Los más demandados fueron los textiles, con un 52%, la alimentación (sector al que se dedica este proyecto), el 51%, y el ocio, el 46%. El sector turístico, por el contrario, perdió 13 puntos porcentuales situándose en un 33%, según el *Observatorio Cetelem e-commerce 2020* [24]. En ese mismo artículo se informa que tras el confinamiento esas estadísticas cambiaron un poco, y es que mientras el turismo y los dispositivos móviles aumentaron mucho sus ventas, el sector de la alimentación permaneció similar. En la Figura 1.1 podemos verlo con más detalle.



Figura 1.1: Compras online tras el confinamiento de 2020

En otro artículo [29], de M. Prieto, de agosto del pasado 2020 encontramos otros datos interesantes. España cuenta con 22,5 millones de compradores online, y, aunque la mayoría combina la compra física con la digital, el 23% lo hacen exclusivamente mediante Internet, siete puntos más que en 2019. Los resultados de una encuesta dicen que, aunque la mayoría de los consumidores tiene intención de volver a su frecuencia de compra física anterior a la pandemia, una cuarta parte de los encuestados pretenden seguir con estos nuevos hábitos.

En cuanto a los supermercados, se ha reducido en un 8% el número de personas que nunca han comprado algún producto de alimentación online. Aún así, el IV Observatorio sobre Comercio Electrónico en Alimentación [1] asegura que existe una mayoría de personas que prefiere adquirir estos productos en establecimientos físicos. Esto se debe a cierta desconfianza hacia el canal online, además de preferir ver los productos en persona para valorar su estado, fecha de caducidad, etc.

Cuando realizamos la compra de productos de alimentación por Internet a cualquier supermercado, se desencadena un serie de procesos, que hay que optimizar, hasta que se entrega el pedido en casa del cliente.

Los procesos que se desarrollan en el almacén del supermercado son los siguientes:

- Cuando llega el pedido hay que saber en cuantas cajas/contenedores va a caber el pedido completo, para ello necesitamos saber la cantidad de productos y las dimensiones de las cajas y de cada producto.
- Los productos son recogidos por personal del supermercado de los estantes donde

están colocados. Por tanto, hay que asignar a cada trabajador (*picker*) uno o varios pedidos que deben recoger. Suelen llevar 4, 8 o 12 contenedores en cada recorrido.

- Se debe indicar a los *pickers* la ruta a seguir para realizar esos recorridos, indicando la ubicación del siguiente producto a recoger, hasta completar la ruta.
- Cada vez que un trabajador coge un producto se le podría proporcionar la ubicación dentro de las cajas/contenedores, para lograr la optimización del espacio.

En este proyecto vamos a centrarnos en optimizar la recogida de los productos (*picking*), y para ello debemos optimizar las rutas que deben seguir los recogedores de los productos (*pickers*). No solo se trata de decidir las rutas que deban seguir los *pickers* con los pedidos que deban recoger en una ruta, sino también hay que decidir cuáles son esos pedidos que el *picker* debe reunir. Si agrupamos los pedidos mediante similitudes, podemos conseguir rutas más cortas por parte de los trabajadores, y por tanto, menores tiempos en el proceso de *picking*. Además, Edward Franzelle, decía en su libro *World-Class Warehousing and Material Handling* (2002) [16] [17], que el trabajo de recoger pedidos en supermercados supone entre el 50 y el 65% de todos los costes en los establecimientos.

Este proceso es en el que más hincapié se está poniendo los últimos años, ya que áreas como el almacenaje están más automatizadas y existe un menor margen de beneficio, pero la tarea de recogida sigue siendo realizada por humanos prácticamente siempre. Un buen sistema que indique a los *pickers* qué pedidos reunir y qué rutas seguir por el almacén puede llegar a ahorrar muchísimas horas de trabajo a la empresa a largo plazo. Muchas empresas siguen realizando estas tareas en casi su totalidad mediante decisiones humanas. Generalmente, es una persona la que le dice a cada *picker* que pedidos debe recoger en el mismo carrito. Una vez que conocen esos pedidos, recogen los productos según un orden arbitrario que ellos mismos deciden, o en un orden lineal marcado por los IDs de los productos, que están colocados en los estantes ordenados por ese ID.

Muy pocos almacenes están optimizados de alguna manera, sin embargo, hace unos años, una gran cadena de supermercados a nivel nacional, optó por la automatización de sus almacenes [4] [7] [5]. Cintas automáticas van recogiendo los productos y llevándolos hacia un punto común, donde los pedidos se ensamblan y se preparan para el envío a los clientes. Este sistema multiplica por cuatro la productividad y eficiencia de los pedidos preparados en tiendas convencionales.

En la mayoría de supermercados, los pedidos se preparan en dos zonas distintas: la tienda abierta, donde los ciudadanos hacen la compra normalmente, y la tienda cerrada, el almacén al que solo acceden los trabajadores. Normalmente se recogen los productos que existan en la tienda abierta primero, terminando con los que sólo existan en el almacén. Sin embargo, en nuestro caso tan sólo usaremos una zona, la tienda

cerrada, para simplificar el problema y centrarnos en los aspectos relevantes de la implementación.

Entre los distintos productos de los supermercados, los refrigerados y los congelados son los más delicados, ya que para su conservación, estos deben ser los últimos en ser recogidos, además de tener que ser almacenados en cajas especiales.

En cuanto al material, cada establecimiento puede tener el suyo propio. Sin embargo, los europalets [2], que podemos ver en la Figura 1.2, se usan en una gran cantidad de almacenes, y sus medidas son de 1.200 x 800 mm. Los tacos miden 145 x 145 mm, y en cuanto al peso, rondan los 25 kg y soportan cargas de hasta 1500 kg (4.000 de manera estática).



Figura 1.2: Europalets

En cuanto a la capacidad total, los *pickers* suelen llevar 4, 8 o 12 cajas, llevando 1, 2 o 3 europalets en carros. Esas cajas pueden ser de distintos tamaños, aunque normalmente se usan de 400 x 600 x 320 mm, tamaño proporcional a los europalets que hemos comentado, para que quepan justamente 4. Estos europalets se apilarían en vertical para poder llevar 8 o incluso 12 contenedores simultáneamente. En la Figura 1.3 podemos ver una caja de estas dimensiones.

1.2. Motivación

Como ya hemos dicho, el modelo de compras está cambiando más y más cada año, aumentando las compras online y disminuyendo las compras en tiendas y supermercados físicos. Es por esto que los almacenes están tratando de optimizar todos los procesos de este tipo de ventas, pero no es una tarea sencilla, ya que cada empresa utiliza un tipo de productos, pedidos, cajas, identificadores, tamaños y configuraciones de sus

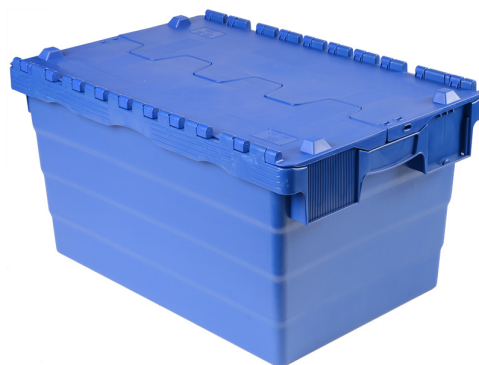


Figura 1.3: Caja tamaño estándar

almacenes, etc.

Todos los procesos pueden ser mejorados, algunos más que otros, como veremos más adelante, lo que significa, al final, una mayor productividad, y por tanto, un mayor beneficio económico para la empresa. Por otro lado, con este tipo de proyectos, también se genera un cierto valor social, ya que, ese aumento de la productividad de las empresas se traduce en un menor tiempo de espera por parte de los clientes en recibir sus pedidos. Además, situaciones como la que hemos comentado acerca de la saturación de algunos establecimientos durante la pandemia se podrían evitar.

También cabe destacar que podemos encontrar un gran valor científico en estos problemas, ya que es un campo que todavía no está muy explorado. Esto puede provocar que mucha gente investigue soluciones para este tipo de problemas en grandes almacenes, resultando así en una mejoría rápida de la productividad en estas empresas.

Por último, existe la posibilidad de que en un futuro las compras online supongan una amplia mayoría frente al modelo de compra tradicional, lo que significaría que muchas empresas abandonarían el esquema que conocemos hasta ahora de tiendas o almacenes para convertirse en establecimientos enfocados en su totalidad a la compra-venta por Internet.

1.3. Objetivos

Como ya hemos comentado, nuestro objetivo es minimizar el tiempo total empleado en la recogida de los productos para completar una serie de pedidos. Para ello, debemos desarrollar una serie de algoritmos con los que decidir qué pedidos recolectará cada *picker* en cada ruta, cuántas cajas serán necesarias para ello, y en qué orden debe recorrer la ruta por el almacén. Para ello, debemos seguir una serie de pasos:

- Describir formalmente el problema: Describir cómo será nuestro almacén, como

serán las cajas, los productos, los pedidos, etc.

- Estudiar el funcionamiento actual en los almacenes en estas grandes empresas, para entender mejor su situación y cómo podríamos mejorar estos procesos.
- Observar el estado del arte de cada uno de los problemas que queremos resolver para barajar todas las soluciones posibles.
- Implementar los algoritmos que decidamos.
- Analizar los resultados, no sólo los tiempos totales en la recolección de los pedidos, sino también los tiempos de ejecución de los algoritmos.
- Si observamos posibles mejoras, implementarlas y analizar de nuevo los resultados.
- Finalmente, obtener una serie de conclusiones acerca del problema y las soluciones que hayamos propuesto.
- Adicionalmente, podemos proponer posibles mejores futuras, no sólo sobre nuestros algoritmos, sino acerca del modelo de trabajo en grandes almacenes, la estructura de éstos, etc.

1.4. Estructura del documento

La estructura del proyecto es la siguiente:

- Capítulo 1: Introducción del proyecto, junto a la motivación del mismo y los objetivos alcanzados.
- Capítulo 2: Estado del arte. En él comenzaremos con una descripción de los algoritmos de búsqueda más usados en problemas de este tipo. Después haremos una definición de nuestro problema y estudiaremos la situación actual de los almacenes tradicionales, así como los algoritmos existentes sobre cada uno de los problemas que queremos resolver.
- Capítulo 3: Algoritmos desarrollados. Explicaremos cada uno de los algoritmos que hemos implementado. Además de su funcionamiento, estructura y métodos incluiremos algunas pruebas simples, para comprobar su funcionamiento.
- Capítulo 4: Resultados. Analizaremos los resultados acerca de los algoritmos desarrollados, así como el tiempo de ejecución de todos ellos.
- Capítulo 5: Conclusiones. Extraemos una serie de conclusiones a partir de esos resultados y del desarrollo de todo el proyecto. Por último propondremos otras posibles mejoras para el futuro y hablaremos de las competencias de la intensificación de computación que hemos trabajado durante el desarrollo del proyecto.

2. Estado del arte y descripción del problema

2.1. Algoritmos de búsqueda

Antes de explicar el problema, y cómo plantear cualquier algoritmo para solucionarlo, vamos a explicar qué son, cómo funcionan y cuáles son los tipos de algoritmos de búsqueda más usados.

2.1.1. ¿Qué es un algoritmo? (6)

Una posible definición de **algoritmo** es: “Un conjunto de órdenes consecutivas que presentan una solución a un problema o tarea”. La palabra algoritmo proviene del matemático árabe Al-Khwarizmi, que en el IX fue reconocido por enunciar paso por paso las reglas necesarias para las operaciones básicas (suma, resta, multiplicación y división) con decimales.

No solo hay algoritmos en el ámbito de las matemáticas y la informática, podemos considerar algoritmos también el manual de un lavavajillas o el prospecto de un medicamento. Sin embargo, hoy en día la palabra algoritmo se relaciona principalmente con la informática. En el ámbito de la programación se les conoce como **algoritmos informáticos**.

Un algoritmo informático [15] consiste en una serie de clases, variables y métodos, que escritos en un lenguaje de programación se ejecutan para resolver un problema. La estructura básica de cualquier algoritmo es la siguiente:

- **Input:** La entrada del algoritmo son los datos que necesita el algoritmo inicialmente para comenzar a operar.
- **Procesamiento:** Todas las operaciones o cálculos lógicos que utiliza el algoritmo para resolver el problema.

-
- **Output:** La salida del algoritmo son los resultados del algoritmo, que se muestran por consola o mediante una interfaz al usuario.

Algunas características de los algoritmos son las siguientes:

- Son **secuenciales**, ya que las operaciones se ejecutan una después de otra.
- Son **precisos**, no pueden ser ambiguos o subjetivos.
- Son **finitos**, ya que deben tener un número limitado de pasos y finalizar en algún momento su ejecución.
- Son **concretos**, pues deben mostrar una solución al problema.
- Son **definidos**, ya que ante los mismos datos de entrada, se deben obtener los mismos datos de salida.
- Son **ordenados y claros**, ya que un código legible y claro es mucho más fácil de entender. Para ello hay que dar un formato consistente al código y recurrir a comentarios para explicar distintos elementos del programa.

Existen muchos tipos de **algoritmos de búsqueda**, que consisten en encontrar una solución con ciertas propiedades en un espacio de datos. Entre ellos destacan los siguientes.

2.1.2. Algoritmos voraces

Los algoritmos voraces (*greedy algorithms*) [32] son algoritmos de búsqueda, que siguiendo una búsqueda heurística toman la decisión óptima en cada momento, pero al tomar una decisión no se puede volver atrás. Algunos ejemplos de algoritmos voraces son el de *Dijkstra* o el de *Kruskal*.

Un problema clásico resuelto mediante algoritmos voraces es el del cambio de monedas, que podemos ver en la Figura 2.1:

“Si tenemos que devolver 36 céntimos a un cliente, y tenemos monedas infinitas de los ocho tipos que existen en la moneda euro (2€, 1€, 50cts., 20cts., 10cts., 5cts., 2cts., y 1cént.), determina qué monedas se deben devolver al cliente para usar el menor número de monedas.”

La solución voraz a este problema consiste en elegir en todo momento la moneda de mayor valor que sea menor que el dinero restante. El resultado es 1 moneda de 20cts., 1 moneda de 10cts., 1 moneda de 5cts., y 1 moneda de 1cént (4 monedas). Gracias a nuestro sistema monetario, un algoritmo voraz siempre encuentra la solución óptima en este problema, pero en otras circunstancias no sería así. Si por ejemplo existieran las monedas de 18cts., este algoritmo comenzaría eligiendo una moneda de 20cts. igualmente, y obtendríamos la misma solución de antes, en lugar de obtener la solución

óptima (2 monedas, ambas de 18cts.).

Los algoritmos voraces sirven para resolver problemas pequeños y muy específicos, y no tienen porqué llegar a soluciones óptimas en problemas complejos.

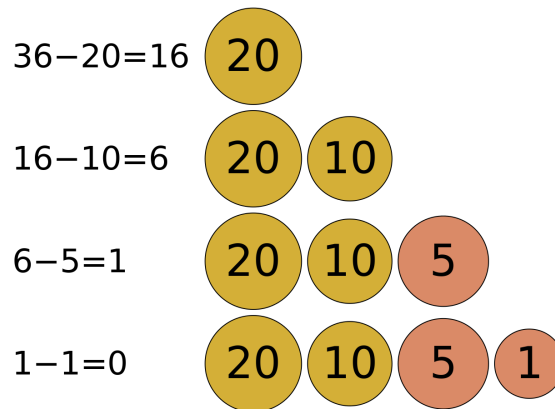


Figura 2.1: Problema del cambio de monedas

2.1.3. Programación dinámica

El concepto básico de la programación dinámica es que la solución de un problema viene dada por la solución de subproblemas. Su máximo representante es Richard Bellman [9], quien inventó estos algoritmos en 1953. El principio de optimalidad de Bellman dice así: “...una secuencia óptima de decisiones que resuelve un problema debe cumplir la propiedad de que cualquier subsecuencia de decisiones, que tenga el mismo estado final, debe ser también óptima respecto al subproblema correspondiente.”.

Un problema clásico resuelto mediante programación dinámica es el de la mochila sin fraccionar, que podemos ver en la Figura 2.2.

“Tenemos una mochila con un peso límite b , y una serie de objetos i con un peso p_i y un valor v_i , donde $i \in I$. Determina que objetos llevar en la mochila sin sobrepasar el límite de peso b y maximizando el valor total de la mochila v .”

Para resolver este problema mediante programación dinámica, tenemos que crear una tabla de resultados intermedios, con tantas filas como kg soporte la mochila, y tantas filas como objetos existan en total. Inicialmente todos los valores de la tabla estarán a 0, ya que tratamos de maximizar (infinito en problemas de minimización). Comenzamos a rellenar por la posición (0,0) y la recorremos de izquierda a derecha y de arriba a abajo. Esto nos asegura el principio de optimalidad de Bellman, ya que


4 kg 10 €	3 kg 40 €	5 kg 30 €	2 kg 20 €
			
8 kg			

Figura 2.2: Problema de la mochila sin fraccionar

si la posición anterior era óptima, la siguiente también lo será. Conforme vamos bajando en la tabla, tenemos que decidir si el objeto actual lo añadimos o no en la mochila.

Nuestro problema del viajante podría resolverse mediante programación dinámica, encontrando siempre buenos resultados.

2.1.4. Vuelta atrás (*backtracking*)

Es una estrategia de búsqueda, normalmente apoyándose en heurísticas, por la cual se puede resolver cualquier problema, pero de manera poco eficiente. Consiste en buscar todas las posibles soluciones, de manera que en algún momento se alcance la solución correcta, por lo que obviamente no ofrece resultados muy efectivos. Algunos problemas típicos resueltos mediante *backtracking* son laberintos, sudokus, o el problema de las ocho reinas (Figura 2.3). Hablamos de poca eficiencia para problemas muy grandes (como el nuestro), en los que éstos algoritmos presentan tiempos computacionales enormes, pero para resolver sudokus o laberintos pequeños es uno de los métodos más usados.

2.1.5. Algoritmos de búsqueda no informada

Se trata de estrategias de búsqueda en las que al evaluar el siguiente estado no conocemos si es mejor o peor que el anterior. Algunos algoritmos de este tipo son: búsqueda en profundidad, búsqueda en anchura, búsqueda en profundidad iterativa, búsqueda en coste uniforme... Son algoritmos más ineficientes en tiempo y memoria que los algoritmos de búsqueda informada, por lo que no debemos plantearnos implementar uno para nuestro problema.

2.1.6. Algoritmos de búsqueda informada

Los algoritmos de búsqueda informada o heurística usan un conocimiento específico del problema para encontrar soluciones de manera más eficiente. Las funciones heu-

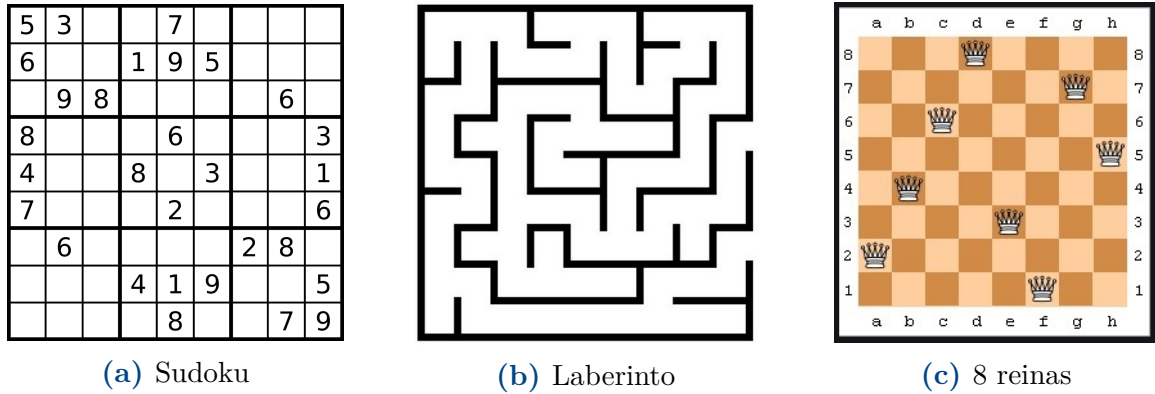


Figura 2.3: Problemas *backtracking*

rísticas son la forma más típica de transmitir conocimiento adicional del problema al algoritmo de búsqueda. Los más conocidos son Primero mejor y A*, que sirven para resolver eficientemente problemas como laberintos o rutas, como nuestro problema a resolver, aunque existen mejores alternativas.

Aunque no vayamos a implementar un algoritmo de este tipo, sería interesante explicar el concepto de heurística, ya que se puede aplicar a prácticamente cualquier tipo de algoritmo. Una heurística es una medida usada en los algoritmos para conseguir buenos resultados. En la Figura 2.4 vemos un ejemplo de heurística.

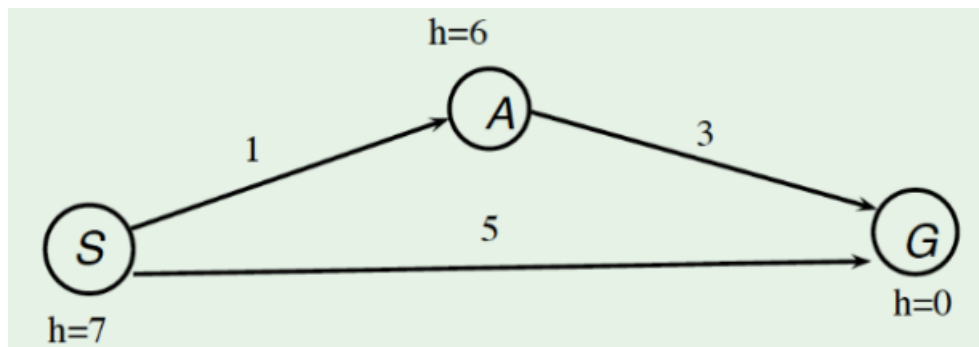


Figura 2.4: Ejemplo de heurística

Una heurística $h(n)$ es **admisible** si para cada nodo n , $h(n) \leq h^*(n)$, donde $h^*(n)$ es el coste verdadero para alcanzar el objetivo desde n .

Es decir, una heurística es admisible si nunca sobrestima el coste real, o sea, es optimista. El ejemplo más básico de heurística admisible en problemas de distancias es

la distancia en línea recta entre dos puntos, ya que la distancia real siempre será igual o mayor a ese valor. (5)

En nuestro problema podríamos usar heurísticas para la de optimización de las cajas, la asignación de los pedidos a los *pickers* o el problema del viajante. Imaginemos que, en nuestro problema, para completar 100 rutas que realizan 10 repartidores, el tiempo más rápido posible que podríamos conseguir es 1 hora. Ese valor óptimo no se puede calcular en un tiempo corto debido a la complejidad computacional, por lo que, si conseguimos aproximarlos a 1h y pocos minutos, habremos conseguido un buen resultado.

2.1.7. Búsqueda con adversario

En todos los algoritmos anteriores participaba un único agente. Este tipo de algoritmos consiste en que dos (o más) agentes compiten por los mismos recursos. El ámbito principal donde podemos encontrar estos algoritmos es en la inteligencia artificial de los videojuegos. El ejemplo más famoso es el de *Deep Blue* [12], la primera máquina que consiguió derrotar a Kasparov en un partido de ajedrez a seis juegos.

Otros juegos donde se pueden usar estos algoritmos son: hundir la flota, backgammon, damas, go, scrabble... La estrategia más usada es la de Minimax, que consiste en maximizar las ganancias de tu siguiente jugada, sabiendo que el rival elegirá la jugada que las minimice.

No son algoritmos útiles para resolver problemas como el nuestro, por lo que no debemos tenerlos en cuenta.

2.1.8. Agentes que aprenden

Se usan en problemas con entornos no deterministas, donde una acción tomada no se ejecuta el 100% de las veces. Esto no es nuestro caso, ya que si nuestro algoritmo le dice a un trabajador que al final del pasillo gire a la derecha, el trabajador lo hará. Los más usados son Iteración de valores y QLearning [22], que es en realidad un algoritmo voraz, ya que cuando tiene que tomar una decisión, siempre toma la que maximiza la utilidad. Por eso mismo, y debido al no determinismo, muchas veces estos algoritmos no convergen en la solución óptima, y hay que establecer un balance entre exploración y explotación.

2.1.9. Optimización combinatoria

Los métodos de búsqueda anteriores exploraban el espacio de búsqueda hasta encontrar un estado objetivo y devolvían el camino para llegar a él. Pero hay problemas

en los que no queremos el camino a la solución, sino la solución en sí misma. Algunos problemas que se resuelven mediante optimización combinatoria son:

- Obtener la configuración de parámetros óptima para maximizar el rendimiento de un Fórmula 1.
- Satisfacción booleana. Consiste en asignar 0 o 1 a las variables de una fórmula lógica para hacerla verdadera o encontrar que no hay solución.
- Problema del coloreo del grafo.
- Problema del viajante. Sin lugar a dudas, el problema del viajante (lo llamaremos TSP, *Traveller Salesman Problem*), es el problema más famoso y estudiado en el ámbito de la optimización combinatoria computacional, lo cual nos indica que ésta es la manera en la que debemos resolver nuestro problema.

Metaheurísticas: Son algoritmos aproximados que en un tiempo razonable proporcionan buenas soluciones al problema, no siempre la óptima. Es importante establecer un balance entre **intensificación** (explotación del espacio actual) y **diversificación** (exploración de otras regiones).

Podemos dividir los tipos de metaheurísticas en:

- **Heurísticas constructivas.** Parten de una solución inicial vacía y van añadiendo componentes hasta construir una solución: GRASP (Greedy Randomized Adapative Search Procedure), Colonias de Hormigas...
- **Heurísticas basadas en trayectorias.** Parten de una solución inicial e iterativamente tratan de reemplazarla por otra con mejor calidad de sus vecinos. Uno de los más usados, y que nosotros implementamos en Sistemas Inteligentes, es el algoritmo *Hill Climbing*, de búsqueda local. Tiene la desventaja de quedarse atrapado en óptimos locales por usar una única solución.
- **Heurísticas basadas en poblaciones.** Evolucionan una población de soluciones iterativamente. El más usado, y que nosotros también implementamos en la misma asignatura, es el **Algoritmo Genético**, de búsqueda global. Soluciona el problema de los óptimos locales gracias a usar muchos individuos, y no solo uno. Consiste en crear muchos individuos aleatorios, elegir a los mejores de ellos, y realizar cruces y mutaciones hasta que el algoritmo converja en un buen resultado.
 - Creamos una **población inicial** de cromosomas aleatoriamente (rutas aleatorias), cada cromosoma formada por distintos parámetros, llamados genes.
 - **Función de fitness**, que determina como de buenos son los individuos.
 - **Selección** de los mejores individuos (padres) para pasar sus genes a la siguiente generación (se puede realizar una selección por torneo, como hicimos en Sistemas Inteligentes).

-
- **Cruce:** Este método nos genera 2 rutas hijas a partir de 2 rutas padres mezclando sus genes.
 - **Mutación:** Algunos individuos sufrirán mutaciones aleatorias en sus rutas (por ejemplo elegir 2 productos al azar e intercambiar sus posiciones), para crear diversidad en la población y evitar la convergencia prematura del algoritmo.
 - **Sustitución de la población:** A partir de la población tras la selección, mutación y cruce, debemos sustituir a la antigua población mediante reemplazo, elitismo o truncamiento.
 - **Repetir** la selección, el cruce, la mutación y la sustitución hasta que la población haya convergido en un buen resultado, devolviéndonos un cromosoma (en nuestro caso, conjunto de las rutas a tomar por los *pickers*).

2.2. Definición de nuestro problema

Una vez explicados brevemente algunos algoritmos de búsqueda clásicos, vamos a definir nuestro problema y cuáles son los algoritmos concretos que se suelen usar para problemas similares.

Nuestro almacén tendrá una disposición similar a la que podemos ver en la Figura 2.5, donde las casillas blancas representan los distintos estantes del almacén, las casillas grises son los pasillos por los que los *pickers* pueden moverse y el punto de partida será la casilla roja.

Inicialmente nuestra idea era tener un número de trabajadores, que dispusieran de 4, 8 o 12 cajas, y en total habría un número determinado de pedidos con sus productos e información. Sin embargo, tras desarrollar los algoritmos y trabajar con las instancias que comentaremos más adelante, no tendremos con varios *pickers*, ni llevarán un número concreto de cajas. Simplemente calcularemos las rutas necesarias, sin asignárselas a ningún *picker*. Al fin y al cabo, una vez establecidas las, por ejemplo, 10 rutas, es indiferente que haya más o menos *pickers* trabajando, lo importante es obtener correctamente cada una de esas rutas con los pedidos que se deben reunir en ellas, el resto se puede adaptar fácilmente a las condiciones de cada almacén.

Para no complicar en exceso el problema del espacio de las cajas, inicialmente íbamos a definir los productos como paquetes con forma de cuboides, es decir, prismas rectangulares, cuyas caras forman entre sí ángulos diedros rectos. Es mucho más sencillo trabajar con figuras rectas que trabajar con formas de productos reales, como bolsas de patatas, botellas de agua, frutas... Sin embargo, y como explicaremos más adelante, hemos decidido trabajar solamente teniendo en cuenta la capacidad que cada *picker* puede llevar, y la capacidad de cada pedido. Es decir, si un *picker* tiene como capacidad un valor de 100, puede llevar tantos pedidos como pueda, mientras que la suma de las capacidades de esos pedidos no supere 100. Hemos elegido esta opción

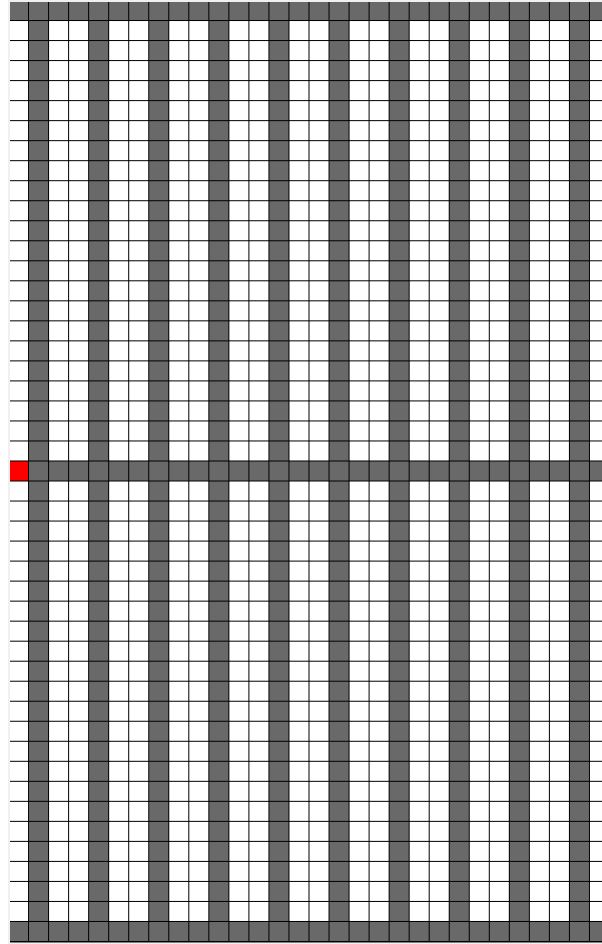


Figura 2.5: Almacén

porque las instancias que hemos usado facilitaban este modo de trabajo, además de que desarrollar un algoritmo de optimización de espacio en los contenedores es muy complejo, digno de un proyecto completo. Sin embargo, como se trata de un problema que se debería resolver en un entorno real en cualquier almacén que desee aumentar su productividad de esta manera, igualmente vamos a estudiar distintas propuestas en el Capítulo 2 acerca de estos algoritmos.

Inicialmente hemos implementado la posibilidad de crear el almacén con las medidas deseadas (por ejemplo 10 pasillos y 20 productos por cada bloque de estantes) y crear los productos aleatoriamente introduciendo la cantidad deseada (por ejemplo 20 productos). De esta manera hemos podido desarrollar el algoritmo de rutas deseado para comprobar que funciona correctamente. Posteriormente, hemos utilizado una serie de instancias obtenidas de [18], donde hay una serie de archivos que indican tanto la configuración de los almacenes, como una serie de pedidos con todos sus productos y la información de los mismos.

Definición formal del problema

Tenemos una serie de pedidos $P = \{p_1, p_2, \dots, p_n\}$, cada uno de estos pedidos necesitará un número determinado de cajas c_p con $p \in P$, este valor se puede calcular de antemano. En nuestro caso, ese número de cajas c_p siempre será 1. Ningún pedido puede superar en capacidad a la capacidad de los *pickers*, y no se puede dividir un pedido en varias cajas, por lo que todos los pedidos se llevarán en 1 caja. Tenemos un conjunto de *batches* que son agrupaciones de pedidos B , cada una de estas agrupaciones tendrá una distancia para ser recogida d_b . El problema consiste en encontrar el mejor subconjunto de *batches* de manera que se recojan todos los pedidos minimizando la distancia total. Para ello diseñaremos un algoritmo que resuelva el *batching* de los pedidos usando como función de evaluación el algoritmo de rutas que también desarrollaremos.

Para entender mejor qué son nuestros *batches*, pongamos un ejemplo. Cabe destacar que siempre que hablemos de capacidades, ya sea del *picker* o de los pedidos, nos referimos a unidades de volumen indefinidas. Ambas capacidades se miden en la misma unidad por lo que se podrían referir a peso o a volumen, nosotros lo trataremos como volumen. Imaginemos que el *picker* tiene una capacidad de 10, y que existen 4 pedidos con capacidades 3, 4, 6 y 7. Un posible *batch* sería agrupar en una ruta los dos primeros pedidos (suman 7 de capacidad), en otra ruta el tercer pedido (6 de capacidad), y en una tercera ruta el último pedido (7 de capacidad). Supongamos que esas tres rutas suman un total de 120 segundos en tiempo de recogida. Sin embargo, es evidente que hay una configuración mejor que esa: si agrupamos el primer y el cuarto pedido obtendremos una ruta con 10 de capacidad, y agrupando el segundo y el tercer pedido obtenemos una segunda ruta con otros 10 de capacidad. Es decir, hemos reducido de 3 rutas totales a 2 rutas totales la solución. Esta segunda configuración de *batches* podría durar 100 segundos (por ejemplo), por lo que habríamos reducido considerablemente el tiempo total de la recogida de los productos. En este ejemplo nos hemos inventado los tiempos de recogida, pero es evidente que dos rutas tendrían un tiempo total de recogida menor al *batch* de las tres rutas. En eso consiste el problema de los *batches*, en encontrar la mejor combinación de pedidos en distintas rutas para reducir el tiempo total de la recogida de productos, utilizando un algoritmo de rutas para calcular como de bueno es un *batch*.

2.3. Instancias

Hemos obtenido una serie de instancias para los pedidos en Optsicom Project, un proyecto de la Universidad Rey Juan Carlos enfocado a optimización de herramientas basadas en metaheurísticas [35]. Tienen propuestas para más de 20 problemas de optimización, en nuestro caso nos interesa el “Order Batching Problem”, [18].

Este problema consiste en agrupar los pedidos recibidos en un set de *batches* de manera que se minimice el tiempo empleado en reunir todos los pedidos, exactamente

igual que nuestro problema. Sin embargo, en su caso, el almacén consiste en múltiples pasillos verticales y tan sólo dos pasillos horizontales, es decir, no tiene un pasillo central como nuestro problema. Se proporcionan dos sets de instancias, sumando en total 144 instancias:

- **Henn and Wäscher, 2012:** Esta colección consiste en un almacén rectangular de 10 pasillos y 90 localizaciones por pasillo. El punto de partida se encuentra en la esquina inferior izquierda, pero esto nos es indiferente, tan sólo queremos conocer datos sobre los pedidos. Existe una distribución ABC y una distribución aleatoria. Los pedidos varían de 40 a 100 (cambiando de 20 en 20) y la capacidad de los *pickers* varía de 30 a 75 (cambiando de 15 en 15).
- **Albareda-Sambola et al., 2009:** Este set consiste en cuatro distintos almacenes rectangulares con el punto de partida en la esquina inferior izquierda o en el centro inferior. Las distribuciones son las mismas (una ABC y una aleatoria), pero los pedidos varían de 50 a 250 (cambiando de 50 en 50) y la capacidad de los *pickers* depende del almacén.

Tras varias pruebas hemos decidido usar las instancias de Henn, ya que son más completas y más fáciles de entender. Las instancias con la distribución ABC tenían un problema, y es que prácticamente un tercio de los productos se acumulaban en los pasillos 0 y 1, mientras que los pasillos 2, 3 y 4 no tenían ni un solo producto. Por ese motivo, hemos utilizado las instancias aleatorias, que reparten completamente al azar todos los productos por el almacén.

Cada instancia consta de dos archivos de texto:

- **Pedidos:** Ofrece toda la información acerca de los pedidos de la instancia. Para cada pedido indica cuál es el número de productos, y para cada uno de estos indica su pasillo y localización (estante).
- **Almacén:** Este otro archivo indica toda la información necesaria para configurar el almacén. Tiene muchos parámetros (tipo, velocidad, ancho y largo de los pasillos, etc.), pero a nosotros sólo nos interesan los siguientes:
 - **Número de pasillos verticales del almacén.**
 - **Número de localizaciones por pasillo.** Respecto a este atributo tuvimos que hacer algún cambio. Debido a que estas instancias están pensadas para almacenes sin pasillo intermedio, pero en nuestro problema sí existe pasillo intermedio, hemos dividido los pasillos por la mitad. Es decir, si en una instancia los pasillos tienen 30 configuraciones, utilizamos 15 para el bloque superior, y 15 para el bloque inferior. En el caso de haber localizaciones impares, por ejemplo 45, dividimos el almacén en 22 localizaciones por bloque, y reubicamos los productos que se encuentren en la última fila a cualquier otra ubicación aleatoria.
 - **Número de pedidos.**
 - **Capacidad del *picker*.**

2.4. Subproblemas a resolver

A la hora de optimizar el tiempo invertido en realizar múltiples rutas en un almacén, nos encontramos con varios subproblemas distintos a resolver, son los siguientes:

- **Optimización de los paquetes en las cajas.** Para cada uno de los pedidos necesitamos saber el menor número de cajas/contenedores en los que podemos acomodar sus productos. Conocemos las dimensiones y el peso de los productos, y el tamaño de las cajas, por lo que podemos calcular la cantidad de cajas necesarias para albergar cada pedido. Como ya hemos dicho, en nuestro caso simplificaremos este proceso utilizando simplemente las capacidades de los *pickers* y de los pedidos. Sin embargo, vamos a estudiar algunas propuestas para tener un mejor entendimiento del problema de cara a un entorno real.
- **Creación de *batches* o lotes.** Una vez hemos hecho el reparto en cajas de cada pedido, tenemos que asignar los pedidos a los *pickers* y decidir qué cajas llevarán en cada ruta. Para encontrar el mejor *batch* posible nos deberemos apoyar en el algoritmo que resuelva el Problema del viajante.
- **Problema del viajante.** Conocido un *batch*, debemos encontrar para cada ruta de los *pickers* el camino más corto que pase por todos los productos a recoger y vuelva al punto de inicio.

2.4.1. Optimización de los paquetes en las cajas

En nuestro caso cada contenedor solo podrá llevar productos de un único cliente. Conocido un pedido, con todos sus productos, el tamaño de cada uno de esos productos, y el tamaño de los contenedores, debemos minimizar el número de contenedores usados para ese pedido. En nuestro caso sería algo más complejo, ya que para minimizar el número de cajas usadas en total, debemos agrupar los pedidos en distintos *batches*. Ese problema lo trataremos más adelante, pero debemos entender que ese *batching* usará este algoritmo de optimización de espacio para decidir qué pedidos pueden ser o no ser agrupados. Se trata de un problema para nada sencillo, con coste computacional NP-completo, y que a día de hoy se sigue investigando ya que todavía hay mucho margen de mejora. Estas dificultades existen para problemas donde los paquetes a colocar en las cajas son ortogonales, por lo que es fácil entender que si intentáramos resolver este problema con formas más complejas: botellas, frutas, bolsas de patatas... la solución se complicaría todavía más.

Resolver este problema de manera eficaz puede suponer un beneficio económico para empresas de alimentación como en nuestro caso, pero también a empresas de envío de cualquier tipo de productos, empresas de reparto donde en lugar de paquetes en cajas tienen que transportar cajas en camiones, transportar carga en aviones, organizar el hardware en dispositivos electrónicos, etc.

Este problema es conocido como *The three-dimensional bin packing problem*, es decir, problema de optimización tridimensional en contenedores (o cajas). Nos referiremos a él como 3D-BPP. Se trata de una generalización del famoso *Bin Packing Problem*, donde sólo existe una dimensión, por lo que es mucho más sencillo. También existe el *Two-dimensional bin packing problem*, más complejo que el anterior, pero no tanto como el 3D-BPP.

En este artículo [25], de Silvano Martello, David Pisinger y Daniele Vigo, tenemos un estudio acerca del 3D-BPP, enfocado a paquetes de formas ortogonales. En él, suponen que las cajas, con sus tres dimensiones (X, Y, Z) proporcionadas, no pueden ser giradas. Es decir, la orientación con la que se indican es la misma en la que deben ir colocadas. Para encontrar mejores soluciones, se deberían evaluar todas las posibilidades (girando en todos los ejes), pero esto conllevaría otro incremento considerable en la complejidad del problema.

El 3D-BPP está muy relacionado con otros problemas de carga:

- *Knapsack Loading*: El problema de la mochila, como ya comentamos anteriormente, consiste en que cada caja tiene un beneficio asociado, y debemos elegir el subconjunto de objetos que quepa en la mochila consiguiendo el mayor beneficio posible. Existen variaciones de este problema según si buscamos una solución voraz, programación dinámica, *backtracking*, ect. Si pensamos en un problema particular, en el que el beneficio de cada objeto es exactamente igual a su volumen, estaríamos hablando del mismo problema, ya que trataríamos de minimizar el espacio [19] [28].
- *Container loading*: En esta versión, todas las cajas tienen que ser colocadas en un solo contenedor, que en este caso tiene una longitud infinita. El problema consiste en minimizar esa longitud del contenedor [20] [10].
- *Bin Packing*: En el 3D-BPP todos los contenedores tienen el mismo tamaño, y se trata de encontrar una solución que minimice el número de contenedores usados. Una aproximación para el 3D-BPP, , fue presentada por Scheithauer [31]. Chen, Lee and Shen [14] generalizaron el problema para contenedores de diferentes tamaños.

En este estudio describen también un algoritmo de ramificación y poda para encontrar la solución a un problema de maximización. En él, conociendo las dimensiones de los paquetes y la caja, tratan de maximizar el volumen ocupado por los paquetes. Asumen que el origen del sistema de coordenadas comienza en la esquina inferior izquierda, al fondo del contenedor. En la Figura 2.6 podemos ver el funcionamiento para el problema en dos dimensiones. La superficie no utilizada se separa de la ya usada mediante una línea discontinua, mientras que las esquinas las indican con puntos.

Presentan, a partir de la propuesta para el 2D-BPP, una generalización no trivial para el caso tridimensional. En el caso del 2D-BPP, la solución pasa por intentar

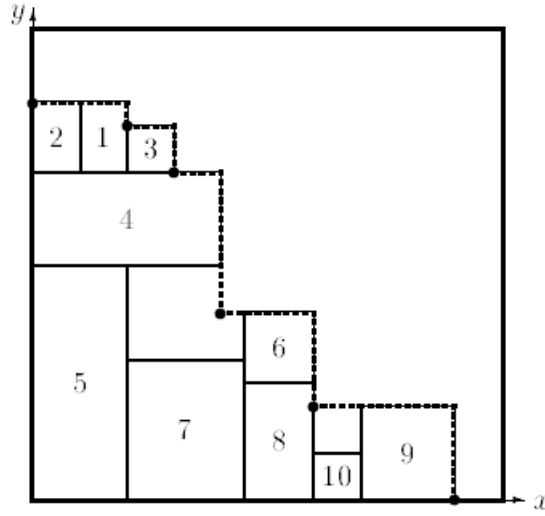


Figura 2.6: 2D-BPP

colocar los paquetes a partir de cada uno de esas esquinas. Su complejidad temporal es $O(|n|)$, siendo n el número de paquetes. Cada vez que se introduce un nuevo paquete, se recalculan esas esquinas para continuar con el siguiente paquete. Existen tres fases en este algoritmo que determina los puntos en las esquinas:

- Primero buscamos los puntos *extremos*, es decir, paquetes en los cuales sus puntos extremos coinciden con un punto donde la línea discontinua cambia de horizontal a vertical.
- Continuamos definiendo puntos en intersecciones entre líneas desde paquetes extremos.
- Por último, los puntos inviables se borran.

Para el enfoque tridimensional, el orden de complejidad es $O(n^2)$, y consiste en aplicar el algoritmo anterior, de dos dimensiones, para todas las “esquinas” en el eje z . En la Figura 2.7 podemos ver un ejemplo de este algoritmo.

Tras este enfoque inicial definen, de manera recursiva, el algoritmo de ramificación y poda para encontrar la mejor manera de llenar un contenedor. En el artículo ([25]) lo explican detalladamente, así como pruebas con pequeñas instancias, algoritmos aproximados y exactos, y experimentos computacionales. Como conclusión, comentan que el algoritmo de ramificación y poda juega un papel muy importante en este problema, y que la solución exacta del 2D-BPP [26] se puede adaptar al problema tridimensional con pequeñas modificaciones.

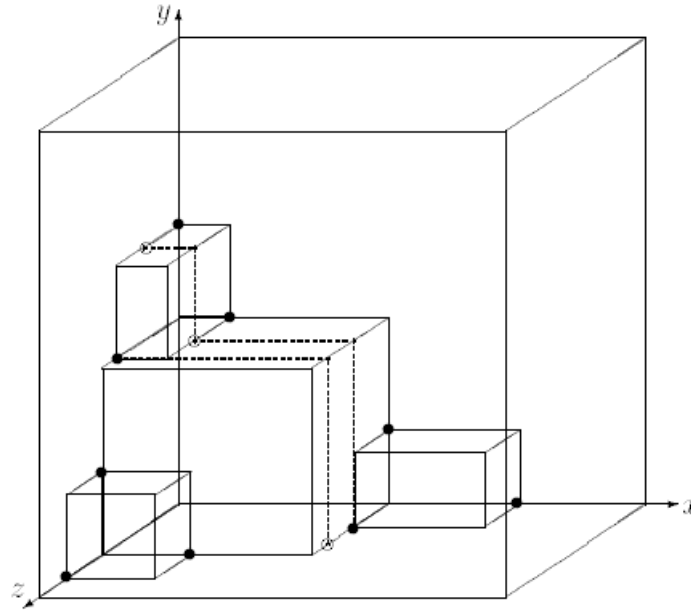


Figura 2.7: 3D-BPP

2.4.2. El problema del camino más corto

El problema de rutas que debemos resolver es el problema del viajante, o TSP. El problema del camino más corto (*Shortest Path Problem*) consiste simplemente en encontrar el camino más corto entre dos puntos dado un grafo. Es decir, en nuestro problema consistiría en encontrar el camino más corto entre dos productos.

El problema del viajante es un caso particular de este problema, en el que hay que encontrar el camino más corto entre el punto de partida y sí mismo visitando todos los nodos del grafo, es decir, todos los productos. Antes de estudiar cómo se puede resolver el problema del viajante, vamos a estudiar el problema del camino más corto, para obtener un mejor conocimiento sobre grafos y este tipo de problemas.

Los algoritmos más usados para resolver el *Shortest Path Problem* son:

- **Algoritmo de Dijkstra** [27]: Es la manera más eficiente de encontrar la distancia más corta entre una serie de puntos iniciales y el resto de vértices. Consiste en ir construyendo un grafo a partir del nodo origen. En cada iteración observamos todos los nodos aún no incluidos en el grafo que están conectados con el último nodo que hemos incluido (en la primera iteración el nodo origen) hasta alcanzar el nodo destino. Existen mejoras, como el particionamiento en subgrafos y la búsqueda bidireccional.

En la Figura 2.8 podemos ver un ejemplo de grafo, y en la Figura 2.9 tenemos la solución paso por paso mediante el algoritmo de Dijkstra.

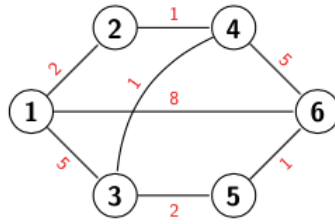


Figura 2.8: Ejemplo de grafo

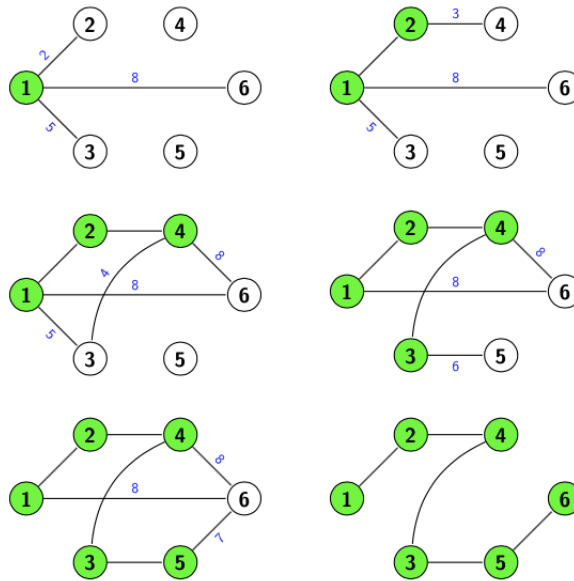


Figura 2.9: Solución con Dijkstra

- **Algoritmo de Bellman-Ford** [27]: Funciona con valores negativos, por lo que en caso de ciclos con algún valor negativo se alcanzaría un tiempo de $-\infty$. En nuestro caso eso no supone un problema ya que todos los caminos tendrán valores positivos. Inicialmente la distancia del nodo origen al resto es infinita, e iremos mejorando ese valor en cada iteración. Hay que realizar un total de $n - 1$ iteraciones, siendo n el número de nodos en el grafo. En la Figura 2.10 podemos ver un grafo de ejemplo con algunos valores negativos.

En la Figura 2.11 podemos ver una tabla con la solución al problema usando el algoritmo de Bellman's Ford.

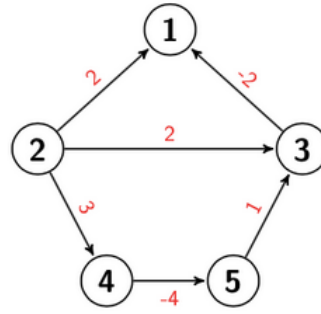


Figura 2.10: Grafo con valores negativos

iteraciones	arco	distancia1	distancia2	distancia3	distancia4	distancia5
0		inf	0	inf	inf	inf
1	2 --> 1	2	0	inf	inf	inf
	2 --> 3	2	0	2	inf	inf
	2 --> 4	2	0	2	3	inf
	3 --> 1	0	0	2	3	inf
	5 --> 3	0	0	2	3	inf
	4 --> 5	0	0	2	3	-1
2	2 --> 1	0	0	2	3	-1
	2 --> 3	0	0	2	3	-1
	2 --> 4	0	0	2	3	-1
	3 --> 1	0	0	2	3	-1
	5 --> 3	0	0	0	3	-1
	4 --> 5	0	0	0	3	-1
3	2 --> 1	0	0	0	3	-1
	2 --> 3	0	0	0	3	-1
	2 --> 4	0	0	0	3	-1
	3 --> 1	-2	0	0	3	-1
	5 --> 3	-2	0	0	3	-1
	4 --> 5	-2	0	0	3	-1
4	2 --> 1	-2	0	0	3	-1
	2 --> 3	-2	0	0	3	-1
	2 --> 4	-2	0	0	3	-1
	3 --> 1	-2	0	0	3	-1
	5 --> 3	-2	0	0	3	-1
	4 --> 5	-2	0	0	3	-1

Figura 2.11: Solución con Bellman's Ford

- **Algoritmo de Floyd-Warshall** [8]: Encuentra el camino óptimo entre todos los puntos usando programación dinámica, pero no es tan eficiente como Dijkstra. Para calcular el camino mínimo en un grafo de V vértices podríamos ejecutar el algoritmo de Dijkstra V veces, para comenzar una vez desde cada nodo, lo que supone un orden $O(V^3)$. Si existen nodos negativos, se podría ejecutar V veces el algoritmo de Bellman, pero conllevaría una complejidad computacional de orden $O(V^4)$.

El algoritmo de Floyd-Warshall usa la metodología de programación dinámica que hemos explicado antes para resolver el problema. Puede obtener el camino mínimo con pesos negativos en orden $O(V^3)$, aunque los ciclos de peso negativo pueden causar fallos en el algoritmo.

- **Algoritmo de Johnson's** [8]: Funciona mejor que el de Floyd si hay pocos nodos. Utiliza el algoritmo de Bellman-Ford para eliminar todas las aristas con peso negativo del grafo. También usa el concepto de reponderar y el algoritmo de Dijkstra, los pasos a seguir son los siguientes:

- 1. Se añade un nodo q al grafo conectado a cada nodo del grafo con una arista de peso nulo.
- 2. Se utiliza el algoritmo de Bellman-Ford para calcular el camino más rápido entre el vértice q y cada uno de los nodos del grafo. Si se encuentra algún ciclo negativo, el algoritmo finaliza.
- 3. A las aristas del grafo original se les actualizan los pesos con los valores calculados de Bellman-Ford.
- 4. Para cada nodo se usa el algoritmo de Dijkstra, a partir de los pesos modificados.

En la Figura 2.12 podemos ver el proceso gráficamente.

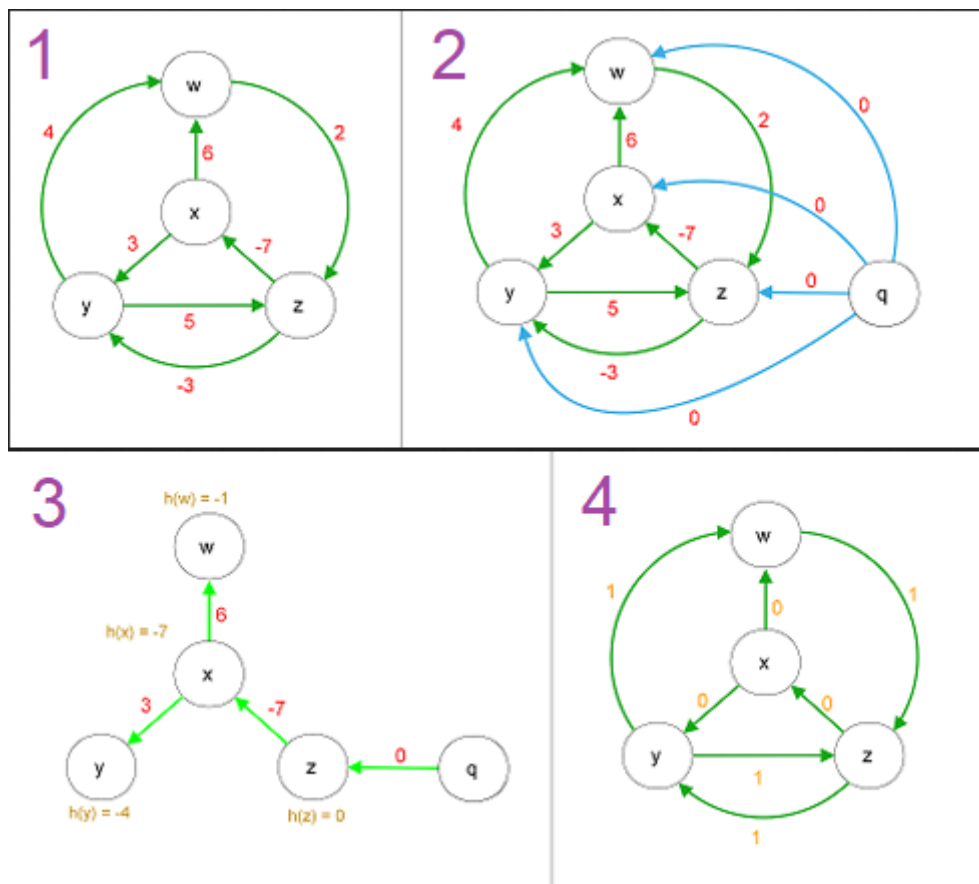


Figura 2.12: Solución con Johnson

2.4.3. Encontrar la ruta más rápida para cada *picker*

En esta sección vamos a hablar sobre el problema de encontrar la ruta más rápida posible, mediante heurísticas, para cada *batch*. Los *pickers*, una vez se les asigna una serie de pedidos con sus productos correspondientes, deben reunirlos recorriendo la menor distancia posible.

Se trata de un claro ejemplo del problema del viajante, al que nos referiremos como TSP (Traveller Salesman Problem), consiste en un viajero que debe visitar todas las localizaciones. Para resolverlo debemos encontrar la ruta más rápida visitando cada localización una vez y volviendo al punto de partida para finalizar. En problemas pequeños se puede resolver de manera sencilla, incluso mentalmente, pero en problemas grandes es muy complejo de resolver, incluso mediante algoritmos, ya que se trata de un problema NP-completo.

Existen dos clases de algoritmos para resolver el TSP: Algoritmos exactos, que encuentran la solución óptima, pero requieren mucho poder computacional; y los algoritmos heurísticos, que obtienen buenos resultados, pero no la solución óptima.

Algunos algoritmos usados típicamente para este problema son:

- **Algoritmo de búsqueda exhaustiva** [11]: Es una de las técnicas más antiguas usadas para resolver problemas, tiene en cuenta todas las posibles rutas, por lo que es un algoritmo exacto. Aunque encontraría la ruta óptima en algún momento, sería muy costoso en cuanto tiempo de ejecución emplearlo en nuestro problema.
Si tenemos un almacén pequeño podemos considerar implementar esta opción, pero el orden de complejidad es exponencial, por lo que en el momento en que aumentemos un poco el número de pasillos del almacén, esta solución daría resultados pésimos.
- **Algoritmo del vecino más cercano** [11]: Consiste en viajar desde cada producto al siguiente producto más cercano que haya que recoger. Es un problema heurístico que puede conseguir un tiempo decente, pero se pueden obtener mejores resultados mediante otros métodos. El proceso paso por paso es el siguiente:
 - 1. Elige el punto de partida (en nuestro caso el extremo izquierdo del pasillo central del almacén).
 - 2. Encuentra el producto más cercano al último visitado y muévete hasta él.
 - 3. Si queda algún producto por recoger vuelve al paso anterior. Si no continúa con el paso 4.
 - 4. Vuelve al punto de partida.

Un estudio donde se muestra su implementación y eficiencia de la Facultad de

Ciencia de Izmir, Turquía: [23]. En él se explica una mejora que se consigue siguiendo estos pasos:

- 1. Elige el punto de partida (en nuestro caso el extremo izquierdo del pasillo central del almacén).
 - 2. Encuentra el producto más cercano y muévete hasta él.
 - 3. Encuentra el producto más cercano a alguno de los dos extremos del camino y actualiza los vértices.
 - 4. Si queda algún producto por recoger vuelve al paso anterior. Si no continúa con el paso 5.
 - 5. Ve de un vértice al otro para completar el grafo (la ruta).
- **Algoritmo voraz:** El algoritmo voraz para resolver el TSP también es conocido como algoritmo de fragmentos múltiples. En el estudio que hemos comentado antes, de la Facultad de Ciencia de Izmir, Turquía: [23], nos muestran una implementación para resolver el TSP con este tipo de metodología. Se trata de construir un grafo gradualmente eligiendo cada vez el arco más corto y añadirlo siempre y cuando no forme un ciclo con menos de N vértices (siendo N el número total de vértices), incremente el grado de ningún nodo en más de 2 o ya exista en el grafo. Se consideran las distancias reales entre los productos, en lugar de en línea recta, por lo que los resultados no son los mejores que podemos conseguir.
- **Colonia de hormigas:** Consiste en que crear una población de w hormigas (en nuestro caso serían rutas), que se generan aleatoriamente y se van mejorando iteración tras iteración eliminando nodos intermedios inútiles. Es un algoritmo heurístico similar a los algoritmos genéticos, pero difícil de implementar. En este estudio se muestra una solución a nuestro problema mediante un algoritmo de colonia de hormigas: [38]. En este otro artículo [13] podemos entender con más detalle cómo funciona este algoritmo. Los pasos a seguir son los siguientes:

- 1. Una hormiga se mueve de manera aleatoria por la colonia (es decir, creamos una ruta aleatoria en nuestro almacén).
- 2. Si encuentra una fuente de comida (en nuestro caso completar una ruta, es decir, reunir todos los productos de un pedido), retorna a la colonia dejando un rastro de feromonas.
- 3. El resto de hormigas son atraídas por estas feromonas y siguen el rastro para llegar al alimento encontrado por la primera hormiga (ruta).
- 4. Al regresar a la colonia, estas hormigas dejan más feromonas, esta vez en un camino más directo que la primera.
- 5. Si existen dos rutas para llegar a un alimento las hormigas elegirán el camino más corto (el que más feromonas tenga) más veces que el camino largo.

- 6. Esto provocará cada vez más proporción de feromonas en el camino corto.
- 7. La ruta larga irá desapareciendo conforme se evaporen las feromonas.
- 8. Finalmente todas las hormigas elegirán el camino más corto.

La implementación es compleja, en la Figura 2.13 podemos ver un esquema de cómo funciona el algoritmo.

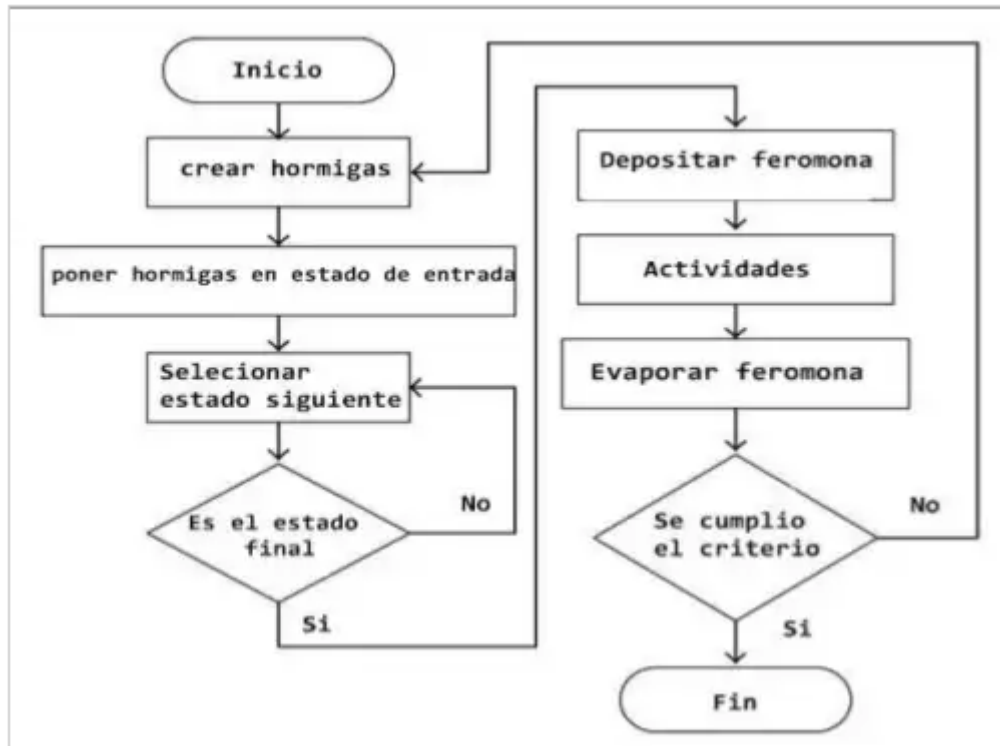


Figura 2.13: Esquema del algoritmo *Ant Colony*

-
- **Algoritmo genético:** Ya hemos explicado el funcionamiento de los algoritmos genéticos. Al igual que el caso del *batching* de los pedidos, es decir, la asignación de pedidos a cada *picker*, el TSP también se puede resolver mediante un algoritmo genético. Consiste en generar una población inicial con rutas aleatorias, en cada iteración seleccionar las mejores e ir mejorando los resultados mediante cruces, mutaciones y sustituciones hasta lograr un buen resultado. Es una buena opción para nuestro problema, aunque no encontraríamos la solución exacta, sino una aproximación, ya que se trata de un algoritmo metaheurístico. Como vamos a implementar un algoritmo genético para el problema del *batching*, vamos a centrarnos en otra solución para el TSP. En este artículo podemos ver una implementación paso por paso de un algoritmo genético sencillo para resolver el problema del viajante: [33].
 - ***Twice around the tree*** [11]: Usa un árbol de expansión (grafo que conecta todos los vértices usando el menor número de arcos) para buscar la ruta óptima generando una lista de vértices avanzando por el árbol.
 - **Algoritmo de Christofides'** [11]: Mejora el algoritmo anterior usando un árbol de expansión mínimo (considerando el peso de los arcos, el árbol tiene el menor peso total) para crear un circuito Hamiltoniano. Un circuito Hamiltoniano [6] es un camino de un grafo, que visita todos los vértices del grafo una sola vez. Esta sería una buena opción para nuestro problema, aunque también es difícil de implementar.

A continuación vamos a hablar sobre un artículo de Kees Jan Roodbergen (Universidad de Groninga, Países Bajos) y René De Koster (Universidad de Rotterdam) [30]. Se trata de un artículo de hace ya dos décadas, donde mostraban distintos algoritmos para resolver el TSP en almacenes junto a sus comparaciones y una serie de conclusiones acerca de cada algoritmo.

En su caso existen varios pasillos centrales, en lugar de uno, y el punto de partida está en la parte inferior izquierda del almacén, en lugar de a media altura, como es en nuestro caso. En la Figura 2.14 podemos observar la configuración de almacén y productos que utilizan a lo largo del artículo. Llamaremos pasillo a los pasillos completos desde un lado al otro del almacén (*pick aisle* en la imagen), y subpasillos a pasillos que atraviesen solamente un bloque de productos (*subaisle*).

La solución óptima para este problema la podemos ver en la Figura 2.15.

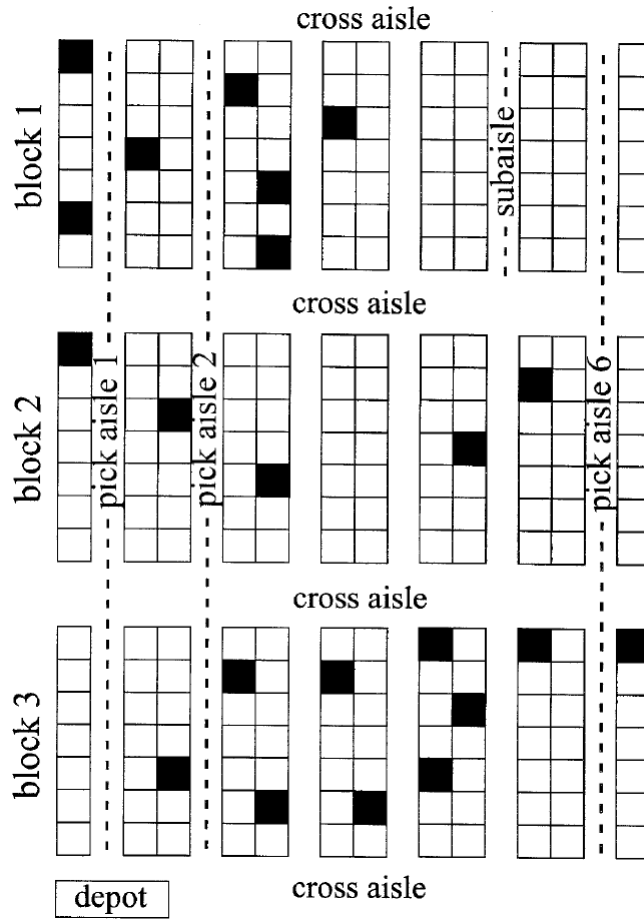


Figura 2.14: Configuración de almacén

- **Programación Dinámica:** Construimos la solución a partir de subsoluciones. Calculamos la distancia más corta entre cada par de puntos teniendo en cuenta solo ese par de puntos, a continuación añadimos otro punto, y calculamos de nuevo la distancia (nos quedamos con la más corta), añadimos otro punto... y así sucesivamente, hasta haber incluido todos los puntos del grafo, obteniendo la solución exacta que buscamos. Es un problema exacto, por lo que si se consigue realizar en un tiempo aceptable, sería una buena opción. Vamos a implementar este algoritmo, por lo que más adelante lo explicaremos en detalle. Por ahora, podemos ver en Figura 2.16 el resultado de aplicar Programación Dinámica al pedido de la Figura 2.14.

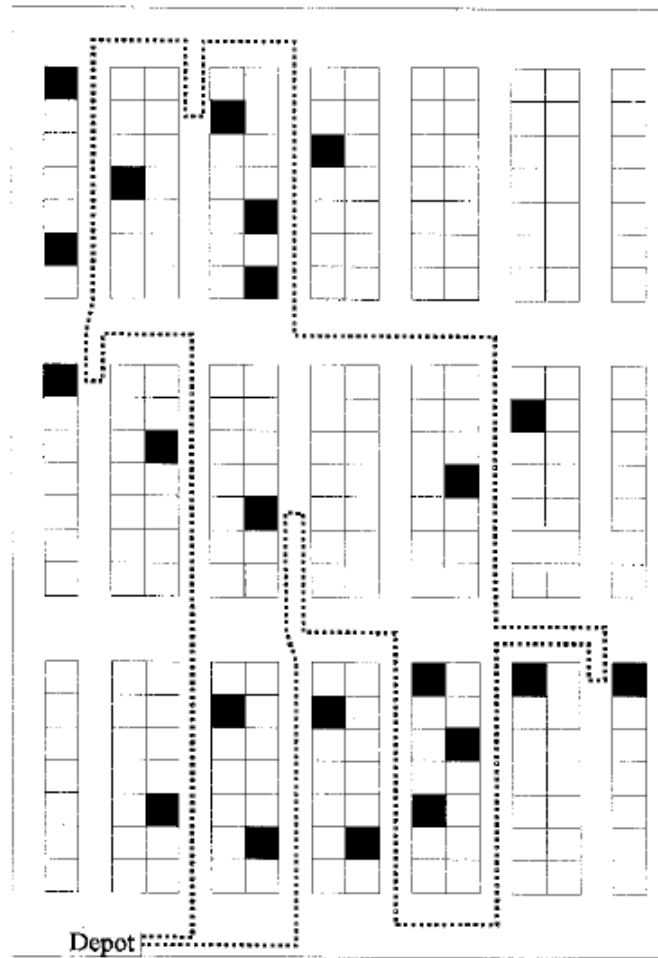


Figura 2.15: Solución óptima

- ***S-shape***: Consiste en recorrer cualquier pasillo completo si tiene algún ítem que se deba recoger, ignorando los pasillos que no tengan ningún producto por recoger. Se trata de una solución heurística, y en el mismo artículo que antes podemos ver una implementación paso por paso. Se trata de una buena solución para almacenes donde existan muchos pasillos centrales, pero no es la opción adecuada para diseños sin pasillo central o como el nuestro, con un único pasillo central. Los pasos a seguir son los siguientes:
 1. Determinar el pasillo más a la izquierda (llamémoslo **L**) y más a la derecha (**R**) con algún *ítem* a recoger.
 2. La ruta empieza yendo a la parte delantera del pasillo de la izquierda.
 3. Ahora avanzamos hasta la parte delantera del subpasillo **L**.
 4. Hacia la derecha hasta encontrar un subpasillo con algún producto a recoger: si es el único subpasillo en este bloque con productos a recoger entras y sales del pasillo por la misma parte. Si hay más de un subpasillo con productos

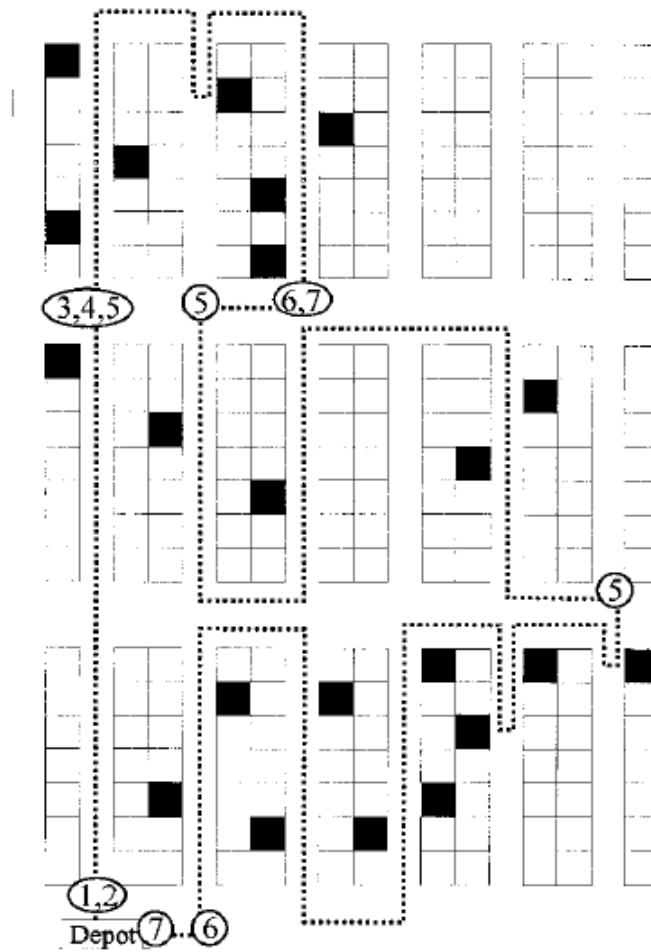


Figura 2.16: Resultado mediante Programación Dinámica

en este bloque lo recorreremos por completo.

5. En este momento, el *picker* está al fondo del bloque actual.
 - (a) Si quedan productos en el bloque actual determinar la distancia al pasillo más a la izquierda y más a la derecha y viajar al más cercano de esos dos. Recorrer el subpasillo por completo y continuar con el paso 6.
 - (b) No quedan productos en el bloque actual. Continúa en el mismo pasillo para llegar al cruce con el siguiente pasillo y continúa en el paso 8.
6. Si quedan productos en este bloque se avanza por el pasillo horizontal hasta el siguiente subpasillo con algún *item* a recoger y se atraviesa por completo. Repetimos el proceso hasta que quede un subpasillo exacto (**P**) con productos a recoger.
7. Ir al subpasillo **P**, recoger los productos e ir a la parte frontal del pasillo. Puede darse el caso de que se atraviesa de la parte posterior a la frontal, o de entrar y salir por la parte frontal.

- En la Figura 2.17 podemos ver el resultado de usar el algoritmo *S-shape* para recorrer el almacén de la Figura 2.14.



- **Largest gap:** Se trata de una heurística en la que se sigue el perímetro de los pasillos entrando cuando sea necesario. Se comienza por el bloque más lejano y sigue bloque por bloque. En el artículo explican la implementación paso por paso:

1. Determinar el pasillo más a la izquierda (llamémoslo **L**) y más a la derecha (**R**) con algún *item* a recoger.
2. La ruta empieza yendo a la parte delantera del pasillo de la izquierda.
3. Ahora avanzamos hasta la parte delantera del subpasillo **L**.
4. Hacia la derecha hasta encontrar un subpasillo con algún producto a recoger: si es el único subpasillo en este bloque con productos a recoger entrar y salir del pasillo por la misma parte. Si hay más de un subpasillo con productos en este bloque lo recorreremos por completo.
5. En este momento, el *picker* está al fondo del bloque actual.
 - (a) Si quedan productos en el bloque actual determinar el subpasillo del bloque actual con algún producto más lejano de la posición actual (llamémosle **Lej**). Continuar en el paso 6.
 - (b) No quedan productos en el bloque actual. Continúa en el mismo pasillo para llegar al cruce con el siguiente pasillo y continúa en el paso 9.
6. Seguir el camino más corto a través del pasillo horizontal visitando todos los subpasillos en los que haya productos por la parte posterior terminando en el último subpasillo del bloque actual. Cada subpasillo se atraviesa por el mayor hueco (*largest gap*). Esto supondrá entrar por algunos pasillos tanto por la parte trasera, izquierda a derecha, como por la parte frontal, de derecha a izquierda.
7. Atravesar el último subpasillo del bloque actual para llegar a la parte frontal del bloque.
8. Realizar el mismo proceso que en el paso 6, pero en lugar de izquierda a derecha, de derecha a izquierda.
9. Si el bloque actual no es el bloque inferior, volver al paso 5.
10. Volver al punto de partida.

En la Figura 2.18 podemos ver la solución utilizando este algoritmo.

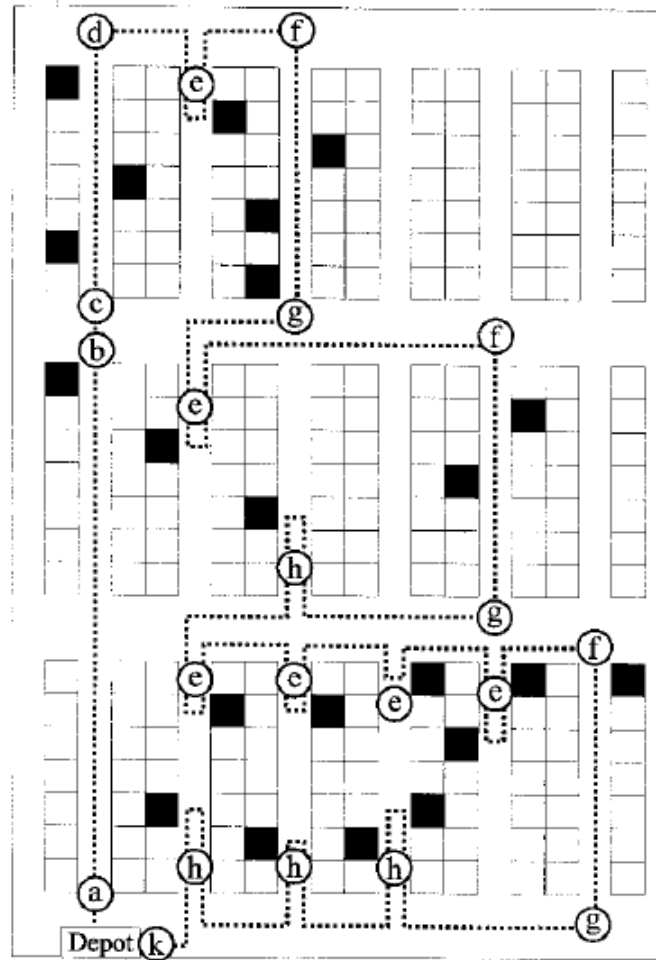


Figura 2.18: Resultado mediante *Largest Gap*

- **Pasillo por pasillo:** De nuevo, se trata de una heurística buena cuando se trata de almacenes con múltiples pasillos centrales, y podemos ver su implementación el mismo estudio. Consiste en recoger todos los productos del pasillo 1, continuar con los del pasillo 2, y así sucesivamente hasta haber recorrido todos. No se recorren los pasillos en toda su longitud, sino hasta donde sea necesario, pudiendo salir de un pasillo por el mismo punto que se ha entrado.

En la Figura 2.19 tenemos la solución utilizando el algoritmo pasillo por pasillo.

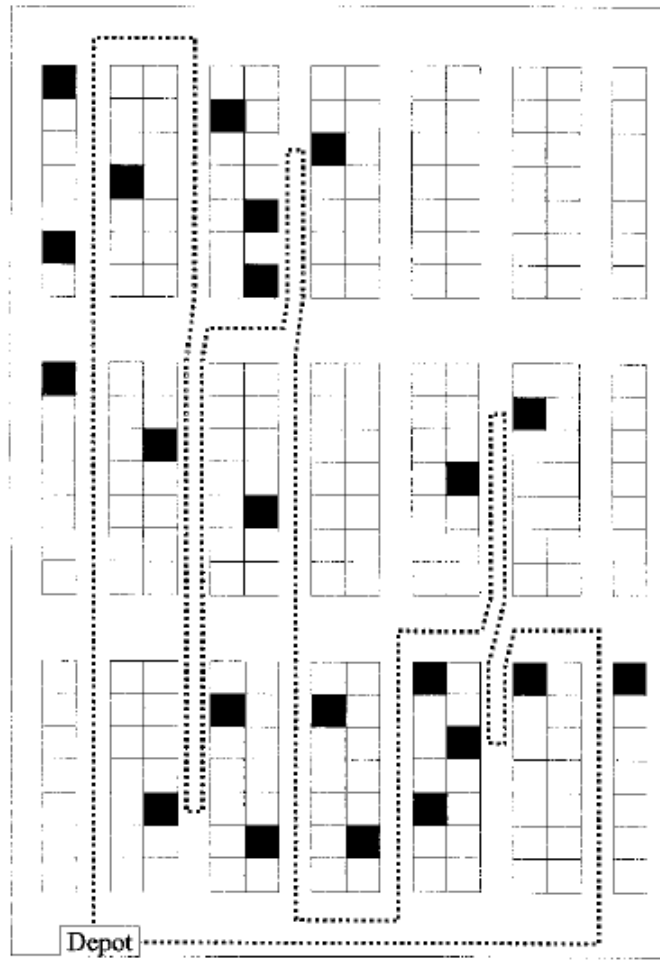


Figura 2.19: Resultado mediante *Aisle by Aisle*

En el artículo realizan un estudio con varias configuraciones y varios pedidos para cada uno de los algoritmos desarrollados. Las conclusiones principales son las siguientes: La mejor opción después de la solución óptima es la programación dinámica, con resultados muy buenos. Le sigue el algoritmo *S-shape*, con resultados peores, y por último los algoritmos *Largest gap* y *Aisle-by-aisle*, con resultados bastante peores.

También hacen experimentos añadiendo un pasillo central adicional, que en ocasiones ofrece mejores resultados (si el almacén es muy grande), y en otras resultados peores (si es pequeño), ya que añadir un almacén central significa aumentar el tamaño total del almacén por la pérdida de espacio.

2.4.4. Asignación de los pedidos a los *pickers*

Lo siguiente que debemos resolver es qué pedidos tienen que reunir los *pickers* en cada ruta. Es muy importante que trabajemos esta parte del algoritmo, ya que si agrupamos distintos productos por cercanía o por tipo de productos (productos estándar, frutas, congelados), podremos reducir mucho más tiempo total que si simplemente nos dedicamos a buscar las rutas óptimas de los trabajadores pero sin elegir qué pedidos reúne cada uno.

Existen muchas posibilidades para resolver este problema, debemos definir correctamente cada elemento y buscar la mejor solución en el espacio de búsqueda. El coste computacional que tardaríamos en evaluar todas las posibilidades es extremadamente alto, ya que si tenemos, por ejemplo, 40 pedidos, y en cada ruta se pueden completar 4 pedidos, existirían 40^4 posibles combinaciones, más de 2.5 millones de opciones.

Por el momento, como ya hemos comentado, la mayoría de supermercados y almacenes no tienen ningún tipo de software para mejorar estos procesos, por lo que se trata de un trabajo humano, ya que existe un encargado que es quién ordena a cada trabajador qué pedidos reunir. Si suponemos un almacén donde puede llevar 4 cajas cada *picker*, lo más probable es que el encargado vaya repartiendo en orden los pedidos, de 4 en 4. Sin embargo, con una correcta implementación podemos calcular configuraciones de pedidos de manera que se ahorre muchísimo tiempo. Puede que no encontremos la solución óptima en cada caso, ya que para plantear este problema debemos considerar búsquedas heurísticas, pero sí encontraremos buenas soluciones en poco tiempo.

Dos soluciones interesantes para nuestro problema son algoritmos genéticos y búsqueda tabú. Mientras que el algoritmo genético siempre alcanza buenas soluciones, la búsqueda tabú proporciona tiempos más reducidos de ejecución. Puede ser interesante implementar ambas opciones y comparar los resultados.

Acerca de la primera propuesta, el algoritmo genético, existe un estudio de 2005, por Chih-Ming Hsu, Kai-Ying Chen y Mu-Chen Chen [21]. En él definen un algoritmo genético simple para resolver el problema del *batching* de los pedidos. Definen cinco restricciones iniciales:

- Minimizar el número de *batches*. Se trata de una buena opción, ya que es muy probable que la solución con menos rutas totales posibles sea la mejor de todas. Sin embargo, puede haber casos específicos donde, por ejemplo, 19 *batches* pueden suponer más tiempo total de recogida que 20 *batches*, si en ese último caso las rutas están mejor agrupadas.
- Limitar el volumen total de los pedidos en un *batch*.
- El volumen total de un único pedido no puede superar la capacidad total del *picker*.
- Todos los pedidos deben ser recogidos.

- No se puede dividir ningún pedido en varios *batches*.

En cuanto a la codificación, usaron un vector de enteros, cada uno de ellos representa un pedido e indican a qué *batch* pertenece. Por ejemplo, el individuo (1, 2, 3, 2, 1) agrupa el primer pedido y el quinto en el *batch* 1, el segundo y el cuarto en el *batch* 2, y el tercero en el *batch* 3, teniendo un total de 5 pedidos y 3 *batches*.

También existe un estudio de 2011 de Sören Koch y Gerhard Wäscher, que no se centran en un simple algoritmo genético con una codificación tradicional. En él, introducen un *Grouping Genetic Algorithm*, que se trata de un algoritmo genético muy enfocado a problemas de agrupamiento, usado sobre todo en problemas de *clustering* de datos. Combinan este algoritmo con un procedimiento de búsqueda local, que resulta en un algoritmo híbrido de gran rendimiento. BUSCAR

2.4.5. Colocación de los productos en las cajas

Cuando un *picker* vaya reuniendo los distintos productos de cada pedido, debe saber en qué caja colocarlo, y cuál es la ubicación exacta dentro de esa caja. Esto es fundamental, ya que la optimización de las cajas que hemos realizado responde a una colocación exacta de los productos en las cajas. Si los trabajadores colocan los productos en las cajas arbitrariamente, no van a encontrar la distribución del tamaño óptima, y por tanto necesitarán más cajas para el proceso, y por ende, más tiempo.

2.5. Situación actual

Los algoritmos que hemos explicado para recorrer rutas y resolver el TSP son soluciones informáticas del problema, que presentan mejoras evidentes en cuanto al tiempo de las rutas de los *pickers*, pero son más difíciles de implementar que las soluciones tradicionales. Vamos a explicar cómo trabajan la mayoría de almacenes en la actualidad, para conocer mejor su situación y cuáles son los métodos que siguen los *pickers* al reunir los productos de los pedidos de sus clientes. Para ilustrar los métodos vamos a tomar como ejemplo el almacén de la Figura 2.20, donde los pasillos están coloreados de marrón claro, el punto de inicio en marrón oscuro y los estantes de los distintos productos en gris. Es un simple ejemplo, ya que en nuestro caso el punto de partida se encuentra en el extremo izquierdo del pasillo central.

Los métodos de recogida más usados son *Order picking*, *Wave picking*, *Zone picking* y *Batch picking* (10).

2.5.1. *Order picking*

Este método es el más lento de todos los que vamos a ver, pero es el más utilizado, sobretodo en pequeños negocios. Consiste en completar un solo pedido en cada ruta.

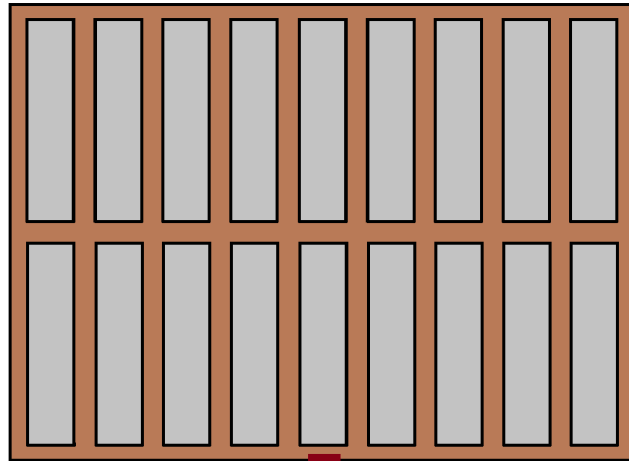


Figura 2.20: Ejemplo de almacén

Las rutas son cortas, pues solo se recorren los pasillos que tengan algún producto del pedido actual. Es altamente ineficiente, debido a la cantidad de recorridos que hay que realizar en grandes almacenes.

Supongamos que tenemos que recoger los productos de la Figura 2.21, representados como círculos rojos. En la Figura 2.22 podemos ver la ruta que habría que tomar para completar ese pedido.

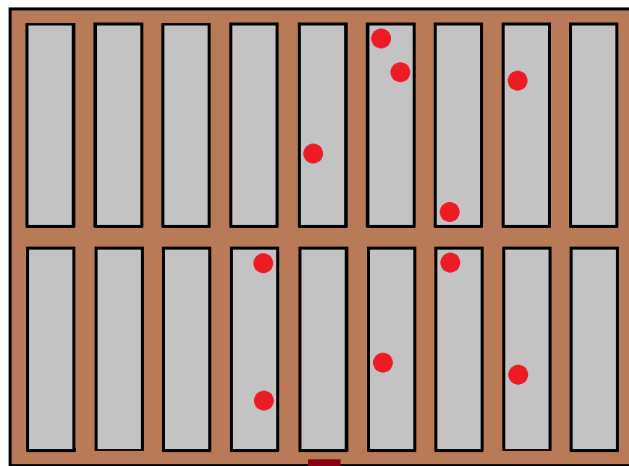


Figura 2.21: Ejemplo de almacén con productos

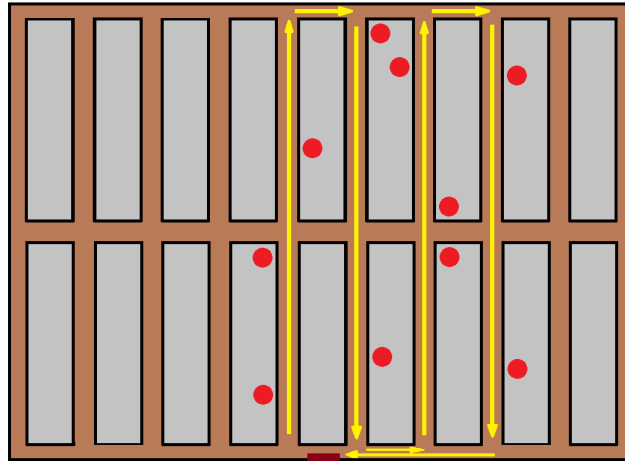


Figura 2.22: Método *order picking*

2.5.2. *Wave picking* (2)

Consiste en asignar a los *pickers* una lista de X pedidos (normalmente 4, 8 o 12) que puedan reunir en un único viaje, es decir, en una vuelta completa por todos los pasillos del almacén. Supongamos otro ejemplo de productos en este caso, donde los *pickers* tienen que recolectar todos los productos de la Figura 2.23.

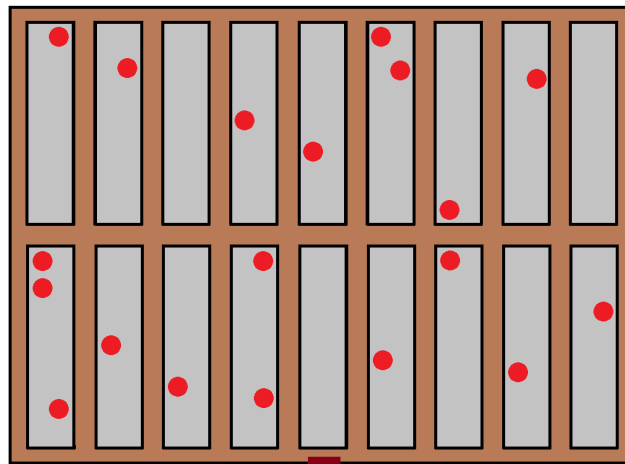


Figura 2.23: Ejemplo de almacén con productos

La ruta mediante el método *wave picking* sería la que podemos ver en la Figura 2.24, que, como es evidente, no parece muy efectiva, aunque es muy sencilla de ejecutar. En el caso de que haya que pasar por todos o casi todos los estantes (como en este

caso) no es una mala opción, pero en la mayoría de ocasiones no ocurrirá esto, sino que habrá que visitar unos pocos estantes de todo el almacén.

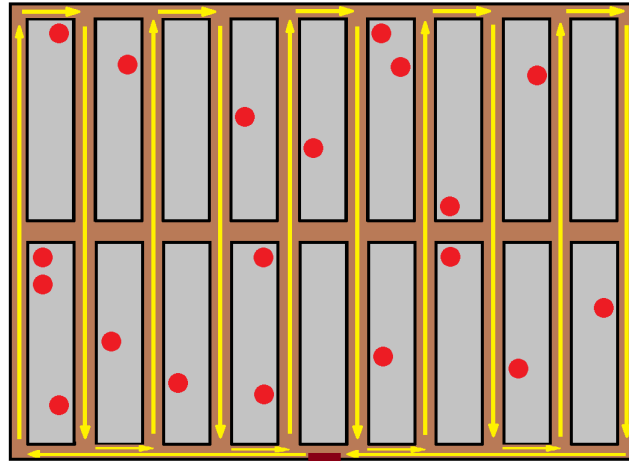


Figura 2.24: Método *wave picking*

Este método reduce el número de viajes de los trabajadores de ida y vuelta, pero se trata de una estrategia muy poco eficiente, porque visita sitios que probablemente se deberían evitar. Un *picker* no puede comenzar su ruta hasta que no tiene una lista con pedidos suficientes, ya que como se recorren todos los pasillos sí o sí, cuantos más pedidos lleve, mejor.

2.5.3. *Zone picking* (3)

También se conoce como el método *pick and pass* (recoge y pasa). Consiste en dividir el almacén en distintas áreas y asignar diferentes *pickers* a cada área. Después, se mueven contenedores entre cada zona para ir completando los distintos pedidos de una sola vez. Es un método más efectivo que el anterior, pero también es más difícil de llevar a cabo en un entorno real debido a esa fase necesaria de consolidación en la que algún trabajador debe agrupar los pedidos y verificar que se hayan completado correctamente.

Supongamos el mismo ejemplo anterior, de la Figura 2.23. Siguiendo el método de *Zone picking*, una manera de dividir nuestro almacén en distintas secciones, sería la que vemos en la Figura 2.25.

En la Figura 2.26 podemos ver las rutas que seguirían los 3 trabajadores.

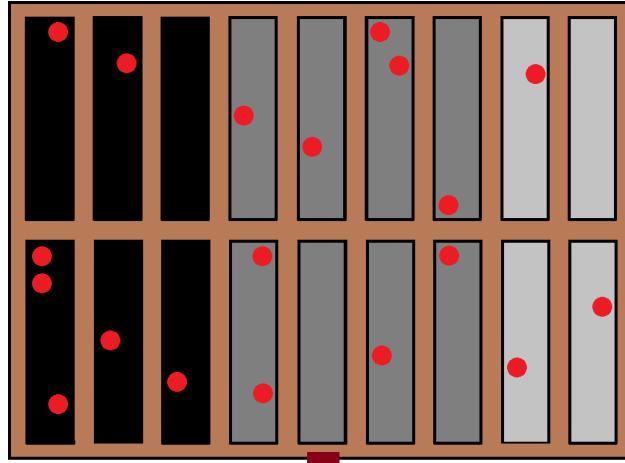


Figura 2.25: Ejemplo de almacén con productos

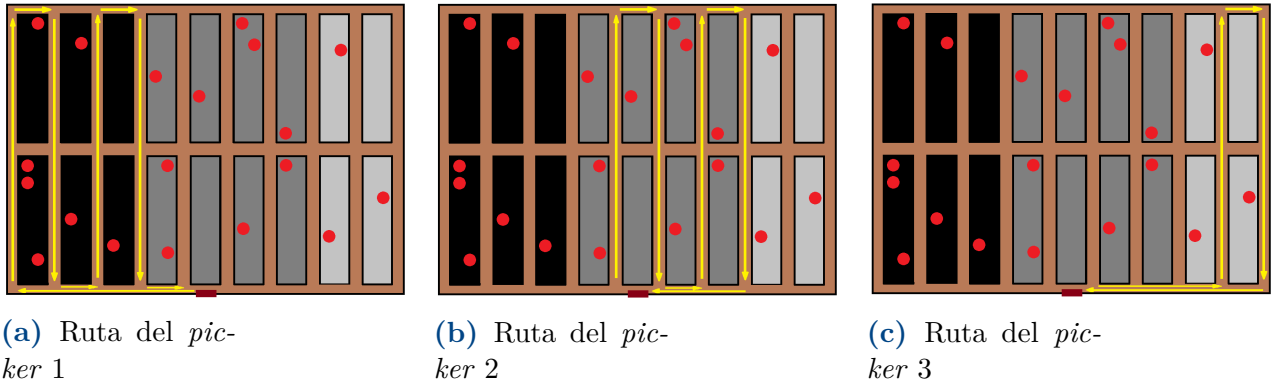


Figura 2.26: Método *zone picking*

2.5.4. *Batch picking* (4)

Imaginemos que hay que recoger todos los productos de la Figura 2.27 y que no se pueden recolectar todos en una sola tanda porque son demasiados.

Si no queremos dividir los pedidos entre distintos trabajadores (*Zone picking*), la única alternativa que nos quedaría sería usar el método *Wave picking*, pero tendríamos que realizar dos recorridos completos para ello. Ahí es donde surge el método *Batch picking*, o recogida por lotes, que consiste en que un *picker* combine productos de distintos pedidos en un recorrido. Por ejemplo, podría realizar dos recorridos parciales, como se ve en la Figura 2.28, de manera que cada estante se visite una sola vez.

Después de realizar los dos recorridos, el *picker* ordenaría los productos y completaría los pedidos. Es un método efectivo, pero también difícil de implementar, como *Zone picking*.

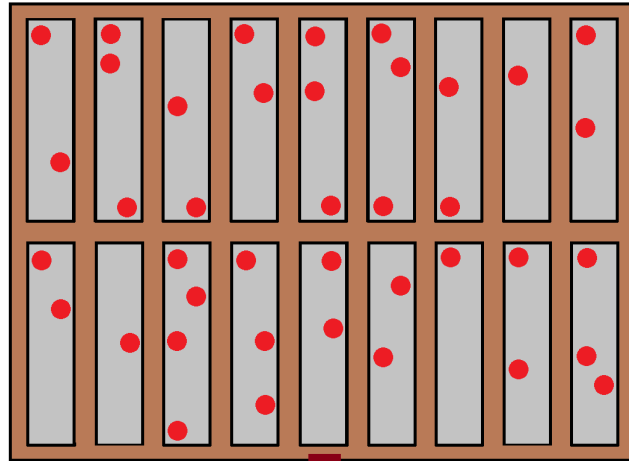
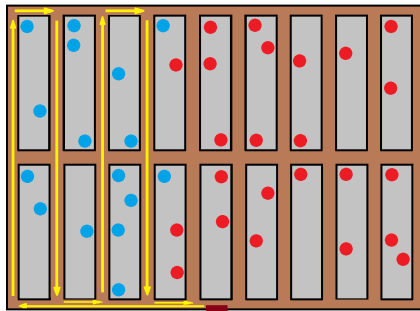
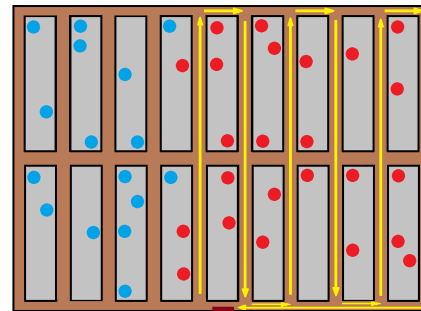


Figura 2.27: Ejemplo de almacén con productos



(a) Primer recorrido



(b) Segundo recorrido

Figura 2.28: Método *batch picking*

2.5.5. Comparación de los distintos métodos

Según un estudio (5) de la empresa LOGIWA, dedicada al software de almacenes e inventarios, el método *Zone picking* y *Batch picking* (siendo muy similares estos dos en cuanto a rendimiento) reducen a la mitad el tiempo empleado por el método *Order picking*, lo que supone el doble de rendimiento. En cuanto al método *Wave picking*, es más efectivo que *Order picking*, pero no tanto como los otros 2 métodos.

3. Implementación

Para resolver este problema, como ya hemos dicho, necesitamos resolver tres sub-problemas que componen el trabajo global: la asignación de los distintos pedidos a reunir por los *pickers*, la optimización de los pedidos en cajas que debemos usar y el cálculo de las rutas de los trabajadores por el almacén.

Además, necesitamos una interfaz para poder observar las rutas finales obtenidas por los algoritmos.

3.1. Interfaz gráfica

Para crear la interfaz gráfica, hemos tomado como referencia una página web del profesor e investigador holandés Kees Jan Roodbergen, [37]. En esta página se puede configurar el almacén de las siguientes maneras (Figura 3.1:

- Número de bloques. Por defecto dos, uno superior y uno inferior, al igual que en nuestro caso. Sin embargo, se puede aumentar este parámetro hasta 27 bloques.
- Número de pasillos, entre 3 y 41.
- Número de localizaciones por pasillo, entre 1 y 12.
- Punto de partida: En la parte inferior de cualquiera de los pasillos indicados. En nuestro caso se encuentra en la izquierda entre los dos bloques.
- Longitud y anchura del pasillo como parámetros opcionales.

Una vez establecida la configuración del almacén, hay que definir el número de productos, que se reparten aleatoriamente por todas las localizaciones, Figura 3.2.

Después se elige el método deseado para resolver la ruta (S-shape, largest gap, aisle-by-aisle, etc.) y por último se observan los resultados, como se puede observar en la Figura 3.3.

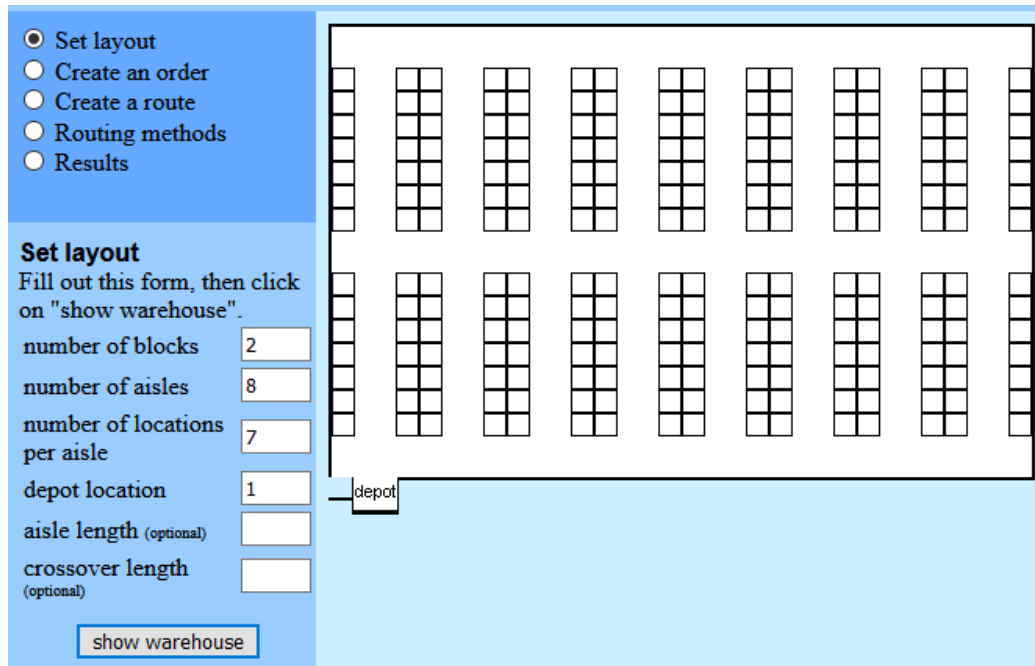


Figura 3.1: Interfaz Roodbergen

En nuestro caso, hemos desarrollado una interfaz similar, como se puede observar en la Figura 3.4.

En este caso, la casilla roja es el punto partida, como ya hemos dicho, en la parte izquierda del pasillo central. Los productos son los cuadrados azules, y los pasillos se indican con las casillas grises, por donde los *pickers* pueden moverse en el almacén. Si resolvemos esta ruta mediante programación dinámica, el resultado que obtenemos es el que se puede ver en la Figura 3.5.

Como se puede observar, con unas flechas azul claro se ve claramente el resultado a seguir por los *pickers* (sabiendo que siempre hay que recorrer todos los productos del bloque superior antes de pasar al bloque inferior). En la parte inferior derecha aparecen los pedidos correspondientes a esta ruta, en este caso el 14, 39, 88 y 53, aunque esto lo explicaremos más adelante.

3.2. *Batching* de los pedidos

Para la asignación de los distintos pedidos a los *pickers* hemos desarrollado un **algoritmo genético** y un algoritmo de **búsqueda tabú**.

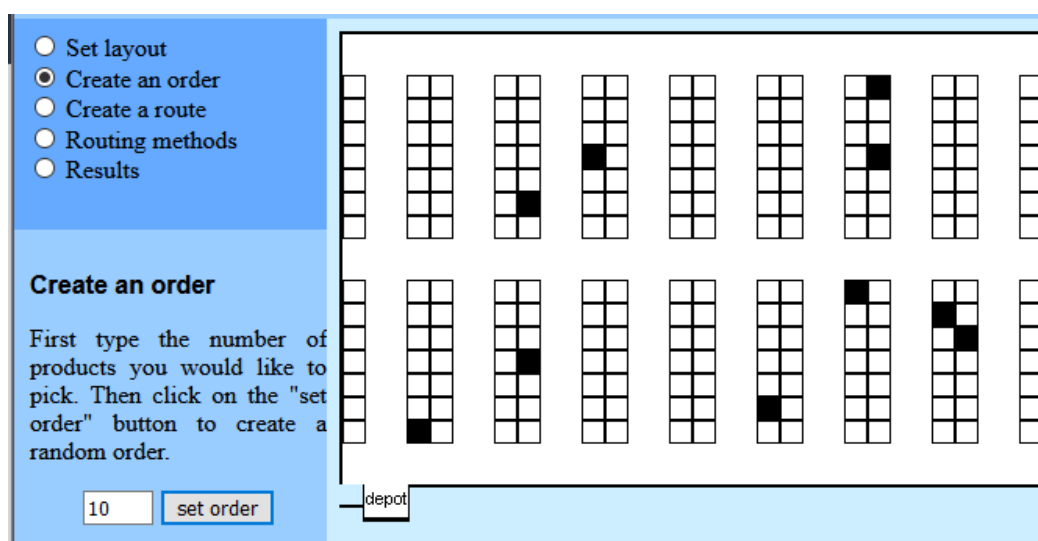


Figura 3.2: Pedido Roodbergen

3.2.1. Algoritmo Genético

En este apartado vamos a describir como hemos implementado nuestro algoritmo genético paso a paso.

Representación de los individuos

En primer lugar, debemos definir como representaremos a los individuos de nuestro algoritmo genético. Pensemos por ejemplo en un algoritmo que sirva para crear laberintos lo más difíciles de resolver posibles. En este caso, los individuos serían simplemente listas de listas de tamaño número de filas x número de columnas, donde cada celda tendría por ejemplo un valor de 0 para espacios libres por los que poder caminar y valor de 1 para los muros, indicando que son celdas por las que no se puede avanzar.

En nuestro problema, lo que necesitamos tener como individuos son posibles *batches* para los pedidos a recoger. Por tanto, la manera en la que hemos codificado nuestra población es como un diccionario compuesto de listas, representando cada una de ellas un individuo. Cada uno de esos individuos está formado por listas, una para cada una de las rutas necesarias para completar todos los pedidos. Esas listas tendrán a su vez dos elementos: la capacidad transportada en esa ruta y los pedidos a recoger la ruta. El diccionario tendrá tantos elementos como el tamaño de la población que definamos. Es decir, para un tamaño de población de cincuenta individuos, el diccionario tendrá cincuenta elementos.

Por ahora, para entender mejor la codificación a la que nos referimos, hemos definido un pequeño ejemplo. Imaginemos que tenemos 5 pedidos (compuestos por un único producto): p1, p2, p3, p4 y p5. La carga de estos pedidos es 5, 3, 7, 2 y 6, respecti-

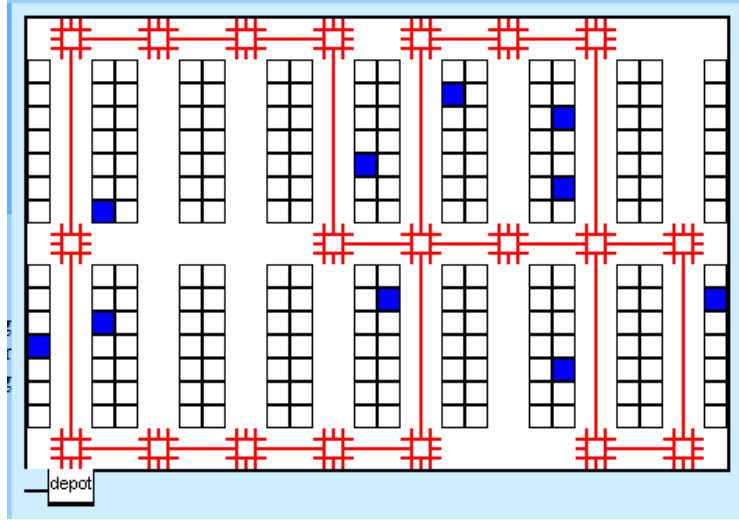


Figura 3.3: Solución Roodbergen

vamente. Digamos que la carga que puede llevar un *picker* como máximo es 10, y que nuestra población tiene un tamaño de 3 individuos. En la Figura 3.6 podemos ver un ejemplo de individuo para este caso.

Creados aleatoriamente, el primer individuo consta de tres rutas (p1-p2, p3-p4 y p5), el segundo individuo lo forman tres rutas (p2-p3, p4-p5 y p1) y el tercero está formado por otras tres rutas (p3, p1-p2-p4 y p5). En este caso, los tres individuos están compuestos del mismo número de rutas. Es decir, para estos tres individuos, son necesarias tres rutas para recoger todos los pedidos necesarios de la instancia. Esto no tiene por qué ser así, cada individuo puede tener un número distinto total de rutas. Una breve explicación para entender la codificación de los individuos: el primer individuo consiste en una ruta de 8 de carga, compuesta por los pedidos 1 y 2 (5 y 3 de carga, respectivamente); una ruta de 9 de carga, compuesta por los pedidos 3 y 4 (7 y 2 de carga, respectivamente); y una ruta de 6 de carga, compuesta por el pedido 5.

Parámetros

Antes de nada debemos definir algunos parámetros importantes para el algoritmo genético:

- **Semilla:** Para todos los sucesos aleatorios que ocurran usaremos una semilla fija, para permitir la reproducibilidad de los experimentos. Las primeras pruebas que hagamos serán con la semilla 1.
- **Tamaño de la población:** Normalmente se usan tamaños de entre 50 y 100 individuos. Nosotros usaremos 50 como tamaño de población inicialmente.
- **Iteraciones:** Se pueden establecer varias maneras para finalizar el algoritmo genético. Nosotros en principio estableceremos un número de iteraciones fijas (por

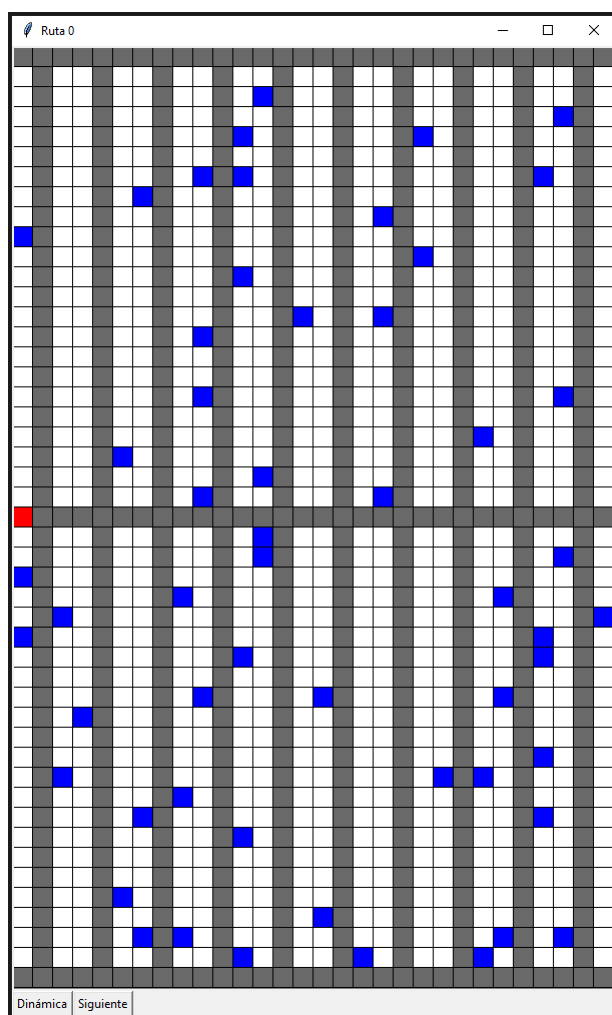


Figura 3.4: Interfaz implementada

ejemplo, 50), al igual que pararemos la ejecución si en las últimas 20 iteraciones el algoritmo no ha mejorado nada.

- **Probabilidad de mutación:** Aunque normalmente se utilizan probabilidades de mutación de 0.05 o 0.1, posteriormente veremos como en nuestro caso necesitamos utilizar una mayor probabilidad.
- **Torneo:** Explicaremos esto más adelante, cuando definamos nuestra función de selección.

Crear la población inicial

Para crear nuestra población inicial declaramos un diccionario vacío (que será nuestra población) y una lista aleatoria de números de tamaño igual al número de pedidos de la instancia actual. A continuación creamos un individuo con una sola ruta vacía, e iremos recorriendo los pedidos para añadirlos en las rutas existentes si caben o en una

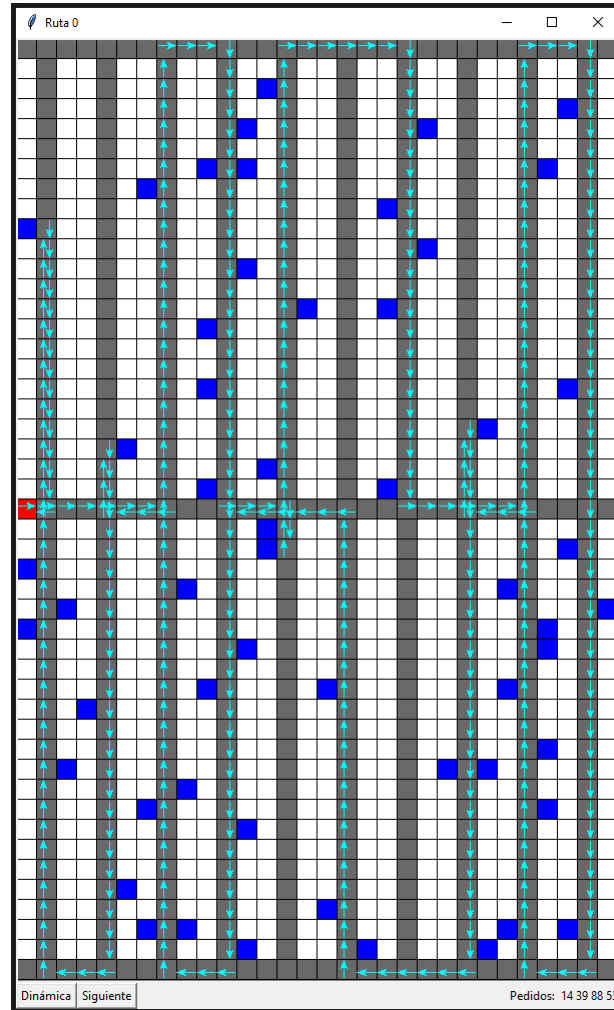


Figura 3.5: Solución en la interfaz implementada

nueva ruta en caso contrario (restamos a la capacidad del *picker* la carga actual de la ruta; el pedido cabe si su carga es menor al resultado de esa resta) . Cada vez que añadimos un pedido a una ruta actualizamos la carga transportada en esa ruta para poder realizar el siguiente cálculo. Actualizar la carga consiste en sumar a la carga actual de la ruta la carga del nuevo pedido. Completado el primer individuo continuamos con el resto hasta completar la población. Es importante comentar que en un diccionario auxiliar almacenamos las listas aleatorias con las que vamos creando los individuos, ya que las necesitaremos posteriormente para la función de cruce. A continuación podemos ver un fragmento de pseudocódigo para entender mejor la función.

```
def crearPoblacionInicial(self):
    ...
    población inicial = {}
    for i in range(len(tamPoblacion):
        creamos un individuo vacío
```



```
individuos = {1:[[8,[pedido1, pedido2]], [9,[pedido3, pedido4]], [6,[pedido5]]],
              2:[[10,[pedido2, pedido3]], [8,[pedido4, pedido5]], [5,[pedido1]]],
              3:[[7,[pedido3]], [10,[pedido1, pedido2, pedido4]], [6,[pedido5]]]}
```

Figura 3.6: Ejemplo de individuo

```
pedidosRandom = lista aleatoria de tamaño igual al número de pedidos
for pedido in pedidosRandom:
    if (el pedido i cabe en alguna ruta de este individuo):
        se añade a esa ruta (en la primera que quepa)
        se actualiza la capacidad de esa ruta
    else:
        se crea otra ruta con ese pedido
        se actualiza la capacidad de esa ruta (carga de ese pedido)
se añade el individuo a la población inicial
return población inicial
```

Función de evaluación

La función de evaluación o *fitness* mide el grado de adecuación de un individuo, es decir, como de bueno es. En nuestro caso, el *fitness* de un individuo será el tiempo total en recorrer todas las rutas. Recorreremos cada uno de los individuos aplicando el algoritmo de rutas deseado para cada una de las rutas de ese individuo. En este caso, utilizaremos el algoritmo de programación dinámica que explicaremos posteriormente. Finalizadas todas las rutas de un individuo, sumaremos el tiempo en completar cada una de ellas para calcular su *fitness*. En un diccionario llamado tiempoIndividuos almacenaremos esos tiempos totales para cada elemento de la población. Cuando hayamos evaluado a todos los individuos, tendremos un valor para evaluar cada uno de ellos y saber cuáles son mejores y cuáles son peores. En el siguiente fragmento de pseudocódigo mostramos el funcionamiento del método:

```
def evaluarFitness(self):
    ...
    tiempo individuos = {}
    for i in range(len(tamPoblacion):
        tiempo = 0
        rutas = todas las rutas de individuo actual
        for ruta in rutas:
            aplicamos el algoritmo de programación dinámica a la ruta actual
            sumamos a tiempo el tiempo en recorrer esa ruta
        se añade ese tiempo a tiempo individuos
    return tiempo individuos
```

Método de selección

El mecanismo de selección es quien se encarga de decidir qué individuos tendrán descendencia en la siguiente población y cuáles no. Existen muchos métodos de selección: basado en *fitness*, basado en rango, selección por torneo determinística, selección por torneo probabilística, etc.

En nuestro caso vamos a utilizar una selección por torneo, ya que este tipo de selección disminuye mucho la presión selectiva. De esta manera se evita que algún “superindividuo” eclipse al resto de individuos. Definiremos un tamaño de torneo de por ejemplo 0.2 x tamaño de la población, es decir, para una población de 100 individuos el torneo será de 20 individuos. El funcionamiento de este método es sencillo: se ejecuta un bucle tantas veces como individuos tenga la población, y en cada ejecución del bucle se escogen al hacer X individuos (en el ejemplo propuesto 20) y de ellos se elige el que mayor *fitness* tenga. De esta manera, los mejores individuos serán elegidos varias veces y los peores individuos serán escogidos pocas veces o ninguna.

Operador de cruce

El operador de cruce nos permite combinar la información de dos padres para crear dos descendientes. El cruce es exploratorio, es decir, nos conduce a individuos con información de los dos padres, pero sin introducir nuevo material genético (no añade diversidad). Algunos cruces para representaciones enteras son: por un punto, por varios puntos o cruce uniforme.

En nuestro caso los individuos son listas de números, definiendo el orden en el que se escogerán para completar las rutas (al rellenar el diccionario de la población). Usaremos un cruce por un punto, que consiste en separar la lista en dos partes, en nuestro caso ese punto será la mitad de la lista. Generaremos el primer hijo con la primera parte del primer padre y la segunda parte del segundo padre, y el segundo hijo de manera inversa.

Pongamos un ejemplo donde tenemos diez pedidos y los dos individuos que nos toca cruzar son los de la Figura 3.7.

2	4	1	9	5	8	6	3	10	7
6	1	2	7	8	10	9	4	5	3

Figura 3.7: Padres operador de cruce

El primer hijo se formará con la primera mitad del primer padre y la segunda mitad del segundo padre (se continúa por el principio del segundo padre hasta conseguir todos

los pedidos). En la Figura 3.8 se puede apreciar mejor cómo se forma el primer hijo.

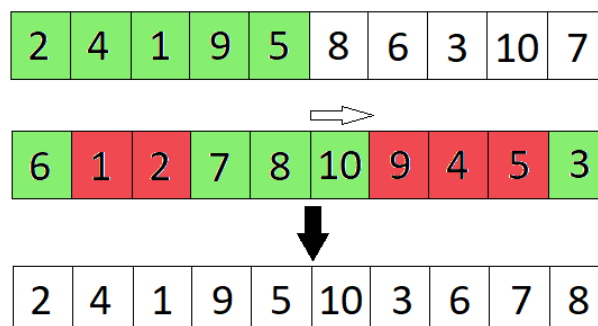


Figura 3.8: Primer hijo operador de cruce

De igual manera, en la Figura 3.9 podemos ver la formación del segundo hijo.

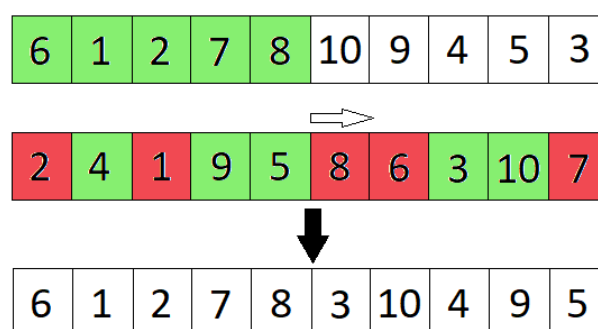


Figura 3.9: Segundo hijo operador de cruce

Operador de mutación

Tras aplicar el cruce a todos los padres de la población, nos toca aplicar el operador de mutación a algunos de los hijos. Al igual que el cruce, aplicaremos estos cambios a las listas que definen a los individuos, para posteriormente formar el diccionario que compone la población.

En cuanto a operadores de mutación, vamos a implementar dos tipos distintos, para evaluar cual de los dos nos proporciona mejores resultados. En primer lugar, un operador que se encargue de coger X pedidos (por ejemplo, un número aleatorio de entre 0 y el 10% del número de pedidos) y sustituirlos por otros pedidos aleatorios de la lista. De esta manera estaríamos alterando un total de entre 0 y el 20% del número

de pedidos. Imaginemos que tenemos diez productos, si aplicamos una mutación de 1 pedido en un individuo, en la Figura 3.10 podemos ver el resultado. Los pedidos 3 y 9 son intercambiados de orden en la lista, lo que introduce diversidad a la población.

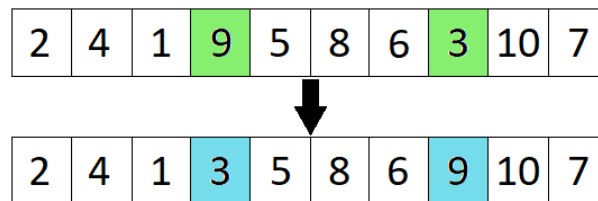


Figura 3.10: Primer operador de mutación

El otro operador de mutación consiste en elegir al azar dos pedidos de la lista e invertir la sublista comprendida entre ellos dos. Si nos imaginamos el mismo individuo que en el caso anterior, el resultado sería distinto, lo podemos ver en la Figura 3.11.

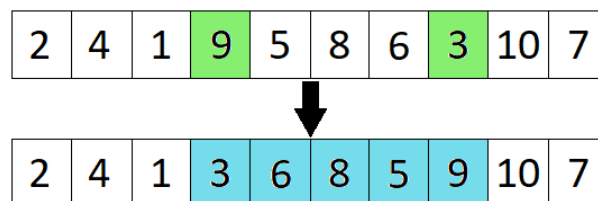


Figura 3.11: Segundo operador de mutación

Así como tendremos que decidir qué operador de mutación usamos finalmente tras hacer algunos experimentos, también tendremos que decidir la probabilidad de mutación, es decir, cuantos hijos de la nueva población mutan. Normalmente no es muy alta, entre 0.05 y 0.1, sin embargo, esto es en casos de problemas donde un solo cambio puede suponer una gran diferencia respecto al individuo anterior, por ejemplo, un laberinto. En ese caso, el sustituir una casilla vacía por un muro puede suponer encontrar una ruta mucho más rápida, o mucho más lenta. Sin embargo, en el caso de nuestro segundo operador de mutación, puede que la diferencia de cambiar un pedido por otro sea nula, así que, para compensar y crear mayor diversidad, tal vez tengamos que usar probabilidades de mutación mucho mayores.

Sustitución de la población

Por último, tras el cruce y la mutación de nuestros individuos, es el momento de la selección de la nueva población. Sea P la población actual y P' la población obtenida mediante selección, cruce y mutación a partir de P , existen varios mecanismos de sustitución de la población: Reemplazo (P sustituye por completo a P'), elitismo (la nueva población se forma con el mejor individuo de P y todos los individuos de P' excepto el peor de ellos) y truncamiento (se ordenan todos los individuos de P y P' y se cogen los n mejores).

La técnica de reemplazo puede hacer que perdamos al mejor individuo si ningún hijo de la nueva población supera al mejor de la anterior. Con truncamiento se pierde diversidad, por lo que vamos a implementar el mecanismo elitista. Previamente guardamos al mejor individuo (en la función *fitness* de la iteración anterior, por lo que simplemente guardamos al peor de esta nueva población y los sustituimos.

Mejora Hash

En problemas pequeños puede no importar repetir algunos cálculos, ya que la variación de tiempo entre hacerlo y no hacerlo puede ser de milisegundos o de segundos. Sin embargo, en problemas como éste, es realmente costoso repetir cálculos como el algoritmo de programación múltiples veces. Hemos realizado una función hash que permite comprobar si una ruta ya ha sido realizada. Antes de calcular una ruta, se ordenan en orden ascendente todos sus pedidos. Por ejemplo, si una ruta consiste en los pedidos 13, 7, 1, 95 y 40; tras ordenarlos obtenemos 1, 7, 13, 40 y 95. Una vez ordenados, comprobamos si ese conjunto de pedidos ya ha sido calculado por el algoritmo de programación dinámica. Si no ha sido calculado, lo hacemos y establecemos la solución como el tiempo de esa combinación de pedidos. Al ordenarlos, conseguimos ahorrar muchos más cálculos, ya que existen muchas combinaciones con esos 5 pedidos, y de esta manera, en lugar de calcular todas, solo necesitamos calcular una. Esto es posible ya que la ruta con los pedidos 1, 7, 13, 40 y 95 es la misma que la ruta con los pedidos 40, 13, 7, 95 y 1; y cualquier ruta con otra combinación de esos pedidos.

Tras unas cuantas pruebas, podemos asegurar que hay una mejora inmensa en cuanto a los tiempos de ejecución. En nuestro problema la solución varía en minutos, pero nuestras instancias tienen entre 40 y 100 pedidos. Imaginemos unos grandes almacenes con 5000 pedidos diarios. En ese caso, la mejora podría suponer ahorrar horas y horas de cálculos innecesarios. En el Capítulo 4, realizaremos varios experimentos para comprobar exactamente cuánto tiempo supone esta mejora hash.

3.2.2. Búsqueda Tabu

En esta sección vamos a describir nuestro segundo algoritmo para el *batching* de los pedidos, con el objetivo de comparar los resultados de ambos posteriormente.

En primer lugar, la búsqueda tabú se enmarca dentro de las estrategias de búsqueda que aceptan soluciones que empeoran el valor de la solución actual. La búsqueda tabú no es un algoritmo en sí, sino un conjunto de ideas que agrupadas resuelven problemas de optimización. Se basa en el uso de memoria, que se utiliza para no repetir trayectorias dentro del espacio de búsqueda. Existen dos tipos de memoria: a corto plazo o reciente, y a largo plazo. La memoria reciente se utiliza para hacer intensificación en la búsqueda, es decir, para buscar soluciones cercanas a la solución actual. La memoria a largo plazo se usa para ir a lugares del espacio de búsqueda que no han sido visitados hasta ahora. En la memoria se guardan atributos de las soluciones, ya que guardar las soluciones completas sería un gasto enorme de memoria.

```
def busquedaTabu(self):
    ...
    crear el individuo inicial (nuestra solución, de momento)
    calcular su fitness
    while (no se cumpla la condición de parada):
        crear los vecinos de la solución actual
        calcular el fitness de los vecinos
        establecer como nueva solución el mejor vecino si es elegible
        actualizar la tabla con la estructura tabú
    ...
```

Los vecinos son los individuos más cercanos al individuo actual. Es decir, en un problema donde se busquen permutaciones de número enteros, los vecinos de un individuo serán aquellos que resulten de intercambiar las posiciones de dos elementos del individuo actual.

En nuestro caso, los individuos tendrán exactamente la misma codificación que en el algoritmo genético: una lista de rutas, donde cada ruta la forman dos elementos; la capacidad de la ruta y una lista con los pedidos a reunir en esa ruta. Los distintos elementos que forman el algoritmo son los siguientes:

Las condiciones de parada son similares a las del algoritmo genético: un número fijo de iteraciones, un umbral de iteraciones el cual detendrá el algoritmo si lleva ese número de iteraciones sin mejorar, una combinación de ambas, etc.

Un vecino es elegible si cumple una de las siguientes condiciones:

- Es el mejor vecino y no está prohibido por la estructura tabú.
- Ofrece una solución mejor que todas las encontradas hasta el momento (en este caso se ignora la condición tabú).

Inicialmente la estructura tabú está compuesta por ceros en su totalidad. Cada vez que un vecino es elegido se debe actualizar la información de la estructura tabú. Se

Tabu structure

	2	3	4	5	6	7
1						
2						
3						
4						
5						
6						

Figura 3.12: Estructura tabú inicial

debe establecer un número de iteraciones en las cuales ese vecino no podrá ser escogido de nuevo (por ejemplo, 3), y se debe reducir en uno el tabú del resto de vecinos.

Imaginemos un problema en el que los individuos se representan como listas de enteros. Aunque en nuestro caso los individuos contengan más información, como la capacidad de cada ruta y los pedidos de cada una de ellas, también se puede entender como una simple lista con los pedidos ordenados (antes de ser distribuidos en cada una de las rutas). Supongamos que existen siete elementos para los cuales hay que encontrar una permutación que minimice el *fitness*. Los vecinos se formarían intercambiando la posición de dos de esos siete elementos. Inicialmente, la estructura tabú será la que se ve en la Figura 3.12.

Imaginemos que tras crear el individuo inicial y calcular su *fitness*, calculamos el de todos sus vecinos y comprobamos que la mejor opción es elegir el vecino que intercambia los vecinos 2 y 5. Tras establecer ese vecino como nueva solución y actualizar la información de la tabla, ésta quedaría como indica la Figura 3.13, indicando que durante las 3 próximas iteraciones no se podrá elegir ese vecino aunque sea el que mejor *fitness* ofrece.

Tras esto, supongamos que en la segunda iteración el mejor vecino es el que intercambia los elementos 4 y 7. Establecemos como 3 las iteraciones tabú del vecino (4,7) y restamos uno a todos los demás vecinos mayores que cero. El resultado será el de la Figura 3.14.

En la tercera iteración, el mejor vecino es el que intercambia los elementos 1 y 3.

Tabu structure

	2	3	4	5	6	7
1						
2				3		
3						
4						
5						
6						

Figura 3.13: Estructura tabú primera iteración

Tabu structure

	2	3	4	5	6	7
1						
2				2		
3						
4						3
5						
6						

Figura 3.14: Estructura tabú segunda iteración

Tabu structure

	2	3	4	5	6	7
1		3				
2				1		
3						
4						2
5						
6						

Figura 3.15: Estructura tabú tercera iteración

Tras actualizar la información como hemos indicado, el resultado sería el que se puede ver en la Figura 3.15.

Este proceso continuaría hasta llegar a la condición de parada, tras lo cual nos quedamos con el mejor individuo encontrado hasta el momento como solución del algoritmo. Algunos problemas se resuelven en dos fases distintas, una que usa la memoria a corto plazo, y otra que usa la memoria a largo plazo. La segunda fase consistiría en, tras finalizar la primera, ejecutar de nuevo el algoritmo penalizando a los vecinos que hayan sido seleccionados más veces. Esto ayuda a salir del espacio de búsqueda local y encontrar otras soluciones en vecinos más lejanos. La manera en la que se realiza este paso es con una tabla de información, similar a la que indica las ejecuciones en las que algunos vecinos no pueden seleccionarse; pero en este caso, para cada vecino se establece como valor el número de veces que se ha seleccionado. En esta fase, al calcular el *fitness* de cada vecino, cada uno de ellos se penaliza más o menos según si se ha explorado más o menos veces, lo cual ofrece mucha diversidad al algoritmo.

Implementación

En nuestra implementación establecemos la capacidad de los *pickers* a la que indica la configuración del almacén o al doble de esta, igual que en el algoritmo genético (explicaremos esto con más detalle en la sección de experimentos). Establecemos un número de vecinos, por ejemplo, el número de pedidos de la instancia. Es decir, si la instancia tiene 40 pedidos, el algoritmo de búsqueda tabú creará 40 vecinos en cada iteración; si tiene 100, serán 100 vecinos. Esto es necesario en problemas grandes como éste, debido a los tiempos de ejecución. En un problema pequeño se pueden calcular

todos los vecinos del individuo actual en un tiempo aceptable, pero en nuestro caso es inviable. Si tenemos 40 pedidos, habrá unos 800 vecinos por iteración, lo cual será lento, pero se podría llegar a considerar. Sin embargo, imaginemos un almacén real donde en un día haya 1000 pedidos. Eso supondría calcular la función *fitness* (el algoritmo de programación dinámica) unas 500.000 veces por iteración. Como no es factible, establecemos el número de vecinos a un valor más aceptable, lo igualamos al número de pedidos de la instancia, para que sea más o menos complejo según el tamaño de la instancia.

Nuestro individuo inicial se crea aleatoriamente, al igual que hacíamos al crear la población inicial en el algoritmo genético. Creamos una lista aleatoria con tantos elementos como pedidos tengamos, y los repartimos en las distintas rutas teniendo en cuenta la capacidad de cada *picker* para no superarla en ninguna ruta.

Nuestros vecinos se calculan de manera distinta al ejemplo que hemos explicado. Si tan sólo intercambiamos dos pedidos de posición, nuestro individuo muchas veces no varía nada, porque al hacer el reparto de las rutas (calculando las capacidades de cada pedido en las rutas) el resultado queda idéntica. Por tanto, usamos el mismo método que explicamos en el algoritmo genético como una posible mutación. Escogemos dos elementos aleatorios de la lista de pedidos e invertimos toda la sublista comprendida entre esos dos pedidos. De esta manera tenemos mucha diversidad entre todos los vecinos, pudiendo alcanzar distintas soluciones en cada iteración.

Nuestra función de *fitness* es, de nuevo, el algoritmo de rutas de programación dinámica. En cada iteración se calcula el tiempo total de cada vecino para elegir al mejor de todos (de los elegibles, como hemos explicado anteriormente). Tras establecer nuestra nueva solución, actualizamos la información tabú como hemos explicado anteriormente (también 3 iteraciones tabú, como en el ejemplo).

Tras repetir este bucle hasta cumplir la condición de parada, se escoge la mejor solución obtenida durante toda la ejecución (no necesariamente la última), así como el individuo que la representa, para poder identificar las distintas rutas que deben seguir los *pickers* con cada uno de los pedidos.

Como hemos explicado antes, estos algoritmos muchas veces están compuestos de dos fases. La segunda fase, de memoria a largo plazo, es útil en problemas como el del ejemplo que habíamos explicado, donde los vecinos se calculan intercambiando la posición de dos elementos en una permutación. Como es un cambio pequeño, puede que a lo largo de muchas ejecuciones se repitan los mismos cambios, bloqueando el algoritmo en un espacio local. Esa segunda fase consiste en generar diversidad penalizando a los vecinos más escogidos. Sin embargo, como nuestros vecinos se calculan invirtiendo toda una sublista, dentro de la permutación (para permitir más variabilidad), esta segunda fase no tiene sentido ya que existe suficiente diversidad con simplemente nuestra primera fase, por lo que no implementamos esa segunda fase.

Por último, hemos implementado una función hash, al igual que en el algoritmo genético, que evita repetir el cálculo de las rutas que ya hayan sido resueltas. Los tiempos de ejecución disminuyen considerablemente, como veremos en los experimentos del Capítulo 4.

3.3. Optimización del número de cajas usadas para los pedidos

En cuanto al problema de la optimización de las cajas, como ya hemos explicado en el desarrollo de los algoritmos de *batching*, se realizará teniendo en cuenta simplemente la capacidad de cada *picker* y la capacidad que conlleva cada uno de los pedidos. En un entorno real se debería desarrollar un algoritmo de optimización como los que hemos estudiado en el Capítulo 2, o utilizar alguno de los existentes en portales como GitHub, de libre uso.

3.4. Cálculo de rutas

Tras el estudio del estado del arte que hemos realizado, podemos asegurar que no es necesario desarrollar distintos algoritmos de rutas, como el *Largest Gap*, el *S-Shape* o el *Aisle-by-Aisle* para calcular su productividad, pues está más que demostrado que la programación dinámica los supera a todos ampliamente. Así pues, hemos desarrollado un algoritmo de este tipo, con ayuda del artículo que ya hemos comentado en el Capítulo 2, de Kees Jan Roodbergen y René De Koster, [30].

3.4.1. Programación dinámica

En esta sección describimos cómo trazar la ruta de un *picker* dado un almacén y los productos que debe recoger, a través de un bloque de estantes i . La ruta empieza en el pasillo de más a la izquierda que contiene algún producto a recoger (*izq*), y acaba en el pasillo de más a la derecha que contiene algún producto (*der*).

Definimos L_j como una ruta parcial que visita todos los productos a recoger en el pasillo vertical j , y definimos dos clases de rutas parciales:

- L_j^a : Ruta parcial que termina al fondo (arriba) del pasillo vertical j .
- L_j^b : Ruta parcial que termina al principio (abajo) del pasillo vertical j .

Existen dos maneras de cambiar de un pasillo vertical j al pasillo vertical $j + 1$:

- t_a : Desde el fondo del pasillo vertical j al fondo del pasillo vertical $j + 1$.
- t_b : Desde el inicio del pasillo vertical j al inicio del pasillo vertical $j + 1$.

Las 4 maneras posibles que distinguimos para recoger todos los productos del pasillo j son:

- t_1 : Atravesar el pasillo j completamente (en cualquiera de las dos direcciones).
- t_2 : No entrar en el pasillo j (solo posible si no hay productos a recoger en el pasillo j).
- t_3 : Entrar al pasillo j desde el fondo y salir por el mismo sitio.
- t_4 : Entrar al pasillo j desde la parte delantera y salir por el mismo sitio.

Con $L_j + t_w$ indicamos que añadimos a la ruta parcial L_j la transición t_w , tratándose tanto de las transiciones horizontales (t_a y t_b), como de las verticales (t_1 , t_2 , t_3 y t_4). La función $c(x)$ nos permite calcular el tiempo que conlleva la ruta indicada como argumento x . Por ejemplo, $c(L_j + t_a + t_b)$ nos proporciona el tiempo necesario para completar la ruta parcial L_j y las transiciones t_a y t_b . Las rutas deben comenzar y finalizar en el *depot*, que en nuestro caso se encuentra en la extremo izquierdo del almacén, entre los dos bloques. Es decir, debemos recorrer uno de los bloques de izquierda a derecha, y el otro bloque de derecha a izquierda, de esta manera obtendremos los mejores resultados posibles. Hemos decidido recorrer primero el bloque superior y posteriormente el bloque inferior. Para recorrer el bloque superior debemos seguir los siguientes pasos:

- 1. Comenzamos en el primer pasillo con algún artículo a recoger (*izq*) con las dos siguientes rutas parciales:
 - L_{izq}^a : Empieza en el nodo b_{izq} (*depot*) y acaba en el nodo a_{izq} mediante la transición t_1 .
 - L_{izq}^b : Empieza en el nodo b_{izq} (*depot*) y acaba en el nodo b_{izq} mediante la transición t_3 .
- 2. Por cada pasillo vertical j entre el primero (*izq*) y el último (*der*) establecemos L_j^a y L_j^b de la siguiente manera.

Si el pasillo j tiene algún producto a recoger:

$$L_j^a = \begin{cases} L_{j-1}^a + t_a + t_4 & \text{si } c(L_{j-1}^a + t_a + t_4) < c(L_{j-1}^b + t_b + t_1) \\ L_{j-1}^b + t_b + t_1 & \text{en otro caso} \end{cases}$$

$$L_j^b = \begin{cases} L_{j-1}^b + t_b + t_3 & \text{si } c(L_{j-1}^b + t_b + t_3) < c(L_{j-1}^a + t_a + t_1) \\ L_{j-1}^a + t_a + t_1 & \text{en otro caso} \end{cases}$$

Si el pasillo j no tiene ningún producto a recoger:

- $L_j^a = L_{j-1}^a + t_a$
- $L_j^b = L_{j-1}^b + t_b$

- 3. Para el último pasillo del bloque superior, no necesitamos una ruta L_r^a , puesto que a continuación vamos a recorrer el bloque inferior del almacén, y queremos estar lo más cerca posible a éste bloque. Por tanto:

$$L_r^b = \begin{cases} L_{r-1}^b + t_b + t_3 & \text{si } c(L_{r-1}^b + t_b + t_3) < c(L_{r-1}^a + t_a + t_1) \\ L_{r-1}^a + t_a + t_1 & \text{en otro caso} \end{cases}$$

Para el caso del bloque inferior tenemos que realizar los pasos explicados en orden contrario, al igual que las transiciones t_a y t_b , que en este caso avanzarán de derecha a izquierda. Necesitamos hacer estas consideraciones, porque como ya hemos dicho, debemos empezar y terminar nuestro recorrido en el *depot*, para lo cual completamos primero el bloque superior (de izquierda a derecha) y posteriormente el bloque inferior (de derecha a izquierda). La ruta finaliza cuando regresemos al *depot* habiendo recogido todos los productos necesarios.

4. Experimentos

En este capítulo vamos a comprobar la efectividad de los algoritmos desarrollados, así como la mejora de las funciones hash que hemos implementado para los algoritmos de *batching*. Una vez hechas algunas pruebas con los algoritmos de *batching*, vamos a realizar una serie de experimentos para encontrar una configuración de parámetros que nos permita obtener los mejores resultados. Como ya dijimos anteriormente, las instancias de la distribución ABC tenían algunos problemas, por lo que usaremos las instancias de distribución aleatoria. Se trata de 16 instancias con sus 16 configuraciones correspondientes. Existen 4 cantidades de pedidos (20, 40, 60 y 80), 4 capacidades para los *pickers* (30, 45, 60 y 75) con lo que tendríamos 16 instancias con todas sus combinaciones posibles.

Cabe destacar que todos los resultados de los experimentos son en segundos, ya sea las soluciones (el *fitness*), que son los segundos que tardarían los *pickers* en recorrer todas las rutas necesarias para completar el *batching* que hemos encontrado; como el tiempo de ejecución, también en segundos.

Para los experimentos se ha utilizado un ordenador con las siguientes características:

- Procesador AMD Ryzen 5 3600X 6-Core 3.80 GHz.
- Memoria RAM de 32 GB.
- Sistema operativo Windows de 64 bits.

4.1. Algoritmo genético

Lo primero que vamos a hacer es establecer los parámetros a usar en el algoritmo genético. Vamos a usar para este experimento tan sólo 4 instancias, una con cada número de pedidos, ya que con esta muestra obtendremos una solución fiable para el conjunto global de instancias. Vamos a probar cada una de esas instancias con todas las combinaciones de los siguientes parámetros:

- Capacidad del *picker*: 1 (capacidad original) o 2 (el doble de la capacidad que indica la instancia, simplemente para observar si existe mucha diferencia entre unos valores y otros).
- Tamaño de la población: 30, 60 y 90 individuos. Tras varias pruebas, hemos detectado que con menos de 20 individuos obteníamos malos resultados, mientras que con poblaciones de más de 80 individuos los resultados no variaban prácticamente, por lo que las soluciones se encuentran entre esos valores.
- Iteraciones: 50 y 100. Además, habrá un umbral fijo que hará que cualquier ejecución del algoritmo genético que lleve 20 iteraciones sin superar al mejor individuo, finalizará esa ejecución, aunque no haya llegado al total de 50 o 100 iteraciones.

Ejecutamos cada una de esas configuraciones con cinco semillas distintas (1, 2, 3, 4 y 5) para que la solución sea más fiable. Los resultados importantes a observar son tanto la propia solución (el mejor individuo en cada caso) como el tiempo de ejecución. Este valor es importante ya que si, a costa de invertir el doble de tiempo en la ejecución del algoritmo, se aumenta en menos de un 1% el *fitness*, tal vez no merezca la pena. El resultado de esas configuraciones se puede observar en la Figura 4.1. Cada configuración muestra la media del *fitness* de las 16 instancias.



Figura 4.1: Fitness según distintos parámetros algoritmo genético

Como se puede observar, no se distingue claramente la diferencia entre los tamaños de población grandes y pequeños ni el número de iteraciones como 50 o 100. Para poder ver mejor como varían los resultados, vamos a acercar las gráficas separándolas en capacidad normal y capacidad doble. Podemos verlo en la Figuras 4.2a y 4.2b.

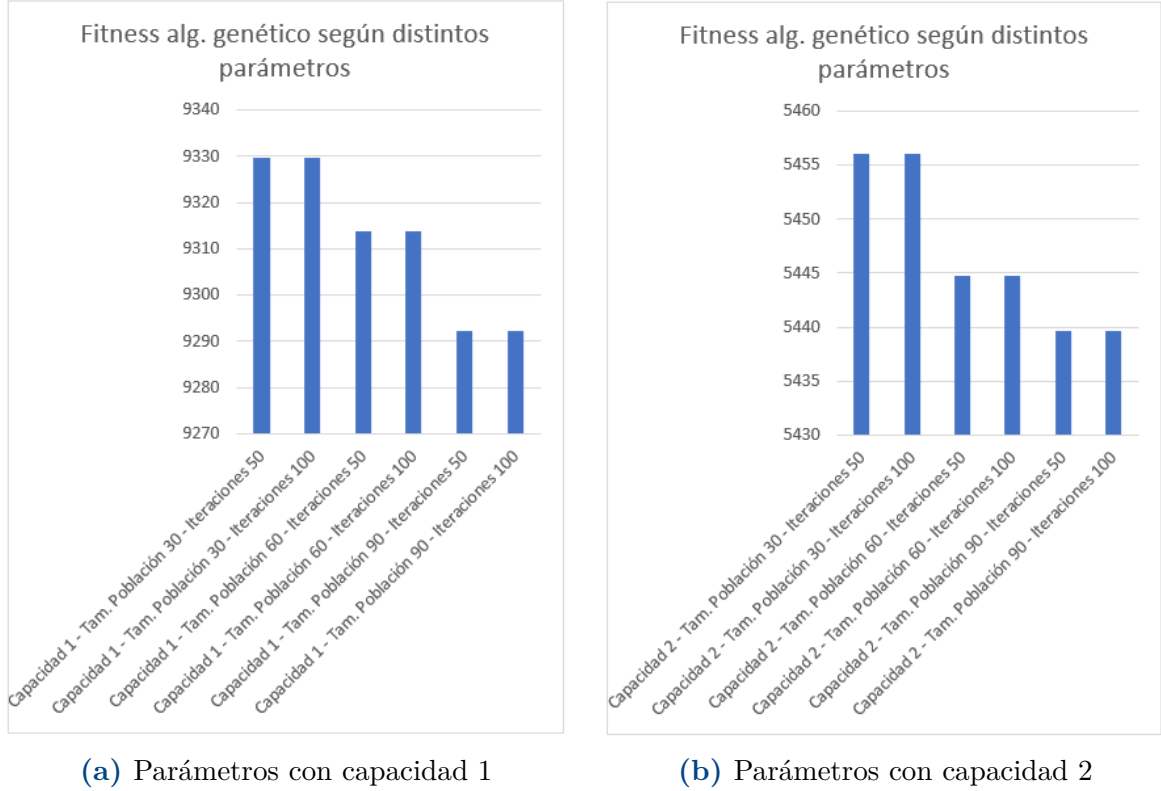


Figura 4.2: Fitness según distintos parámetros algoritmo genético

En cuanto al tamaño de la población, se ve claramente como en ambas gráficas (capacidad 1 y capacidad 2) se mejora el *fitness* al utilizar 90 individuos por población, en lugar de 30 o 60. Por ello, y tras comprobar que a partir de los 70 u 80 individuos las soluciones no mejoran, fijaremos 90 como tamaño de la población.

Respecto al número de iteraciones, se ve claramente como en ningún caso el *fitness* mejora por utilizar 100 iteraciones en lugar de 50. Esto significa que antes de llegar a las 50 iteraciones la solución ya ha sido encontrada prácticamente siempre. Por ese motivo, y para ahorrar tiempo de ejecución (la mitad de tiempo en cada iteración), fijamos como 50 el número de iteraciones para los demás experimentos. Seguiremos usando ese umbral de 20 iteraciones con el cual el algoritmo finalizará cualquier iteración que no mejore la solución durante 20 iteraciones.

En cuanto a las capacidades (la que indica cada instancia o el doble de ésta), podemos decir que la primera de ellas nos va a dar más margen de beneficio. Mientras que con la primera capacidad los resultados varían en casi 40 segundos (9329'52 - 9292'15), la capacidad doble tan solo varía unos 16 segundos (5456'08 - 5439'62). Por tanto, como la capacidad estándar nos va a dar más juego, la fijaremos como parámetro para el resto de experimentos.

Vamos ahora a establecer la probabilidad de mutación. Para los primeros experimentos hemos usado 0.5 como probabilidad de mutación, porque anteriormente habíamos realizado unas pruebas iniciales y ese valor daba buenos resultados. Vamos a realizar una prueba con varias probabilidades de mutación con todas las instancias, para comprobar que esa probabilidad de mutación nos da los mejores resultados. Vamos a resolver las 16 instancias de nuevo, con los parámetros que acabamos de fijar (60 como tamaño de población, 50 iteraciones y la capacidad de los *pickers* que indique cada instancia. Utilizamos de nuevo 5 semillas para obtener resultados fiables, y en este caso vamos a resolver las instancias con tres probabilidades de mutación distintas: 0'05, 0'2 y 0'5. Podemos observar el resultado en la Figura 4.3.

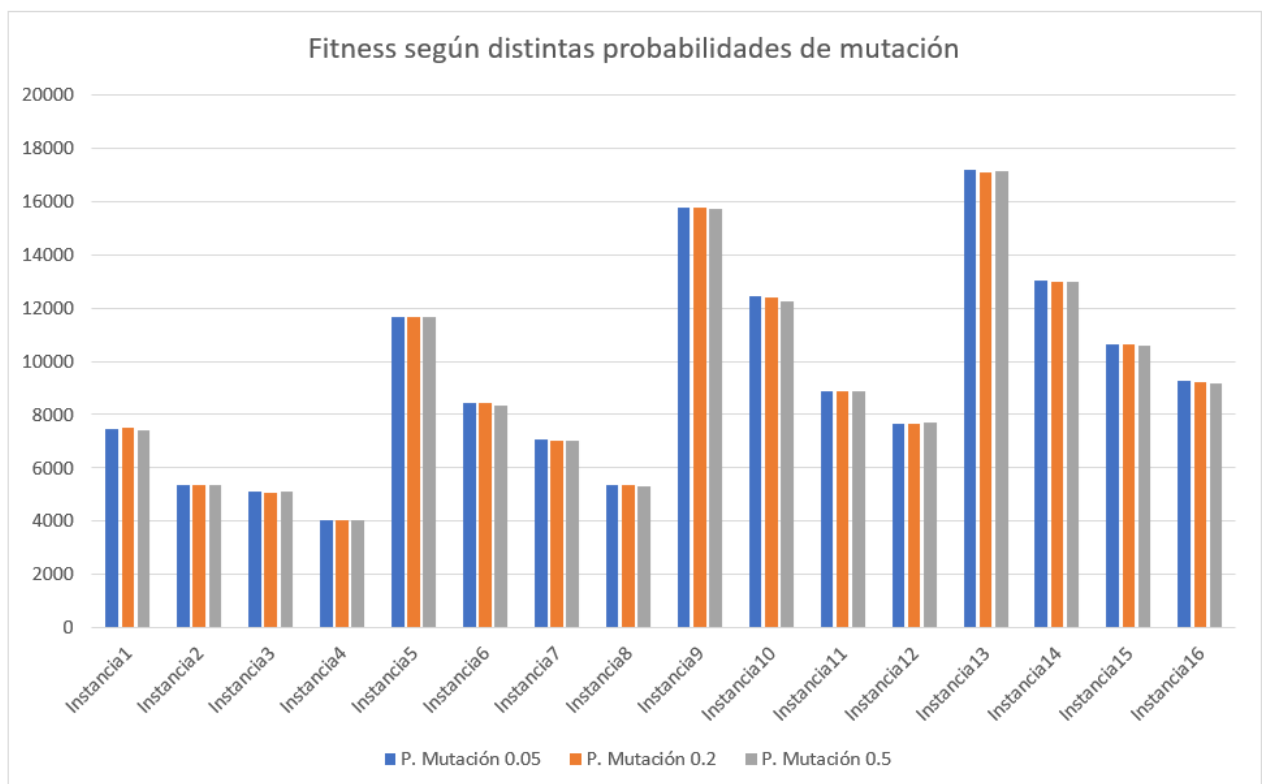


Figura 4.3: Fitness según distintas probabilidades de mutación al algoritmo genético

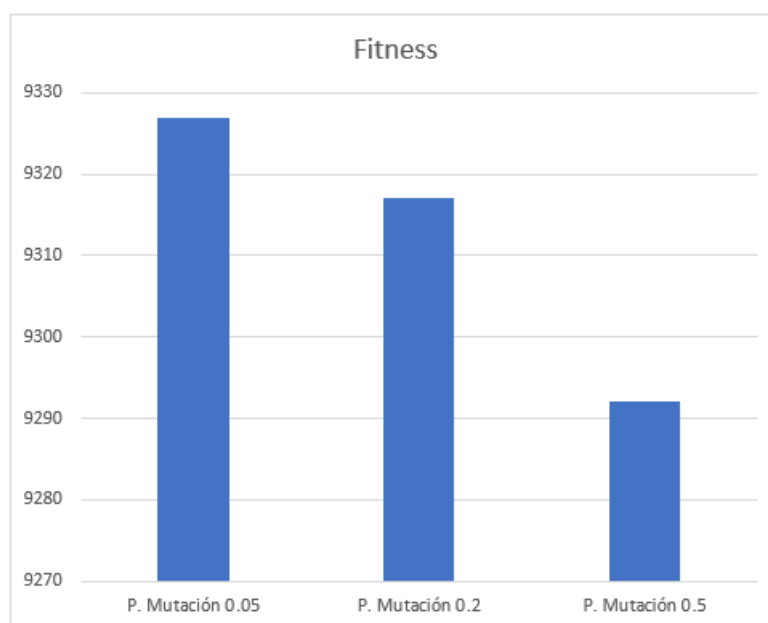


Figura 4.4: Fitness medio según distintas probabilidades de mutación algoritmo genético

Como prácticamente no se aprecian las diferencias entre unos valores y otros, vamos a crear una gráfica con simplemente las medias de *fitness* de las 16 instancias para cada valor de la probabilidad de mutación. En la Figura 4.4 se puede observar como la probabilidad de mutación 0.5 es la mejor de todas las propuestas.

Como se puede observar, en prácticamente todas las instancias las probabilidades bajas de mutaciones dan mejores resultados. Por ese motivo, vamos a utilizar como probabilidad de mutación 0.5 para el resto de experimentos.

A continuación, vamos a comparar la versión del algoritmo genético que no usa la función hash con la que sí lo usa. Para ello ejecutamos los algoritmos con cada una de las 16 instancias con 10 semillas distintas (del 1 al 10), para obtener soluciones fiables. Los resultados se pueden observar en la Figura 4.5.

A simple vista se puede apreciar como la función hash mejora indudablemente los tiempos de ejecución de la versión sin hash. Si nos fijamos en la Figura 4.6 se aprecia mejor la diferencia. Mientras que sin hash el tiempo de ejecución medio de cada iteración son 72'93 segundos, utilizando la función hash se reduce a 30 segundos. Esto supone una mejora del 143'1%, por lo que usaremos esta función hash para el experimento final.

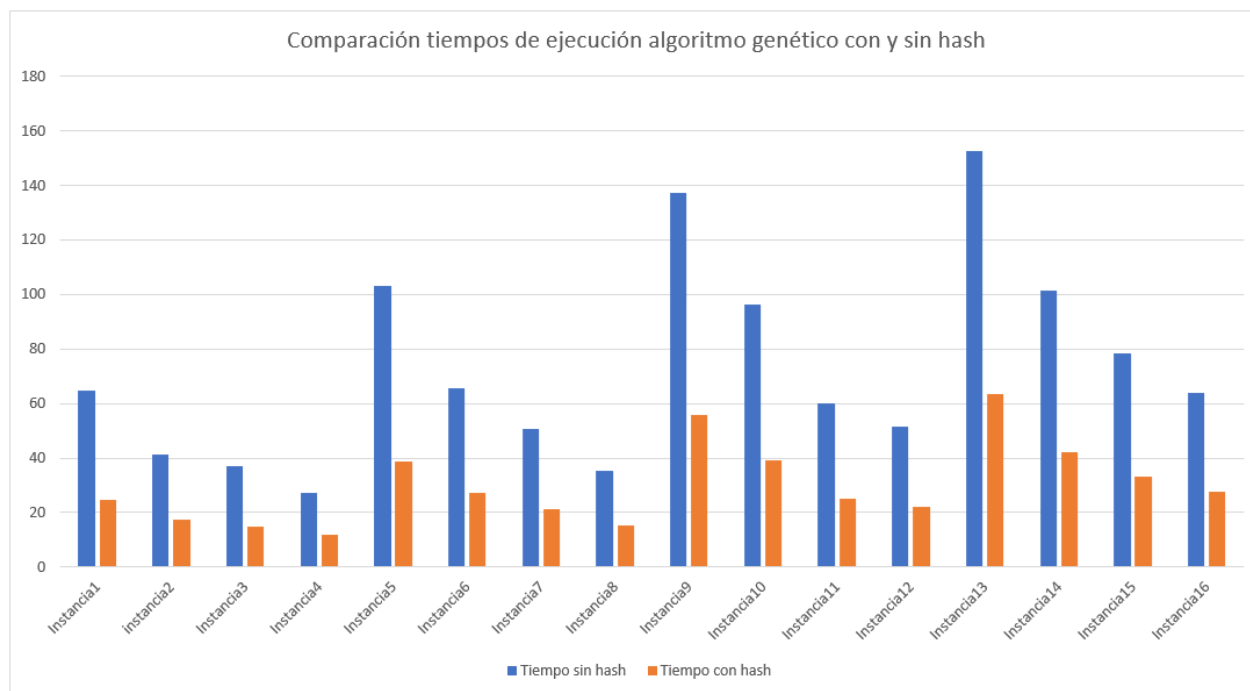


Figura 4.5: Comparación algoritmo genético con y sin hash

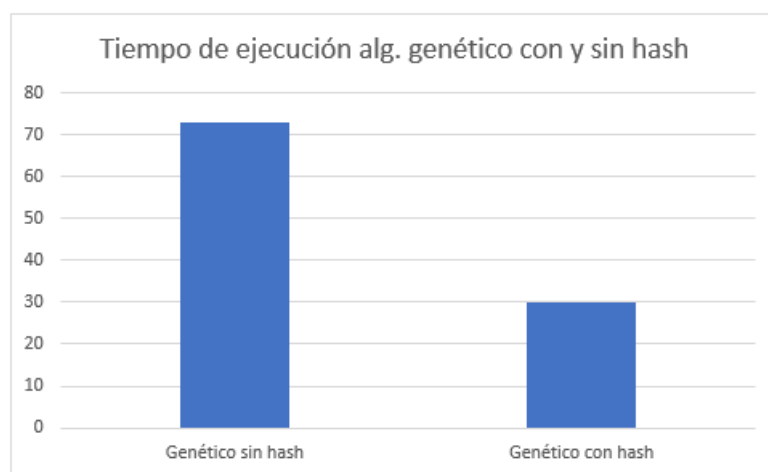


Figura 4.6: Comparación tiempos de ejecución medios algoritmo genético con y sin hash

4.2. Búsqueda tabú

Respecto a la búsqueda tabú, existen tres parámetros que debemos configurar:

- El valor tabú que indica cuántas iteraciones se debe bloquear un vecino en el caso de ser elegido lo fijamos a 3, ya que es un valor estándar usado en este tipo de algoritmos.
- En cuanto al número de iteraciones, inicialmente habíamos hecho pruebas con entorno a 30 iteraciones. Los resultados no mejoran prácticamente a partir de ahí, pero como estamos tratando de comparar dos algoritmos muy distintos (algoritmo genético y búsqueda tabú), hemos decidido fijar un número de iteraciones que provoque que los tiempos de ejecución sean similares entre ambos algoritmos de *batching*. Tras varias pruebas, fijamos el número de iteraciones a 50, ya que de esta manera (y con el número de vecinos que a continuación indicaremos) se consiguen tiempos de ejecución parecidos a los del algoritmo genético, lo que los hace estar en una mayor igualdad de condiciones.
- Por último, el número de vecinos. Como ya explicamos en el Capítulo 3, en problemas grandes no es factible calcular todos los vecinos del individuo actual en cada una de las iteraciones. Para alcanzar tiempos de ejecución similares al algoritmo genético, hemos duplicado el número de vecinos que establecimos inicialmente. Habíamos indicado que el número de vecinos sería igual al número de pedidos de la instancia, sin embargo, podemos fijarlo al doble de ese valor, para conseguir resultados algo mejores y observar si los tiempos de ejecución son similares a los del algoritmo genético, buscando esa igualdad de condiciones que comentábamos.

Vamos a comparar el uso de la función hash al igual que hemos hecho con el algoritmo genético. Utilizando de nuevo 10 semillas para resolver las 16 instancias, obtenemos los tiempos de ejecución que indica la Figura 4.7.

De nuevo, se puede apreciar como la función hash reduce considerablemente los tiempos de ejecución de la versión sin hash. Si nos fijamos en la Figura 4.8 se aprecia aún más la diferencia. Mientras que sin hash el tiempo de ejecución medio de cada iteración son 149,1 segundos, utilizando la función hash se reduce a 57,2 segundos. Esto supone una mejora del 160%, todavía mayor que en el caso del algoritmo genético. Utilizaremos esta versión para la comparativa final.

Sin embargo, nos hemos dado cuenta de que no hemos logrado esa igualdad en los tiempos de ejecución con ambos algoritmos de *batching*. Mientras que el algoritmo genético tardaba unos 70 segundos por iteración en su versión normal, y 30 en la versión con hash; la búsqueda tabú disponía del doble de tiempo para encontrar sus soluciones, 149 segundos en la versión sin hash y 57 en la versión con hash. Por ese motivo, vamos a realizar de nuevo las pruebas de la búsqueda tabú utilizando como número de vecinos el número de pedidos, en lugar del doble de esa cantidad, lo que reducirá el tiempo de

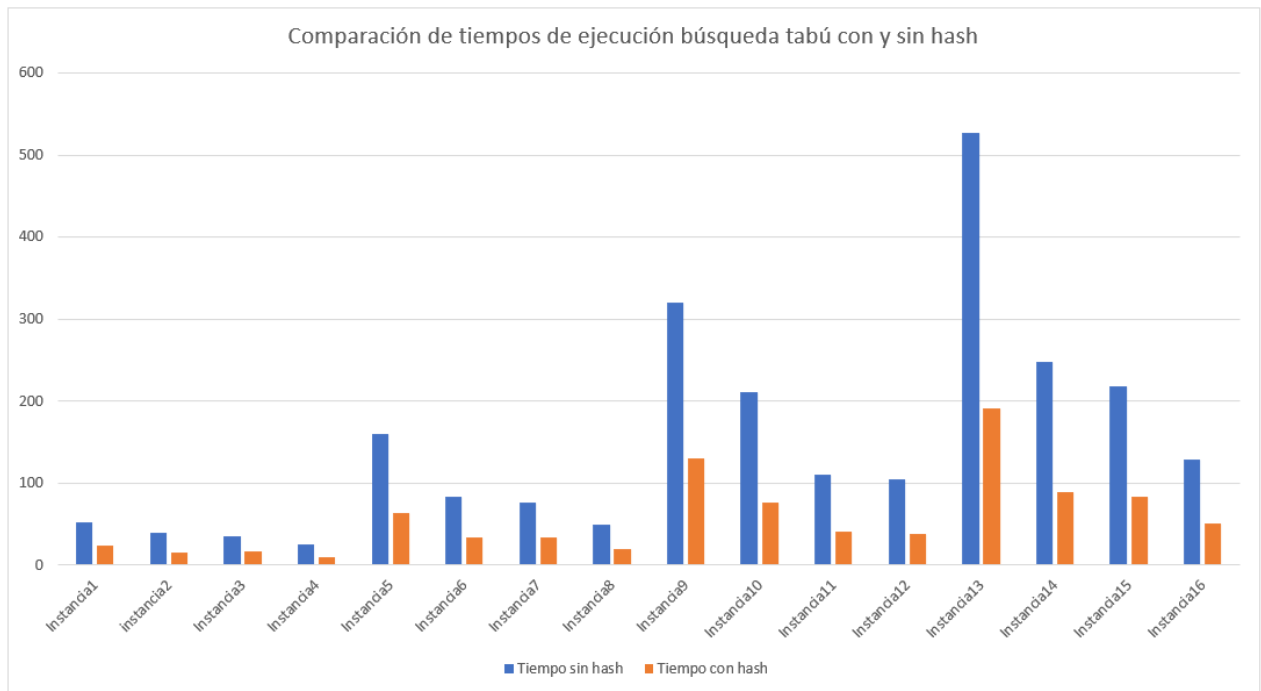


Figura 4.7: Comparación búsqueda tabú con y sin hash

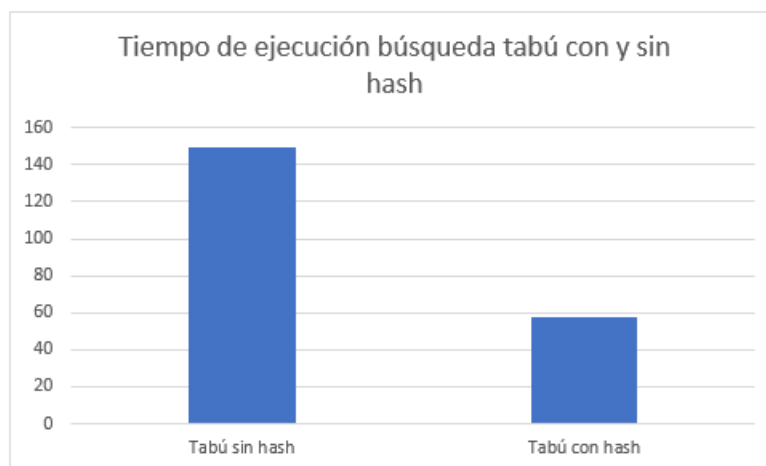


Figura 4.8: Comparación búsqueda tabú con y sin hash

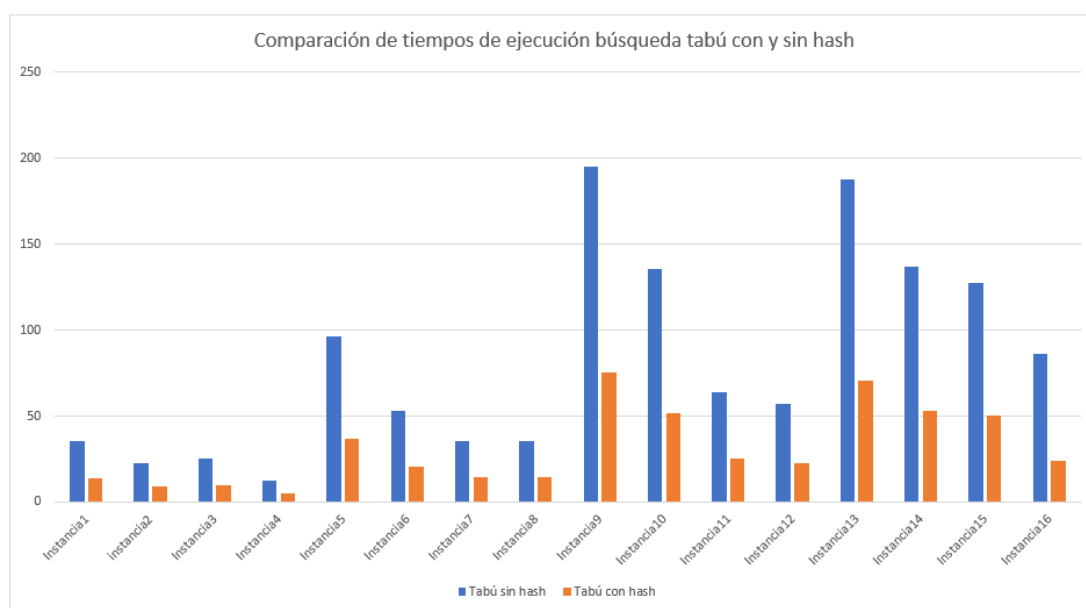


Figura 4.9: Comparación búsqueda tabú con y sin hash mitad de vecinos

ejecución de cada iteración. En la Figura 4.9 se pueden observar los nuevos tiempos de ejecución.

De nuevo, se ve claramente la eficacia de la función hash. Vamos a comparar la media total de los tiempos de ejecución, se puede ver en la Figura 4.10. En este caso, los tiempos son 81'63 y 31'13, lo cual supone una mejora del 162'2%, lo cual es muy similar a la que habíamos obtenido anteriormente. En este caso los tiempos de ejecución se parecen más a los del algoritmo genético, por lo que tenemos unas condiciones más igualadas, y podemos realizar el experimento final.

4.3. Experimento final

Como experimento final, queremos comparar las soluciones obtenidas por ambos algoritmos de *batching*. No sólo eso, también añadiremos un tercer participante al experimento: los almacenes tradicionales. Como ya comentamos en el Capítulo 1, la inmensa mayoría de grandes almacenes no optimizan el proceso de *batching* de los pedidos. Es decir, los asignan por orden de llegada, por ID, manualmente por algún trabajador, o de alguna otra manera que no optimice la productividad. Por ese motivo, vamos a comparar junto a los dos algoritmos de *batching*, la opción de usar ningún algoritmo de este tipo. Es decir, creamos un individuo aleatorio, asignamos las rutas en el orden que corresponda, y las calcularemos mediante el algoritmo de programación dinámica, al igual que hacen los algoritmos de *batching*. Aunque en la mayoría de almacenes ni

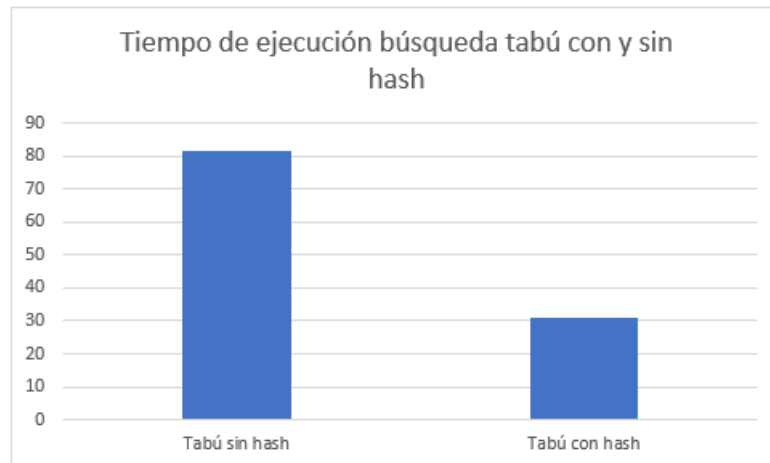


Figura 4.10: Comparación tiempos de ejecución medios búsqueda tabú con y sin hash mitad de vecinos

quiera usen un algoritmo de rutas para optimizar ese proceso, nosotros vamos a hacerlo para igualar las condiciones con los dos algoritmos de *batching*.

Tras crear esta clase sin *batching* y hacer alguna prueba, podemos comparar los tiempos de ejecución respecto a las clases que usan algoritmos de *batching*. En la Figura 4.11 se puede ver como evidentemente esta última versión es cientos de veces más rápida que las versiones que utilizan algoritmos de *batching*, aunque evidentemente obtiene peores soluciones.

Por último, vamos a comparar las soluciones de cada una de las tres versiones. Hemos calculado el tiempo total en recoger todos los pedidos de las 16 instancias tras resolverlos con 10 semillas distintas. Para el algoritmo genético y la búsqueda tabú tan solo hemos usado las versiones con hash, ya que los resultados son los mismos, mientras que nos ahorramos mucho tiempo de ejecución. El resultado se puede observar en la Figura 4.12.

Se ve claramente que el algoritmo genético es la mejor opción en este caso, seguido por la búsqueda tabú, y por último, la versión sin *batching*. Vamos a calcular las medias para apreciar mejor la diferencia entre las tres versiones. En la Figura 4.13 se puede observar el resultado.

El algoritmo genético obtiene una solución media de 9304'75 segundos, la búsqueda tabú 9529'64 segundos y la versión sin *batching* 9747 segundos. Es decir, el algoritmo genético supone una mejora del 2'42% respecto a la búsqueda tabú, y de un 4'75% respecto a la versión de los almacenes tradicional, sin ningún proceso de *batching*.

Imaginemos un almacén tradicional donde trabajan 10 *pickers* a la semana, 35 hora-

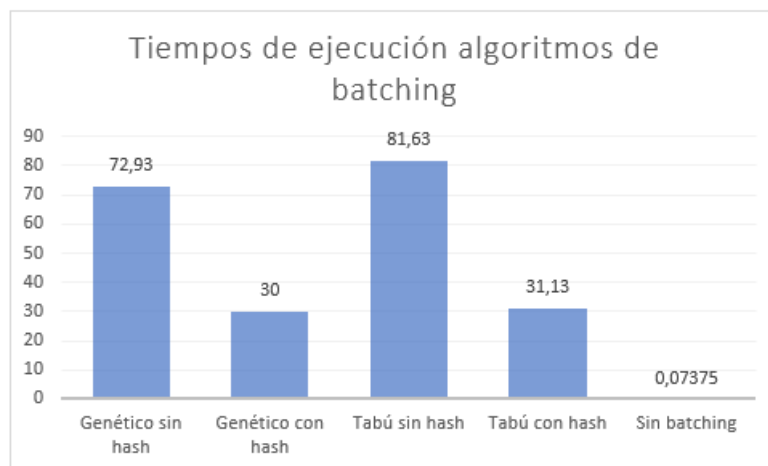


Figura 4.11: Comparación tiempos de ejecución entre las tres versiones

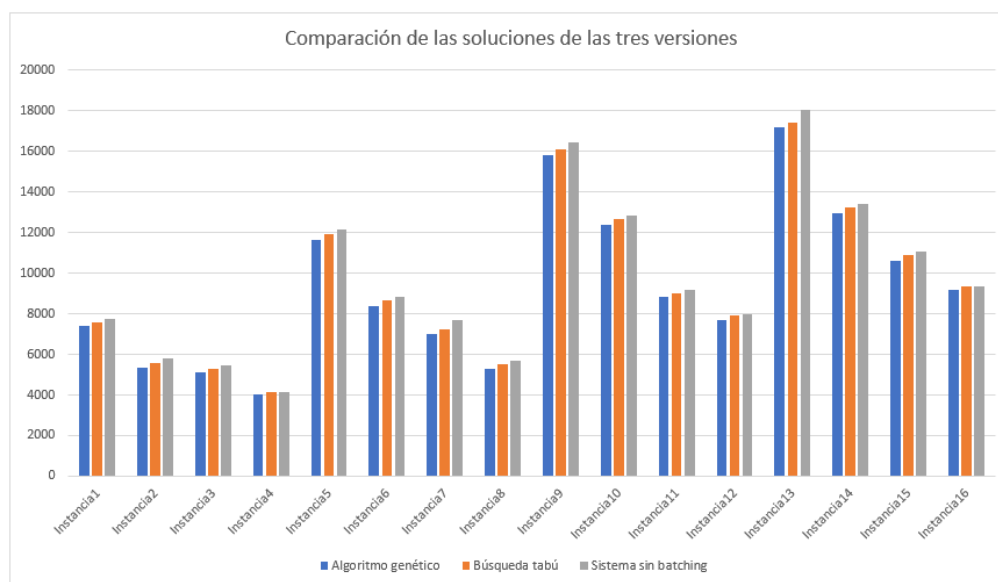


Figura 4.12: Comparación tiempos de las soluciones entre las tres versiones

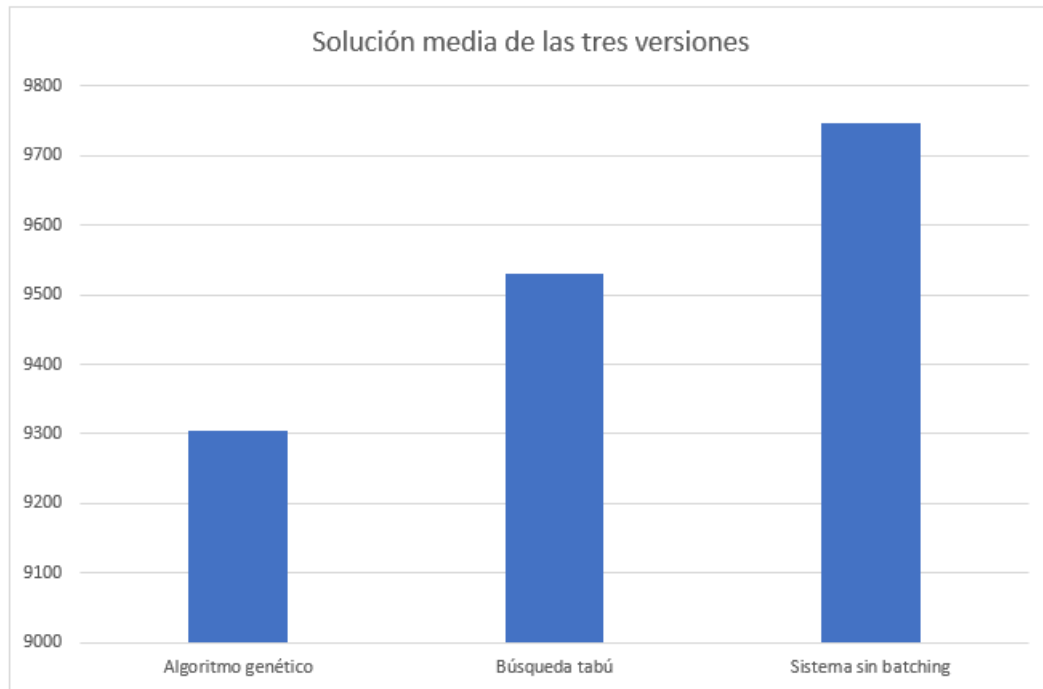


Figura 4.13: Comparación media tiempos de las soluciones entre las tres versiones

s/semana cada uno. A lo largo de un mes eso hacen un total de 10500 horas de trabajo de recogida. La mejora del 4'75% reduciría esas horas a 10001 horas, es decir, 500 horas menos de trabajo. Esto sería en el caso de utilizar esta versión del algoritmo genético. Añadiendo nuevas mejoras al algoritmo, y disponiendo de tiempos de ejecución mucho mayores, los resultados que se pueden esperar son mucho mejores a los logrados con esta versión.

En cuanto a la búsqueda tabú, podemos afirmar que proporciona resultados mejores a la versión tradicional, concretamente un 2'28% mejores. Sin embargo, no ofrece resultados tan buenos como el algoritmo genético porque la búsqueda tabú no es tan eficiente en problemas grandes como éste.

Además estos experimentos se han llevado a cabo teniendo en cuenta que los almacenes tradicionales usen el algoritmo de programación dinámica para calcular las rutas de sus *pickers*, cuando no es el caso. Es decir, si comparásemos nuestras soluciones con los datos reales de un almacén, donde las rutas se realizan por orden de ID de los productos o arbitrariamente por parte de los *pickers*, la mejora que observaríamos sería mucho mayor.

5. Conclusiones

En este último capítulo vamos a hablar sobre algunas conclusiones acerca de los algoritmos desarrollados, propuestas para trabajos futuros y las competencias de la intensificación de computación que hemos trabajado durante la realización de este proyecto.

5.1. Conclusiones

En este proyecto hemos abordado un problema del mundo real, la optimización de pedidos en un almacén. Este proceso lo componen el *batching* de los pedidos, la optimización de los pedidos en cajas y la optimización de las rutas que los trabajadores deben seguir.

Este software es un comienzo de lo que podría ser una herramienta muy útil para cualquier empresa que trabaje con grandes almacenes. Aunque queden temas por tratar, mejoras posibles para los algoritmos, mejor definición de las instancias y los almacenes, etc. se trata de un acercamiento a un posible software que cualquier empresa podría utilizar para conseguir grandes beneficios económicos.

Los dos procesos más importantes que hemos resuelto que ocurren durante la recogida de los productos son:

- *Batching* de los pedidos: Repartimos los pedidos en las distintas rutas que los *pickers* deben seguir, tanto con un algoritmo genético como con una búsqueda tabú.
- Las rutas seguidas por los *pickers*. Utilizando el algoritmo de programación dinámica, se asignan las rutas con los pedidos a los trabajadores, que las siguen de manera óptima ahorrando así mucho tiempo.

Tras desarrollar cada algoritmo y comparar sus rendimientos, podemos afirmar que el algoritmo genético obtiene mejores resultados que el sistema sin *batching* de la mayoría de almacenes tradicionales, y también mejores que la búsqueda tabú que hemos

implementado. Por ese motivo, creemos que sin duda sería interesante que cualquier gran almacén apostara por automatizar el proceso de *batching*, ya que vería incrementada su productividad considerablemente. Por otro lado, también creemos que sería conveniente que automatizaran las rutas que deben seguir los *pickers* por el almacén durante la recogida de los pedidos. Con estas implementaciones al final conseguiríamos reducir el número de horas totales de trabajo y crear por tanto un beneficio económico para la empresa, los objetivos iniciales que nos habíamos propuesto.

5.2. Trabajo futuro

A lo largo de este proyecto han surgido muchas ideas que podrían mejorar la solución del problema, no sólo optimizando aún más los algoritmos, también conceptos más generales sobre los almacenes.

En primer lugar, el aspecto más necesario que no hemos resuelto en el proyecto, debido a complejidad y falta de tiempo, es la optimización de los productos en las cajas. Se trata de un problema muy complejo, y el desarrollo de un buen algoritmo de este tipo podría ser perfectamente un trabajo de fin de grado en sí mismo. Sin embargo, existen muchos algoritmos de libre uso bajo licencia MIT en portales como GitHub. En un desarrollo de software profesional para una empresa que trabaje con grandes almacenes, es necesario tratar este tema ya que se puede ahorrar no solo material (cajas), sino también tiempo ya que utilizar menos contenedores para los pedidos implica directamente realizar menos rutas totales para reunir todos los pedidos en una jornada.

Además de usar las dimensiones de los productos, también sería conveniente utilizar el peso de los mismos, ya que en un entorno real, los productos más pesados deben ir debajo de los más ligeros. Resolver el problema teniendo en cuenta las dimensiones y los pesos puede suponer un gran reto.

Otro tema interesante sería desarrollar otro tipo de algoritmos de *batching*, como por ejemplo un algoritmo de colonia de hormigas, para estudiar más opciones y comparar los resultados.

Aparte de los algoritmos, existen otros temas interesantes que acercarían el problema más hacia un caso real en grandes almacenes, y se alejarían del modo teórico en el que hemos desarrollado el proyecto. Por ejemplo, se podrían obtener los detalles completos y precisos de un almacén real. Esto comprende:

- Distribución del almacén. En lugar de una sola sección, con un pasillo central, la mayoría de almacenes tienen distintos elementos que los hacen más complejos: más pasillos centrales, estantes en los pasillos centrales, un almacén de puerta abierta y otro de puerta cerrada (con distintos productos en algunas ocasiones).

- Distintos tipos de productos. En todos los almacenes existen distintos tipos de productos, ya sea por tamaño, peso, urgencia (en el caso de supermercados), etc. Muchas veces esos productos se organizan en varias secciones, lo que hace que no todos los productos se reúnan indistintamente en una sola ruta sin ninguna condición añadida. Por ejemplo, en un supermercado, se deben recoger primero los productos normales, después los productos frescos y por último los congelados.
- Instancias reales. Teniendo la estructura real de un almacén y la posición y las características de cada producto, sería interesante trabajar con instancias reales, donde aplicar todo lo anteriormente explicado. Esto, además, podría servir para mejorar el almacén en otro aspecto distinto al que trata este proyecto: informatizando estos procesos se podrían identificar relaciones entre productos y así reorganizar los elementos del almacén con el objetivo de ahorrar más tiempo durante la recogida de los productos. No sólo se podría estudiar una reestructuración de los pedidos, sino también del almacén, añadiendo o quitando pasillos centrales, así como modificando el ancho de los pasillos para estudiar si aumenta o disminuye la productividad.

Por último, sería interesante desarrollar una aplicación completamente funcional, no sólo una interfaz como la que hemos creado. Disponiendo de las instancias reales del almacén, se podría desarrollar una base de datos con toda la información de los productos (dimensiones, peso, localización, etc.) y la información de los pedidos. Con esa base de datos, el último paso sería desarrollar una aplicación que los *pickers* usaran para recorrer las distintas rutas, completando así todos los aspectos relativos a nuestro proyecto.

5.3. Competencias adquiridas

Durante el desarrollo de este trabajo se han trabajado varias competencias de la intensificación en la que el proyecto se identifica, la computación.

[CM1] *Capacidad para tener un conocimiento profundo de los principios fundamentales y modelos de la computación y saberlos aplicar para interpretar, seleccionar, valorar, modelar, y crear nuevos conceptos, teorías, usos y desarrollos tecnológicos relacionados con la informática.*

Con este proyecto, hemos mejorado nuestras capacidades frente a un problema real, como es el caso de la gestión de pedidos en almacenes. A diferencia de cualquier ejercicio o práctica del grado, en este proyecto no hay ninguna guía ni información que te indique qué desarrollar, que tecnologías usar o como interpretar la información encontrada. Es el problema más grande al que me he enfrentado, y el que creo que es el más cercano a lo laboral desde que estudio informática.

[CM2] *Capacidad para conocer los fundamentos teóricos de los lenguajes de programación y las técnicas de procesamiento léxico, sintáctico y semántico asociadas, y*

saber aplicarlas para la creación, diseño y procesamiento de lenguajes.

Durante este proyecto he ampliado mis capacidades en cuanto a programación, sobre todo en el lenguaje Python, pero también indirectamente en cualquier lenguaje orientado a objetos. También he aprendido a crear interfaces gráficas con Python, cosa que nunca había necesitado y que puede ser de mucha utilidad. También he mejorado mis capacidades de comprensión en otros lenguajes, ya que durante todo el desarrollo del proyecto hemos estudiado muchas propuestas de otros estudiantes y científicos en Java, C++, C, pseudocódigo, etc.

[CM3] *Capacidad para evaluar la complejidad computacional de un problema, conocer estrategias algorítmicas que puedan conducir a su resolución y recomendar, desarrollar e implementar aquella que garantice el mejor rendimiento de acuerdo con los requisitos establecidos.*

Tras analizar muchos algoritmos de búsqueda para el *batching* de los pedidos y el cálculo de las rutas, creemos que la elección de los algoritmos desarrollados ha sido la adecuada, consiguiendo buenos resultados y comparando el rendimiento de los dos algoritmos.

[CM4] *Capacidad para conocer los fundamentos, paradigmas y técnicas propias de los sistemas inteligentes y analizar, diseñar y construir sistemas, servicios y aplicaciones informáticas que utilicen dichas técnicas en cualquier ámbito de aplicación.*

De nuevo, los algoritmos desarrollados son sistemas inteligentes que utilizamos para resolver el problema. Así mismo, hemos implementado una interfaz que usa esos sistemas inteligentes para mostrar de manera más clara las soluciones obtenidas y facilitando así el trabajo de los *pickers* en un entorno real.

[CM5] *Capacidad para adquirir, obtener, formalizar y representar el conocimiento humano en una forma computable para la resolución de problemas mediante un sistema informático en cualquier ámbito de aplicación, particularmente los relacionados con aspectos de computación, percepción y actuación en ambientes o entornos inteligentes.* Esta competencia es una de las más importantes y trabajadas a lo largo de la intensificación de computación. En el futuro laboral es muy importante entender lo que el cliente quiere y traducir a un lenguaje de programación para resolver el problema que te pidan computacionalmente. En este caso, esta competencia ha sido trabajada en muchos casos: entender el problema, saber codificar el almacén, saber interpretar las instancias y trabajar con ellas, desarrollar cada uno de los algoritmos aplicados a nuestro problema a partir de la teoría de cada uno de esos algoritmos, etc.

[CM6] *Capacidad para desarrollar y evaluar sistemas interactivos y de presentación de información compleja y su aplicación a la resolución de problemas de diseño de interacción persona computadora.*

La interfaz implementada, aunque sea simple, y no una aplicación real con acceso a una base de datos, etc. es un sistema interactivo que cumple con el objetivo propuesto, y presenta la información de manera clara al cliente consiguiendo así esa interacción persona-ordenador.

[CM7] *Capacidad para conocer y desarrollar técnicas de aprendizaje computacional y diseñar e implementar aplicaciones y sistemas que las utilicen, incluyendo las dedicadas a extracción automática de información y conocimiento a partir de grandes volúmenes de datos.*

Tal vez esta competencia haya sido la que menos hemos fortalecido en el proyecto. Hemos utilizado un set de instancias obtenido manualmente de un proyecto en Internet, pero no de manera automática ni hemos trabajado con grandes volúmenes de datos. Existen otros campos como la minería de datos, en los que sí se trabaja mucho más con este tipo de datos, pero en nuestro caso no es así, ya que se trata de un problema concreto acerca de un almacén. Si este proyecto avanzara hacia un trabajo más grande, con almacenes e instancias reales como hemos comentado anteriormente, sí se podría hablar de grandes volúmenes de datos.

5.4. Probar código

Hemos subido a un repositorio en GitHub el código fuente, una copia de esta memoria y una pequeña descripción de las clases. Está disponible en <https://github.com/Alexferjona/TFG>

Bibliografía

- [1] E. AGRO, “IV Observatorio para la Evolución del Comercio Electrónico en Alimentación: hacia la proximidad”, 2020. dirección: <https://www.youtube.com/watch?v=NEk0d2Zt2hI>.
- [2] L. M. Alcantarilla, “Origen del palet europeo”, 2017. dirección: <https://www.lopezmarin.com/es/paginas/ver/que-es-el-europalet>.
- [3] F. Alonso, “La compra ‘online’ de alimentación y textil crece un 50% por el Covid-19”, 2021. dirección: <https://revistas.eleconomista.es/transporte/2021/enero/la-compra-online-de-alimentacion-y-textil-crece-un-50-por-el-covid-19-CG5967235>.
- [4] Anonymous, “Mercadona: un centro logístico para 46.000 paletas”, 2014. dirección: <https://www.mecalux.es/articulos-de-logistica/mercadona-centro-logistico-46000-paletas>.
- [5] —, “Witron se adjudica la automatización de un nuevo almacén de Mercadona”, 2015. dirección: <https://www.alimarket.es/logistica/noticia/182204/witron-se-adjudica-la-automatizacion-de-un-nuevo-almacen-de-mercadona>.
- [6] —, “Ciclo de Hamilton”, 2016. dirección: <http://m4tem4ticasdiscret4s.blogspot.com/2016/11/ciclo-de-hamilton.html>.
- [7] —, “Mercadona estrena supermercado online con estanterías para picking de Mecalux”, 2018. dirección: <https://www.mecalux.es/noticias/almacen-estanterias-picking-mercadona-supermercado-online>.
- [8] —, “Algoritmo de Floyd-Warshall”, 2020. dirección: http://arodrigu.webs.upv.es/grafos/doku.php?id=algoritmo_floyd_warshall.
- [9] —, “Principio de optimalidad de Bellman”, 2020. dirección: <http://diccionario.raing.es/es/lema/principio-de-optimalidad-de-bellman>.

-
- [10] E. E. Bischoff y M. D. Marriott, "A comparative evaluation of heuristics for container loading", *European Journal of Operational Research*, vol. 44, n° 2, págs. 267-276, 1990. dirección: <https://www.sciencedirect.com/science/article/pii/037722179090362F>.
- [11] R. Bowles, "Warehouse Optimization - Algorithms For Picking Path Optimization", 2020. dirección: <https://www.logiwa.com/blog/picking-path-optimization-algorithm>.
- [12] M. Campbell, A. J. H. Jr. y F.-h. Hsu, "Deep Blue", *Artificial Intelligence*, vol. 134, n° 1-2, págs. 57-83, 2002. dirección: <https://www.sciencedirect.com/science/article/pii/S0004370201001291>.
- [13] F. S. Caparrini, "Algoritmos de hormigas y el problema del viajante", 2018. dirección: <http://www.cs.us.es/~fsancho/?e=71>.
- [14] C. S. Chen, S. M. Lee y Q. S. Shen, "An analytical model for the container loading problem", *European Journal of Operational Research*, vol. 80, n° 1, págs. 68-76, 1995. dirección: <https://www.sciencedirect.com/science/article/pii/037722179400002T>.
- [15] C. Davidson, "What Is A Programming Algorithm?", 2020. dirección: <https://www.indicative.com/data-defined/programming-algorithm/>.
- [16] E. Franzelle, "World-Class Warehousing and Material Handling", 2002. dirección: <https://pdfcoffee.com/world-class-warehousing-and-mat-edward-h-frazellepdf-pdf-free.html>.
- [17] E. Franzelle, "World-Class Warehousing and Material Handling", 2002. dirección: <https://www.amazon.com/-/es/Edward-Frazelle/dp/0071376003>.
- [18] M. Gallego, "Order Batching Problem", 2016. dirección: <http://grafo.etsii.urjc.es/optsi.com/obp/>.
- [19] H. Gehring, K. Menschner y M. Meyer, "A computer-based heuristic for packing pooled shipment containers", *European Journal of Operation Research*, vol. 44, n° 2, págs. 277-288, 1990. dirección: <https://www.sciencedirect.com/science/article/pii/037722179090363G>.
- [20] J. A. George y D.F. Robinson, "A heuristic for packing boxes into a container", *Computers and Operations Research*, vol. 7, n° 3, págs. 147-156, 1980. dirección: <https://www.sciencedirect.com/science/article/pii/0305054880900015>.
- [21] C.-M. Hsu, K.-Y. Chen y M.-C. Chen, "Batching orders in warehouses by minimizing travel distance with genetic algorithms", *Computers in Industry*, vol. 56, n° 2, págs. 169-178, 2005. dirección: <https://doi.org/10.1016/j.compind.2004.06.001>.
- [22] B. Jang, M. Kim, G. Harerimana y J. W. Kim, "Q-Learning Algorithms: A Comprehensive Classification and Applications", *IEEE*, vol. 7, págs. 133 653-133 667, 2019. dirección: https://www.researchgate.net/publication/335805245_Q-Learning_Algorithms_A_Comprehensive_Classification_and_Applications.
-

- [23] G. Kizilates-Evin y F. Nuriyeva, “On the Nearest Neighbor Algorithms for the Traveling Salesman Problem”, *Advances in Intelligent Systems and Computing*, vol. 225, 2013. dirección: https://www.researchgate.net/publication/289195926_On_the_Nearest_Neighbor_Algorithms_for_the_Traveling_Salesman_Problem.
- [24] L. Marsán, “El Observatorio Cetelem de eCommerce 2020”, 2020. dirección: <https://elobservatoriocetelem.es/observatorio-cetelem-ecommerce-2020>.
- [25] S. Martello, D. Pisinger y D. Vigo, “The Three-Dimensional Bin Packing Problem”, *Operations Research*, vol. 48, nº 2, págs. 256-267, 1998. dirección: https://www.researchgate.net/publication/2353632_The_Three-Dimensional_Bin_Packing_Problem.
- [26] S. Martello y D. Vigo, “Exact Solution of the Two-Dimensional Finite Bin Packing Problem”, *INFORMS*, vol. 44, nº 3, págs. 388-399, 1997. dirección: https://www.jstor.org/stable/2634676?seq=1#metadata_info_tab_contents.
- [27] Mbalsells, “Camino más corto entre nodos”, 2019. dirección: <https://aprende.olimpiada-informatica.org/algoritmia-dijkstra-bellman-floyd-warshall>.
- [28] D. Pisinger, “Heuristics for the container loading problem”, *European Journal of Operation Research*, vol. 141, págs. 383-392, 2002. dirección: <http://www.cmesinofrasassi.it/dmdocuments/load%20packing%20optim.pdf>.
- [29] M. Prieto, “La explosión del comercio electrónico”, 2020. dirección: <https://www.expansion.com/economia-digital/2020/08/20/5f3d852f468aeb11628b45c3.html>.
- [30] K. J. Roodbergen y R. de Koster, “Routing order pickers in a warehouse with a middle aisle”, *European Journal of Operation Research*, vol. 133, nº 1, págs. 32-43, 2001. dirección: <https://www.sciencedirect.com/science/article/abs/pii/S0377221700001776>.
- [31] G. Scheithauer, “A Three-dimensional Bin Packing Algorithm”, *J. Inf. Process. Cybern.*, vol. 27, nº 5/6, págs. 263-271, 1991. dirección: https://www.researchgate.net/publication/220287826_A_Three-dimensional_Bin_Packing_Algorithm.
- [32] A. Sharma, “Basics of Greedy Algorithms”, 2020. dirección: <https://www.hackerearth.com/practice/algorithms/greedy/basics-of-greedy-algorithms/tutorial/>.
- [33] E. Stoltz, “Evolution of a salesman: A complete genetic algorithm tutorial for Python”, 2018. dirección: <https://towardsdatascience.com/evolution-of-a-salesman-a-complete-genetic-algorithm-tutorial-for-python-6fe5d2b3ca35>.

-
- [34] R. B. Thomsen, “19 datos estadísticos de e-commerce que puedes incluir en tu estrategia de marketing”, 2020. dirección: <https://sleeknote.com/es/blog/datos-estadisticos-e-commerce>.
- [35] U. of Valencia scientists y U. R. J. C. scientists, “Opticom. Optimization of Complex Systems”, 2006. dirección: <http://grafo.etsii.urjc.es/opticom/index.php>.
- [36] T. S. Vicente, “La compra en los supermercados «online» se dispara casi un 50% en marzo por el estado de alarma”, 2020. dirección: https://www.abc.es/economia/abci-compra-supermercados-online-dispara-casi-50-por-ciento-marzo-estado-alarma-202004081446_noticia.html.
- [37] I. Warehouse, “Q-Learning Algorithms: A Comprehensive Classification and Applications”, 2008. dirección: <http://www.roodbergen.com/warehouse/index.php>.
- [38] J. Yu, R. Li, Z. Feng, A. Zhao, Z. Yu, Z. Ye y J. Wang, “A Novel Parallel Ant Colony Optimization Algorithm for Warehouse Path Planning”, *Journal of Control Science and Engineering*, vol. 2020, 2020. dirección: <https://www.hindawi.com/journals/jcse/2020/5287189/>.