Resolviendo el Order Picking Problem en almacenes irregulares

Grado en Ingeniería de Computadores

Proyecto Fin de Grado

Universidad Politécnica de Madrid

ETSI SISTEMAS INFORMÁTICOS

AUTOR
Humberto Carlos Duarte Peña
Tutor: Sergio Gil Borrás

Índice

Índice de Tablas

Índice de Figuras

Objetivo

Con el desarrollo de este proyecto se tiene una serie de metas y objetivos específicos enunciados a continuación:

Diseñar y formular el modelo para optimizar el Order Picking Problem(OPP) pero específicamente en almacenes irregulares. Tomando como objetivo minimizar los parámetros de tiempo de cómputo y distancia de la solución y con las restricciones propias impuestas por este problema.

También se tiene como meta implementar algoritmos basados en vecindad con la finalidad de mejorar los resultados que entregan las investigaciones sobre este problema de optimización en entornos irregulares. Para esto se desarrollan dos versiones de estos algoritmos, uno más simple como el BVNS el cual implementa una estructura básica de las metaheurísticas adaptándose a la geometría irregular y otro más complejo siendo el GVNS mejorando el algoritmo anterior pues realiza búsquedas locales más exhaustivas generando por lo consecuente el uso de mayores recursos con el fin de obtener mejores resultados.

Evaluar y comparar la eficiencia algorítmica tanto en tiempo de cómputo como de la calidad de la solución de diferentes algoritmos para resolver el problema de rutas para la recogida de pedidos en almacenes logísticos es un objetivo de una importancia. Complementario a esta comprobación de la eficiencia surge la necesidad de estimar la sensibilidad de los parámetros de cada algoritmo como el tamaño del almacén, su disposición, la cantidad de perdidos entre otros parámetros comunes a cada implementación. La finalidad de comparar una variedad de algoritmos es poder determinar las fortalezas y debilidades de cada algoritmo al enfrentarse a las diferentes instancias que se puede encontrar en entornos reales buscando el mejor balance entre calidad de la ruta y coste computacional.

Otro objetivo que se tiene es contribuir con el estudio del OPB en almacenes irregulares pues sobre estos existe relativamente poca investigación a diferencia de los almacenes regulares. Tratando de conseguir que no solo se amplíe la literatura sino también impulsar y motivar futuras investigaciones en esta aérea.

Por lo que optimizar este proceso logístico provoca un beneficios tanto para aquellas empresas pequeñas con almacenes hasta aquellas multinacionales como Amazon o Aliexpress reduciendo costes y tiempos lo cual se traduce en resolver más pedidos en menor tiempo y por lo consecuente más beneficios.

Resumen

Principalmente nos centraremos en resolver un problema de optimización característico de los almacenes. En los almacenes logísticos se llevan a cabo una diversa variedad de actividades las cuales generan problemas de optimización aunque en este documento nos enfocaremos principalmente en el problema de la generación de rutas en la recogida de pedidos. Este problema en específico es estudiado en la literatura y reconocido con el nombre de OPP. El primero es aquel que se realiza una búsqueda heurística mediante A*. El segundo método empleado proviene de una investigación realizada por un estudiante de la Universidad WUZI de Beijing [1] en la cual se plantea utilizar algoritmos genéticos para dar soluciones a esta cuestión. Por último tenemos dos algoritmos de la misma familia, por una parte tenemos Basic Variable Neighborhood Search (BVNS) y General Variable Neighborhood Search (GVNS). Ambos pertenecen a la familia de algoritmos Variable Neighborhood Search (VNS) los cuales son algoritmos metaheurísticos utilizados para resolver problemas de optimización mediante métodos de búsqueda local. Para comprobar cómo se desenvuelven estos algoritmos en diferentes entornos fueron diseñados inicialmente estructuras de almacenes irregulares con dimensiones pequeñas con el fin de determinar que las rutas que se siguen son los adecuados para posteriormente utilizar almacenes con un mayor volumen de artículos que se asemejan más a reales. Por otra parte para aportar una mayor variabilidad a estos diseños se utilizan diferentes disposiciones encontradas y estudiadas en la literatura acerca de almacenes irregulares entre las cuales tenemos las distribuciones chevron, fishbone, grid o parrilla, radial entre otros.

Las soluciones de estos algoritmos se comparan mediante la distancia que debe recorrer y el tiempo que tarda cada algoritmo en llegar a una solución. Para uso de los algoritmos anteriormente mencionados se plantea en un contexto de almacenes logísticos en donde se contienen estanterías con productos diversos. Por lo tanto tendremos una serie de nodos que representan el mapa del almacén y una serie de pedidos siendo los artículos que el trabajador del almacén debe recoger coincidiendo con el nodo del almacén. Al igual que para cada problema de optimización para el OPP existe un método exacto para poder determinar la solución en instancias pequeñas. La mejor solución, nos indica una cota pues nuestros algoritmos serán capaces de dar siempre soluciones iguales o peores permitiendo servir como referencia. La problemática de estos algoritmos exactos que requieren de un mayor tiempo de cómputo y en los problemas NP-difíciles el tiempo de computo crece de forma exponencial, debido a esto surge la necesidad de encontrar soluciones que a pesar de no ser las mejores puedan ser obtenidas en un tiempo razonable que permita emplear estas soluciones en entornos reales.

En cuanto a los resultados observables el algoritmo A* devuelve soluciones muy sólidas en entornos pequeños y controlados y a pesar de no dar las mejores soluciones en mapas de grandes dimensiones entrega soluciones en apenas unos milisegundos siendo mucho más rápido que el resto. De forma complementaria GVNS presentan un mayor equilibrio con aquellas instancias grandes utilizando una mayor cantidad de

recursos computaciones, causando que tarden una mayor cantidad de tiempo pero siendo el coste de devolver unos resultados respectivamente mejores.

Abstract

We will mainly focus on solving an optimization problem characteristic of warehouses. In logistics warehouses, a diverse variety of activities are carried out, which generate optimization problems, however in this document, we will focus primarily on the problem of route generation in order picking. This specific problem is studied in the literature and is recognized under the name OPP. The first approach consists of performing a heuristic search using A*. The second method used, comes from research conducted by a student at WUZI University in Beijing [1], in which the use of genetic algorithms is proposed to provide solutions to this issue. Finally, we have two algorithms from the same family: Basic Variable Neighborhood Search (BVNS) and General Variable Neighborhood Search (GVNS). Both belong to the family of Variable Neighborhood Search (VNS) algorithms, which are metaheuristic algorithms used to solve optimization problems through local search methods. To verify how these algorithms perform in different environments, irregular warehouse structures with small dimensions were initially designed to determine that the routes followed were appropriate, and subsequently, warehouses with a larger number of items were used to better resemble real cases. Moreover, to provide greater variability to these designs, different layouts found and studied in the literature on irregular warehouses were used, among which are the chevron, fishbone, grid or mesh, and radial configurations, among others.

The solutions of these algorithms are compared based on the distance that must be traveled and the time each algorithm takes to reach a solution. The previously mentioned algorithms are applied in the context of logistics warehouses containing shelves with various products. Therefore, we will have a series of nodes representing the warehouse map and a series of orders, with the items that the warehouse worker must pick corresponding to the warehouse nodes. As with any optimization problem, for the OPP there exists an exact method to determine the solution in small instances. The best solution provides a bound, as our algorithms will always be capable of producing solutions that are equal to or worse than this one, serving as a reference. The problem with these exact algorithms is that they require greater computational time, and for NP-hard problems, the computation time grows exponentially. For this reason, it becomes necessary to seek solutions that, while not necessarily optimal, can be generated within a reasonable amount of time, making them practical and applicable in real-world scenarios.

In terms of observable results, the A* algorithm delivers very solid solutions in small and controlled environments, and although it does not provide the best solutions in large-scale maps, it produces results in just a few milliseconds, making it much faster than the others. Complementarily, GVNS presents a better balance for larger instances, using a greater amount of computational resources, which causes it to take more time, but at the cost of returning comparatively better results.

Lista de acrónimos

>> **OPP:** Problema de recogida de pedidos **(**Order Picking Problem)

>> **OBP:** Problema de recogida de pedidos por lotes **(**Order Batching Problem)

>> **WMS:** Sistemas de gestión de almacenes

>> **IoT:** Internet de las cosas

>> **RFID:** Identificador de radio frecuencia

>> **MILP**: Programación Lineal Entera Mixta

>> **VND**: Variable Neighborhood Descent

>> **VNS**: Variable Neighborhood Search

>> **BVNS**: Basic Variable Neighborhood Search

>> **GVNS**: General Variable Neighborhood Search

>> **GRASP**: Procedimiento codicioso de búsqueda adaptativa aleatorizada

>> **CO2**: Dióxido de carbono

>> **IDE**: Entorno de desarrollo Integrado

>> **GA**: Algoritmo genético

>> **openJDK**: Open Java Development Kit (Kit de desarrollo de Java)

1. Introducción

Con la creciente automatización y digitalización de los procesos logísticos, se han puesto en el punto de mira de gran parte de los investigadores los procesos que son llevados a cabo en los almacenes. El hecho de realizar cada proceso de la forma más eficiente genera resolver diferentes procesos de optimización, que influyen directamente en los beneficios económicos, siendo esto uno de los factores que más han impulsado las investigaciones en estos sectores. Los problemas de optimización empiezan desde la selección de la forma que tendrá, organizar como se van a distribuir las zonas de almacenamiento, pasillos y equipos, generando problemas como áreas irregulares que no pueden usarse eficientemente, los pasillos deben tener cierta coherencia cuando no son paralelos o ángulos que no son rectos y la existencia de obstáculos que fragmentan el espacio y limitan los libres movimientos por los pasillos. Otro problema típico de optimización es la asignación de los artículos en el almacén pues cada colocación en concreto genera tiempos de recogida y recorrido por los operarios diferentes, cuando se tiene una demanda de productos definida, la colocación de estos permite disminuir la distancia requerida para recoger los pedidos. Otro problema típico consiste en cómo se van a planificar las rutas en las cuales se recojan los artículos de un determinado pedido utilizando la menor ruta posible, siendo este problema en el cual nos centraremos y enfocaremos el desarrollo de este. También es un problema destacable y recurrente la gestión actualizada del inventario pues existe continuamente un flujo de productos que entran y otro que salen, con productos perecederos los cuales no pueden estar almacenados de forma permanente. El período de tiempo que se requiere reponer cada artículo es diferente por lo que puede influir en la gestión de las existencias y añadiéndole un grado de dificultad a este problema. Los procesos de almacenamiento y los problemas de optimización que en este tienen lugar exigen soluciones que requieren combinar las tecnologías con diseños flexibles y algoritmos personalizados, haciendo que sin estas optimizaciones aumenten de manera significativa los costes operativos y de los costes a todos los productos y servicios relacionados a esta actividad.

Entre la variedad de complicaciones que nos podemos enfrentar, la eficiencia en la planificación de rutas dentro de los almacenes ha sido la seleccionada por importancia e impacto en diferentes aspectos. Esto es especialmente crucial en entornos industriales donde los espacios de almacenamiento presentan estructuras irregulares, caracterizadas por pasillos no uniformes, obstáculos fijos y áreas de acceso restringido. La identificación de rutas óptimas para el tránsito de operarios o vehículos autónomos se convierte, por ende, en un desafío computacional significativo. Tales escenarios dan lugar a lo que se denomina el problema de rutas para la recogida de pedidos (OPP, por sus siglas en inglés), el cual es una variante del clásico Problema del Agente Viajero (TSP). En este caso, el objetivo consiste en visitar un conjunto específico de nodos donde se encuentran los artículos del pedido pero en un mismo instante se debe elegir cual nodo debe ser el siguiente en ser visitado y respetando las restricciones impuestas por la topología del almacén específico.

En la literatura en relación al problema de optimización el cual abordaremos tenemos métodos exactos los cuales devuelven la solución óptima pero teniendo problemas con la escalabilidad pues para estos problemas NP-completo devuelve solucionen en cantidades de tiempo de forma exponencial. Los problemas NP-completos son aquellos donde si una respuesta es correcta es eficiente, pero hallar la mejor solución requiere cantidades de tiempo potencialmente exponencial y siendo uno de los más complejos para este tipo de problemas y al ser el OPP en almacenes irregulares un problema de este tipo es clave buscar otras perspectivas que conlleven un coste computacional menor. De esta perspectiva encontramos diferentes aproximaciones algunas basadas en modelos matemáticos, algoritmos heurísticos incluso algoritmos genéticos. Teniendo en cuenta las virtudes y limitaciones de los algoritmos que tenemos en la literatura podemos detectar una brecha y la posibilidad de mejorar los resultados con algoritmos metaheuristicos basados en vecindarios pues estos destacan por su efectividad, facilidad de implementación y bajo coste computacional. Esto representa una brecha las cual puede ser utilizada para entregar soluciones más cercanas al óptimo sin llegar a ser computacionalmente inviables en sistemas en tiempo real.

Esta propuesta de emplear algoritmos basados en vecindad viene con el objetivo de mejorar las soluciones entregadas por los algoritmos actuales en la literatura, especialmente en almacenes que presentan disposiciones irregulares, llevando a evaluar diferentes factores como calidad de la solución o el tiempo de procesamiento.

Tras esta introducción, queda que esta memoria se organiza de la siguiente manera quedando como la primera sección la Introducción de conceptos y escenarios vistos durante el desarrollo. En la sección 2 tenemos un estudio y análisis del estado de las investigaciones relacionadas con almacenes irregulares y las diferentes formas de resolver el OPP. En la sección 3 tenemos con mayor profundidad los detalles del problema que planteado y el cual se pretende resolver. Teniendo acotado el problema pasamos a la sección 4 en la cual se enfoca en las técnicas o algoritmos que ayudaran a resolver el problema y el proceso específico que conlleva cada uno. En el siguiente capítulo coincidiendo con la número 5 se tiene una explicación precisa de los pasos seguidos para poder seleccionar cual problemas de optimización se va a pretender resolver, las investigaciones acerca de este problema y como tratar de resolver los resultados de estas investigaciones visitadas con anterioridad. Las soluciones algorítmicas necesitarán entornos simulados sobre los cuales tomar restricciones típicas del problema como es el almacén con la cantidad de estanterías que tiene, su topografía y los costes que conlleva el movimiento de un nodo a otro nodo dentro de este almacén y una lista de pedidos con los artículos que son necesarios recoger.

Teniendo el problema y como solucionar este problema podemos pasar a definir y enunciar los resultados obtenidos, además de que las conclusiones que se pueden tomar a partir de estos resultados. Para cualquier proyecto es necesario una planificación de costes y plazos de tiempo que estarán a la disposición de la realización

de este proyecto, correspondiendo con el capítulo número 8 de esta memoria. Para el capítulo 10 se destina para el impacto social y medioambiental que trae consigo la implementación de rutas más eficientes en la recogida de pedidos en almacenes irregulares. Para el capítulo 11 tenemos las conclusiones de los resultados que hemos obtenidos, si son viables implementar estas soluciones en almacenes reales y a nivel general que ha sido el desarrollo de este proyecto. El siguiente capítulo es destinando para profundizar en las futuras mejoras que se puedan implementar a nuestra solución con el objetivo de tener unos resultados que se adapten de forma flexible al entorno que es cambiante en su naturaleza, para dar soluciones más acertadas según las condiciones específicas. Para concluir en las últimas sección es se tienen las referencias con enlaces a todos los documentos y materiales utilizados durante todo el desarrollo de este proyecto.

2. Estado del Arte

Con el crecimiento de grandes almacenes logísticos por empresas distribuidoras de productos hacen muy necesario optimizar el proceso tanto de almacenamiento como de recogida, evitando tanto costes adicionales, pérdidas de tiempo innecesarias, en general cualquier ineficiencia.

La búsqueda de rutas óptimas en almacenes es un tema sobre el cual se han realizado numerosas investigaciones debido al impacto directo que tiene sobre la eficiencia y los costes en la logística. El objetivo es encontrar el recorrido más corto que pase por un conjunto determinado de ubicaciones donde se encuentran los artículos de un pedido. Dichas rutas son utilizadas mayormente por operarios humanos pero en ocasiones también por sistemas autónomos, guiando a la siguiente cuestión. ¿Por qué a pesar de los avances en las automatizaciones en la gran mayoría de los sectores el recurso humano sigue siendo el mayoritario? [12]Esto se debe a la facilidad que tiene el ser humano para adaptarse al entorno dinámico pues pueden surgir bloqueos en los pasillos, averías en las herramientas o materiales, fenómenos medio ambientes o problemas en el stock por diversos motivos haciendo que el entorno de recogida no sea un entorno aislado o estático. También los trabajadores humanos son más flexibles a las variaciones de la demanda permitiendo tener trabajadores temporales en estas temporadas de alta solicitud de pedidos además que la inversión inicial es mucho menor incluso en almacenes pequeños. Al trabajo humano se le facilitan ciertas operaciones logísticas gracias a integraciones con la tecnología para optimizar aún más el desempeño de los operadores. Uno de estos sistemas es el WMS por ejemplo el WAMAS-5 el cual permite configuraciones de recogida de pedidos de forma única o múltiples, escanear los pedidos por códigos de barras y una navegación mejor optimizada. [6] En otros documentos se proponen el uso de modelos matemáticos y heurísticos con el objetivo de resolver la asignación de las ubicaciones y rutas en almacenes con estructuras generales, regulares e irregulares. Afianzando que el diseño estructurado de los pasillos facilita la aplicación de modelos exactos y metaheurísticas, como GVNS y algoritmos genéticos, que logran soluciones cercanas al óptimo en tiempos razonables.

En gran parte de los trabajos se centran en almacenes con diseño regular, caracterizado por la disposición paralela de los pasillos y la incorporación de tendencias como integraciones de tecnologías como el WMS para optimizar el inventario, el camino y la gestión de operaciones, también la incorporación de sistemas IoT y sensores para monitorear en tiempo real de condiciones ambientales y ubicar productos mediante RFID para el análisis de estos datos con el objetivo final de hacer predicciones de la demanda. También las automatizaciones parciales permiten tener al mismo tiempo usuarios humanos y usuarios robóticos en un mismo espacio desarrollando tareas complementarias. Al ser espacios regulares sus disposiciones modulares permiten reconfigurar espacios según las estaciones o variaciones en la demanda.

Recientemente se han comenzado a explorar el rendimiento de algoritmos y políticas de ordenamiento en almacenes con estructuras no tradicionales, como pueden ser almacenes con distribución fishbone, flying-V entre otros en donde no existen limitaciones geométricas de los almacenes regulares. Con estos diseños se pretende reducir las distancias en los caminos de operaciones de recogida especialmente en aquellas donde se aplican políticas de colocación de productos por clases, el cual es conocido como distribución ABC. Una de estas ideas es el uso de modelos estocásticos para determinar la distancia esperada en almacenes irregulares y almacenamiento en la forma ABC, comparando diferentes estrategias, demuestran que bajo ciertas condiciones la estrategia de entrar por un pasillo, recoger los artículos necesarios y retornar por el mismo, minimiza las distancias en entornos irregulares. También se demuestra por estos métodos estocásticos que los almacenes fishbone dan mejoras de hasta un 20% en recorridos simples y este rendimiento decrece frente a distribuciones tradicionales cuando la lista de pedidos es extensa. Este tipo de almacenes introduce problemas adicionales al obtener rutas debido a las geometrías. A pesar de que los estudios en relación a los almacenes irregulares han crecido y se observa una necesidad de evaluar la complejidad estructural de estos entornos cómo se comportan diversos algoritmos ante la complejidad estructural de estos entornos.

Al igual que la disposición de los almacenes, el cómo se colocan los objetos en el almacén provocan una mejora o un empeoramiento de las soluciones en dependencia de los pedidos específicos. Esto despierta el interés, llevando a cabo investigaciones acerca las cualidades de cada distribución. Entre las distribuciones más comunes encontramos el almacenamiento aleatorio, el basado en el volumen de los artículos y por clases ABC, siendo este ampliamente estudiado por su equilibrio entre eficiencia y facilidad de implementación.

En las diferentes [5] fases de la recogida de un pedido, entre el 55% y el 75% del tiempo total empleado es ocupado por la recogida de los pedidos, siendo uno de los más intensos en mano de obra y costos. Los pasillos no octogonales, los diagonales y los variables complican el proceso de calcular una distancia más corta de un punto a otro.

Tradicionalmente tenemos algoritmos como Dijkstra y A Estrella los cuales han sido utilizados para obtener formas de llegar de un punto a otro especialmente en entornos cuya orografía se encuentra bien definida. Estos nos devuelven la distancia mínima entre 2 puntos. Pero no genera una ruta mínima que atraviese todos los puntos como se busca en el TSP o en el OPP. Aunque ambos tienen una baja complejidad algorítmica, por ejemplo ambos en el peor de los casos ambos presentan una complejidad algorítmica de $O(n^2)$ y para Dijkstra cuando se implementa una cola de prioridades tal y como se implementa en este proyecto presenta una complejidad de $O(A \log(V))$ siendo A el número de aristas y V el número de vértices mientras que por otro lado el A estrella tiene una complejidad de $O(b^d)$ donde d es la profundidad del camino y b es el factor de ramificación siendo. El factor de ramificación es un concepto

proveniente de estructuras de datos en árbol representado el número promedio de nodos hijos que tiene cada nodo dentro de un grafo o árbol [4].

En la literatura podemos encontrar también otras perspectivas heurísticas clásicas para resolver el problema del OPP para almacenes regulares como la heurística S-shape [26] y la return [27]. El S-shape consiste en atravesar cada pasillo en donde se encuentre un objeto de forma completa en forma de "S". Siendo utilizado en almacenes con topología Fishbone y ordenamiento de productos por clases pero estando por dejado de lo óptimo en estos diseños irregulares debido a los patrones fijos de movimiento. Por otro lado el return consiste en entran en un pasillo recoger todos los productos necesarios en este y posteriormente retroceder por la misma dirección por donde se entró. Pero demostrando que el S-shape entrega rutas más cortas entre un 29 y un 90 % en los modelos fishbone en comparación con el algoritmo return.

Para resolver problemas de optimización podemos recurrir a modelos de programación lineal (LP). Para este problema podemos encontrar en la literatura modelos MILP esta variante de LP resuelve modelos donde existen variables de decisión con números enteros y números continuos. Esto es una técnica de optimización que se utiliza con la finalidad de resolver problemas donde existen variables de decisión son números enteros y otros son continuos. Esta herramienta es usada en sectores como la logística y el transporte, telecomunicaciones, finanzas, planificación de productos permitiendo modelar y resolver un mayor volumen de problemas ya que intervienen tanto decisiones discretas como continuas. Estos modelos matemáticos, se suelen emplear en instancias de reducido tamaño pues para aquellas de gran tamaño el tiempo de cómputo se dispara, no haciéndose viable su uso para problemas de esta envergadura.

En cuanto a los algoritmos evolutivos [24] como los algoritmos genéticos [25] han mostrado una buena capacidad para explorar espacios de soluciones, permitiendo llegar a soluciones en entornos complejos o irregulares. A pesar de no garantizar que la solución dada es las más óptimas pero su flexibilidad y rapidez para adaptarse son muy beneficiosas para aplicaciones que disponen de un tiempo limitado.

[2] En 1997 P. Hansen y N. Mladenovic presentan el algoritmo VNS un algoritmo de búsqueda basado en metaheurísticas. Este algoritmo presenta características que le proporcionan ventajas al enfrentarse a este problema pues son capaces de escapar de mínimos locales mediante la exploración a través de nodos vecinos favoreciéndoles en lugares no estructurados como pueden ser los almacenes irregulares o en lugares donde tenemos obstáculos dinámicos.

Teniendo en cuenta el estado actual del estudio de los almacenes irregulares surge el planteamiento de poner a prueba algoritmos para obtener algoritmos más robustos que se adapten de forma genérica a los diseños irregulares y no a un grupo en específico. La optimización de almacenes irregulares requiere una sinergia entre la búsqueda de rutas, la búsqueda global y el refinamiento local. Con nuestra integración

se espera reducir la distancia total de las rutas y mejorar la eficiencia de estos almacenes con topologías más complejas.

3. Descripción Del Problema

Entre las operaciones que tienen lugar en almacenes modernos, el cumplimiento eficiente de la recogida de los pedidos de los clientes implica generar una ruta que permita a un encargado o un robot visitar todos los artículos necesarios en la ruta más corta posible. La complejidad de encontrar una ruta aumenta cuando el diseño es irregular y dicha ruta debe pasar por cada elemento en un pedido siendo una problemática que se asemeja al Problema del viajero. En este documento se discute los desafíos y las compensaciones asociadas con cuatro algoritmos comunes utilizados para abordar este problema: A*, Algoritmos Genéticos (GA) y Variable Neighborhood Search (VNS).

Encontrar una ruta que visite todos los artículos en un orden con la mínima distancia y uso de CPU es un problema multiobjetivo difícil. Si bien Dijkstra y A* proporcionan precisión pero son menos escalables. En cambio, GA y VNS ofrecen aproximaciones flexibles y a menudo más lentas. La elección del algoritmo depende de la complejidad del almacén, las limitaciones de tiempo y los recursos computacionales.

Para este problema al igual que otros problemas conllevan limitaciones que deben seguir para poder considerar como válidas las soluciones entregadas. Para la ruta solución debe iniciar y terminar en el nodo Depot o en donde se depositan los artículos recolectados y el paso de un nodo a otro solo es posible si existe dicha conexión de estos nodos en la estructura donde se representan los pasillos y la distancia que hay de un nodo a otro. Para la lista de pedidos después de recoger todos los artículos de este de debe volver al nodo depot o de salida, el camino puede pasar por este nodo sin restricciones pero después de completar un pedido no puede recoger otro artículo hasta que vuelva a dicho nodo inicial. La lista de pedidos debe ser tratada según el orden que traen en la lista y sin poder mezclar artículos de diferentes pedidos.

Los almacenes logísticos están compuestos por estanterías y según la distribución de estas forman la disposición de los pasillos. En la literatura podemos encontrar distintas formas de organizar almacenes irregulares entre los cuales tenemos la distribución caótica o de forma aleatoria, distribución en V, distribución fishbone siendo muy similares.

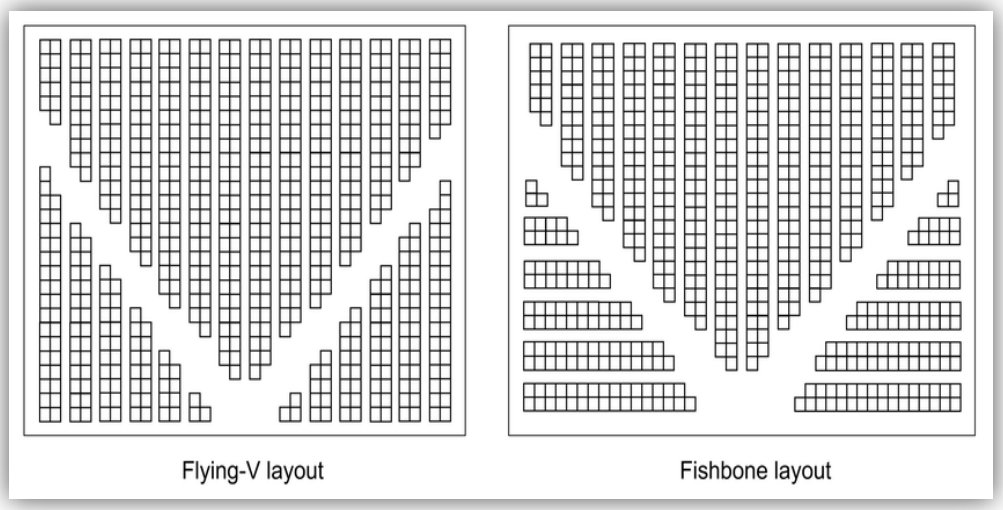

Figura 1: Modelo V o chevron en la imagen de la izquierda y fishbone a la derecha [11].

También podemos encontrar mezclas de distintas disposiciones utilizando rotaciones y traslaciones generando almacenes irregulares entre los cuales podemos encontrar la combinación de almacenes regulares donde si tomamos únicamente la parte izquierda o derecha nos queda un almacén regular. Por otra parte tenemos disposiciones como las que tienen forma de red, donde se da la impresión de regularidad pues los pasillos forman cuadrículas pero cada cuadrícula se encuentra rotada en comparación la cuadrícula adyacente. Para finalizar tenemos el modelo radial que se define por caminos que convergen en un centro.

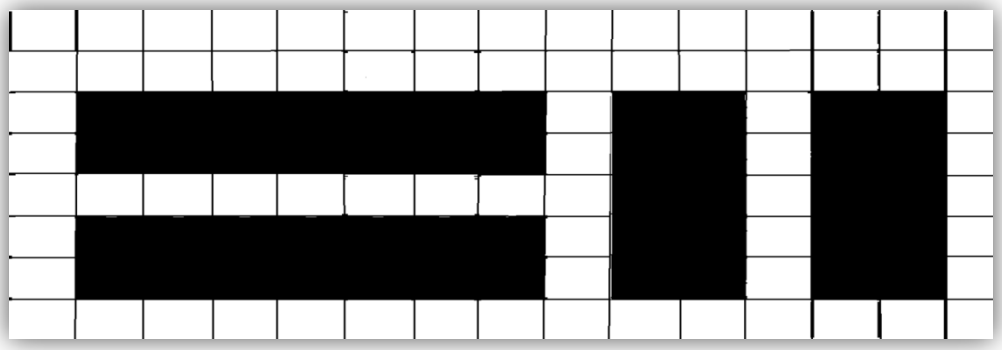

Figura 2: Almacén irregular formado por dos secciones regulares

Incluso la colocación de los objetos en las estanterías no se realiza de forma aleatoria en cualquier hueco libre sino que en la literatura podemos encontrar estudios sobre el impacto que provoca cada forma de agrupar los artículos en el problema que provoca su posterior recogida, encontrando organización por familia de productos que consiste en almacenar aquellos que son similares o suelen pedirse juntos cerca, organización por demanda estacional que consiste durante la estación del año los productos se colocaran en posiciones diferentes según su demanda, organización fija donde cada producto ocupa su posición concreta, organización caótica donde a diferencia del anterior ocupan posiciones aleatorias, organización ABC que consiste en colocar los pedidos más frecuentes más cerca de la salida entre otros tipos de organización.

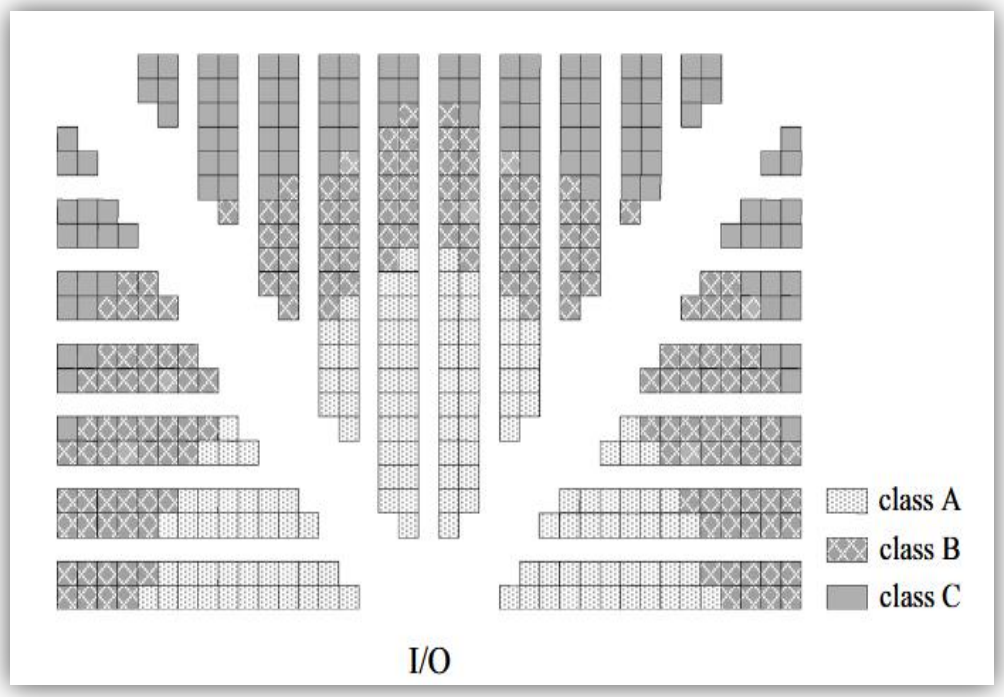

Figura 3: Distribución ABC de productos [10]

4. Propuesta algorítmica

La resolución de problemas de optimización ha sido un campo principal en la investigación y computación desde mediados del siglo XX, entre estos problemas como la planificación de rutas de un vehículo, planificación de horarios y el diseño de redes logísticas son algunos ejemplos que se pueden modelar como problemas de optimización. Con el paso del tiempo se ha ido pretendiendo optimizar instancias de estos problemas cada vez con un mayor tamaño, haciendo que los enfoques exactos se vuelven inviables debido al elevado coste computacional. Durante este contexto surgen distintas familias de algoritmos cada cual con sus motivaciones específicas.

Para este problema en específico o cualquier otro existen diversos enfoques o caminos para llegar a soluciones más o menos aceptables según la perspectiva que se opte. En este documento se han empleado los algoritmos que se pueden encontrar a continuación:

- Algoritmos clásicos de búsqueda

Dijkstra: Tanto este algoritmo como el A* se seleccionaron debido a que representan dos de los métodos más clásicos para realizar búsquedas no informadas sobre estructuras de grafos. Este algoritmo encuentra el camino más corto entre un nodo y otro, utilizando una cola de prioridad para explorar nodos con el menor coste acumulado pero debido a que no contiene información heurística recorre nodos innecesarios provocando un aumento en el tiempo de CPU especialmente en entornos muy grandes o dispersos.

Dijkstra se relaciona con el siguiente pseudocódigo:

```
1  function Dijktra(Graph. source):
2
3      for each vertex v in Graph.Vertices:
4          dist[v] <- Infinity
5          prev[v] <- undefined
6          add v to Q
7      dist[source] <- 0
8
9      while Q is not empty:
10         u <-vertex in Q with minimun dist[u]
11         Q.remove(u)
12
13         for each arc (u,v) in Q:
14             alt <- dist[u] + Graph.Edges(u,v)
15             if alt < dist[v]:
16                 dist[v] <- alt
17                 prev[v] <- u
18
19     return dist[], prev[]
```

Figura 4: Pseudocódigo del algoritmo Dijkstra [5]

Se necesita inicializar dos arrays, ambos con la cantidad de vértices del grafo donde uno contiene la distancia hasta el vértice actual explorado iniciándose a infinito para aquellos nodos que no han sido descubiertos y otro que contiene el siguiente nodo a visitar según el camino más corto desde el vértice actual. También se inicializa otro array donde se indica que nodos ya han sido visitados para evitar volver a pasar por ellos. El flujo normal comienza poniendo la posición actual del array de distancias a 0.

Se agrega a la lista de prioridad en nodo inicial de donde se partirá a buscar el nodo objetivo. Se añade la distancia acumulada desde el origen hasta el nodo actual la cual se almacena en uno de los arrays inicializados más la distancia desde el nodo actual a cada nodo vecino. Si la distancia obtenida es menor que la distancia guardada en la variable inicial. Se desencola el siguiente nodo en la lista de prioridad y si el nodo es el final podemos parar en caso de no ser así se repite el bucle hasta que la cola de prioridad este vacía o se encuentre el nodo objetivo.

Dijkstra no es capaz de mínimos locales y se encarga de devolver el menor coste entre dos nodos de un grafo.

A*: El surgimiento del algoritmo A* se encuentra ligado al estudio de la inteligencia artificial simbólica y la necesidad de resolver los problemas de búsqueda de caminos. Fue propuesto originalmente en 1968 y su objetivo inicial no se enfocaba en problemas de optimización sino más bien la búsqueda de rutas óptimas en grafos. La motivación era combinar la eficiencia de las heurísticas con la obtención de soluciones óptimas garantizada por algoritmos como Dijkstra.

La característica especial que tiene A* es que se introduce la evaluación f(n) = g(n) + h(n) representando el coste acumulado desde el nodo inicial y con una función heurística admisible que estima el costo hacia el destino, asegura encontrar la solución óptima reduciendo de forma radical el espacio de búsqueda. En contextos de planificación representa la diferencia entre obtener la solución en un tiempo considerable o no encontrar una solución. Este planteamiento tuvo gran aceptación debido a que la simplicidad y eficiencia, sin embargo un error frecuente en su implementación es la selección de una heurística no admisible, sobrestimando el coste y haciendo que no se pueda garantizan que la solución que se obtiene sea la óptima. Otra de las dificultades que presenta la implementación de este modelo es el consumo de memoria pues se debe mantener una cola de prioridad de todos los nodos abiertos.

A estrella es una variante informada de Dijkstra, utilizando una heurística para determinar el coste consecuente hasta el destino. A pesar de estas ventajas no se queda exento de ciertas limitaciones pues su rendimiento está directamente relacionado con localidad del heurístico además que presenta una mayor complejidad en memoria debido a que mantiene diversas rutas abiertas al mismo tiempo.

Se opta por su implementación debido a que es una versión del tradicional algoritmo de Dijkstra pero con ventajas sobre este al ser una búsqueda informada.

Al igual que Dijkstra no es capaz de escapar de los mínimos locales y devuelve un camino óptimo hasta el destino, no necesariamente devuelve el camino óptimo, para que esto suceda la función heurística debe ser admisible y no sobre estimar. La heurística utilizada consiste en determinar una matriz con las distancias mínimas utilizando el algoritmo de Floyd-Warshall ya que nos permite obtener de forma eficiente estas distancias. Esta aproximación es admisible y consistente por lo que garantiza que este algoritmo encuentre rutas óptimas pues no sobreestima los costos. También existen heurísticas más simples empleadas en las etapas iniciales la cual consistía en devolver la distancia si existía una conexión directa entre un nodo y otro y si no existía una conexión devuelve 0, con esta heurística también se cumplía que no se sobreestimaban los pesos.

El flujo general se describe mediante el siguiente pseudocódigo.

```
// Algoritmo A*
function A*(inicio, meta)
    abiertos = {inicio}
    cerrados = {}
    g[inicio] = 0
    f[inicio] = g[inicio] + h(inicio)
    repeat
        actual = nodo en abiertos con menor f
        if actual = meta then
            return ReconstruirCamino(actual)
        end if
        eliminar actual de abiertos
        agregar actual a cerrados
        for cada vecino en Vecinos(actual) do
            if vecino en cerrados then
                continue
            end if
            tentativo_g = g[actual] + costo(actual, vecino)
            if vecino no en abiertos then
                agregar vecino a abiertos
            else if tentativo_g >= g[vecino] then
                continue
            end if
            padre[vecino] = actual
            g[vecino] = tentativo_g
            f[vecino] = g[vecino] + h(vecino)
        end for
    return fallo
end function
```

Figura 5: Pseudocódigo del algoritmo A [5]*

Para su funcionamiento general se necesita inicializar una cola de prioridades donde se guaran aquellos nodos explorados pero no han sido evaluados. También se inicializan dos arrays con tantas posiciones como vértices tiene el grafo, en uno se almacena el coste conocido más bajo hasta ese nodo en concreto y otro que almacena la estimación total del coste del camino pasando por este nodo y para que se guarden las distancias más cortas está Variable debe ser inicializada a infinito o a Integuer.MAX_VALUE.

En cada bucle de la iteración principal se saca el siguiente elemento de la cola de prioridades el cual tiene la menor estimación total. Si el nodo seleccionado es el nodo final se reconstruye el camino que se ha seguido para llegar a este nodo mediante el método cameFrom.

En caso de que el nodo extraído no sea el objetivo se procede a explorar los vecinos y se calcula una distancia acumulada desde el inicio hasta este vecino y si está distancia es menor que la que se tenía previamente inicializada se actualiza el valor está nueva distancia. Si el vecino no se encuentra en la cola de prioridades se agrega para que pueda ser evaluado en situaciones futuras. Si se llega al punto de intentar de intentar pasar al nodo siguiente y la cola se encuentra vacía, se dice que no se encuentra una ruta y falla, de otra forma en cuanto se llega al objetivo se detiene la ejecución y se lanza el camino por el cual se ha llegado al nodo destino.

En el contexto especifico de almacenes irregulares, la eficiencia de este algoritmo recae de forma crítica en la calidad de la discretización de la disposición interna del almacén y de la heurística seleccionada que ayude a guiar la búsqueda hacia

- Algoritmos evolutivos

Los algoritmos genéticos toman idea de la biología y la teoría de la evolución de Charles Darwin como la mutación, las poblaciones, genes y entrecruzamiento. Estos algoritmos gracias a las mutaciones y entrecruzamiento realizan evitan caer en mínimos locales, son muy flexibles permitiendo optimizar múltiples criterios además de no necesitar funciones heurísticas bien definidas. Fueron formalizados en la década de 1970 teniendo como inspiración procesos en organismos vivos y con la motivación de resolver problemas complejos de optimización en los cuales la estructura matemática no está claramente definida. A diferencia de otras perspectivas que buscan únicamente un candidato, gracias a que estos algoritmos trabajan en poblaciones permiten evaluar diferentes perspectivas en una misma iteración. Los individuos que conforman una población van evolucionando con el tiempo y continuas iteraciones sobre los procesos de estos. La idea de trasladar los conceptos de entrecruzamiento, mutación y selección permiten encontrar mejores soluciones sin la necesidad de modelos exactos y aportando flexibilidad permitiendo que puedan ser empleados en problemas discretos, continuos, multiobjetivo sacrificando la garantía de que la solución obtenida sea la óptima.

Entre las limitaciones que presentan esta familia de algoritmos tenemos que no garantizan que la solución es la más óptima, requieren muchas generaciones para obtener buenos resultados, requiriendo mucha más carga computacional además de presentar una alta sensibilidad a los parámetros provocando desde una lenta convergencia a la solución hasta un comportamiento oscilante. Para este algoritmo se toma de referencia el artículo científico [1] “A New Genetic Algorithm for Order-Picking of Irregular Warehouse” para clonarlo y comparar las soluciones con las nuestras. El flujo normal de un algoritmo genético es el siguiente:

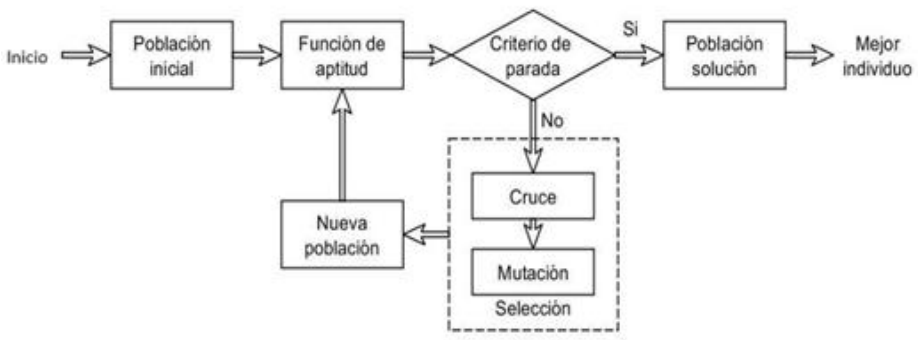

Figura 6: Flujo de un Algoritmo Genético [7]

Para iniciar la población inicial se genera a partir del pedido, cada artículo del pedido es un gen de cada individuo y el orden de los genes es el orden en que se van a recoger los artículos. En la investigación de la cual se toma esta idea no se especifica la cantidad de individuos que debe tener cada población por lo que se toma como poblaciones de 4 individuos partiendo de 1 pedido. Para el primer individuo se forma de la asignación directa de cada artículo, el segundo mediante el orden inverso al primero. El tercero consiste en iniciar por la mitad de la lista de artículos hasta el final seguido del inicio y el resto de aquellos que se encuentran en el individuo y por último el cuarto individuo se forma invirtiendo la segunda parte de la lista formando el inicio y el resto es la primera mitad invertida. Esto hace que nos quede en siguiente código:

Para la función de aptitud solo necesitamos que siempre se encuentre en cada gen un artículo del pedido, por lo que no provoca codificación adicional para el problema que estamos tratando.

El cruce y la mutación son las formas que tiene el algoritmo para escapar de mínimos locales debido a las variaciones y desordenamientos que hace a la solución. La mutación consiste en que el 80% de los individuos sufrirán cambios, se selecciona una subcadena que será invertida, quedando de la siguiente forma en nuestra implementación:

Para él entrecruzamiento al igual que para la mutación el 80% de los individuos sufrirán entrelazamiento. Este proceso se realiza seleccionando inicialmente dos individuos y el inicio y el fin de una subcadena, se intercambia la subcadena del primer individuo por la del segundo y se eliminan los genes repetidos. Por ejemplo teniendo los individuos A = [1, 2, 8, 9, 12, 10, 6, 3, 5,14, 4, 7, 13, 11] y B = [10, 8, 13,

11, 3, 6, 4, 9, 12, 14, 1, 2, 5, 7], se seleccionan las subcadenas y se usa el carácter “|” para delimitar la región solicitada quedando A= [1, 2, 8, 9, 12, 10, | 6,3, 5, 14, |4, 7, 13, 11] y B= [10, 8, 13, 11, 3, 6,| 4, 9, 12, 14,| 1,2, 5, 7], se intercambian las regiones y nos queda A'= [1, 2, 8, 9, 12, 10, | 4, 9, 12, 14, |4, 7, 13, 11] y B'= [10, 8, 13, 11, 3, 6,| 6, 3, 5, 14,| 1, 2, 5, 7]. Ahora al mover elementos nos quedan genes repetidos los cuales se tienen que eliminar y reemplazar por aquellos que desaparecen en el intercambio dejando como resultado final de este proceso A'= [1, 2, 8, 3, 5, 10,| 4, 9, 12, 14, |6, 7, 13, 11], and B'=[10, 8, 13, 11, 9, 4,| 6, 3, 5,14,| 1, 2, 12, 7].

Proceso de Entrecruzamiento

Figura 7: Flujo del Entrecruzamiento. Fuente Propia

Para eliminar estas repeticiones se eliminan aquellos elementos repetidos fuera del segmento, eliminamos esta repetición intercambiando el elemento externo al segmento por el elemento que existía anteriormente en la misma posición donde encuentra el elemento repetido.

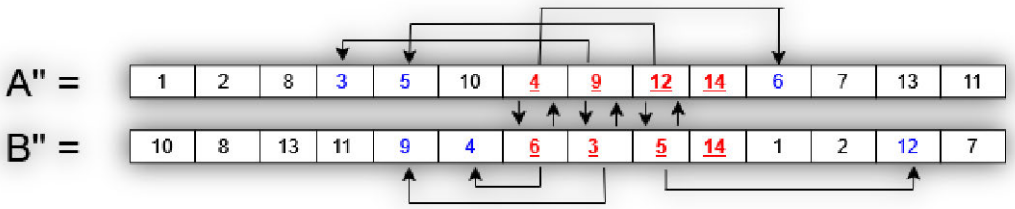

Figura 8: Flujo del Entrecruzamiento. Fuente Propia

Como el nodo 14 es el mismo en ambos segmentos, no se duplica por lo que hace innecesario estos movimientos.

Para determinar que individuos son mejores que otros o para simplemente determinar si las mutaciones y entrelazamientos provocan alguna mejoría se necesita determinar la calidad o función de ajuste donde el artículo nos da la siguiente fórmula:

$$F_j = 2 - 2 \times \frac{OrdJ - 1}{N - 1}$$

Figura 9: Formula del grado de entrenamiento o grado de calidad [1]

Donde tenemos que OrdJ es la distancia total al recoger los artículos según el orden de los genes, N es la cantidad de genes que tiene cada individuo y Fj es la calidad del individuo j. Para obtener la distancia entre un artículo de un gen y su adyacente se utiliza Dijkstra visto previamente.

En cada ciclo de entrenamiento del genético aquel que presenta peor calidad no pasan a la siguiente generación y estos espacios que dejan son ocupados por aquellos con mejor calidad siendo el 90% aquellos que pasan a la siguiente generación.

Para la condición de parada como se pretende comparar con otros algoritmos se pretende que se mantenga entrenando y dando ciclos durante un tiempo relativamente igual que el del resto de algoritmos para comprobar la calidad de sus salidas para tiempos similares de entrenamiento.

- Algoritmos Metaheurísticos basadas en búsquedas locales

VNS (Variable neighborhood search): Es un algoritmo que surge a finales de los años 90 como una respuesta de forma metódica a una de las limitaciones fundamentales de los algoritmos basados en metarheurísticas basadas en búsquedas locales. Estas búsquedas locales tienen como objetivo escapar de los óptimos locales de forma más controlada sin la necesidad de heurísticas sofisticadas, haciendo que sea conveniente su implementación en la resolución de problemas NP-completo. El hecho de estar reiteradamente cambiando de vecindario y no quedarse en la misma ayuda a diversificar las búsquedas. Estos algoritmos presentan otros beneficios como su facilidad de implementación y lo bien que se adaptan a múltiples problemas de combinatoria siendo de gran importancia para adaptar estos algoritmos a las dificultades topográficas de los almacenes irregulares.

[2] Es un método meta-heurístico para resolver un conjunto mediante optimización combinatoria. Explorando la distancia con los vecinos de una solución y se mueve hacia ellos sí y solo si se obtiene una mejoría. El método de búsqueda local se aplica de

forma repetida para llegar a las soluciones. Este algoritmo fue diseñado para soluciones aproximadas de problemas de optimización tanto continuos como discretos. VNS sistemáticamente cambia de vecino en dos fases. En primer lugar cuando desciende a encontrar un óptimo local y por último una perturbación en la fase en la que intenta salir del valle correspondiente mediante shaking(Agitación).

VNS es meta-Heurístico el cual se encuentra de forma sistemática mejorando el cambio de un nodo a su vecino. Este algoritmo está construido sobre las siguientes percepciones:

Un mínimo local con respecto a un nodo no es necesariamente un mínimo local de otro nodo. Un mínimo global es un mínimo local con respecto a todos los nodos.

Para muchos problemas, el mínimo local con respecto a uno o varios nodos están relativamente cerca uno de otro. BVNS y GVNS son algoritmos muy similares derivados del VNS siendo BVNS es más simple, realizando una única búsqueda local mientras que GVNS es más complejo debido a que realiza tanto múltiples como búsquedas locales adaptativas. Para estas dos versiones diferentes tenemos estas implementaciones:

El pseudocódigo del BVNS:

```
1  // Basic Variable Neighborhood Search
2  function BVNS (solution, kmax,tmax)
3      repeat
4          k = 1
5          while h <= kmax do
6              S' = shake(solution,k)
7              S* = Improvement(S')
8              k = NeighborhoodChange(solution.S*,k)
9          end while
10     until t < tmax
11     return solution
12 end function
```

Figura 10: Pseudocódigo del BVNS

El método shake se utiliza para escapar de los mínimos locales y existen muchas formas de hacerlo pero se usa el intercambio de uno a uno de forma aleatoria.

Por otro lado la función Improvement se encarga de explorar los vecinos para mejorarla de forma local. Esta búsqueda local se basa en intercambios de uno a uno hasta obtener el mejor resultado, intentando encontrar una solución y en caso de no hacerlo devuelve la solución inicial.

La función NeighborhoodChange es la que se encarga del cambio de vecindarios según la calidad de las búsquedas, comparando la solución nueva con la anterior y si la nueva es mejor que la anterior se reinicia la variable k a uno y en caso contrario de que no sea mejor se aumenta el valor de dicha variable k.

El pseudocódigo del GVNS:

```
1  // General Variable Neighborhood Search
2  function GVNS (solution, kmax,tmax)
3      repeat
4          k = 1
5          while h <= kmax do
6              S' = shake(solution,k)
7              S* = VND(S',N1,...Ni)
8              k = NeighborhoodChange(solution.S*,k)
9          end while
10     until t < tmax
11     return solution
12 end function
```

Figura 11: Pseudocódigo del GVNS

El GVNS al ser una versión del VNS tiene funciones similares al BVNS como puede ser el Shake y NeighborhoodChange pero se diferencia del básico pues no solo realiza una búsqueda local sino que realiza varias y se queda con la que mejor calidad tiene.

```
1  // Variable Neighborhood Descent
2  function VND (solution, N1,... Ni)
3      k = 1
4      kmax = i
5      best = solution
6      repeat
7          solution' = LocalSearch(best,Nk)
8          if eval(solution') < eval(best) then
9              best = solution'
10             k = 1
11         else
12             k = k + 1
13         end if
14     until k > kmax
15     return best
16 end function
```

Figura 12: Pseudocódigo del VND

El VND (Variable Neighborhood Descent) no es más que un método determinístico para buscar de forma sistemática mediante búsquedas locales y llegar a un mínimo local con respecto a los esquemas de vecindad.

Para este VND se utilizan 3 distintos tipos de búsquedas locales como por ejemplo el ya utilizado en el BVNS siendo intercambios uno a uno y tomando el mejor cambio. También tenemos el segundo el cual es un reordenamiento de subsegmentos de forma aleatoria, tomando la primera mejora y por último esta búsqueda hace inserciones de forma aleatoria tomando la primera mejora.

- Algoritmo exacto

Para poder determinar las soluciones óptimas las cuales sirven como un comparador para los algoritmos previamente implementados. Para almacenes irregulares podemos obtener este algoritmo exacto generando un modelo matemático del almacén utilizado y la lista de pedidos a comprobar. Teniendo estos modelos se hace uso de la herramienta Gubori [23] obtenemos los valores asociados tanto del camino más corto como del tiempo de cómputo utilizado. Esto lleva a plantear una cuestión, si mediante el exacto se obtiene la mejor solución ¿Por qué plantearse utilizar otras aproximaciones con soluciones peores? Pues esto se debe al problema que estamos tratando de resolver, el opp es un problema con una complejidad np-completo donde para enfrentarse a problemas con este tipo de complejidades se suele optar por el uso de heurísticas debido a que los resultados que suelen entregar se asimilan bastante sin generar el coste computacional desproporcionado de conlleva la ejecución de un modelo exacto.

Tal y como para el resto de problemas de optimización existen modelos exactos que permiten obtener la solución óptima a un problema específico. La principal desventaja de estos algoritmos es la elevada cantidad de tiempo que requieren por lo que para entornos reales y problemas de gran escala se vuelven ineficientes pues no solo es importante que los resultados sean lo más acertado posibles sino que también las soluciones sean obtenidas en un periodo de tiempo razonable para poder utilizar estos resultados.

El OPP es una versión del TSP donde se elimina las restricciones y existen diversos métodos que permiten transformar instancias de un problema a otro. Esto demuestra que están intrínsecamente ligados. Siendo de gran importancia a la hora de obtener un modelo matemático que pueda ser utilizado en la obtención de un algoritmo exacto. En el caso concreto del problema al que nos enfrentamos tiene características como la existencia de un solo picker o encargado de la recogida de los paquetes, los artículos tienen una posición predefinida en el almacén y que el operario encargado de realizar

esta labor inicia en el nodo depot y finaliza en este mismo. Facilitando que se transforme de forma más sencilla de un modelo OPP a un modelo TSP.

Inicialmente es necesario construir el grafo que representa los pasillos del almacén generándose a partir de los datos que se tienen guardados sobre el almacén. Partiendo de este grafo se determinan puntos relevantes como el depot y las ubicaciones de las ubicaciones de los artículos del pedido, se calcula la matriz de distancias entre los nodos determinados anteriormente y teniendo esta matriz de distancias y los nodos relevantes para este problema se define un TSP completo.

Teniendo el problema definido como se desea, debemos obtener un modelo matemático que lo defina, existiendo para esto una gran variedad de formulaciones y utilizando uno sencillo como MTZ (Millerf-Tucker-Zemlin). Para finalizar preparamos un entorno de simulación para la herramienta comercial Gubori, utilizando esta como entrada. Para realizar esta labor existe una variedad de herramientas mercantiles que realizan esta misma función como Cplex o localsolver haciendo que se cuente con una variedad opciones para poder resolver este problema y obtener un algoritmo exacto.

5. Metodología

El método utilizado en el proceso de este proyecto mediante un proceso iterativo basado en el método científico iterando entre investigación y desarrollo orientándolo a la optimización de las rutas de recogida de pedidos dentro de entornos logísticos. Describiendo su estructura en la figura 13. El hecho de optimizar estos procesos se traducen en cuantiosos beneficios para la empresa. Una de las tareas realizadas diariamente en estos lugares es la recogida de productos siendo uno de los cuales se destina la mayor cantidad de tiempo en donde encontramos el problema del orden de selección en que se recogerán esos pedidos.

Inicialmente se comienza en la fase de "Identificación del problema" detectando el problema que queremos mejorar. Entre la diversidad de problemas de optimación que existen optamos por enfocar el proyecto en la optimización de las rutas de la recogida de productos y específicamente en entornos irregulares.

Teniendo como objetivo optimizar este proceso en almacenes se tiene que conocer el estado actual de las investigaciones sobre este tema por lo que pasamos a la siguiente fase siendo la de "Estudio del estado del arte", para tener tanto un punto de partida como una comparación con las soluciones que somos capaces de dar. En la literatura se plantean diversas soluciones algunas utilizadas también en los almacenes regulares como s-shape y otras perspectivas que surgen como el basado en cruzamiento aleatorio, mediante modelos estocásticos, modelos matemáticos o heurísticas adaptativas como el GRASP , hasta incluso algoritmos genéticos. También con esta investigación se aclaró algunos de los diseños irregulares más frecuentes y más investigados. Con este estudio encontramos que cada disposición contiene una complejidad diferente tanto entre distintas disposiciones irregulares como con las disposiciones regulares. Con este estudio se llega a determinar que estas disposiciones conllevan una complejidad significativamente superior que los almacenes regulares incluso en diferentes distribuciones de almacenes irregulares las complejidades encontradas son diferentes.

De las investigaciones estudiadas debemos seleccionar un modelo tal y como indica la fase a la pasamos siendo "Seleccionar un modelo". Este modelo será tomado como referencia para intentar mejorar sus resultados utilizando otra perspectiva. Entre las soluciones encontradas en la gran variedad de investigaciones se optó por tomar el algoritmo genético y trataremos de obtener resultados mejores que los que se entrega por este algoritmo. Para implementarlo lo clonaremos de forma detallada para replicar su comportamiento, pues el hecho de generar variaciones en su comportamiento podría causar soluciones anómalas y que no sean válidas las comparaciones. Teniendo este algoritmo implementado es necesario poder compararlo por lo que se opta por implementar un algoritmo heurístico simple, optando por el clásico algoritmo de A*.

Teniendo únicamente estos algoritmos implementados no podemos comprobar su funcionamiento por lo tanto para poder resolver el problema de la recogida se

necesita que existan unos almacenes e instancias de lista de pedidos a recoger. Por lo que podemos pasar a la fase de “Instancias pequeñas de prueba” la cual e pretende crear un grupo de instancias sencillas con el fin de poder comprobar de forma más directa como está funcionando todo el sistema. Teniendo los almacenes y los pedidos ya es posible empezar a implementar las diferentes formas de obtener los caminos.

En este punto tenemos todo el entorno necesario para poder desarrollar nuevos algoritmos y unas instancias donde simularla permitiéndonos determinar la calidad de las soluciones de cada algoritmo y si hemos cumplido con el objetivo de mejorar las soluciones. Por lo cual se pasa a la fase de “Propuesta de un nuevo Algoritmo” donde debemos identificar algún algoritmo que de mejores resultados que de esto surge nuestra hipótesis de que utilizando algoritmos basados en vecindad se obtendrían mejores resultados en estos problemas de optimización y combinatoria que por los devueltos por los algoritmos de A* y genético ya implementados. Para implementar este algoritmo de vecindad los cuales presentan una enorme cantidad de variantes por lo que se opta por el BVNS es la versión más simplificada de este algoritmo de vecindad (VNS) y el GVNS el cual presenta una mayor cantidad métodos de búsquedas locales por lo que suele ofrecer mejores resultados ya que realiza mejores exploraciones.

En esta fase se implementan ambos algoritmos basados en vecindad. Teniendo las implementaciones de los algoritmos que queremos comprobar lanzamos una serie de entornos en los cuales se obtienen la información que deseamos optimizar, siendo que el camino que es utilizado para recoger los productos debe de ser el menor posible al igual que el tiempo empleado en la CPU para obtener estas rutas. Estas pruebas consisten en una serie de almacenes y una lista con los pedidos que hay que recoger en este almacén. Para ofrecer una mayor fiabilidad en las estadísticas que se toman estas instancias deben constar de una variedad y cantidad como para que las estadísticas que se tomen sean representativas de la población y no de individuos específicos.

Con estos datos obtenidos debemos analizar los valores medios en la fase de “Evaluación de resultados”, mínimos y mínimos locales para obtener como se están comportando y variando las soluciones planteadas por cada algoritmo. En caso que las soluciones obtenidas no fueran las esperadas o no superan las soluciones entregadas por el algoritmo genético, tenemos que volver a reformular una hipótesis volviendo a la fase de “Propuesta de un nuevo algoritmo” siguiendo el recorrido que nos ha llevado hasta este punto de tal forma que estamos refinando y validando este algoritmo propuesto. Hasta el punto que tenemos un algoritmo con la calidad adecuada como para cumplir los objetivos planteados desde el inicio del proceso llegando al fin de este flujo.

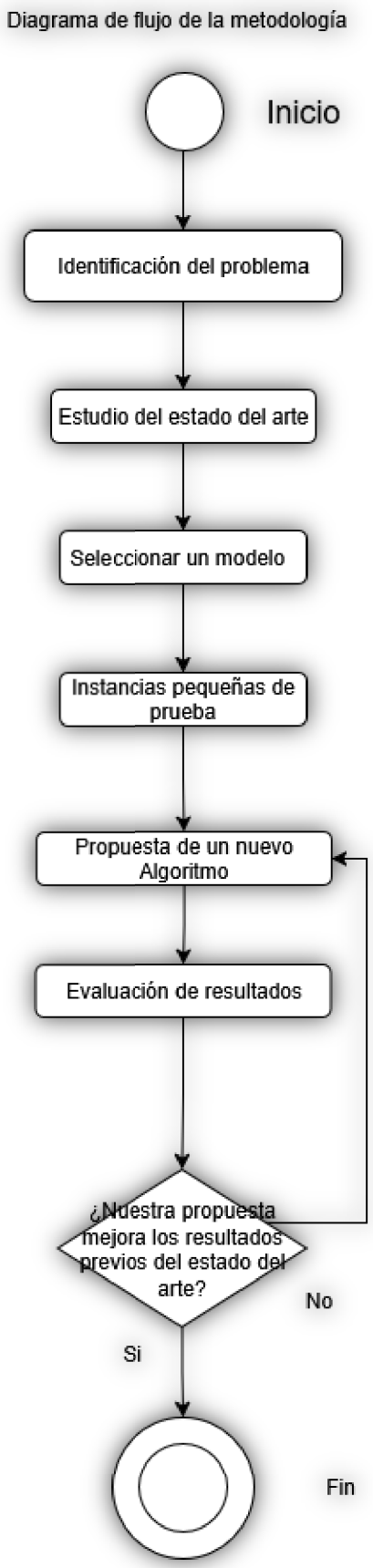

Figura 13: Diagrama de Flujo de la metodología seguida

6. Conjuntos de instancias

Para las instancias utilizadas donde se realizan las búsquedas de rutas utilizando las diferentes formas de obtener dicho recorrido se forman de dos partes. Por un lado se tiene el almacén, lugar donde se ubican los productos que se pretenden almacenar en estantes y las colocaciones específicas de los estantes provocan distribuciones en concreto. Los almacenes son almacenados mediante nodos almacenando los lugares donde se recogen y la distancia entre dos puntos de recogida adyacentes se toma como estándar de una unidad de distancia. Para obtener la distancia entre los diferentes nodos se colocan nodos invisibles para determinar la cantidad de unidades que separan un nodo del otro. Por otro lado tenemos la lista de pedidos, se tiene una variedad instancias de 30,60 y 90 pedidos. Cada pedido está formado por un número aleatorio entre 1 y 8 artículos. Las instancias se forman mediante la combinación de un almacén con una lista de pedidos. Estas instancias son almacenadas en ficheros JSON.

De forma general es importante indicar características esenciales de los almacenes pues estas instancias influyen en la solución obtenida y es esencial tener una variedad de estos para que los resultados obtenidos no dependan de tamaños o diseños específicos.

También es destacable que los primeros almacenes presentan unos tamaños y capacidades considerablemente bajos debido a que su objetivo más que parecerse a entornos reales su diseño es para validar y probar los procesos y sistemas del proyecto en un entorno controlado antes de escalar a operaciones más grandes.

Nombre del almacén	Número de artículos	Tamaño (m^2)
fishboneS50	50	183,75
PaperN3_161	161	520
PaperN8_234	234	493
chevronS50	50	240
warehouse_graph_chevrom_21A_256P	256	3010
warehouse_graph_chevrom_27A_384P	384	3360
warehouse_graph_chevrom_27A_400P	400	3600
warehouse_graph_chevrom_39A_952P	952	5280
warehouse_graph_chevrom_47A_1280P	1280	8160
warehouse_graph_circular_30A_244P	244	1680
warehouse_graph_circular_36A_396P	396	2100
warehouse_graph_circular_42A_416P	416	2520
warehouse_graph_circular_54A_912P	912	3840
warehouse_graph_circular_66A_1276P	1276	5160
warehouse_graph_fishbone_32A_388P	388	2888
warehouse_graph_fishbone_34A_250P	250	3010
warehouse_graph_fishbone_40A_402P	402	4060
warehouse_graph_fishbone_44A_1258P	1258	6237

warehouse_graph_fishbone_48A_928P	**928**	**6528**
warehouse_graph_grid_21A_244P	**244**	**3120**
warehouse_graph_grid_26A_394P	**394**	**4002**
warehouse_graph_grid_27A_400P	**400**	**3306**
warehouse_graph_grid_36A_918P	**918**	**5670**
warehouse_graph_grid_43A_1272P	**1272**	**6728**
warehouse_graph_radial_18A_248P	**248**	**3360**
warehouse_graph_radial_20A_386P	**386**	**3360**
warehouse_graph_radial_22A_414P	**414**	**3360**
warehouse_graph_radial_24A_902P	**902**	**5800**
warehouse_graph_radial_26A_1250P	**1250**	**7540**

Tabla 1: Características esenciales de los almacenes

Para nuestros primeros almacenes, se implementaron una variedad de diseños seleccionados con detenidamente. Esto nos lleva a utilizar configuraciones bien conocidas y estudiadas con profundidad por muchos de investigadores como el Chevron y el Fishbone, así como otros diseños innovadores. La mayoría de estas instancias fueron extraídas directamente de investigaciones previas sobre almacenes regulares e irregulares. La finalidad de estos almacenes es poder comparar rigurosamente los resultados de nuestras propias implementaciones con los hallazgos existentes en la literatura académica y la industria.

Teniendo en cuenta los tamaños, disponemos de almacenes pequeños y manejables, con capacidades para 50, 161 y 234 artículos. Estas dimensiones más reducidas fueron elegidas deliberadamente ya que nos permiten establecer entornos altamente controlados. Estos entornos acotados facilitan el seguimiento del flujo de las operaciones y con esto verificar que tanto el camino trazado como la distancia total de ese camino se corresponden con el almacén y la lista de pedidos que estamos comprobando. Permitiendo comprobar que los artículos se recogen de manera óptima y que se respetan todas las restricciones impuestas, ya sea por la naturaleza específica del problema o por la propia forma física del almacén.

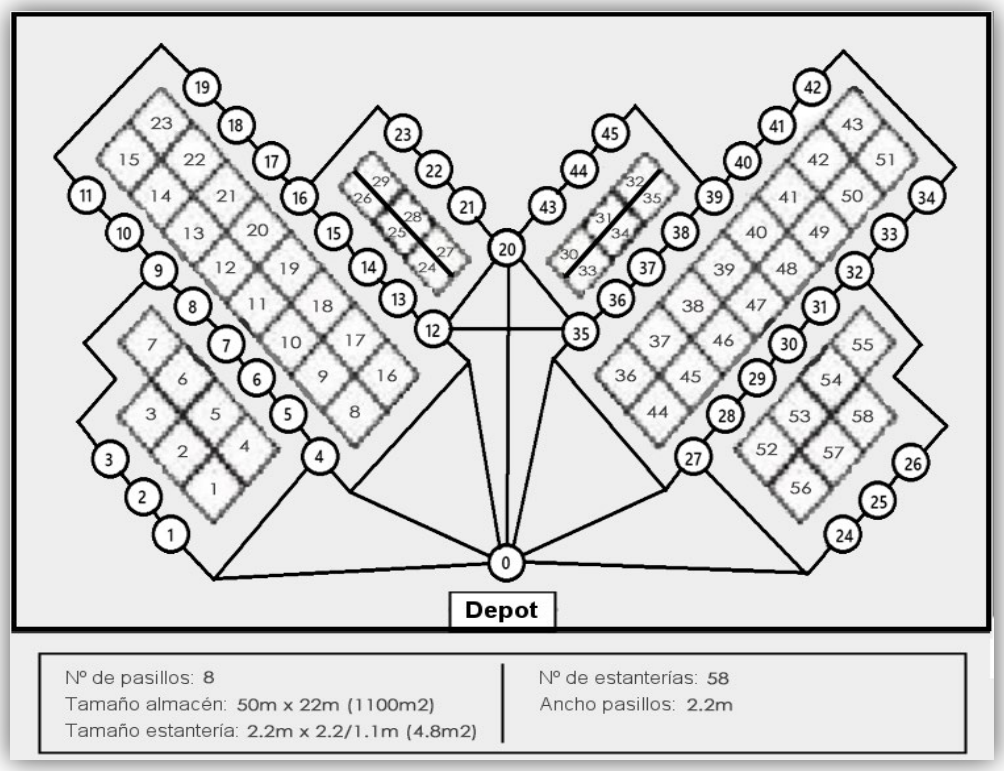

Figura 14: Almacén Irregular con distribución Chevron reducido para pruebas

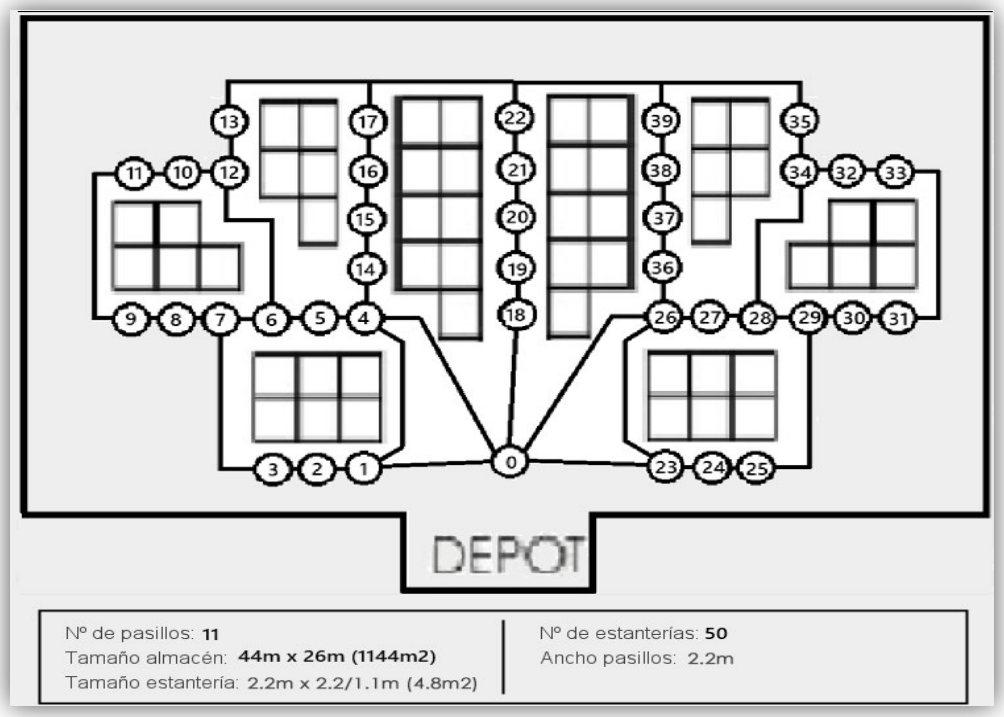

Figura 15: Almacén Irregular con distribución fishbone reducido para pruebas

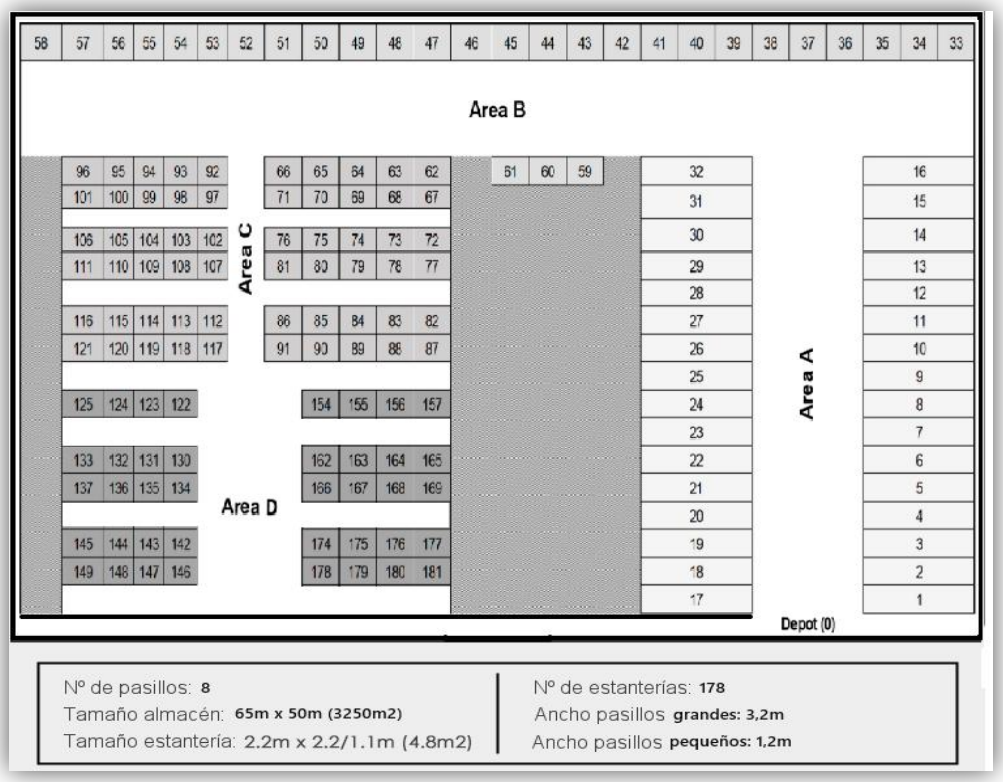

Figura 16: Almacén irregular extraído de un artículo científico [3] basico

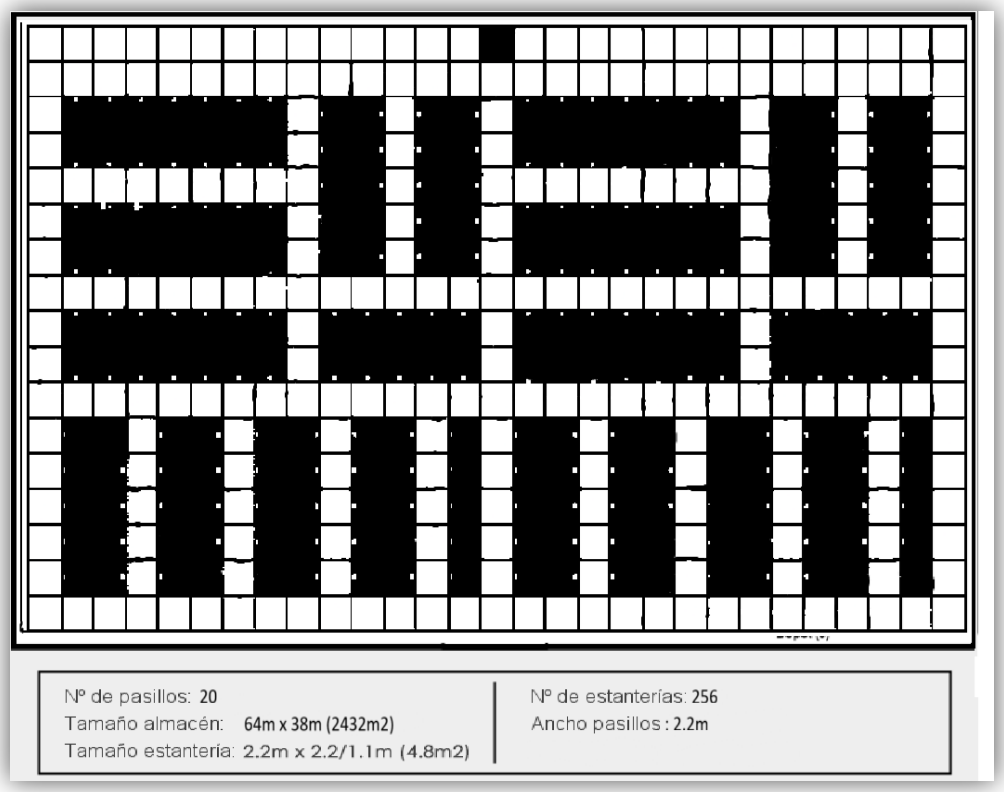

Figura 17: Almacén del artículo científico [1] reducido para pruebas

Además de los almacenes simulados, tenemos almacenes basados en entornos reales permitiendo el análisis de estos almacenes siendo imprescindible para nuestro proyecto. En este punto encontramos complejidades y diversidades significativamente mayores. En estos diseños, no solo se tiene una mayor variedad sino que también una mayor cantidad de espacio disponible abriendo un amplio abanico de posibilidades para la optimización.

Hemos observado y trabajado con diseños ya conocidos y probados en la literatura como puede ser el Chevron y el Fishbone, pero aplicados a escalas muchos mayores si lo comparamos con sus contrapartes simuladas. Además de estas colocaciones, encontramos y analizamos la eficiencia de otras como el circular, el radial y el de parrilla. Cada diseño individual presenta sus propias ventajas y complicaciones en cuanto a la accesibilidad a las estanterías, flujo de los trabajadores por los pasillos y utilización del espacio.

Otro punto destacable de estos entornos es la gran variabilidad en los tamaños de los almacenes, tamaños de las estanterías y cantidad de productos que puede albergar un almacén. Para ilustrar esta variabilidad, hemos generado almacenes que contienen desde 244, 248, 250, 256, 384, 386, 388, 394, 396, 400, 402, 414, 902, 416, 912, 918, 928, 952, hasta 1250, 1258, 1272, 1276 y 1280 artículos o espacios.

En esta diversidad de configuraciones y escalas es lo que nos permite desarrollar y aplicar soluciones en múltiples entornos. Con un espectro tan amplio de geometrías y capacidades, podemos realizar un análisis exhaustivo, con el objetivo de obtener una descripción detallada del rendimiento de cada algoritmo frente a los problemas específicos presentados por cada tipo de geometría en estos almacenes. Este enfoque nos permite identificar las mejores soluciones para maximizar la eficacia y la productividad en entornos irregulares a nivel general sin importar su diseño específico.

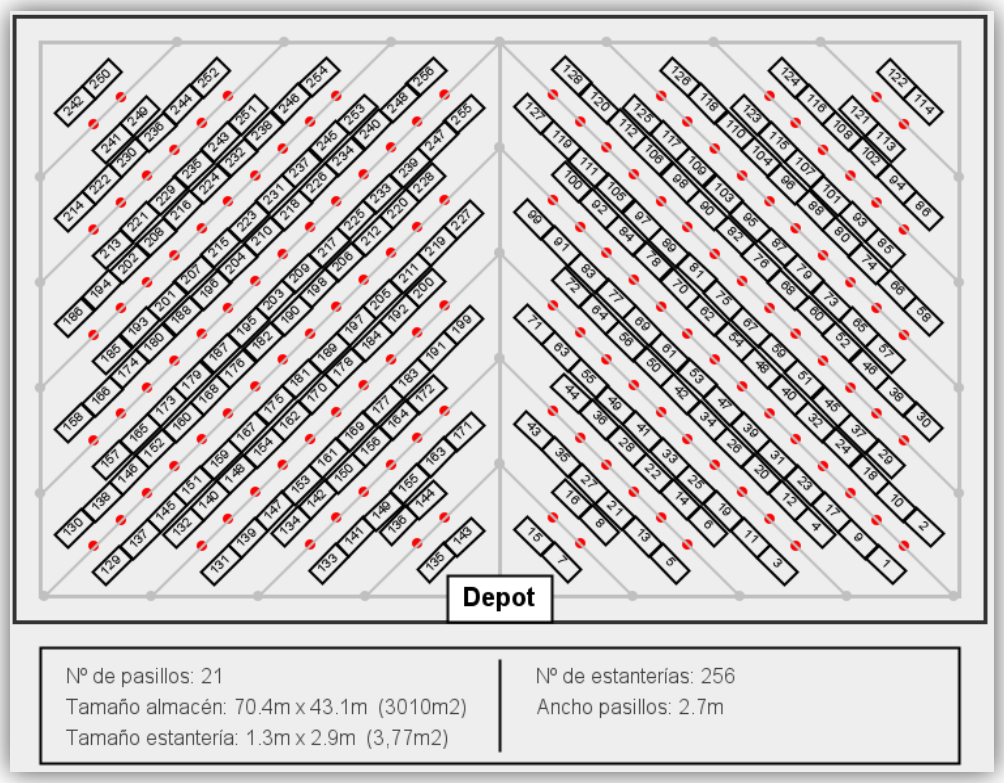

Figura 18: Almacén Irregular con distribución Chevron

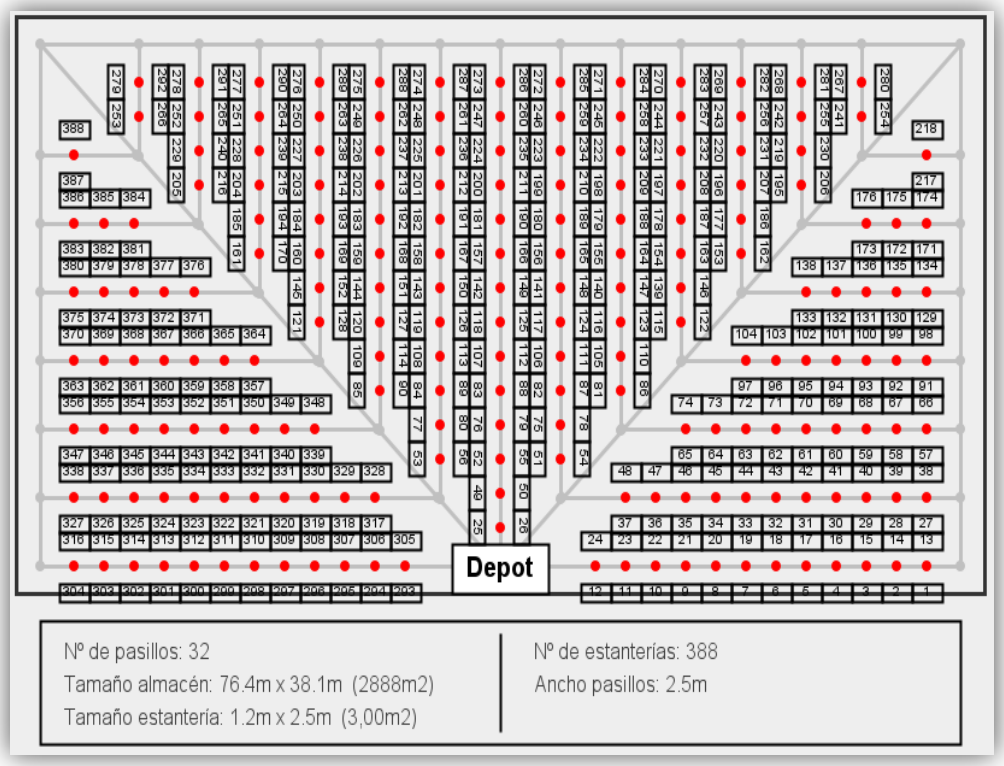

Figura 19: Almacén Irregular con distribución Fishbone

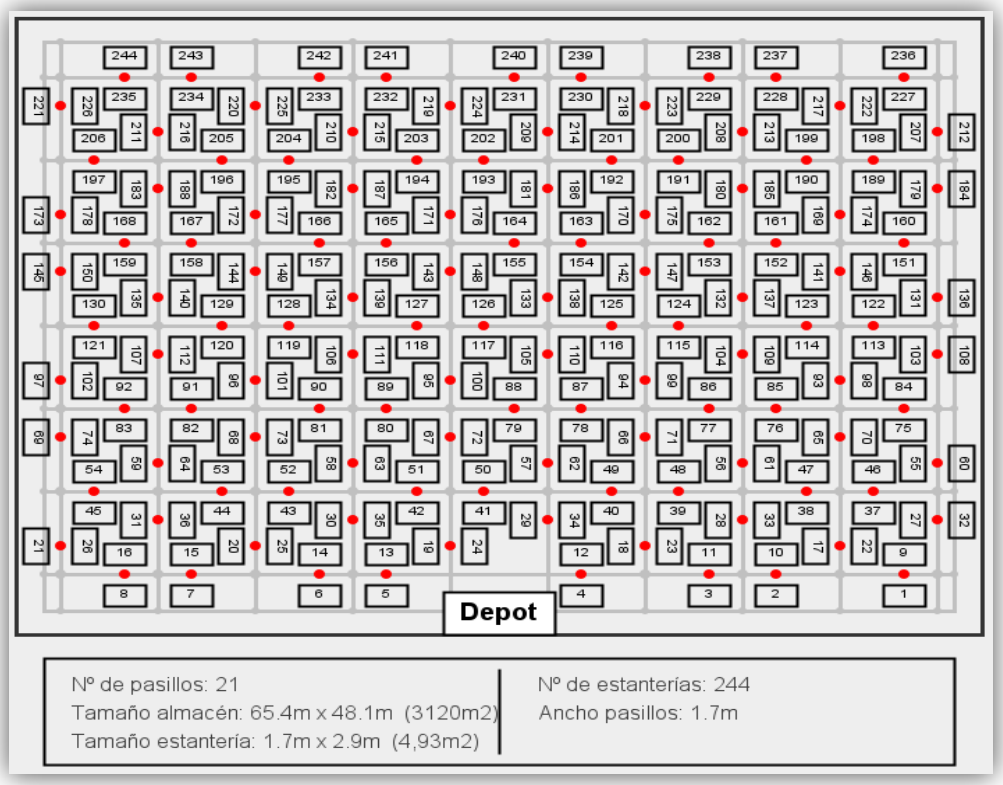

Figura 20: Almacén en disposición de red

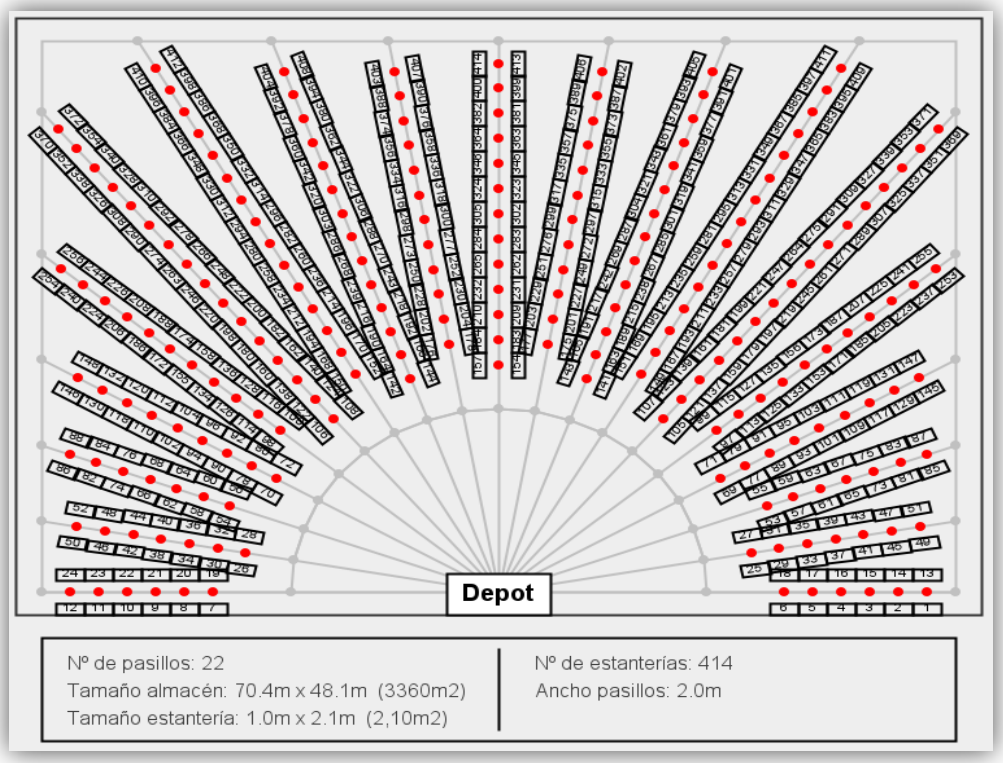

Figura 21: Almacén en disposición radial

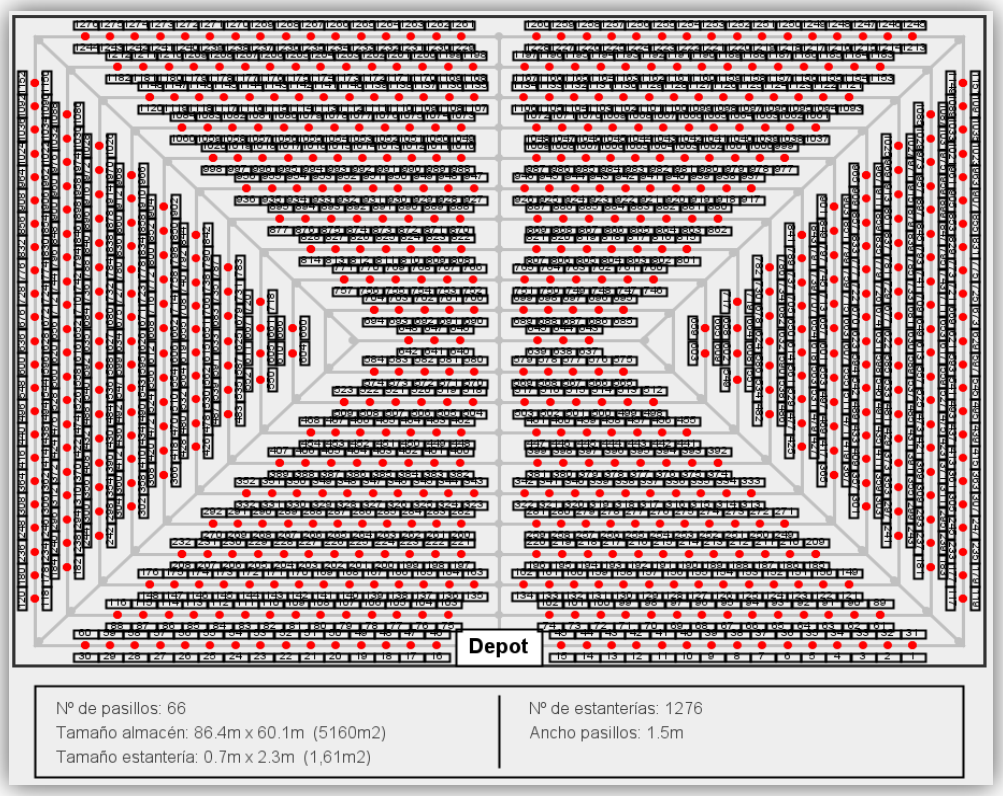

Figura 22: Almacén en disposición circular

Las instancias de pedidos presentan una característica especial, pues como se puede ver en los almacenes algunos de ellos caben apenas 50 artículos mientras que en otros 1200, pues en lugar de generar una gran cantidad de pedidos específicos para cada almacén se opta por usar pedidos generales que se convierten según el almacén que se van a utilizar. Esto ahorra tanto almacenamiento como tiempo de cómputo al evitar tener que cargar variables y leer ficheros pues ya las variables se encuentran cargadas y se cambian mediante operaciones más sencillas. Para generar los pedidos se generan de forma normal pero para almacenes muy grandes mucho mayores que los que tenemos, al leer el artículo que se quiere se le hace el módulo según la cantidad de artículos que se almacenan en la instancia del almacén leída. De esta forma sin importar el tamaño del almacén siempre que su tamaño sea menos que el de los pedidos generados se podrá llegar a cualquier artículo del inventario.

Estructura de la lista de Pedidos

Teniendo siempre N un número siempre mayor a 1 y nunca mayor que 8, por otro lado M puede ser muy variables con valores entre 30, 40, 50, 60, 80, 90, 100, 150,200 y 250.

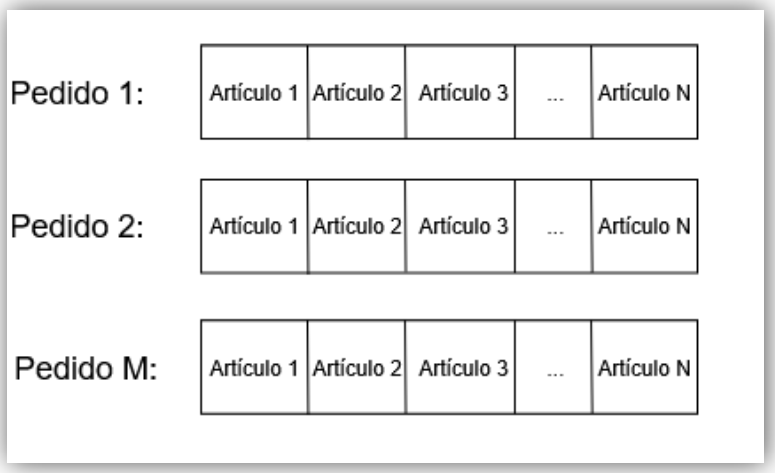

Figura 23: Ejemplo de Lista de pedidos adaptado al almacén de la figura 9.

Por ejemplo tomando los pedidos del archivo Pedido_30_8.json nos queda:

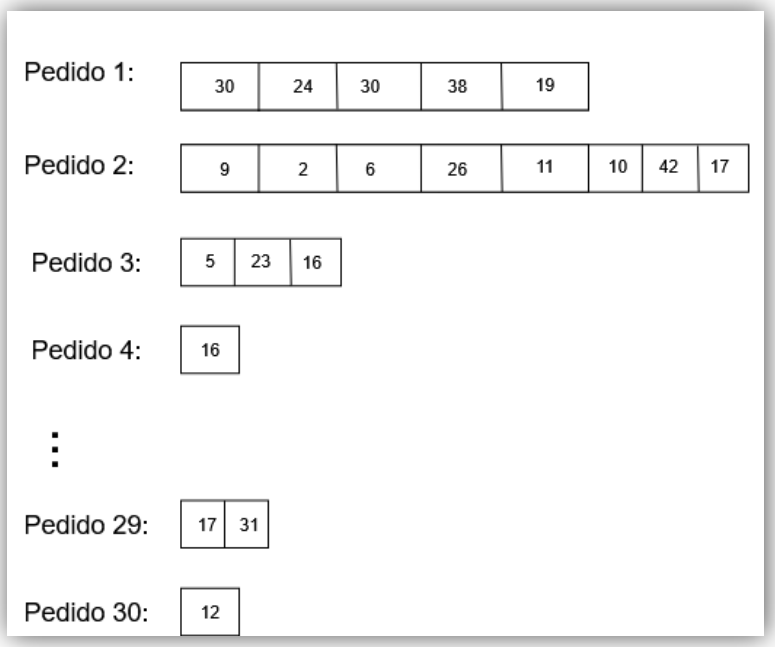

Figura 24: Lista de pedidos del archivo Pedido_30_8.json.

La necesidad de ver de forma visual la topografía del almacén y como se determina una ruta teniendo en cuenta el pedido y el algoritmo en cuestión lleva a la creación de un visualizador de instancias. Este visualizador se encarga de mostrarnos la disposición de los nodos del grafo, con nodos rojos que indican nodos que no contienen ningún artículo de la lista de pedidos a diferencia de los nodos verdes que indican que desde ese nodo se recoge algún artículo. Los caminos cambian de color para indicar que la ruta pasa por esas uniones. En la parte superior del visor cuenta con dos menús desplegables uno para seleccionar cuál de los cuatro algoritmos implementados se desea representar el camino y el otro menú del que se dispone es para seleccionar de la lista de pedidos seleccionada si se desea ver la ruta de recogida de toda la lista o de algún pedido en específico.

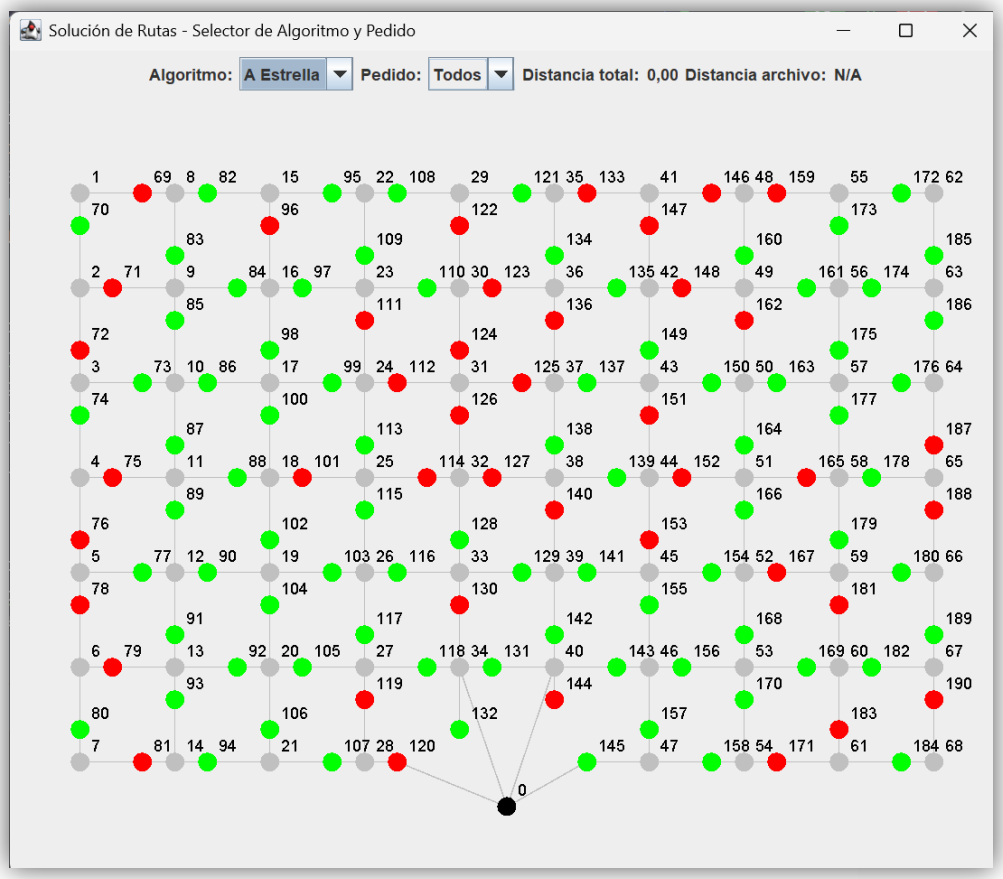

Figura 25: Visualizador de instancias.

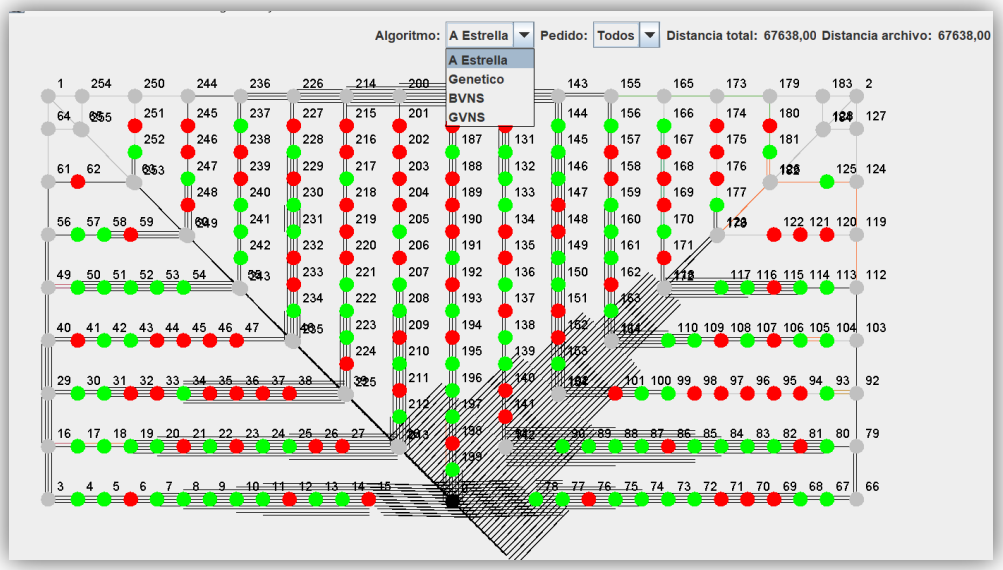

Figura 26: Selector de algoritmos del visor.

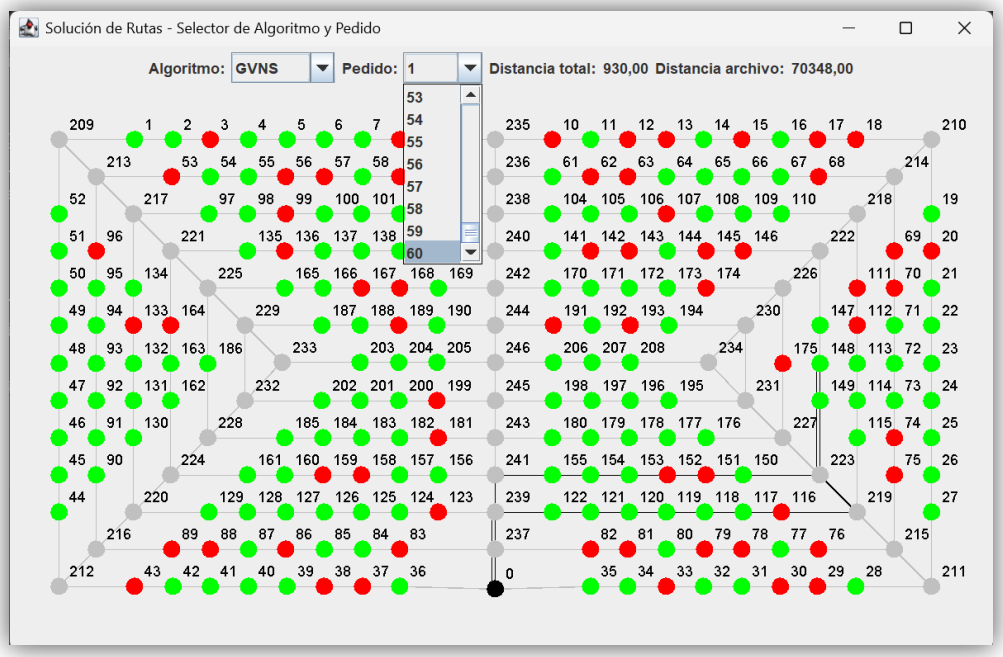

Figura 27: Selector de pedido del visor

7. Experimentación

Para poder cumplir el objetivo final de este documento se realizaron una serie de pruebas y experimentos para permitir evaluar el desempeño de los algoritmos al resolver instancias del Problema de Orden de Recogida en almacenes irregulares. La función de estas pruebas no es únicamente comparar la calidad de sus soluciones, sino que también que sirve como pruebas de caja blanca para evaluar que la implementación de dichos algoritmos ha sido de forma adecuada.

Para ponerlos a prueba se utilizan una variedad tanto de mapas de almacenes irregulares, como de pedidos. Cada instancia es diferente entre sí, pues a pesar de usar dos listas de pedidos con la misma cantidad de pedidos como los artículos a recoger son distintos se genera que los caminos sean distintos y consecuentemente la distancia total a recorrer también lo sea.

Para definir los almacenes usamos un array de listas enlazadas, cada índice del array representa el nodo actual en el que nos encontramos del mapa y la lista que contiene esta posición del array contiene los nodos vecinos y el coste en distancia que conlleva movernos a este nodo. También necesitamos poder saber qué artículo se recoge desde cada nodo o incluso en algunos casos si no se recoge ningún artículo, estos datos se guardan en una estructura similar pero en este caso se guarda en la lista el valor característico de cada artículo correspondiéndose con un valor entero positivo y mayor que 0. Los mapas tienen tamaños muy variables, algunos de 50, 60, 161, 234, 256, 384, 400, 952 y hasta de 1280 artículos, en cuanto a la forma de los almacenes tenemos algunos con distribución chevron, fishbone, radial, circular, grid y otros con estructuras no definidas, cada disposición con varios tamaños. Los pedidos son almacenados en una lista enlazada de arrays. Cada lista de pedidos está conformada por una cantidad igual de variable de pedidos, encontrando tamaños de 30, 40, 50, 60, 80, 90, 100, 150, 200 o 250, y cada pedido individual contienen una cantidad variable de artículos diferentes. Al leer los pedidos se deberá traducir desde el indicador ser artículo al nodo del mapa donde se puede recoger dicho artículo.

Para la implementación tanto de las instancias anteriormente mencionadas como de los pedidos y los algoritmos se implementan usando el lenguaje de programación JAVA, para aprovechar que es un lenguaje orientado a objetos, la gestión de la memoria mediante el recolector de basura facilitando las tareas de implementación y comparación de los resultados.

Para este experimento utilizamos un ordenador portátil con 8GB de RAM, un procesador Intel(R) Core™ i7-1065G7 CPU con 1.30GHz de frecuencia de reloj y con una tarjeta gráfica integrada Intel® Iris® Plus Graphics de 128 MB. Además este equipo tiene una licencia de Windows 11 Home esto hace que se destinen recursos del dispositivo para mantener el sistema operativo y el programa en cuestión no posea la totalidad de los recursos.

El objetivo de estos algoritmos es encontrar el camino donde se recogen todos los pedidos recorriendo la menor distancia en el menor tiempo posible. Sacando estos valores de los escenarios de pruebas siendo definido cada entorno como la combinación de un mapa y una única lista de pedidos. A partir de cada instancia se obtienen las distancias y el tiempo empleado en cada algoritmo y con estos datos se saca el la distancia mínima entre las soluciones de los algoritmos implementados y la desviación local. A partir de aquí podemos determinar los promedios de tiempos de ejecuciones, de distancias, mínimos y mínimos locales.

Inicialmente compararemos los primeros algoritmos basados en heurísticas implementados anteriormente con el fin de obtener cual desarrolla las mejores soluciones y para posteriormente utilizarlo para que sea comparado con el algoritmo exacto para determinar las diferencias en cuanto al tiempo de cómputo empleado por cada uno pues el exacto siempre devuelve la solución óptima haciendo que el resto de implementaciones devuelvan soluciones iguales o aproximadas pero nunca mejores que las devueltas por este algoritmo.

Tras refinar y corregir los comportamientos no esperados o erróneos se obtienen las siguientes tendencias al analizar los datos finales. El algoritmo A* es un algoritmo es determinista por lo que por una misma entrada devuelve siempre la misma salida y con estos datos podemos apreciar que el algoritmo más rápido, incluso llegando a indicar que ha tardado 0 ms, esto se debe a que los almacenes son tan pequeños que es capaz de obtener las rutas en un tiempo muy pequeño. Esta velocidad trae como consecuencia que la solución que entrega se aleja del óptimo y se alejan cada vez más con diseños más complejos. Por contraparte el algoritmo genético es estocástico por lo que quiere decir que las soluciones que devuelve pueden variar para una misma instancia, en ocasiones mejoran las soluciones entregadas por A* y su calidad depende del número de generaciones y de las poblaciones. El BVNS el primero de aquellos basados en vecindarios presenta una mayor estabilidad, utilizando cantidades de tiempo similares al anterior mejora las rutas en almacenes tanto pequeños como medianos. Siendo un buen punto medio entre estabilidad y tiempo de cómputo a pesar de tener soluciones ligeramente peores que su derivado GVNS. El que mejores resultados ofrece es el GVNS además de que es de los más estables, mejorando al BVNS gracias a que cuenta con múltiples etapas de shaking o agitación de la solución para intentar escapar de los mínimos locales. Para poder establecer una igualdad de condiciones se le permiten tiempos similares de ejecución al BVNS, GVNS y el genético.

Después de utilizan instancias de prueba y realizar las pruebas pertinentes de comprobar que los algoritmos funcionan como deberían, podemos pasar a obtener los resultados de utilizar instancias ya probadas en la literatura. Entre estas instancias ya investigadas tenemos las instancias de Albareda y de Henn llegando a tener 25 almacenes con tamaños, diferenciados en cinco conjuntos con tamaños similares. En cuanto a las lista de pedidos tenemos una total de 144 listas cada una asociada a uno de los cinco grupos de tamaño relacionado a los almacenes dando un total de 720

entornos como resultado de combinar un almacén con una lista en donde ambos pertenecen al mismo rango de tamaños. Con estos datos obtenidos de lanzar la simulación de estas instancias se llega a la conclusión de que para la mayoría de las instancias el camino que nos brinda el algoritmo GVNS es el mejor, manteniendo la calidad de las soluciones con el consecuente aumento del tiempo de utilización de la CPU. En cuanto a los algoritmos genéticos y los basados en vecindades como el GVNS y el BVNS se ejecutan durante cantidades de tiempo similares para poder compararlos de forma más justa pues cuanto más tiempo se le facilite en la ejecución mejores resultados tienen a dar pero sin ser mejores que los óptimos entregados por algoritmos exactos.

Con estas instancias basadas en entornos reales se evalúa como los algoritmos utilizados escalan los resultados con el aumento de la complejidad, el tamaño del almacén o el aumento del tamaño de la lista de pedidos. El A* demostró ser el más rápido incluso para almacenes complejos o con una alta cantidad de pedidos a recoger pero con la consecuente pérdida de calidad en las soluciones pues entrega los peores resultados. Por otro lado el algoritmo genético y el BVNS se ejecutan durante tiempos similares tanto entre sí como en comparación con el algoritmo GVNS, a pesar de que en la gran mayoría de los escenarios como podemos ver en la Tabla 2, en algunas escasas instancias el genético suele tener mejores resultados y en otros algo más frecuentes el BVNS ser el que más se acerca al óptimo. Mediante los datos de la tabla podemos determinar que los valores medios del GVNS son un 40,84% mejores que los que da el A* sin embargo solo un 9,95% mejores que la media obtenida por el genético y un 1,37% mejores si se toma como referencia el BVNS.

	A Estella	Genético	BVNS	GVNS
Desv.(%)	64,55%	8,68%	1,28%	0,12%
AVG(s)	706,87	464,40	423,97	418,17
CPUTime(s)	0,1	135,0	137,4	146,2
#Best	0	5	106	609

Tabla 2: Resultados obtenidos por los algoritmos (usando las instancias de Albareda y Henn)

Los métodos exactos dan la mejor solución la cual nunca podrá ser superada por nuestros algoritmos implementados. Para obtener estos modelos exactos se usó el modelo exacto de Gubori dando los óptimos resultados para cada instancia puesta a prueba. Con las comparaciones de los algoritmos anteriores podemos determinar que el GVNS es aquel que mejores comportamientos ha tenido gracias a la estabilidad que ha presentado entre tiempo de uso de la CPU y longitud de la ruta. Mediante esta comparación y los datos obtenidos por ambos algoritmos nos permiten llegar a ciertas conclusiones. Podemos determinar que en ninguno de los escenarios el

GVNS llega a dar la solución óptima sino que siempre da rutas más largas que el exacto, dando este rutas en promedio un 69,45% mejores y requiriendo un 65,39% más de tiempo.

	GVNS	Gubori
Dev.(%)	11,02%	0,00%
AVG(s)	470416	143689
CPUTime(s)	143,6	415,2
#Best	0	400

Tabla 3: Resultados de comparar el GVNS y el exacto de Gubori (usando las instancias de Albareda)

A continuación se han extraído de las estadísticas obtenidas de la simulación con el fin de mostrar la evolución del tamaño de los caminos con las variaciones de complejidad impuestas por la disposición del almacén, la cantidad de pedidos que caben en el mismo y el tamaño de la lista de pedidos a recoger siendo estos los factores más influyentes en la distancia de la ruta. En el eje horizontal tenemos la cantidad de productos que contiene el alancen y para cada tamaño se tienen cuatro listas de pedidos con diferentes tamaños como 100, 150, 200, 250 y 50 pedidos. Las listas se van utilizando respetando la secuencia anterior. Por otro lado en la barra vertical tenemos una representación de la distancia dada por los algoritmos.

Soluciones para almacenes Chevron

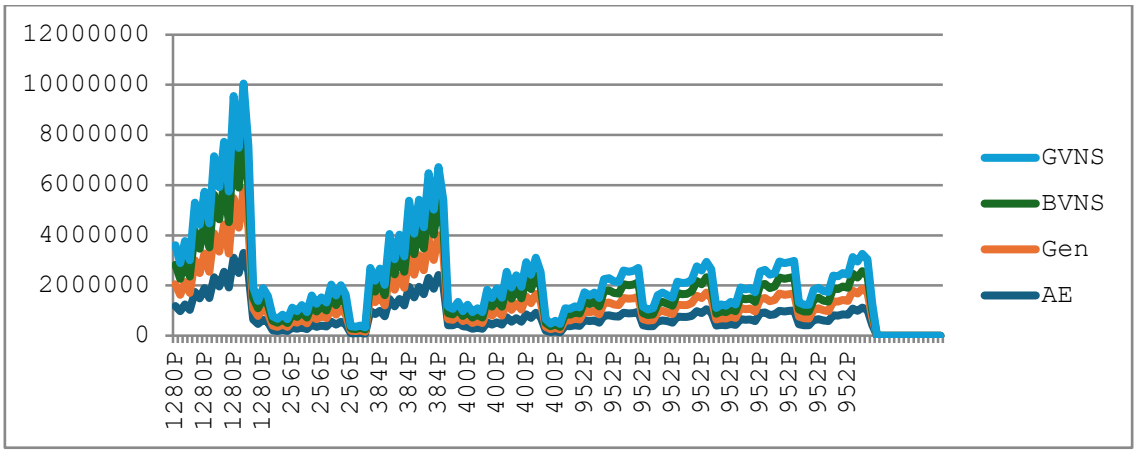

Figura 28: Tiempo de recogida en milisegundos de los almacenes chevron

Tiempo de recogida en milisegundos de los almacenes Fishbone

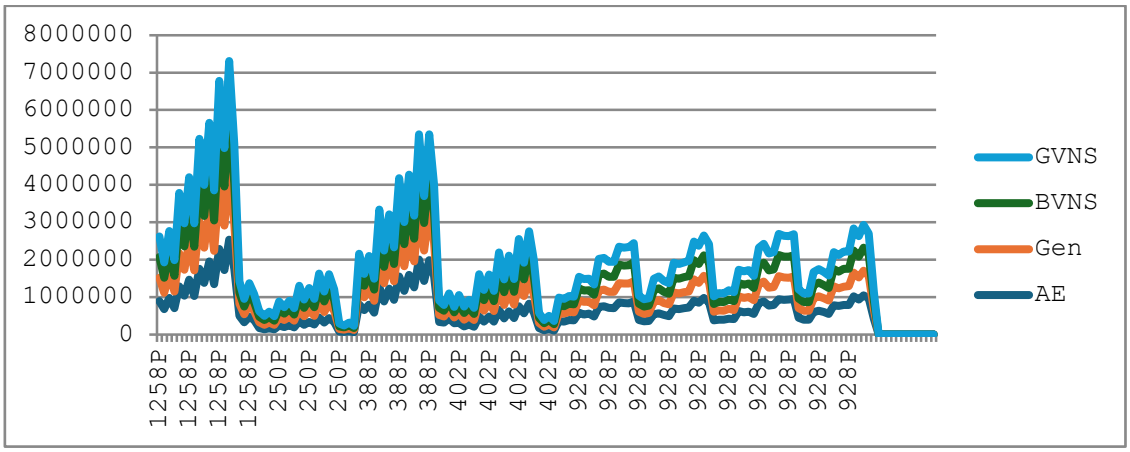

Figura 29: Tiempo de recogida en milisegundos de los almacenes Fishbone

Analizando las gráficas obtenidas y como se observa en las imágenes anteriores podemos llegar a detectar patrones que se repiten en cada almacén, la lista de pedidos que tienen la misma cantidad de pedidos se mantienen acotados en unos rangos, el aumento de la cantidad de pedidos genera un pico en la distancia acumulada. Para dos almacenes con la misma distribución y distintos tamaños aquellos que son más grandes generan las mismas variaciones pero desplazando las funciones de forma que las rutas tienen más distancia para los mismos pedidos. Un ejemplo de esto es que para dos distribuciones de almacenes diferentes como los ya presentados como el chevrom y el fishbone con una cantidad de pedidos similares como podemos ver 250 y 256 y un mismo pedido, la variación de complejidad estructural del almacén causa variaciones en la solución.

8. Análisis estadístico

Tras obtener los valores que se pretenden optimizar y apreciar a rasgos generales como se comportan y como tienden a desarrollarse las soluciones en cada instancia, podemos hacer algunos test estadísticos que nos indiquen de forma más detallada además de obtener combativas más fiables que la simple observación. Para esto utilizaremos dos pruebas no paramétricas, siendo la primera la prueba de Friedman el cual es utilizado para analizar los datos de medias repetidas principalmente cuando no se cumplen los principios de normalidad y homogeneidad en las varianzas. La normalidad se determina que no se cumple con ver las gráficas estadísticas obtenidas podemos comprobar que no se corresponde con la distribución de una función normal y la homogeneidad de las varianzas se demuestra que no se cumple pues las varianzas entre los diferentes grupos de una población que se comparan no son iguales pues cada instancia es independiente al igual que cada algoritmo utilizado sobre esta instancia. La segunda prueba no para métrica es la prueba de Wilcoxon utilizada para comparar el rango medio de dos poblaciones determinadas y determina si existen diferencias entre ellas refiriéndonos a una diferencia estadísticamente significativa. Para facilitar la realización de estos test hacemos uso de la herramienta IBM SPSS Statistics pues esta aplicación ya contiene estas pruebas ya implementadas.

Con las pruebas de Friedman sobre las soluciones obtenidas por los algoritmos tenemos un p-valor o valor de significancia asintótica menor a 0,01 evidenciando que no se cumple la condición nula significando que las diferencias entre las poblaciones no son causadas al azar. Se utiliza como punto de referencia para el p-valor 0.01 pues si es menor a este indica que existe menos del 1% de probabilidades de obtener los valores si se cumple la hipótesis nula. A pesar de esto esta prueba solo nos indica que existen diferencias entre las soluciones de los algoritmos pero no entre cuales algoritmos, debido a esto se deben incorporar pruebas complementarias como la ya mencionada prueba de Wilcoxon.

Al igual que en Friedman la prueba de Wilcoxon requiere que no sigan distribuciones normales y la homogeneidad en las varianzas. Ejecutando y comparando cada algoritmo utilizado vamos obtenido si se cumple o no la condición nula. Quedando una significancia asintótica para cada algoritmo menor a 0.01 demostrando que cada conjunto de soluciones es estadísticamente diferente entre sí.

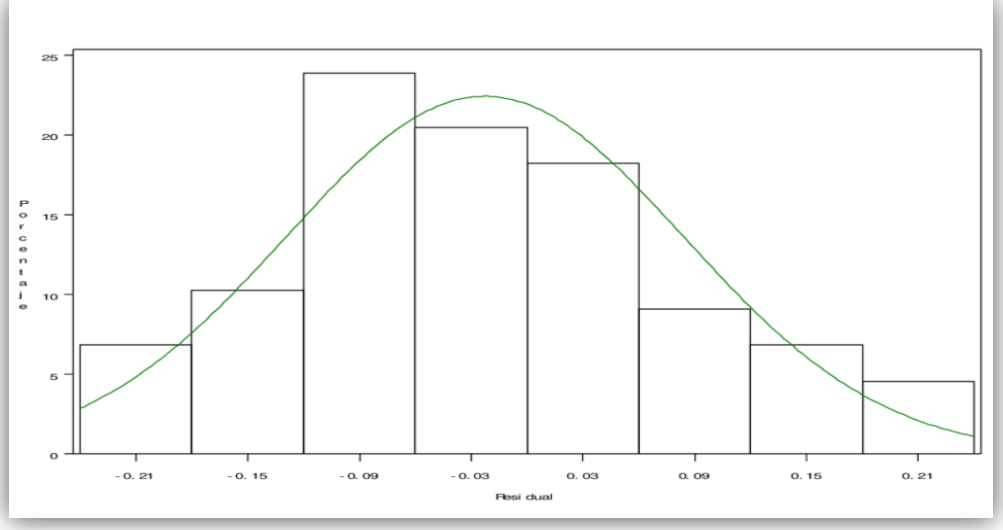

Figura 30: Grafica de la función de normalidad [21]

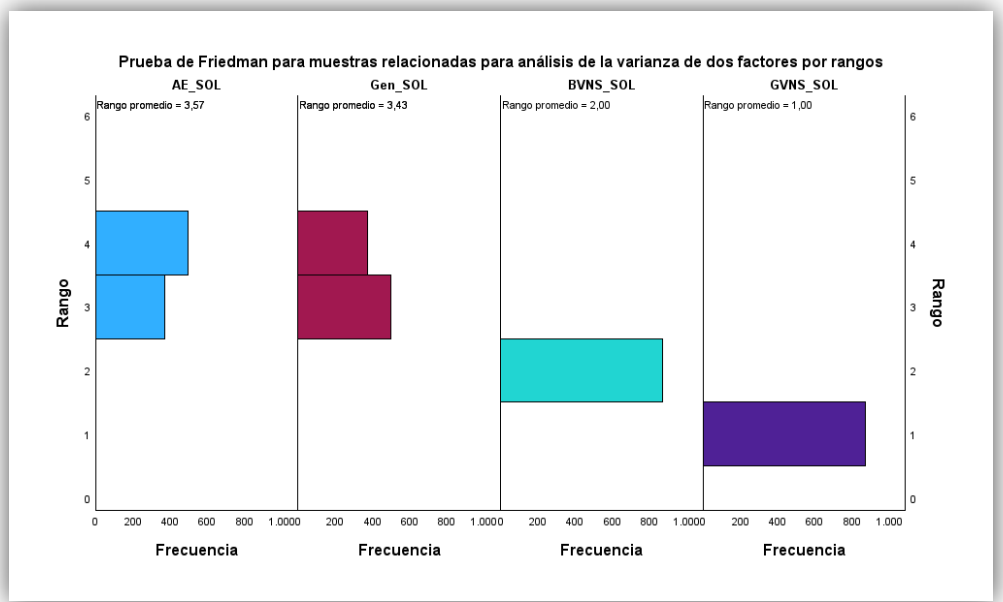

Figura 31: Prueba de Friedman para la varianza

Ejecutando las pruebas de frieman para los algoritmos implementados se obtienen valores por rangos promedios para el algoritmo A estrella de 3,57 siendo el que peor desempeño plantea, el genético un valor muy aprocimado de 3,43 y por parte de los basados en vecindad el BVNS presenta un rango promedio de 2,00 y el VNS valores de 1,00 y por lo tanto el de mejor desempeño. El BVNS presenta el desempeño más equilibrado y siendo superior a los genéticos y heurísticos como el A*. Con estos datos podemos llegar a concluir que este algoritmo es el

que mejor se adapta a este contexto y demuestra una ventaja significativa frente al resto de alternativas son aquellas estrategias de búsqueda local variable.

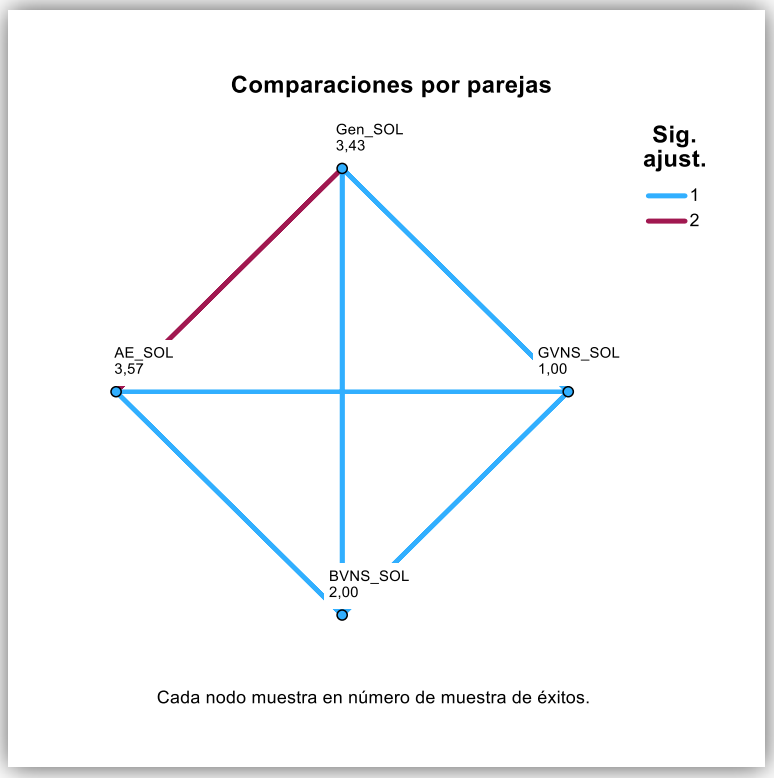

Figura 32: Comparación por parejas de las pruebas de Wilcoxon

Mediante las pruebas de Wilcoxon consiste en comparaciones pareadas entre algoritmos tras obtener los valores medios de la prueba de Frieman pues estos valores serán importantes para esta comparación y nos lleva a la conclusión de que el algoritmo GVNS parece ser el mejor algoritmo, pero sin p-valores o tamaños del efecto, no puede concluirse rigurosamente. Los datos disponibles solo permiten hipótesis cualitativas:

9. Planificación y presupuesto del proyecto

El hecho de determinar de forma exacta el tiempo total empleado para culminar este proyecto conlleva una complejidad muy elevada pues la velocidad en que avanza es inherente a cada individuo además de que las los inconvenientes que surgen durante el desarrollo provocan retrasos inesperados en la planificación inicial.

Para cumplir el objetivo de este proyecto y determinar rutas de recogida óptimas no es suficiente tomar una única perspectiva como la mejor por lo que se usan 4 algoritmos diferentes para determinar que aproximación ofrece mejores resultados. Simular estos algoritmos para un escenario nos daría una solución a un problema específico por lo que tendremos que simular una variedad de almacenes con formas y tamaños diferentes y listas de pedidos con cantidades variables de artículos, para permitirnos tener una visión general de las soluciones. Se documentan estos resultados y se validan generando estadísticas.

Etapa/Semana	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Planificación y Comprobación																		
Desarrollo del primer entorno																		
Búsqueda clásica																		
Algoritmo genético																		
BVNS y GVNS																		
Instancias																		
Visualización																		
Documentación																		
Validación																		

Tabla 4: Diagrama de Gannt con los tiempos estimados

Etapa/Semana	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Planificación y Comprobación																				
Desarrollo del primer entorno																				
Búsqueda clásica																				
Algoritmo genético																				
BVNS y GVNS																				
Instancias																				
Visualización																				
Documentación																				
Validación																				

Tabla 5: Diagrama de Gannt con los tiempos reales

El desarrollo de este proyecto conlleva un desembolso monetario para poder pagar una determinada serie de estructuras y servicios que se ajustan tanto al presupuesto de la empresa que desempeña el proyecto como a los gastos concretos de su realización. Se planifica desempeñar el trabajo destinado un total de 35 horas semanales durante aproximadamente 20 semanas. Los gastos lo podemos dividir en gastos directos siendo aquellos que son ocasionados de forma directa por el proyecto entre los cuales encontramos los salarios destinados para los empleados siendo solo uno para nuestro caso específico. El salario de un desarrollador en el lenguaje de programación JAVA es de [9] 13.33 **€,** si durante el proyecto se destinan 700 horas quedando un salario de 8397,9 € de salario neto para nuestro trabajador. Con este salario neto para el trabajador tenemos que agregarle la cotización a la seguridad social por el empleado, siendo un porcentaje variable a la fiscalidad de cada País. En el caso específico de España esta cotización es del 28,30% en contingencias comunes y siendo un 23,60% por parte del empleador y 4,7 por parte del empleado, esto trae como consecuencia que se tengan que destinar otros 1981,9 € en cotizaciones a la seguridad social.

La realización total de este proyecto se puede dividir en etapas o módulos y para cada una se estima una cantidad de tiempo. Inicialmente tenemos que conocer acerca del estado de la literatura de los almacenes irregulares y los modelos utilizados actualmente para obtener rutas estimando 70 horas para desarrollar esta labor, de forma que nos permita desarrollar unas bases sólidas. Teniendo unas bases se puede

empezar con las primeras etapas del código creando los primeros almacenes, las listas de pedidos de prueba, las interfaces de algoritmos de búsqueda para poder utilizar diferentes algoritmos de forma que al agregar nuevos no influyan en la ejecución y utilizando principios SOLID. Teniendo los primeros entornos se puede empezar con los diferentes algoritmos de búsqueda clásica más específicamente el A estrella, necesitando 45 horas para preparar este algoritmo y la primera integración y los flujos entre las instancias y las búsquedas son de la forma que se desea. Para el siguiente algoritmo a implementar siendo el algoritmo genético necesitando 45 horas. Para finalizar los métodos de búsqueda tenemos aquellos basados en vecindad y que partes de la implementación se pueden reutilizar de algoritmos previos requeriremos aproximadamente 20 horas de trabajo. Teniendo el estilo e instancias iniciales para validar el funcionamiento correcto es necesario crear un conjunto de instancias más amplio y con una mayor variedad, llegando a parecerse a aquellos que podemos encontrar en entornos reales requiriendo para esta etapa 87 horas. Llegando a las fases finales del desarrollo surge la necesidad de no solo obtener una estadística de la calidad de cada ejecución sino que también poder visualizar el almacén que cuestión, en donde se encuentran los artículos que se necesita recoger dentro de ese almacén, el camino que devuelve cada algoritmo además de la distancia de dicho camino, requiriendo 24 horas. Durante todas las fases de implementación es conveniente destinar el tiempo pertinente para realizar las pruebas de validación e integración pertinentes para cada módulo planeando destinar 175 horas. Por último la confección de esta memoria se plante disponer de 94 horas.

En relación con las infraestructuras la empresa cuanta con el alquiler de una oficina de forma mensual de 1100 € al mes [19] por lo que el mantenimiento del inmobiliario no corre por cuenta de la empresa. En cuanto a los materiales y equipamientos se cuenta con un servidor alojado en las oficinas el cual conlleva un desembolso inicial de 800 € y un ordenador portátil para cada usuario, con un valor de 379 € [20]. Estos dispositivos son los necesarios para desarrollar todo el proyecto de forma íntegra.

En cuanto a los costes indirectos son aquellos que no son atribuidos de forma directa al proyecto pero influyen en el funcionamiento general del desarrollo del proyecto. Entre estos gastos tenemos el personal de administración y soporte que son aquellos que gestionan los recursos humanos y materiales que recibe un proyecto y tienen un salario de 13,2 € por cada hora y por la duración del proyecto siendo 8316 € de salario y 1962,576 € en cotizaciones a la seguridad social por cada empleado destinado a la gestión. En cuanto a los costos energéticos son variables a las etapas del año debido a que en el verano se hace más uso del aire acondicionado y en invierno se hace más uso de la calefacción pero oscilando entre 180 y 250 € en gastos de energía mensuales. Para tener acceso a internet y fibra se tiene un contrato con una empresa de telecomunicaciones destinando 50 € mensuales.

Para prever los problemas que puedan surgir de forma inesperada provoca que se destine 825,44 € coincidiendo con el 5% de los gastos directos y tomando como margen de ganancia empresarial del 5% nos quedan 1395,6408 €.

En cuanto a los costes por herramientas utilizadas viene en correlación con lo que se determine el encargado, pues existen opciones que no requieren un gran desembolso. Por ejemplo para las licencias de IDE muchas de ellas son licencias comerciales pero tenemos opciones gratuitas como IntelIJ IDEA Community Edition. Por otro lado no es imprescindible poseer servidores dedicados, con una gran capacidad ni de almacenamiento de ni procesamiento computacional pues con un ordenador portátil se puede realizar la labor sin agregar una complejidad adicional. En contraste a esto se ha utilizado la herramienta IBM SPSS Statistics la cual es una aplicación de licencia privada la cual tiene un coste de 87,67 €.

Categoría	Concepto	Tiempo destinado	Coste (€)
Gastos directos	Salario desarrollador (13,33 €/h)	700 h	8.397,90
	Seguridad Social (28,3% sobre salario)	—	1.981,90
Infraestructura y equipos	Oficina (1.100 €/mes × 5 meses)	5 meses	5.500,00
	Servidor local	—	800,00
	Portátil	—	379,00
Herramientas software	Licencia de la herramienta IBM SPSS Statistics	—	87,67
Gastos indirectos	Administración y soporte (13,2 €/h)	631 h	8.316,00
	Seguridad Social administración (23,6%)	—	1.962,58
	Energía (215 €/mes en promedio)	5 meses	1.075,00
	Internet y fibra (50 €/mes)	5 meses	250,00
Contingencias	5% de los gastos directos	—	825,44
Margen empresarial	5% sobre total	—	1.395,64
TOTAL ESTIMADO			**30.970,13**

Tabla 6: Tabla de valores del presupuesto

10. Impacto social y medioambiental

Al optimizar las rutas de recogida en almacenes no solo se reducen los costos operativos al poder realizar una recogida de pedidos recorriendo una menor distancia, por lo tanto un menor tiempo, sino que también conlleva beneficios sociales y ambientales. Al minimizar la distancia total de viaje para los operadores o robots, estos algoritmos reducen el consumo de energía, especialmente en sistemas automatizados que dependen de la electricidad o el combustible. Esto contribuye directamente a reducir las emisiones de carbono y apoya la logística sostenible.

Desde una perspectiva social, la planificación inteligente del camino reduce la tensión física para los trabajadores humanos al tener que estar menos tiempo desplazándose, mejorando las condiciones ergonómicas, la satisfacción de los trabajadores y reduciendo la posibilidad de lesiones. Proporcionado caminos más cortos se obtienen menos pasos y menos esfuerzo físico tomando especial relevancia en aquellos almacenes de grandes dimensiones pues los trabajadores caminan varios kilómetros al día. Con estos cambios se trafico mejora la seguridad de los empleados debido a que evitan la exposición en zonas de de maquinarias y por lo tanto menos riesgo. Estas situaciones podrías llevar las situaciones de permitir la inclusión de personas mayores, con discapacidad leve o movilidad reducida en tareas de recogida ayudando a ampliar la diversidad en estos lugares de trabajo.

Con respecto al medio ambiente en almacenes muy grandes que hacen uso de vehículos como montacargas u otros tipos reduce los gastos de combustibles fósiles o electricidad. En los casos de uso de combustibles fósiles se disminuye las emisiones de CO2 el cual es el principal causante del efecto invernadero y en caso de reducir el uso eléctrico permite la independencia de fuentes de energía no renovables. Estos algoritmos a medida que se trabaja con almacenes más grandes aumenta el tiempo que tarda en dar una solución así como la carga computacional pero son fácilmente integrables en ordenadores convencionales a diferencia de grandes modelos de inteligencia artificial que requieren inmensas cantidades de potencia de cálculo, generando una gran dependencia eléctrica y generando mucho calor el cual es disipado con enormes cantidades de agua. En las operaciones a gran escala, incluso pequeñas mejoras en la eficiencia de los caminos pueden tener un efecto acumulativo significativo tanto sobre la sostenibilidad ambiental como sobre el bienestar de los empleados.

11. Conclusiones

Partiendo con el objetivo de este proyecto el cual es encontrar diferentes formar de resolver el orden y el camino que se debe tomar para recoger los artículos que forman un pedido en almacenes irregulares y para llegar a estos caminos se utilizan diferentes perspectivas y poder determinar cual ofrece los mejores resultados.

Entre los diferentes algoritmos empleados el cual ofrece los mejores resultados aunque con la consecuencia de emplear un mayor tiempo de cómputo es el GVNS, pero siendo una diferencia no significativa pues la diferencia de tiempos es 1300 ms aproximadamente, comparando el promedio de este algoritmo y el que utiliza la menor cantidad de tiempo. El algoritmo genético presenta una mayor flexibilidad frente a entornos complejos a diferencia del método heurístico A* pero a pesar de esto no garantiza que sus resultados sean mejores pues al tener parámetros muy agresivos con pocos ciclos la solución puede variar de forma muy brusca. Para contrastar los algoritmos BVNS y GVNS son más estables en cuanto a la calidad de la solución.

Como métricas para contabilizar la calidad de cada método buscando que el camino tenga la menor distancia y se consuma el menor tiempo posible en determinar dicho camino anteriormente mencionado. Los valores obtenidos se encuentran entre un rango de valores influyendo en el resultado principalmente el tamaño del almacén y el tamaño de los pedidos.

Los resultados obtenidos se obtienen simulando los mapas de los almacenes de forma estática pero hay una diferencia con escenarios reales donde las condiciones como el tráfico, bloqueos temporales ocurren de forma dinámica llegando a afectar la solución total. Esto nos da un punto de partida para poder expandir y continuar el desarrollo de este proyecto.

Con esto podemos llegar a concluir que para resolver el problema definido los algoritmos metaheurísticos basados en vecindarios devuelven mejores resultados con respecto al resto de algoritmos que podemos encontrar en la literatura por lo que este trabajo aporta una estrategia eficiente la cual mejora la calidad de las rutas en entornos con alta complejidad topológica como es la de los almacenes irregulares y donde otros métodos tradicionales pierden eficacia.

12. Líneas futuras

El hecho de resolver este problema de forma estática e invariante en el tiempo no se puede alejar más de la realidad pues los almacenes logísticos son entornos de continuo movimiento pues también existen variaciones en el stock, que el almacén tenga posiciones en las que no tiene ningún producto o incluso que colisionen en el mismo espacio dos operarios haciendo que las rutas que se indican requieren más tiempo o más distancia debido a estos y otros muchos factores. También seria considerable agregar productos perecederos los cuales podrán estar almacenados hasta una fecha determinada, además de agregar restricciones de peso y tamaño a los artículos que puede transportar un operario y con esta restricción pasamos a resolver un problema distinto del OPP al OBP. Por lo tanto con el fin de fortalecer la selección de rutas en tiempo real, se pretende agregar esta información como contexto a los algoritmos de búsqueda ya planteados anteriormente para que estos métodos de búsqueda se adapten a las situaciones cambiantes del ambiente. Gracias a estos parámetros las rutas se adaptan a congestiones en los pasillos, bloqueos temporales o reconfiguración en los pasillos. De la misma manera resulta interesante evaluar las técnicas para analizar entornos colaborativos donde se encuentran circulando tanto personal como sistemas autónomos. Para finalizar se planea introducir mediciones de parámetros ambientales y sociales para no solo mejorar la distancia y el tiempo en recorrer o calcular el camino sino también para contribuir a una operación logística más sostenible.

Por otra parte al ejecutar de forma síncrona los cuatro algoritmos para determinar el camino combinado con cada instancia puede llegar a consumir mucho tiempo de cómputo, por ejemplo en nuestro caso tenemos 29 mapas de almacenes, cada uno con sus formas, tamaños y cantidad de artículos combinado a que a que cada uno se puede implementar una lista de pedidos cualquiera de las 30 que contamos haciendo un total de 870 instancias únicas e independientes. Una forma relativamente sencilla de ahorrar en cuanto al tiempo de ejecución sería implementar paralelismo o pseudo-paralelismo en casos extremos donde se cuente con un procesador monohilo. Cada algoritmo combinado con una instancia forma una sección secuencial y cada sección puede ser paralelizable de tal forma que se puede evaluar una mayor cantidad de instancias en el mismo tiempo.

13. Referencias

[1] MarketsandMarkets, «A New Genetic Algorithm for Order-Picking of Irregular Warehouse 2009,» [En línea]. Available: https://ieeexplore.ieee.org/document/5200079

[2] VNS .[En línea]. Available: http://www.decom.ufop.br/prof/marcone/projects/ppm676-17/LNCS-2019-ICVNS2018-MDVRP.pdf

[3] I. C. Lago, «Solving Location Assignment and Order Picker-Routing Problems in Warehouse Management» 2023. [En línea]. Available: https://www.mdpi.com/2075-1680/12/7/711

[4] Branching Factor .[En línea]. Available: https://www.opentrain.ai/glossary/branching-factor

[5] ALGORTIMO A* .[En línea]. Available: https://nlaredo.tecnm.mx/takeyas/Apuntes/Inteligencia%20Artificial/Apuntes/tareas_alumnos/A-Star/A-Star(2005-II-A).pdf

[6] Optimization of the Storage Location Assignment and the Picker-Routing Problem by Using Mathematical Programming .[En linea]. https://www.mdpi.com/2076-3417/10/2/534

[7] Flujo de un Algoritmo Genético .[En linea]. Available: https://www.researchgate.net/figure/Figura-1-Diagrama-de-flujo-de-un-algoritmo-genetico-simple_fig1_310731033

[8] Labor diaria de los trabajadores en un almacen .[En linea]. Available: https://www.youtube.com/watch?v=0C1hHVt5pHA&t=587s

[9] Desarrollador java: salario promedio en España, 2025 .[En linea]. Available: https://es.talent.com/salary?job=desarrollador+java

[10] Stochastic models of routing strategies under the class-based storage policy in fishbone layout warehouses .[En línea]. Available: https://www.nature.com/articles/s41598-022-17240-w

[11] Imagen del modelo chevron y el fishbone .[En línea]. Available: https://www.researchgate.net/figure/The-flying-V-left-and-fishbone-layout-right_fig2_352898953

[13] Experimental analysis of manual order picking processes in a Learning Warehouse .[En línea]. Available: https://proc.logistics-journal.de/article/view/859

[14] Warehouse design and management: A literature review .[En línea]. Available: https://www.tandfonline.com/doi/full/10.1080/00207543.2017.1371856

[15] The impact of IoT on warehouse management .[En línea]. Available: https://www.mdpi.com/1424-8220/23/4/2213

[16] El papel de los algoritmos en la gestión de almacenes .[En línea]. Available: https://www.generixgroup.com/es/blog/gestion-de-almacenes-funciones-algoritmos

[17] Sobre MILP (Mixed Integer Linear Programming) de NAG .[En línea]. Available: ttps://www.addlink.es/noticias/nag/3374-sobre-milp-mixed-integer-linear-programming-de-nag

[18] The Floyd-Warshall Algorithm on Graphs with Negative Cycles .[En línea]. Available: https://www.researchgate.net/profile/StefanHougardy/publication/220110597_The_FloydWarshall_algorithm_on_graphs_with_negative_cycles/links/5a533894aca2725638c7ef03/The-Floyd-Warshall-algorithm-on-graphs-with-negative-cycles.pdf

[19] Bases y tipos de cotización .[En línea]. Avaliable: https://www.seg-social.es/wps/portal/wss/internet/Trabajadores/CotizacionRecaudacionTrabajadores/36537

[20] Oficina de alquiler en Avenida Marconi, 1, Villaverde Alto .[En línea]. Avaliable: https://www.fotocasa.es/es/alquiler/oficina/madrid-capital/villaverde-alto/183903622/d

[21] Portátil HP EliteBook .[En línea]. Avaliable: https://www.ofertaspc.com/ordenadores-portatiles-reacondicionados/402-4636-hp-elitebook-840-g7-i5-10310u-de-14-ram-16-gb-ssd-256-gb.html#/72-estado_estetico-buen_estado

[22] Grafica de la función normal .[En línea]. Avaliable: https://www.researchgate.net/figure/Figura-46-Representacion-grafica-de-normalidad-de-la-distribucion-de-la-densidad-versus_fig11_50204833

[23] Modelo exacto de Gubori .[En línea]. Avaliable: https://www.gurobi.com/

[24] Algoritmos evolutivos para optimización multiobjetivo: un estudio comparativo en un ambiente paralelo asíncrono .[En línea]. Avaliable: https://sedici.unlp.edu.ar/handle/10915/22476

[25] OPTIMIZACIÓN MEDIANTE ALGORITMOS GENÉTICOS .[En línea]. Avaliable: https://www.researchgate.net/profile/Pablo-Estevez-2/publication/228708779_Optimizacion_Mediante_Algoritmos_Geneticos/links/0912f51111f82b2a61000000/Optimizacion-Mediante-Algoritmos-Geneticos.pdf

[26] A route-selecting order batching model with the S-shape routes in a parallel-aisle order picking system. European Journal of Operational Research, .[En línea]. Avaliable: https://ideas.repec.org/a/eee/ejores/v257y2017i1p185-196.html

[27] Order-picking in a rectangular warehouse: A solvable case of the traveling salesman problem. Operations Research.[En línea]. Avaliable: https://www.researchgate.net/publication/242931999_Order-Picking_in_a_Rectangular_Warehouse_A_Solvable_Case_of_the_Traveling_Salesman_Problem

ANEXOS

Anexo 1

Código de implementación del algoritmo de Dijkstra para determinar el coste en distancia que genera el ir desde un nodo “strart” a un nodo “end”.

```
private int  getJourney(int start, int end){
    int[] distances = new int[numNodos];
    boolean[] visited = new boolean[numNodos];
    int[] prev = new int[numNodos];

    Arrays.fill(distances, Integer.MAX_VALUE);
    Arrays.fill(prev, -1); // Inicialmente no hay predecesores
    distances[start] = 0;

    PriorityQueue<int[]> pq = new
PriorityQueue<>(Comparator.comparingInt(a -> a[1]));
    pq.add(new int[]{start, 0});

    while (!pq.isEmpty()) {
        int[] current = pq.poll();
        int node = current[0], cost = current[1];
        //System.out.println("Nodo: "+node+" Distancce: "+cost);
        if (node == end){    // Ya encontramos el camino más
corto
            break;
        }

        if (visited[node]) continue;
        visited[node] = true;
        Iterador iterador = mapa[node].getIterador();
        while (iterador.hasNext()) {
            Nodo neighbor = iterador.next();
            if (!visited[neighbor.getClave()]) {
                int newCost = cost + neighbor.getDistancia();
                //System.out.println("Vecino:
"+neighbor.getClave()+"  Distancia: "+newCost);
                if (newCost < distances[neighbor.getClave()]) {
                    distances[neighbor.getClave()] = newCost;
                    prev[neighbor.getClave()] = node;
// Guardamos el nodo anterior
                    pq.add(new int[]{neighbor.getClave(),
newCost});
                }
            }
        }
    }

    // Si no se encontró camino
    if (distances[end] == Integer.MAX_VALUE) {
        return -1;
```

```
    }

    // Reconstruir el camino
    prevP.clear();
    for (int at = end; at != -1; at = prev[at]) {
        prevP.add(at);
    }
    Collections.reverse(prevP); // Invertir para obtener en
orden correcto

    return distances[end];
}
```

Anexo 2

Implementación base del algoritmo A* para determinar el camino entre un artículo y otro dentro de un determinado almacén.

```
private List<Integer> aEstrellaSimple(ListaCalificada[] grafo,int
inicio,int destino) {
    int n = grafo.length;
    PriorityQueue<node> openSet = new PriorityQueue<>();
    Map<Integer, Integer> cameFrom = new HashMap<>();
    int[] gScore = new int[n];
    Arrays.fill(gScore, Integer.MAX_VALUE);
    gScore[inicio] = 0;

    // f(n) = g(n) + h(n)
    int fScoreInicio = gScore[inicio] + distancias[inicio][destino];
    openSet.add(new node(inicio, fScoreInicio));

    while (!openSet.isEmpty()) {
        node actual = openSet.poll();
        int nodo = actual.id;

        if (nodo == destino) {
            return reconstruirCamino(cameFrom, destino);
        }

        Iterador iterador = grafo[nodo].getIterador();
        while (iterador.hasNext()) {
            Nodo vecino = iterador.next();
            int vecinoId = vecino.getClave();
            int tentativoGScore = gScore[nodo] +
vecino.getDistancia();

            if (tentativoGScore < gScore[vecinoId]) {
                cameFrom.put(vecinoId, nodo);
                gScore[vecinoId] = tentativoGScore;
                int fScore = tentativoGScore +
distancias[vecinoId][destino];
                openSet.add(new node(vecinoId, fScore));
            }
        }
    }

    return new ArrayList<>(); // No path found
}
```

Anexo 3

Implementaciones de las partes que componen un algoritmo genético

3.1 Implementación de cómo se genera la población inicial a partil de la lista de pedidos.

```
private void madePopulation(int[] list){
    System.out.println("Pedido :"+Arrays.toString(list));
    int k = list.length-1;
    int half = list.length/2;
    int tamanio = list.length;
    population = new int[4][list.length];
    for (int i = 0; i < list.length; i++){
        population[0][i] = list[i];
        population[1][i] = list[k];
        k--;
    }
    int[] firstHalf = Arrays.copyOfRange(list, 0, half);
    int[] secondHalf = Arrays.copyOfRange(list, half, tamanio);

    int[] p2 = new int[tamanio];
    System.arraycopy(secondHalf, 0, p2, 0, secondHalf.length);
    System.arraycopy(firstHalf, 0, p2, secondHalf.length,
firstHalf.length);
    int[] p3 = new int[tamanio];
    for (int i = 0; i < secondHalf.length; i++) {
        p3[i] = secondHalf[secondHalf.length - 1 - i];
    }
    for (int i = 0; i < firstHalf.length; i++) {
        p3[secondHalf.length + i] = firstHalf[firstHalf.length -
1 - i];
    }
    population[2] = p2;
    population[3] = p3;
}
```

3.2 Implementación de cómo se realizan los cambios por las mutacionesas

```
private void madeMutatation(){
    float mutation = 0.8F;
    for (int i = 0; i < population.length; i++){
        float w = r.nextFloat();
        if ( w <= mutation){
            int minshape = Math.abs(r.nextInt()) %
population[0].length;
            int maxshape = Math.abs(r.nextInt()) %
population[0].length;

            if (minshape > maxshape){
```

```
                int aux = maxshape;
                maxshape = minshape;
                minshape = aux;
            }
            int point = maxshape;
            for (int x = minshape; x <= maxshape/2; x++){
                int aux = population[i][x];
                population[i][x] = population[i][point];
                population[i][point] = aux;
                point--;
            }
        }
    }
}
```

3.3 Implementación del cruzamiento

```
private void madeCrossover(){
    if (population[0].length > 1) {
        float crossoaver = 0.8F;
        int genA = -1;
        int genB = -1;
        int aux = 0;
        int minShape = -1;
        int maxShape = -1;
        while (genB == -1) {
            if (r.nextFloat() < crossoaver) {
                if (genA == -1) {
                    genA = aux;
                } else {
                    genB = aux;
                    aux = 4;
                }
            }
            aux++;
            if (aux < 4) {
                aux = 0;
            }
        }
        do {
            minShape = Math.abs(r.nextInt()) %
population[0].length;
            maxShape = Math.abs(r.nextInt()) %
population[0].length;
        } while (minShape == maxShape);

        if (minShape > maxShape) {
            int var = maxShape;
            maxShape = minShape;
            minShape = var;
        }
        interGenchange(genA, genB, minShape, maxShape);
    }
```

```
}
private void interGenchange(int peopleA, int peopleB, int lower,
int upper){
    List<Integer> listA = new ArrayList<>();
    List<Integer> listB = new ArrayList<>();
    int indice = -1;

    for (int i = lower; i <= upper;i++){
        listA.add(population[peopleA][i]);
        listB.add(population[peopleB][i]);
    }
    HashSet<Integer> commonValues = new HashSet<>(listA);
    commonValues.retainAll(listB); // Retener solo los elementos
comunes

    // Eliminar los elementos comunes de ambas listas
    listA.removeIf(commonValues::contains);
    listB.removeIf(commonValues::contains);
    for (int k = 0; k < population[0].length; k++ ) {
        if (k >= lower && k <= upper) {
            int auxiliar = population[peopleA][k];
            population[peopleA][k] = population[peopleB][k];
            population[peopleB][k] = auxiliar;
        } else {
            if (listB.contains(population[peopleA][k])) {
                indice = listB.indexOf(population[peopleA][k]);
                population[peopleA][k] = listA.get(indice);
            }

            if (listA.contains(population[peopleB][k])) {
                indice = listA.indexOf(population[peopleB][k]);
                population[peopleB][k] = listB.get(indice);
            }
        }
    }
}
```

3.4 Implementación de cómo se obtiene la calidad de cada individuo de la población.

```
private double qualiyDigree(){
    int best = 0;
    int N = pedidos.length -1;
    if(N == 0){
        N +=1;
    }
    for (int i=0;i< population.length; i++){
        fitness[i]= 0;
        int ini = depot;
        path.clear();
        for (int j = 0;j< population[0].length; j++) {
            fitness[i] += (getJourney(ini, population[i][j]));
            path.addAll(prevP);
            if (!path.isEmpty()) {
```

```
                ini = path.removeLast();
            }
        }
        fitness[i] += getJourney(ini,depot);
        path.addAll(prevP);
        fitnessMat.add(path);
        fitness[i] = 2 - (2 * ((fitness[i]-1)/N));
        if(i != best && fitness[i]>fitness[best]){
            best = i;
        }
    }

    return fitness[best];
}
```

3.5 Implementación de los cambios para generar una nueva población a partir de la anterior

```
private void pass2NxtGen(){
    double bestfit = Double.MIN_VALUE;
    int index = 0;
    for (int j =0; j<fitness.length; j++){
        if(fitness[j] > bestfit){
            bestfit = fitness[j];
            index = j;
        }
    }
    for (int[] ints : population) {
        float pass2NxtGen = 0.9F;
        float w = r.nextFloat();
        if (w > pass2NxtGen) {
            //System.out.println("El gen "+
Arrays.toString(ints)+" se reemplaza por el de mejor Calidad");
            System.arraycopy(population[index], 0, ints, 0,
population[0].length);
        }
    }
}
```

3.6 Implementación para que el genético se ejecute durante un tiempo determinado.

```
long ini = System.currentTimeMillis();
long act = System.currentTimeMillis();
while ((act - ini)< 7){
    this.madeCrossover();
    this.madeMutatation();
    this.pass2NxtGen();
    double actQuallity = qualiyDigree();
    if (actQuallity >= prevQuality){
        mejor = true;
```

```
    }
    ciclo++;
    act = System.currentTimeMillis();
}
```

Anexo 4

Código utilizado para implementar el algoritmo BVNS

4.1 Código utilizado para la ejecución principal de este algoritmo

```
public int[] bvns(int [] solution, int kmax, long tmaxMillis) {
    long startTime = System.currentTimeMillis();
    long act = System.currentTimeMillis();
    while ((act - startTime) < tmaxMillis) {
    int k = 1;
        while (k <= kmax) {
            int [] shaken = shake(solution, k);
            int [] improved = improvement(shaken);
            k = neighborhoodChange(solution, improved, k);
        }
        act = System.currentTimeMillis();
    }
    return solution;
}
```

4.2 Implementación del método utilizado para salir de los mínimos locales.

```
private int [] shake(int[] solution,int nk) {
    if (solution.length > 1) {
        for (int i = 0; i < nk; i++) {
            int idx = aleatorio.nextInt(solution.length);
            int idxAux = aleatorio.nextInt(solution.length);
            int auxiliar = solution[idxAux];
            solution[idxAux] = solution[idx];
            solution[idx] = auxiliar;
        }
    }
    double cost = evaluate(solution);
    return solution;
}
```

4.3 Código utilizado para intentar buscar una mejora mediante un vecindario local

```
private int [] improvement(int [] solution){
    for (int i = 0; i < solution.length; i++) {
        for (int val = 0; val < solution.length; val++) {
            if (val != i) {
                int oldVal = solution[i];
                solution[i] = solution[val];
                solution[val] = oldVal;
                newCost = evaluate(solution);
```

```
                if (newCost < cost) {
                    cost = newCost;
                } else {
                    solution[val] = solution[i];
                    solution[i] = oldVal;
                }
            }
        }
    }
```

4.4 Código empleado para intercambiar las soluciones en caso de encontrar una mejor solución

```
private int neighborhoodChange(int [] current, int [] candidate,
int nk){
    if (newCost < cost) {
        current = candidate;
        cost = newCost;
        return 1; // Restart neighborhood
    } else {
        return nk + 1; // Go to next neighborhood
    }
}
```

Anexo 5

5.1 Variable utilizada para almacenar las diferentes búsquedas locales

```
private final List<Function<int[],int[]>> neighborhoods =
Arrays.asList(
        GVNS::localSearch1to1,
        GVNS::localSearchRepSP,
        GVNS::localSearchRnd
);
```

5.2 Código principal del VND

```
private int[] VND(int [] solution,int n){
    int k = 0;
    int kmax = neighborhoods.size();
    int[] best = solution;

    while (k < kmax) {
        Function<int[],int[]> Nk = neighborhoods.get(k);
        int [] newSolution = Nk.apply(best.clone());
        if (evaluate(newSolution) < evaluate(best)) {
            best = newSolution;
            k = 0; // Volver al primer vecindario
        } else {
            k++; // Probar con el siguiente vecindario
        }
    }
    return best;
}
```

5.3 Tres Búsquedas locales Implementadas:

1_ Búsqueda local con intercambios uno a uno y toma el mejor cambio

2_ Búsqueda local por reordenamiento aleatorio de subsecuencias hasta obtener la primera mejora

3_ Búsqueda local de intercambios aleatorias hasta obtener la primera mejora

```
private static int[] localSearch1to1(int [] solution) {
    double coste = evaluate(solution);
    double newCost = Double.MAX_VALUE;

    for (int i = 0; i < solution.length; i++) {
        for (int val = 0; val < solution.length; val++) {
            if (val != i) {
                int oldVal = solution[i];
                solution[i] = solution[val];
                solution[val] = oldVal;
                newCost = evaluate(solution);
```

```
                if (newCost < coste) {
                    coste = newCost;
                } else {
                    solution[val] = solution[i];
                    solution[i] = oldVal;
                }
            }
        }
    }
    return solution.clone();
}
private static int[] localSearchRepSP(int [] solution){
    int n = 0;
    int[] path;
    double coste = evaluate(solution);
    double newCoste = 0;
    do {
        path = solution;
        if (path.length < 4) {
            return solution.clone(); // No tiene sentido si hay
pocos nodos
        }

        int start = aleatorio.nextInt(path.length - 2);
        int end = start + 1 + aleatorio.nextInt(path.length -
start - 1);
        List<Integer> subPath = new ArrayList<>();
        for (int i = start; i <= end; i++) {
            subPath.add(path[i]);
        }
        Collections.shuffle(subPath);

        for (int i = start; i < end; i++) {
            path[i] = subPath.get(i - start);
        }
        newCoste = evaluate(path);
        n++;
    }while (coste < newCoste && n < solution.length);
    if (coste < newCoste){
        return solution.clone();
    }else {
        return path.clone();
    }
}

private static int[] localSearchRnd(int [] s){
    double valIni = evaluate(s);
    double valAct = Double.MAX_VALUE;
    int mxValue = s.length, index = 0,nxIndex = 0;
    if (mxValue > 1) {
        do {
            do {
                index = aleatorio.nextInt(mxValue);
                nxIndex = aleatorio.nextInt(mxValue);
            } while (index == nxIndex);
            int aux = s[index];
```

```
            s[index] = s[nxIndex];
            s[nxIndex] = aux;
            valAct = evaluate(s);
            if (valAct > valIni) {
                s[nxIndex] = s[index];
                s[index] = aux;
                index++;
            }
        } while (valIni < valAct && index < mxValue);
    }
    return s.clone();
}
```

Anexo 6

Fragmento de la Tabla de datos totales con la ejecución de todas las instancias y los algoritmos implementados siendo el A estrella, el genético, BVNS y GVNS obteniendo de cada uno la distancia total del camino obtenido y el tiempo empleado en determinar dicho camino para cada instancia en concreto.

Instance	AE_SOL	AE_Time	Gen_SOL	Gen_Time	BVNS_SOL	BVNS_Time	GVNS_SOL	GVNS_Time
chevrom256P_30_0	88141	47	89249	637	74395	652	51209	491
chevrom256P_30_1	77426	9	77475	589	66837	573	45704	346
chevrom256P_30_2	76584	11	76993	625	68097	571	40349	360
chevrom256P_30_3	73540	10	69987	590	63874	603	46546	348
chevrom256P_30_4	80469	10	78892	614	68287	612	44467	397
chevrom256P_30_5	68031	14	69916	618	59920	614	45303	381
chevrom256P_30_6	73473	12	71889	598	62226	596	42538	357
chevrom256P_30_7	75432	11	69742	558	64084	526	44214	351
chevrom256P_30_8	75901	11	73608	614	67279	566	43487	506
chevrom256P_30_9	88280	8	84300	597	70135	603	45997	422
chevrom256P_60_0	148711	10	141780	1219	125402	1221	89926	703
chevrom256P_60_1	154793	0	149525	1223	130833	1220	91989	793
chevrom256P_60_2	154023	6	144730	1214	127951	1208	88691	794
chevrom256P_60_3	154944	7	155570	1224	130294	1222	90998	673
chevrom256P_60_4	150196	0	142836	1207	126382	1204	90549	792
chevrom256P_60_5	170954	11	162507	1223	138242	1218	96861	764
chevrom256P_60_6	173842	8	165517	1222	145360	1218	89660	773
chevrom256P_60_7	169766	9	160803	1225	140261	1209	91913	669
chevrom256P_60_8	163412	8	154860	1211	135909	1221	94376	772
chevrom256P_60_9	155977	0	151652	1220	133821	1218	89480	808
chevrom256P_90_0	219532	0	214048	1822	187778	1816	125343	1045
chevrom256P_90_1	229684	10	218348	1832	193310	1818	130342	1046
chevrom256P_90_2	251159	10	240869	1828	210035	1667	134168	1141
chevrom256P_90_3	229605	9	220137	1833	190102	1825	135120	1105
chevrom256P_90_4	246321	0	232000	1816	203987	1828	135240	1292
chevrom256P_90_5	234381	10	223288	1832	193934	1828	132232	1167
chevrom256P_90_6	255950	0	239109	1832	211345	1796	143357	1210
chevrom256P_90_7	253592	11	243987	1830	207008	1830	142686	1222
chevrom256P_90_8	243041	0	233008	1795	198856	1702	137164	1186
chevrom256P_90_9	251781	0	244240	1821	209264	1805	141968	1097
...	...	...	...	...	...	...	...	
Promedio	173206	27	171502	1213	149680	1273	104810	1292

Tabla 7: Fragmento de Datos de tiempos y distancias

Anexo 7

Fragmento de la Tabla de datos con la distancia mínima y la desviación local de cada algoritmo con respecto al mínimo en cada instancia tomado a partir de los datos de distancias obtenidas para cada instancia y cada algoritmo.

Instance	MIN	AE-MIN-LOC	Gen_MIN_LOC	BVNS-MIN-LOC	GVNS-MIN-LOC
chevrom_30_0	51209,0	72,12%	74,28%	45,28%	0,00%
chevrom_30_1	45704,0	69,41%	69,51%	46,24%	0,00%
chevrom_30_2	40349,0	89,80%	90,82%	68,77%	0,00%
chevrom_30_3	46546,0	57,99%	50,36%	37,23%	0,00%
chevrom_30_4	44467,0	80,96%	77,42%	53,57%	0,00%
chevrom_30_5	45303,0	50,17%	54,33%	32,26%	0,00%
chevrom_30_6	42538,0	72,72%	69,00%	46,28%	0,00%
chevrom_30_7	44214,0	70,61%	57,74%	44,94%	0,00%
chevrom_30_8	43487,0	74,54%	69,26%	54,71%	0,00%
chevrom_30_9	45997,0	91,93%	83,27%	52,48%	0,00%
chevrom_60_0	89926,0	65,37%	57,66%	39,45%	0,00%
chevrom_60_1	91989,0	68,27%	62,55%	42,23%	0,00%
chevrom_60_2	88691,0	73,66%	63,18%	44,27%	0,00%
chevrom_60_3	90998,0	70,27%	70,96%	43,18%	0,00%
chevrom_60_4	90549,0	65,87%	57,74%	39,57%	0,00%
chevrom_60_5	96861,0	76,49%	67,77%	42,72%	0,00%
chevrom_60_6	89660,0	93,89%	84,61%	62,12%	0,00%
chevrom_60_7	91913,0	84,70%	74,95%	52,60%	0,00%
chevrom_60_8	94376,0	73,15%	64,09%	44,01%	0,00%
chevrom_60_9	89480,0	74,31%	69,48%	49,55%	0,00%
chevrom_90_0	125343,0	75,15%	70,77%	49,81%	0,00%
chevrom_90_1	130342,0	76,22%	67,52%	48,31%	0,00%
chevrom_90_2	134168,0	87,20%	79,53%	56,55%	0,00%
chevrom_90_3	135120,0	69,93%	62,92%	40,69%	0,00%
chevrom_90_4	135240,0	82,14%	71,55%	50,83%	0,00%
chevrom_90_5	132232,0	77,25%	68,86%	46,66%	0,00%
chevrom_90_6	143357,0	78,54%	66,79%	47,43%	0,00%
chevrom_90_7	142686,0	77,73%	71,00%	45,08%	0,00%
chevrom_90_8	137164,0	77,19%	69,88%	44,98%	0,00%
chevrom_90_9	141968,0	77,35%	72,04%	47,40%	0,00%
…	…	…	…	…	…
Promedio	104837,0	74,79%	73,26%	47,48%	0,00%
Nº de veces siendo el mejor		0	0	0	869

Tabla 8: Fragmento de Datos mínimos y mínimos locales

Anexo 8

Fragmento de la tabla de distancias optimas y tiempo de ejecución de las instancias utilizando el modelo exacto de Gubori.

N°	Warehouse	Pedido	SumaTotalDistancias	Tiempo (s)
1	warehouse_graph_circular_30A_244P_W1.json	Pedidos_W1_100_0.json	124970,6033	0,7606746
2	warehouse_graph_fishbone_34A_250P_W1.json	Pedidos_W1_100_0.json	145050,343	0,3921076
3	warehouse_graph_grid_21A_244P_W1.json	Pedidos_W1_100_0.json	144806,6658	0,3646534
4	warehouse_graph_radial_18A_248P_W1.json	Pedidos_W1_100_0.json	147206,2641	0,3249346
5	warehouse_graph_chevrom_21A_256P_W1.json	Pedidos_W1_100_0.json	184319,0246	0,3773925
6	warehouse_graph_circular_30A_244P_W1.json	Pedidos_W1_100_1.json	113499,4491	0,3499849
7	warehouse_graph_fishbone_34A_250P_W1.json	Pedidos_W1_100_1.json	117619,597	0,3922321
8	warehouse_graph_grid_21A_244P_W1.json	Pedidos_W1_100_1.json	107747,7048	0,3825858
9	warehouse_graph_radial_18A_248P_W1.json	Pedidos_W1_100_1.json	124230,3207	0,4772201
10	warehouse_graph_chevrom_21A_256P_W1.json	Pedidos_W1_100_1.json	158330,4078	0,3951913
11	warehouse_graph_circular_30A_244P_W1.json	Pedidos_W1_100_2.json	126653,6651	0,3559197
12	warehouse_graph_fishbone_34A_250P_W1.json	Pedidos_W1_100_2.json	143092,3657	0,3561219
13	warehouse_graph_grid_21A_244P_W1.json	Pedidos_W1_100_2.json	130654,7228	0,3166508
14	warehouse_graph_radial_18A_248P_W1.json	Pedidos_W1_100_2.json	152565,5646	0,3280136
15	warehouse_graph_chevrom_21A_256P_W1.json	Pedidos_W1_100_2.json	199973,1736	0,3673458
	…	…	…	…
		Promedio:	415207,2907	1854,57163

Tabla 9: Fragmento de la tabla de algoritmos exactos de Gubori