

TRABAJO DE FIN DE MÁSTER

Optimización del llenado manual de contenedores con paquetes heterogéneos

Jose Gustavo Quilca Vilcapoma

Máster Universitario en Estadística Computacional y Ciencia de Datos para la Toma
de Decisiones

Instituto Centro de Investigación Operativa

Curso 2023-2024

Optimización del llenado manual de contenedores con paquetes heterogéneos

Jose Gustavo Quilca Vilcapoma

Máster Universitario en Estadística Computacional y Ciencia
de Datos para la Toma de Decisiones

Instituto Centro de Investigación Operativa

Universidad Miguel Hernández de Elche

Curso 2023/2024

Palabras clave:

Optimización, Problema de llenado de contenedores, Algoritmo Genético

Tutor: Javier Alcaraz Soria

Agradecimientos

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetur id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Jose Gustavo Quilca Vilcapoma

Abstract

The transportation of goods in international trade commonly utilizes containers, where efficient usage is crucial to minimize costs and carbon emissions. This paper addresses the combinatorial optimization problem of packing containers with packages of various sizes and shapes, known for its NP-hard complexity, through a metaheuristic based on genetic algorithms. The focus is on manual loading by operators, considering realistic constraints such as package rotation and contiguity. The study proposes improvements in the loading process that enhance the effectiveness of the metaheuristic, allowing for high-quality solutions in reduced times. The method's validity is demonstrated through computational experiments, highlighting its ability to handle complex configurations and offering an efficient and effective framework for the logistical challenge of container loading.

Resumen

El transporte de mercancías en el comercio internacional se realiza comúnmente mediante contenedores, cuyo uso eficiente es crucial para minimizar costos y emisiones de carbono. Este trabajo aborda el problema de optimización combinatoria de llenar contenedores con paquetes de diferentes tamaños y formas, conocido por su complejidad NP-dura, mediante el uso de una metaheurística basada en algoritmos genéticos. Se enfoca en el llenado manual por operarios, considerando restricciones realistas como la rotación y contigüidad de los paquetes. El estudio propone mejoras en el proceso de llenado que potencian la eficacia de la metaheurística, permitiendo obtener soluciones de alta calidad en tiempos reducidos. La validación del método se demuestra a través de experimentos computacionales, destacando su capacidad para manejar configuraciones complejas y ofreciendo un marco eficiente y efectivo para el desafío logístico del llenado de contenedores.

Abbreviations

LHS Left Hand Side

MF Mean-Field

RHS Right Hand Side

MSE Mean Squared Error

Índice

Abstract	v
1. Introducción	1
2. Revision de la literatura	3
2.1. Origen del CLP	3
2.2. Formulaciones del CLP	3
2.3. Metodologías de solución	5
2.4. Software usado en la industria	6
3. Formulación del problema	9
3.1. Definición formal del problema	12
3.2. Procedimiento de carga manual	13
4. Metaheurística	15
4.1. Codificación de soluciones	15
4.2. Función de evaluación	16
4.2.1. Algoritmo de llenado manual	16
4.2.2. Unión de subespacios	19
4.2.3. Eliminación de subespacios inaccesibles	20
4.2.4. Eliminación de subespacios profundos	21
4.2.5. Algoritmo de llenado manual adaptado	22
4.2.6. Mejora adicional al algoritmo de llenado	23
4.3. Población inicial	26
4.4. Selección	27
4.5. Cruce	28
4.6. Mutación	31
4.7. Elitismo	33
4.8. Algoritmo genético	33

5. Estudio experimental	35
5.1. Generación de datos de prueba	35
5.2. Diseño del experimento	37
5.3. Resultados y análisis	38
5.3.1. Valor aportado por las soluciones	38
5.3.2. Tiempo de ejecución	39
5.3.3. Rendimiento	40
6. Conclusiones	43
A. Appendix I	44
B. Appendix II	44
Referencias	45

1. Introducción

En el comercio internacional, el transporte de mercancías se realiza principalmente a través de contenedores de carga. Los contenedores son cajas de acero de forma rectangular que se utilizan para transportar mercancías en barcos, trenes y camiones. Los contenedores son una forma eficiente y segura de transportar mercancías, ya que permiten que las mercancías se carguen y descarguen rápidamente y se almacenen de manera segura durante el viaje. Los contenedores vienen en diferentes tamaños y capacidades, y se utilizan para transportar una amplia variedad de mercancías, incluyendo productos manufacturados, materias primas, alimentos, etc. Para un buen aprovechamiento del espacio y la capacidad de carga de los contenedores, es importante que las mercancías se carguen de manera eficiente y se aproveche al máximo el espacio disponible.

En muchos casos, estas mercancías se encuentran en cajas o paquetes de diferentes tamaños, formas y pesos. Optimizar el llenado de dichos paquetes en los contenedores es un problema importante en la industria de la logística y el transporte ya que puede tener un impacto significativo en los costos y la eficiencia de la cadena de suministro. Por otro lado el mejor aprovechamiento del espacio y la capacidad de carga de los contenedores puede ayudar a reducir el número de viajes necesarios para transportar las mercancías, lo que puede reducir los costos de transporte y las emisiones de carbono asociadas (Parreño et al. 2008).

El problema de llenado de paquetes en contenedores es un problema de optimización combinatoria que ha sido ampliamente estudiado en la literatura. Así mismo, problemas similares se pueden observar en distintas industrias, como el llenado de paquetes en camiones, carga de pallets, carga en almacenes, entre otros, donde la colocación de cajas dentro de otras cajas más grandes es una tarea que se realiza con frecuencia. El llenado de contenedores consiste en colocar paquetes de diferentes tamaños y formas en un contenedor de manera que se utilice el espacio disponible de la mejor manera posible, cumpliéndose ciertas restricciones (peso, estabilidad, etc.) y optimizando uno o más objetivos, entre los que puede estar el minimizar el espacio no utilizado o maximizar el beneficio asociado a la carga transportada, por poner sólo algunos ejemplos.. Este problema ha sido clasificado como NP-duro (Pisinger 2002), lo que significa que muchas veces para instancias grandes de paquetes no existe un algoritmo de tiempo polinomial que pueda resolverlo de manera exacta. Por ello, muchos autores han propuesto diferentes enfoques heurísticos y metaheurísticos para resolver este problema de manera aproximada.

Existen diferentes variantes del problema de la carga de contenedores con paquetes, dependiendo de las restricciones y objetivos específicos que se consideren. Algunas de las variantes más estudiadas incluyen el uso de paquetes homogéneos, paquetes heterogéneos, paquetes rotativos, paquetes frágiles, entre otros. En este trabajo, nos enfocaremos en restricciones derivadas de un caso de uso real que se da cuando la carga es realizada por uno o varios operarios, es decir una carga manual, cuyo principal objetivo es facilitar el proceso de la carga poniendo énfasis en las limitaciones que un operador humano pueda tener. Para esto se considera el uso de paquetes de baja heterogeneidad, que consiste en grupos de paquetes que comparten ciertas características, como el tamaño, el peso, el costo, es decir, paquetes que pueden ser cargados por una persona sin necesidad de maquinaria. También consideraremos restricciones de rotación, que indican que los paquetes pueden ser girados en ciertas direcciones para aprovechar mejor el espacio disponible y restricciones de contigüidad, que indican que los paquetes del mismo grupo deben ser cargados de manera contigua.

En este trabajo, se propone una metaheurística basada en el algoritmo genético para resolver el problema de la carga manual de contenedores con paquetes heterogéneos. El algoritmo genético es una técnica de optimización que se basa en la evolución biológica y que ha sido ampliamente utilizada para resolver problemas de optimización combinatoria. El algoritmo genético es un enfoque de búsqueda poblacional que mantiene una población de soluciones candidatas y utiliza operadores genéticos como la selección, el cruce y la mutación para generar nuevas soluciones a partir de las soluciones existentes. El algoritmo genético es un enfoque flexible y versátil que ha demostrado ser efectivo para resolver una amplia variedad de problemas de optimización combinatoria.

Para evaluar las soluciones que el algoritmo genético va generando, se ha implementado un algoritmo que simula el llenado manual que realizaría un operario, teniendo en cuenta todos los condicionantes que supone este proceso. Por ejemplo, el operario no puede alcanzar ciertas zonas del contenedor si no hay espacio suficiente para pasar o si otros paquetes que han sido colocados anteriormente le obstruyen el paso.

Este trabajo se enfoca en resolver el problema de llenado de contenedores considerando un único contenedor y paquetes débilmente heterogéneos, esta clasificación también es conocida como Three-dimensional Single Large Object Placement Problem (SLOPP) cuya clasificación fue propuesta por Wäscher et al. (2007), sin embargo muchos de los trabajos en la literatura que resuelven este problema no consideran todas restricciones prácticas que se presentan en la industria. (Şafak y Erdoğan 2023) también mencionan que encontrar una disposición óptima de los paquetes en los contenedores mientras se satisfacen una variedad de restricciones del mundo real es una tarea no trivial.

Con esta falta de enfoques en la literatura con un conjunto realista de restricciones prácticas, nuestro objetivo es proponer un método metaheurístico de buen rendimiento que sea capaz de resolver problemas con un número razonable de número de paquetes y garantizar resultados consistentes de buena calidad. Consideramos las restricciones prácticas que tienen como origen del hecho de ser un llenado manual de los paquetes en el contenedor. El método de solución propuesto consiste en un enfoque de llenado priorizando el espacio más al fondo, más debajo y más a la izquierda, compuesto por un simulación de llenado por ordenador y unas propuestas de mejoras en los procesos de llenado. El método está estructurado de tal manera quede claro y facilite el procedimiento de carga para el operario.

Las contribuciones de este trabajo son las siguientes:

- Se propone un método de solución basado en una metaheurística de buen rendimiento para resolver el problema de llenado de contenedores con un conjunto realista de restricciones prácticas derivadas de un caso de uso real de carga manual.
- Se propone una mejora en el proceso de llenado que ayuda a la metaheurística a encontrar soluciones de mejor calidad en menos tiempo.
- Se presenta una serie de experimentos computacionales que muestran la eficacia del método propuesto en diferentes instancias del problema.

El resto de este trabajo está organizado de la siguiente manera. En la sección 2, se presenta una revisión de la literatura relacionada con el problema de la carga de contenedores y sus variantes. En la sección 3, se presenta la definición del problema en particular. En la sección 4, se describe la metaheurística propuesta para resolver el problema. En la sección 5, se presenta un estudio computacional para evaluar el desempeño con diferentes configuraciones. Finalmente, en la sección 6, se presentan las conclusiones y las direcciones futuras de investigación.

2. Revision de la literatura

El problema de llenado de paquetes en contenedores es también conocido en la literatura como Container Loading Problem o CLP, ha sido ampliamente estudiado desde los años 60's (Barnett y Kynch (1967)), una de las definiciones más sencillas lo realiza George y Robinson (1980) quién lo define como encontrar posiciones adecuadas para colocar las cajas en el contenedor de tal manera que todas las cajas puedan ser colocadas en el contenedor sin superponerse de tal modo que se maximice la utilización del espacio. A continuación se presentan algunas revisiones de la literatura que se han realizado sobre el problema.

2.1. Origen del CLP

El problema de llenado de contenedores tiene su origen en el campo del transporte y la logística. Surge de la necesidad de empacar eficientemente objetos en contenedores o vehículos mientras se optimiza la utilización del espacio. Aunque tiene un origen en la industria, el problema ha sido ampliamente objeto de investigación en la comunidad académica, debido a su complejidad, quienes también los clasifican como un problema de optimización combinatoria derivado de otros problemas de optimización como el problema de cortes y empaçado de objetos Alvarez-Valdes et al. (2018), el problema de la mochila de Queiroz et al. (2012) entre otros.

Existen muchos tipo de problemas de llenado de contenedore, presentaremos a continuación algunas de las clasificaciones más comunes que se han propuesto en la literatura.

2.2. Formulaciones del CLP

Muchos de los tipos de problemas formulados en torno al CLP pueden categorizarse en dos tipo principales, los problemas de llenado con restricciones básicas y los problemas de llenado con restricciones prácticas o reales. Los problemas con restricciones básicas son aquellos que consideran restricciones simples, por ejemplo que los paquetes no pueden ser superpuestos y que deben ser colocados dentro de los límites del contenedor, también llamado restricciones de viabilidad del empaquetado Scheithauer (2017). Los problemas con restricciones prácticas consideran restricciones más realistas como por ejemplo restricciones de estabilidad, restricciones de rotación, restricciones de contigüidad, restricciones de peso, entre otras.

Bortfeldt y Wäscher (2013) hizo una revisión de los distintos tipos de restricciones que se han considerado en la literatura, entre las que se encuentran:

CLP con múltiples contenedores: También conocido como el problema de llenado de contenedores múltiples, es una variante del CLP en la que se tienen varios contenedores y se busca llenarlos con un conjunto de paquetes. El objetivo es minimizar el número de contenedores utilizados, maximizando la utilización del espacio en los contenedores.

Algunas de las subvariantes de este problema incluyen el uso de contenedores del mismo tamaño o de diferentes tamaños, por ejemplo Single Bin-Size Bin Packing Problem (SBSBPP) se enfoca en llenar un conjunto de contenedores de un solo tamaño con un conjunto de paquetes (Ren et al. (2011)), mientras que Multiple Bin-Size Bin Packing Problem (MBSBPP) se enfoca en llenar un conjunto de contenedores de diferentes tamaños con un conjunto de paquetes (Zhao et al. (2016)).

CLP con restricciones de estabilidad: La estabilidad se refiere a la capacidad de los paquetes de mantenerse en su lugar ya sea en reposo o durante el transporte, evitando movimientos no deseados que puedan dañar la carga o el contenedor. Las restricciones de estabilidad se refieren a las condiciones que deben cumplirse para garantizar que la carga sea estable, como por ejemplo que

los paquetes no aplasten, se deslicen o no se caigan.

Bortfeldt y Wäscher (2013) identifican dos tipos de restricciones de estabilidad: estática y dinámica, señalando que la mayoría de los estudios se concentran en la primera y pocos en la estabilidad dinámica. Critican las métricas de estabilidad existentes por ser insuficientes y no reflejar adecuadamente la estabilidad dinámica real, lo que puede llevar a evaluaciones incorrectas.

A. Galvão Ramos et al. (2015) propone un enfoque para determinar métricas de estabilidad más realistas y aborda la importancia de la estabilidad en las operaciones de carga de contenedores, destacando su impacto en la satisfacción del cliente, la eficiencia operacional y la seguridad de los paquetes y de los operarios.

CLP con restricciones de rotación: Las restricciones de rotación se refieren a la capacidad de los paquetes de ser rotados en diferentes direcciones, lo que puede aumentar la utilización del espacio y mejorar la estabilidad de la carga. Bortfeldt y Wäscher (2013) señalan que es un tipo de restricción común en la literatura y que se ha investigado ampliamente, categorizando las restricciones de rotación en 6 tipos:

- Caso 1: No se permite rotar las cajas, que deben colocarse en una única orientación fija tanto vertical como horizontalmente.
- Caso 2: Solo se permite una orientación vertical fija para cada tipo de caja, pero se pueden rotar 90° en el plano horizontal.
- Caso 3: Las cajas pueden rotarse libremente en el plano horizontal y tienen restricciones limitadas en la orientación vertical.
- Caso 4: Las cajas pueden tener hasta cinco orientaciones prohibidas en cualquier dirección, permitiendo una amplia variedad de restricciones.
- Caso 5: Todas las cajas son completamente rotativas sin restricciones de orientación, proporcionando el máximo grado de libertad.

CLP con paquetes heterogéneos: La heterogeneidad de los paquetes se refiere a la variabilidad en las dimensiones, formas y pesos de los paquetes, lo que puede complicar el proceso de empaquetado y afectar la utilización del espacio. Wäscher et al. (2007) clasificaron los problemas de llenado de contenedores según la variedad de artículos grandes y pequeños, relacionándolo con el número de diferentes tipos de artículos en el problema y lo clasificaron como: homogéneo (un solo tipo), fuertemente heterogéneo (muchos tipos) o ligeramente heterogéneo (pocos tipos).

Zhao et al. (2016) propusieron una clasificación más detallada basada en la homogeneidad de los contenedores y las cajas, identificando seis subtipos de problemas de llenado de contenedores:

- Problema de corte de stock de tamaño único (SSSCSP) si los contenedores son idénticos y las cajas son ligeramente heterogéneas.
- Problema de empaque de contenedor de tamaño único (SBSBPP) si los contenedores son idénticos y las cajas son muy heterogéneas.
- Problema de corte de stock de múltiples tamaños (MSSCSP) si tanto los contenedores como las cajas son ligeramente heterogéneas.
- Problema de empaque de contenedor de múltiples tamaños (MBSBPP) si los contenedores son ligeramente heterogéneos y las cajas muy heterogéneas.
- Problema de corte de stock residual (RCSP) si los contenedores son muy heterogéneos y las cajas ligeramente heterogéneas.

- Problema de empaque de contenedor residual (RBPP) si tanto los contenedores como las cajas son muy heterogéneos.

2.3. Metodologías de solución

Muchos autores han propuesto diferentes enfoques heurísticos y metaheurísticos para resolver este problema de manera exacta o aproximada. A continuación se presentan algunas de las metodologías de solución que se han propuesto en la literatura.

Métodos exactos: Los métodos exactos son aquellos que garantizan encontrar la solución óptima al problema, sin embargo, estos métodos pueden ser computacionalmente costosos y no escalables para problemas grandes. Muchos autores han intentado resolver el CLP con restricciones prácticas de manera exacta, pero la mayoría ha encontrado que los métodos exactos no son adecuados para problemas de gran tamaño. Por ejemplo Junqueira et al. (2012) mencionan que tuvieron problemas de desbordamiento de memoria al intentar resolver instancias con mas de 20 tipos. Más recientemente Nascimento et al. (2021) propusieron un enfoque exacto basado en modelos de programación lineal entera y programación por restricciones para resolver el problema de llenado de contenedores, demostrando que pudieron resolver instancias de hasta 10 tipos de paquetes y hasta 110 paquetes en un tiempo razonable.

Métodos heurísticos: Los métodos heurísticos son aquellos que no garantizan encontrar la solución óptima, pero pueden proporcionar soluciones de buena calidad en un tiempo razonable. Los métodos heurísticos son ampliamente utilizados para resolver problemas de optimización combinatoria, como el CLP, debido a su eficiencia y escalabilidad. Algunos de los métodos heurísticos constructivos más comunes que se han propuesto en la literatura son: heurística en base a la construcción de paredes (George y Robinson 1980), con una versión mejorada por Pisinger (2002) y heurística en base a la construcción por capas (Bischoff y Ratcliff 1995), con una versión mejorada por Ranck Júnior et al. (2019).

Métodos metaheurísticos: El término metaheurística fue acuñado en los años 80s por F. Glover (1986). Una metaheurística puede entenderse como una metodología que incluye estrategias maestras capaces de guiar la búsqueda hacia la solución óptima global. Se consideran más complejas y eficientes que los algoritmos heurísticos simples porque exploran áreas en el espacio de soluciones que van más allá de las exploradas por las heurísticas simples, las cuales tienden a centrarse en encontrar una única solución óptima local. Luego Sörensen y F. W. Glover (2013) lo definió como "Un marco algorítmico de alto nivel e independiente del problema que proporciona un conjunto de directrices o estrategias para desarrollar algoritmos de optimización heurísticos". Más recientemente Martí et al. (2024) definen una metaheurística no como un algoritmo específico con pasos precisamente definidos, sino como un conjunto más o menos consistente de ideas de alto nivel. Estas ideas sirven como directrices para desarrollar algoritmos de optimización heurísticos ajustados a problemas específicos. En esta definición, se enfatiza la flexibilidad que tiene el diseñador del algoritmo para elegir las características específicas de su método, y se destaca que puede requerirse un grado considerable de "ingeniería" para adaptar el marco de metaheurística a fin de resolver problemas de optimización concretos.

Algunos de los métodos metaheurísticos más comunes que se han propuesto en la literatura para resolver el problema de llenado de contenedores son:

- **Algoritmos Genéticos (GA):** Se describe como un algoritmo de búsqueda inspirado en la evolución natural (Goldberg 2013), útil para resolver muchos problemas de optimización debido a su robustez y capacidad de manejar complejidades diversas. Los GA se utilizan en el contexto del CLP para manejar restricciones como el equilibrio de carga, considerado como una restricción estricta en algunos estudios. Por ejemplo algunos autores que han usado GA para resolver el problema de llenado de contenedores son António G. Ramos et al. (2018),

Gonçalves y Resende (2012), Kang et al. (2012a) entre otros.

- **Algoritmos de Enjambre de Partículas (PSO):** Se describe como un algoritmo de optimización basado en la inteligencia de enjambre, como el de los pájaros o peces. Por ejemplo algunos autores que han usado PSO para resolver el problema de llenado de contenedores son Kuo et al. (2023), Domingo et al. (2012), Cano y Torra (2010) entre otros.

2.4. Software usado en la industria

En la industria existen varios software que permiten resolver el problema de llenado de contenedores, algunos de los más conocidos son:

Cargo Manager: Según Zhao (2017), Cargo Manager (CM) es un ejecutable independiente diseñado para empaquetar contenedores siguiendo un orden de prioridad específico, comenzando por la parte trasera del contenedor. El proceso de empaquetado utiliza una serie de reglas de colocación, probando cada artículo en el orden establecido; si un artículo no es adecuado, se pasa al siguiente, volviendo a considerar los artículos no adecuados en futuras colocaciones. Los artículos de mayor prioridad se empaquetan primero y es posible configurar el sistema para que se adhiera estrictamente a las prioridades de empaque, asegurando que todos los artículos de una prioridad actual se empaquen antes de comenzar con la siguiente.

El proceso principal de empaquetado en CM utiliza diferentes métodos heurísticos constructivos que varían desde heurísticas básicas hasta métodos más complejos y consumidores de tiempo. El objetivo es maximizar el uso del espacio del contenedor formando "muros" con los artículos, donde cada bloque consiste en cajas del mismo tipo y orientación. A medida que se colocan los artículos, los espacios que ocupan ya no están disponibles, generándose nuevos espacios cúbicos que pueden fusionarse con espacios previos. Los métodos difieren en cómo seleccionan los espacios y las cajas, y las restricciones aplicadas al tamaño de los bloques.

En una etapa adicional, CM utiliza la heurística más básica para intentar empaquetar tantos artículos restantes como sea posible. Esta etapa no es adecuada para problemas de múltiples destinos o donde se requiere nivelación de carga, y excluye artículos pesados o frágiles. La solución óptima se determina comparando el volumen y la longitud utilizados por diferentes métodos, seleccionando como mejor aquella solución que ocupe mayor volumen o, en caso de igual volumen, utilice menos longitud. Los usuarios pueden decidir cuántas heurísticas se evalúan, y en caso de seleccionar un solo método, su solución se convierte automáticamente en la solución final. En la imagen 1 se muestra la interfaz gráfica de Cargo Manager.

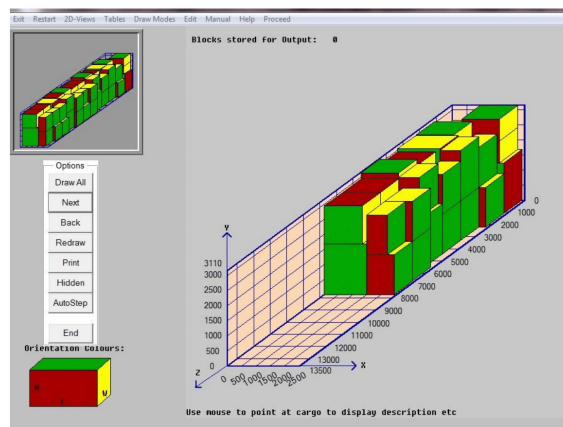


Figura 1: La interfaz gráfica de Cargo Manager permite a los usuarios empaquetar contenedores siguiendo un orden de prioridad específico, maximizando el uso del espacio y minimizando la longitud utilizada (Zhao 2017).

PackageCargo: PackageCargo es una herramienta de apoyo a la decisión, diseñada para abordar el problema de carga de contenedores (CLP), que es esencial en la logística de transporte (Martínez-Franco et al. 2020). Desarrollada como una aplicación de código abierto usando el motor de juegos Unity, esta herramienta permite calcular, visualizar y guardar patrones eficientes de empaque, además de estimar métricas de estabilidad de la carga mediante modelos matemáticos y simulaciones físicas. Su objetivo principal es ofrecer un sistema utilizable tanto en entornos industriales como académicos, permitiendo a los usuarios modificar el marco de trabajo según sus necesidades, lo que ahorra tiempo en desarrollo de software y fomenta la contribución comunitaria para su mejora continua.

El software utiliza un diseño modular que incluye componentes para la visualización de soluciones, simulación de la estabilidad de la carga, y optimización de los patrones de empaque. La arquitectura de PackageCargo facilita la integración de algoritmos de optimización para generar patrones eficientes y su simulación correspondiente para evaluar la estabilidad, usando el motor de físicas PhysX de Nvidia. Esta estructura permite una fácil extensión y adaptación del software, fomentando la innovación y colaboración entre investigadores y profesionales del sector.

PackageCargo no solo ofrece una alternativa competitiva a soluciones comerciales, sino que también se posiciona como una plataforma valiosa para la investigación y el desarrollo en el ámbito del CLP. Al proporcionar una herramienta accesible y extensible, promueve el avance en la comprensión y solución de problemas complejos de carga, beneficiando tanto a la comunidad académica como a la industrial. En la imagen 2 se muestra la interfaz gráfica de PackageCargo.

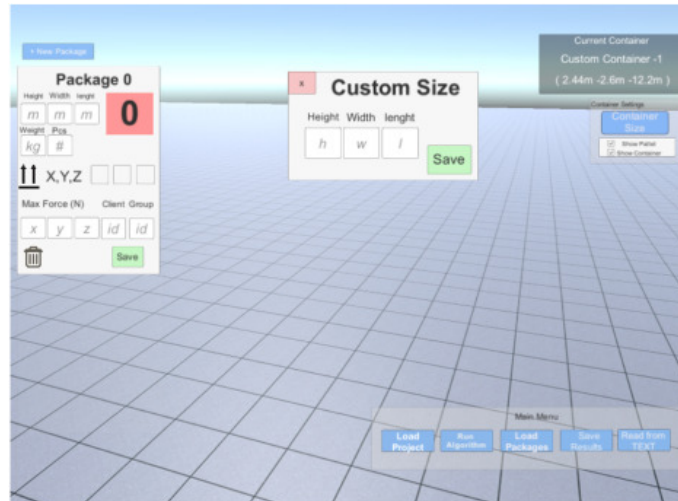


Figura 2: La interfaz gráfica permite al usuario definir variedades de carga, dimensiones, restricciones y cantidades. Esta información se convierte en un archivo de entrada para los algoritmos generadores de patrones de empaque (Martínez-Franco et al. 2020).

LoadCargo: LoadCargo es un software comercial diseñado para facilitar la planificación y optimización de la carga en contenedores y camiones. Utiliza tecnología en 3D para ofrecer visualizaciones interactivas que mejoran la precisión de las simulaciones de carga. Este sistema admite tanto planificaciones automáticas como manuales, proporcionando herramientas flexibles para diversos requerimientos logísticos (LoadCargo.in 2024).

El software es accesible en múltiples plataformas a través de Adobe Air, compatible con sistemas operativos como Windows, Mac OS X y Linux. Soporta tanto unidades métricas como imperiales y se integra con sistemas EDI (Intercambio Electrónico de Datos) para facilitar la importación de datos. Además, incluye características como la visualización del centro de gravedad, exportaciones de planificaciones a varios formatos y herramientas para la construcción de pallets, entre otras.

LoadCargo es eficaz para optimizar el uso del espacio en contenedores y camiones, aunque mencionan que presenta limitaciones como la incapacidad de manejar cargas de formas irregulares o volúmenes muy altos de cajas. Sin embargo, para operaciones estándar, ofrece soluciones robustas que mejoran la eficiencia en la carga y reducen los costos operativos. En la imagen 3 se muestra la interfaz gráfica de LoadCargo.

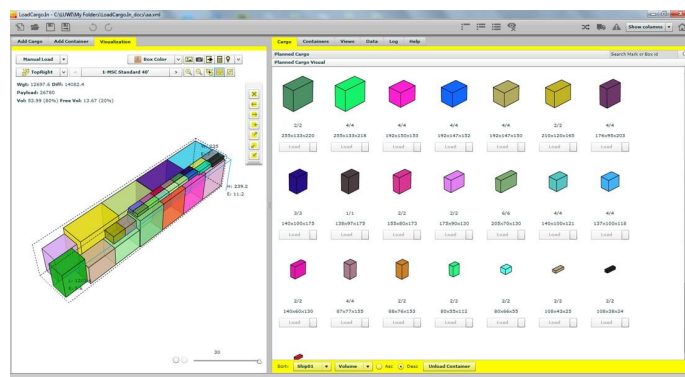


Figura 3: La interfaz gráfica de LoadCargo permite a los usuarios visualizar y planificar la carga de contenedores y camiones, mejorando la eficiencia y reduciendo los costos operativos (LoadCargo.in 2024).

3. Formulación del problema

El desafío presentado se basa en una situación real relacionada con una empresa que gestiona un elevado volumen de pedidos de cajas de diferentes tipos. Cada semana, la empresa busca determinar cuántas cajas de cada tipo debe enviar en un contenedor para maximizar el beneficio total de dicho envío.

El desafío principal al que se enfrenta la empresa es optimizar el proceso de envío de cajas, donde cada una proporciona un ingreso específico. El objetivo es maximizar el uso del espacio en los contenedores para obtener el mayor beneficio posible. Dado que las cajas tienen distintos tamaños, pesos y valores, se plantea una dificultad para estimar de manera eficaz cómo llenar completamente un contenedor sin desperdiciar espacio. Este desperdicio no solo implica una pérdida financiera, sino que también tiene un impacto ambiental negativo, ya que cada viaje consume combustible y genera emisiones contaminantes.

Para abordar este desafío, es crucial considerar el proceso manual de carga del contenedor. El procedimiento actual de carga no está optimizado y no considera la disposición óptima de las cajas, lo que lleva a una carga subóptima y un desperdicio de espacio. Por tanto, es necesario revisar y mejorar este procedimiento para asegurar que se aproveche al máximo el espacio disponible en los contenedores.

El procedimiento de carga manual debe ser meticulosamente evaluado y optimizado. Este aspecto es crucial porque los operarios son los responsables de organizar físicamente las cajas en el contenedor. Un procedimiento manual de carga bien diseñado garantizaría que se aproveche cada centímetro disponible del contenedor, reduciendo así el espacio vacío y aumentando la rentabilidad del envío.

Respecto a las restricciones que se generan debido a la carga manual, se consideran las siguientes:

Los paquetes que son cajas de forma rectangular, pueden variar en tamaño, peso y valor, pero han sido concebidos previamente para que puedan ser cargados manualmente es decir que no son paquetes muy grandes o pesados.

Los paquetes que comparten el mismo tamaño, peso y valor estrictamente se consideran del mismo tipo, dos paquetes pueden tener el mismo tamaño y peso pero distinto valor, lo que los convierte en tipos diferentes. El valor de un paquete no depende de su tamaño o peso, es decir que un paquete sea más grande y pesado que otro no implica que sea de mayor valor y viceversa.

Los paquetes llegan a la puerta del contenedor agrupados por tipo y en un orden específico, los paquetes pueden apilarse unos sobre otros independientemente de su tipo ya que las cajas lo soportan y sus pesos no son muy dispares, pero se debe asegurar la estabilidad de la carga. Por ejemplo en el la Figura 4 se muestra un ejemplo de cómo se apila un tipo de paquete encima de otro.

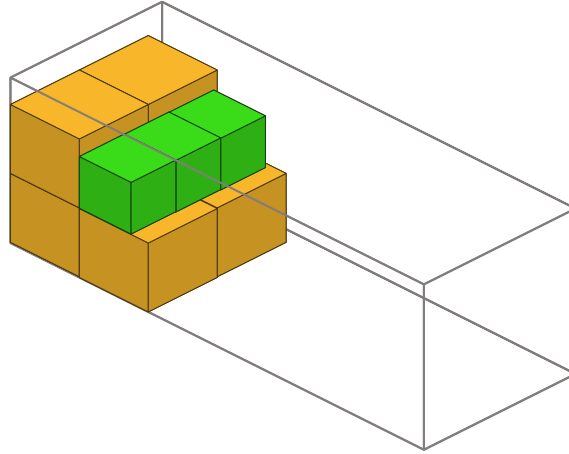


Figura 4: Ejemplo de cómo se apila un tipo de paquete encima de otro.

Para asegurar la estabilidad de la carga, por ejemplo un paquete más grande no puede estar encima de uno más pequeño, es decir un paquete siempre debe tener una base sobre la que se apoye, en la Figura 5 se muestra un ejemplo de una carga inestable.

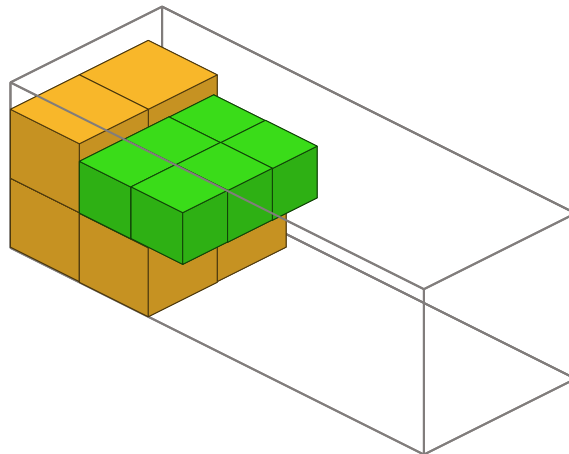


Figura 5: Ejemplo de una carga inestable.

Para facilitar la carga manual se considera de que todos los paquetes de un mismo tipo deben mantener la misma orientación, es decir que no se pueden colocar paquetes de un mismo tipo en diferentes orientaciones, por ejemplo en la Figura 6 se muestra un ejemplo de cómo no se deben colocar los paquetes, ya que dificultaría al operario seguir dicho procedimiento, además que aumentaría el riesgo de desperdiciar espacio o de que la carga sea inestable.

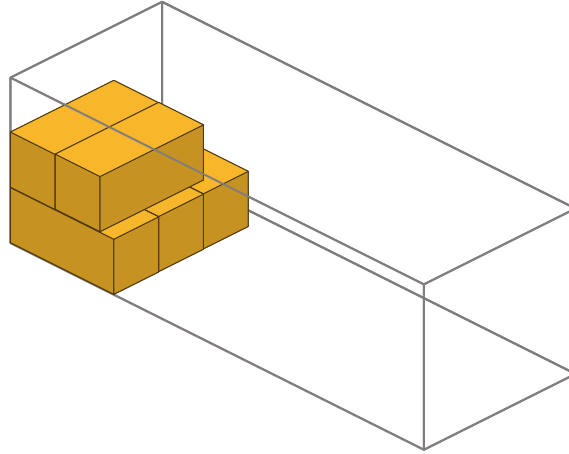


Figura 6: Ejemplo de cómo los paquetes de un mismo tipo tienen distinta orientación.

Muchas de las cajas están diseñadas para ser apiladas y soportar un gran peso encima siempre y cuando se respete la indicación de mantener una posición mirando hacia arriba, por lo que los paquetes solo pueden ser girados en un eje, por ejemplo en la Figura 7 se muestra un mismo tipo de paquete girado en un eje.

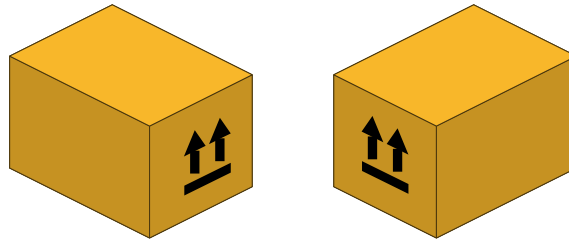


Figura 7: Ejemplo de cómo los paquetes de un mismo tipo pueden ser girados en un eje.

Para evitar la fatiga del operador de levantar los paquetes, la empresa suele usar cintas o bandas transportadoras, para aprovechar su uso, esto implica que los paquetes deben ser colocados en primer lugar lo más profundo posible del contenedor, es decir que los paquetes que están siendo cargados, deben ser colocados en la parte más alejada de la puerta del contenedor, de este modo también se evita que los paquetes obstruyan el ingreso del personal de carga al contenedor. En la Figura 8 se muestra un ejemplo de cómo se puede hacer uso de una cinta transportadora que desliza los paquetes hacia el fondo del contenedor, mientras el contenedor se va llenando la cinta se va moviendo en sentido contrario.

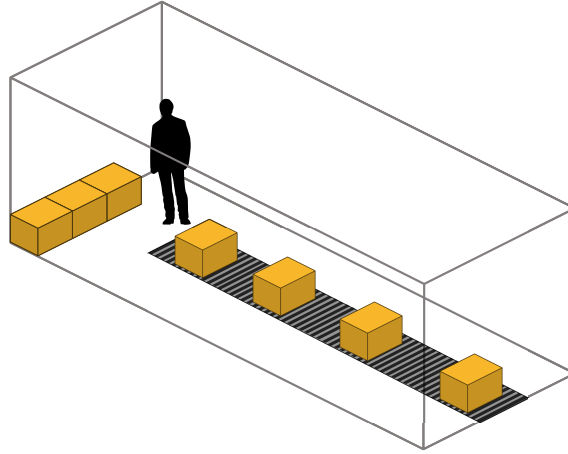


Figura 8: Ejemplo del uso de una cinta transportadora.

La empresa ha implementado un procedimiento específico para guiar a los operadores en la colocación de paquetes dentro del contenedor, con el objetivo de cumplir con todas las condiciones preestablecidas. No obstante, este procedimiento aún no está optimizado y no garantiza la disposición óptima de los paquetes, lo que conduce a una carga subóptima y al desperdicio de espacio.

El desafío que se enfrenta es conocer con anticipación la cantidad de paquetes por tipo que deben cargarse en el contenedor, así como definir el orden de carga y la rotación de cada tipo de paquete. El objetivo es lograr una disposición que no solo cumpla con las restricciones de espacio y los requerimientos del llenado manual, sino que también maximice el valor económico de la carga transportada y la eficiencia en el uso del espacio .

3.1. Definición formal del problema

El problema de la carga manual de paquetes en un contenedor se define formalmente de la siguiente manera:

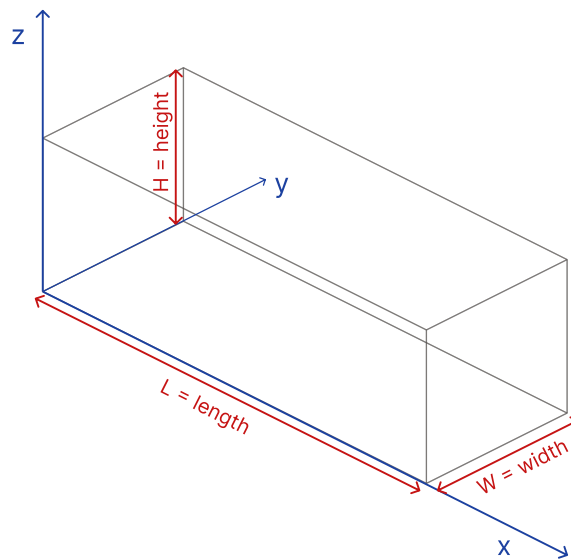


Figura 9: Contenedor con dimensiones W , L , H

Siendo un contenedor una caja de forma rectangular, de ancho W , largo L , alto H , en la figura 9 se muestra un contenedor con sus dimensiones, y una capacidad de carga P , se tiene definido unos tipos de paquetes también de formas rectangulares $t \in T = \{0, 1, 2, \dots, n\}$, donde cada tipo t posee ciertas dimensiones de ancho w_t , largo l_t , alto h_t , también posee un peso p_t y un valor v_t , además se conoce la cantidad máxima q_t^{max} de paquetes que dispone la empresa por cada tipo.

En este problema, consideramos que W, L, H y $P, w_t, l_t, h_t, p_t, q_t^{max}$ son enteros positivos, los cuales podrían ser representados en unidades de medida centímetros, milímetros, kilogramos, litros, entre otros.

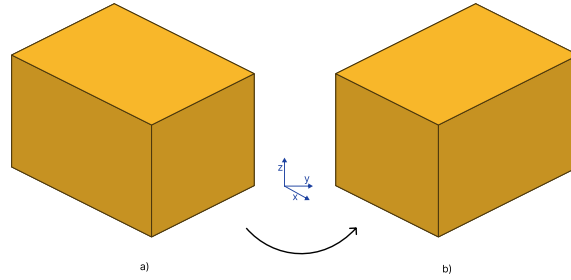


Figura 10: Rotación de un paquete en el eje x

Se tienen restricciones de rotación debido al enfoque de carga manual, en el cual se establece que $\forall r \in r_t, r \in \{0, 1\}, t \in T$ donde 0 representa que el tipo no se encuentra girado y 1 que el tipo está girado 90 grados en el eje x , esto se puede ver en la figura 10 a) sin rotación y b) con rotación. Esto implica que los anchos y largos pueden intercambiarse, mientras que la altura no puede ser modificada.

Por otro lado para facilitar la carga manual, se debe disponer de un orden de carga o_t para cada tipo t , donde $o_t \in O = \{0, 1, 2, \dots, n\}$, que indica el orden en el que se debe cargar cada tipo de paquete en el contenedor.

El problema consiste en determinar la cantidad de paquetes por cada tipo a cargar \tilde{q}_t (la cual se encuentra en torno a la cantidad máxima por tipo, $0 \leq \tilde{q}_t \leq q_t^{max}$) y el orden de carga de cada tipo o_t con determinada rotación r_t , de tal modo que se pueda obtener la disposición óptima de los paquetes en el contenedor, asegurando el cumplimiento de las restricciones relacionadas al espacio disponible. Además, se busca maximizar en primer lugar el costo total de la carga ($\max \sum_{t \in T} v_t \cdot \tilde{q}_t$) y en segundo lugar maximizar la utilización del espacio del contenedor ($\max \sum_{t \in T} w_t \cdot l_t \cdot h_t \cdot \tilde{q}_t$).

3.2. Procedimiento de carga manual

Para estandarizar la carga manual y facilitar el modelado por ordenador, así como simplificar la labor del operador, se han establecido una serie de supuestos, restricciones y normas que deben seguirse para optimizar la carga de paquetes en el contenedor.

En relación con la definición del problema, se establecen las siguientes suposiciones acerca de los paquetes:

- Los paquetes son cajas de forma rectangular.
- Los paquetes pueden variar en tamaño, peso y valor.
- Los paquetes presentan tamaños y pesos razonables para ser cargados manualmente.

- Los paquetes que comparten el mismo tamaño, peso y valor estrictamente se consideran del mismo tipo.
- Dos paquetes pueden tener el mismo tamaño y peso pero distinto valor, lo que los convierte en tipos diferentes.
- El valor de un paquete no depende de su tamaño o peso, es decir que un paquete sea más grande y pesado que otro no implica que sea de mayor valor y viceversa.
- Los paquetes llegan al contenedor agrupados por tipo y en un orden específico.
- Los paquetes pueden apilarse unos sobre otros, independientemente de su tipo, pero se debe asegurar la estabilidad de la carga.
- Cada tipo de paquete tiene una cantidad fija deseada que debe ser cargada en el contenedor.
- Todos los paquetes de un mismo tipo deben mantener la misma orientación.
- Los grupos de paquetes llegan en bloques del mismo tipo o de forma secuencial, por ejemplo, a través de cintas transportadoras.

Las suposiciones relacionadas con el operario son las siguientes:

- Uno o varios operarios realizan la carga de los paquetes en el contenedor de forma manual.
- El operario recibe indicaciones previas sobre cómo cargar los paquetes en el contenedor, incluyendo el orden, la cantidad y la orientación de cada tipo de paquete.
- Las indicaciones también podrían especificar los espacios que deberán quedar vacíos en el contenedor, los cuales pueden ser llenados con material de relleno para evitar que los paquetes se muevan durante el transporte.
- Las indicaciones previas proporcionadas al operario son el resultado de la solución del problema de optimización de la carga.
- El objetivo del operario es seguir las indicaciones previas de manera eficiente y precisa, sin necesidad de tomar decisiones adicionales sobre la disposición de los paquetes.

El procedimiento de carga manual se basa en la combinación de las suposiciones, restricciones y reglas mencionadas anteriormente, con el objetivo de lograr una carga eficiente y organizada de los paquetes en el contenedor. Este procedimiento se implementa siguiendo una metodología específica que guía al operario en la colocación de los paquetes, asegurando que se cumplan todas las condiciones establecidas.

En el siguiente capítulo se presentará un algoritmo metaheurístico para resolver el problema de optimización de la carga manual de paquetes en un contenedor, considerando las restricciones de espacio del contenedor, así como las restricciones de rotación y orientación de los paquetes.

4. Metaheurística

4.1. Codificación de soluciones

Existen diferentes maneras de codificar las soluciones para el problema de llenado de contenedores. Algunos de los más interesantes son por ejemplo Bortfeldt y otros Gehring y ortfeldt 1997 que propusieron una codificación basada dobles o simplemente dos listas de igual longitud, la primera representa la secuencia de llenado de los paquetes y la segunda indica la rotación de cada paquete. De manera similar Xianbo y otros [2] propusieron el uso de la primera lista como la secuencia de llenado de cada tipo de paquete y la segunda como la rotación de cada tipo de paquete.

Sin embargo, para las restricciones en nuestro problema, esta codificaciones no son adecuadas ya que necesitamos considerar que las cantidades de un tipo de paquete no son fijas sino que pueden variar de un rango mínimo a un máximo. Por lo tanto, proponemos una codificación basada en 3 listas de igual longitud, la primera lista representa la secuencia de llenado de cada tipo de paquete, la segunda lista indica la rotación de cada tipo de paquete y la tercera lista indica la cantidad de paquetes de cada tipo que se llenan en el contenedor.

Por ejemplo, si tenemos 4 tipos de paquetes $T = \{1, 2, 3, 4\}$, todos los tipos con un mínimo de 0 paquetes y el tipo 1 con un máximo de 10 paquetes $q_1 \in [0, 10]$, el tipo 2 con un máximo de 30 paquetes $q_2 \in [0, 30]$, el tipo 3 con un máximo de 20 paquetes $q_3 \in [0, 20]$ y el tipo 4 con un máximo de 22 paquetes $q_4 \in [0, 22]$, una solución factible para el problema de llenado de contenedores podría ser la siguiente:

Tipo	2	1	4	3
Rotación	0	1	1	0
Cantidad	27	10	18	13

Figura 11: Codificación de una solución con 4 tipos de cajas

La primera lista indica que el tipo 2 se llena primero, luego el tipo 1 seguido del tipo 4, y finalmente el tipo 3. La segunda lista indica que el tipo 2 no se rota, el tipo 1 se rota, el tipo 4 se rota y el tipo 3 no se rota. La tercera lista indica que se llenan 27 paquetes del tipo 2, 10 paquetes del tipo 1, 18 paquetes del tipo 4 y 13 paquetes del tipo 3. En la Figura 2 se muestran dos ejemplos adicionales sobre cómo se usa la codificación propuesta para representar soluciones para el problema.

Las tres listas mencionadas son fundamentales para el algoritmo de llenado descrito en la sección 2. Este algoritmo llena el contenedor de manera determinista, siguiendo estrictamente las restricciones establecidas por el problema. Específicamente, una lista determinada siempre resulta en un único método de llenado del contenedor. Si, después de llenar el contenedor, quedan paquetes sobrantes, se procederá a descartar aquellos que no caben, lo que requiere una actualización o corrección de la lista de cantidades de cada tipo de paquete. Este proceso asegura la factibilidad de la solución.

Otra ventaja de esta codificación es la flexibilidad que ofrece para adaptar el problema a diferentes restricciones, por ejemplo si se desea considerar una cantidad fija de paquetes de un tipo, simplemente se puede fijar el mínimo y el máximo de ese tipo de paquete. De igual manera, la estructura soporta si se desea considerar otros tipos de rotaciones, además de las consideradas en el presente trabajo.

Este tipo de codificación permite representar soluciones factibles para el problema de llenado de contenedores, ya que considera la cantidad de paquetes de cada tipo que se llenan en el contenedor. Además, esta codificación es fácil de implementar y de entender, lo que facilita la implementación del algoritmo de llenado y del algoritmo de optimización para resolver el problema.

4.2. Función de evaluación

La función de evaluación es una parte fundamental de cualquier algoritmo de optimización, ya que permite evaluar la calidad de las soluciones generadas por el algoritmo. Para el problema de llenado de contenedores, la función de evaluación se puede definir como la suma de los valores de los paquetes que se llenan en el contenedor, es decir, se busca maximizar el valor total de la carga que se llena en el contenedor.

Un paso previo importante antes de calcular el valor total de la carga es verificar si la solución generada es factible, es decir, si al agregar todas las cajas en los ordenes y las posiciones indicadas en la solución, se cumple que todas son colocadas en el contenedor sin superar su capacidad. Para ello, se propone un algoritmo que simula el procedimiento de llenado manual, el cual verifica si una solución generada es factible y si no lo es, lo corrige para que sea factible.

4.2.1. Algoritmo de llenado manual

El algoritmo de llenado propuesto está basado en el método Deepest Bottom Left with Fill (DBLF) propuesto por Karabulut y otros Karabulut y İnceoğlu 2005, cuyo uso se ha extendido y varios autores han realizado propuestas para mejorarlo o adaptarlo a determinados contextos por ejemplo Wang y otros Wang y Chen 2010, Kang y otros. Kang et al. 2012b. El algoritmo propuesto en el presente trabajo está enfocado en cumplir las restricciones y adaptarse al contexto de un llenado manual de paquetes en un contenedor.

La idea básica del algoritmo DBLF es que los paquetes comienzan a ser llenados en el contenedor de forma secuencial, se prioriza que el paquete se coloque en las siguientes posiciones:

- La posición más profunda del contenedor, que ayuda a llenar primero los espacios más alejados de la puerta y evitar obstaculizar el ingreso del operador humano al contenedor.
- La posición más baja del contenedor, se da naturalmente debido al efecto de la gravedad, un paquete no podría ser colocado en una posición alta si no se ha llenado primero las posiciones más bajas.
- La posición más a la izquierda del contenedor, aunque no es una restricción fuerte, podría usarse el criterio de priorizar la posición más a la derecha si se considera necesario, lo crucial es mantener la consistencia al elegir una de estas dos direcciones. Para el caso de este trabajo se ha elegido la posición más a la izquierda.

Al inicio del procedimiento un paquete se coloca en la posición más profunda, más baja y más a la izquierda del contenedor vacío, luego al colocar el paquete en el contenedor, el espacio restante se divide en tres nuevos subespacios, la parte superior, la parte lateral y la parte frontal, en la figura 12 se muestra de cómo se divide el espacio restante en el contenedor al colocar un paquete.

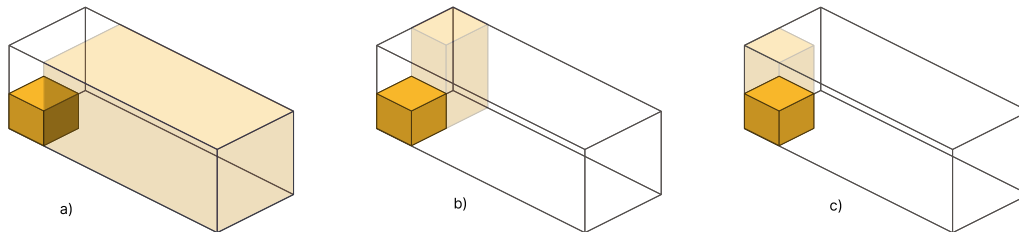


Figura 12: División del espacio restante en el contenedor al colocar un paquete. a) Subespacio frontal, b) Subespacio lateral, c) Subespacio superior.

Cada subespacio es considerado como un nuevo contenedor y se repite el proceso de colocar el siguiente paquete en uno de los subespacios creados. Para elegir el siguiente subespacio donde colocar el siguiente paquete, se usa el mismo criterio de priorización DBL, es decir, se elige el subespacio más profundo, más bajo y más a la izquierda, luego se coloca el paquete en dicho subespacio y se repite el proceso.

En la figura 13 se muestra un ejemplo de cómo se coloca un segundo paquete en el contenedor siguiendo el algoritmo DBLF.

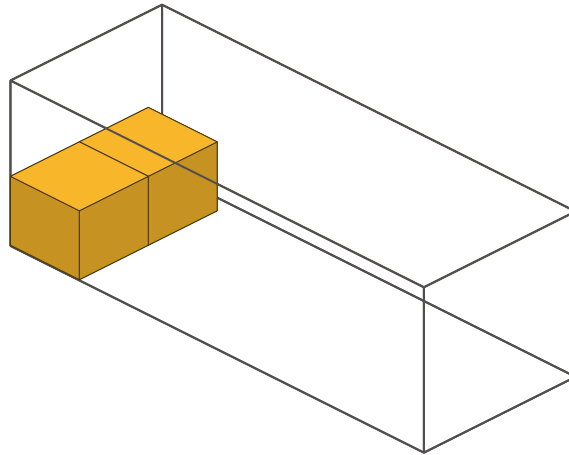


Figura 13: Ejemplo de colocación de un segundo paquete en el contenedor luego de elegir el último subespacio lateral.

Para este segundo paquete, se ha elegido el anterior subespacio lateral, el cual fué el subespacio más profundo, más bajo y más a la izquierda, siguiendo el criterio de priorización DBL, luego se coloca el paquete en dicho subespacio y se repite el proceso de dividir el espacio restante. Como resultado de esta subdivisión, se obtienen en este caso solamente dos subespacios, la parte superior y la parte lateral, ya que no queda espacio frontal para dividir. En la figura 14 se muestra estos dos nuevos subespacios.

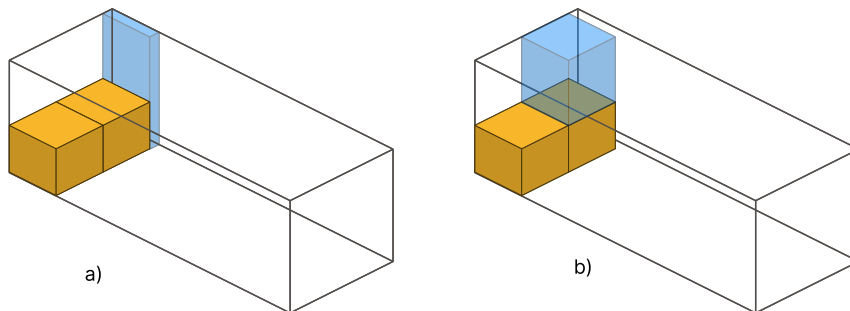


Figura 14: División del espacio restante en el contenedor al colocar un segundo paquete. a) Nuevo subespacio lateral, b) Nuevo subespacio superior.

Este procedimiento se repite hasta que se hayan colocado todos los paquetes en el contenedor o no se pueda colocar más paquetes debido a restricciones de espacio. En la figura 15 se muestra como este procedimiento se asemeja a un tipo de construcción de un muro el cuál es otro método de llenado de contenedores.

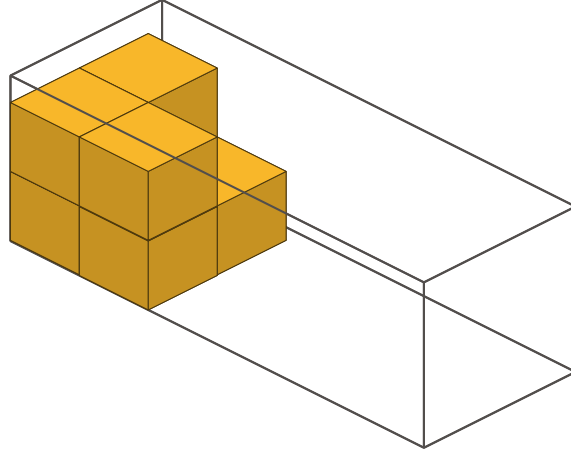


Figura 15: Ejemplo de colocación de más paquetes en el contenedor

El Algoritmo 1 muestra el procedimiento de llenado manual de paquetes en un contenedor basado en el algoritmo DBLF.

Algoritmo 1 Algoritmo de llenado manual de paquetes en un contenedor

```

1: Paquetes  $\leftarrow$  lista de paquetes ordenados por tamaño
2: Inicialización: dblf  $\leftarrow$  lista inicializada con el espacio total del contenedor
3: Contenedor  $\leftarrow$  lista vacía para almacenar los paquetes colocados
4: for paquete  $\in$  Paquetes do
5:   SubespacioOptimo  $\leftarrow$  buscar el subespacio más adecuado en dblf
6:   if SubespacioOptimo  $\neq$  null then
7:     Contenedor.add(colocar(paquete, SubespacioOptimo))
8:     NuevosSubespacios  $\leftarrow$  dividir(SubespacioOptimo, paquete)
9:     dblf.remove(SubespacioOptimo)
10:    dblf.extend(NuevosSubespacios)
11:   else
12:     print "No se encontró espacio para el paquete."
13:   end if
14: end for
15: return Contenedor

```

El el Algoritmo 1, en la línea 1: se inicializa una lista llamada *Paquetes* que contiene los paquetes ordenados. Este orden será determinado previamente por el algoritmo de optimización genético, el cual se detallará en la próxima sección. En la línea 2: Se inicializa una lista llamada *dblf*, que representa los subespacios libres en el *Contenedor*. Inicialmente, esta lista contiene un único subespacio que es el contenedor entero. En la línea 3: Se crea una lista vacía llamada *Contenedor* donde se almacenarán la posición y tamaños de los paquetes que se vayan colocando. Para la parte del proceso de llenado, en la líneas 4: El ciclo for recorre cada paquete en la lista de paquetes. En la línea 5: Se busca en *dblf* el primer subespacio disponible que sea suficientemente grande para el paquete. La búsqueda tiene en cuenta que el subespacio debe ser el más profundo, más bajo y más a la izquierda posible donde el paquete pueda caber. En la línea 6: Si tiene una condicional por si se encuentra o no un subespacio adecuado. De continuar con el procedimiento En la línea 7: La función *colocar* ubica el *paquete* en la posición más profunda, más baja y más a la izquierda del *SubespacioOptimo* y devuelve la posición del paquete en el *Contenedor*, la función *add* agrega el valor devuelto a la lista *Contenedor*. En la línea 8: El subespacio original *SubespacioOptimo* donde se colocó el paquete se divide hasta en tres nuevos subespacios menores usando la función *dividir*. En la línea 9: se elimina el subespacio original *SubespacioOptimo* de *dblf* y en la línea 10: se agregan los nuevos subespacios a la lista *dblf*. En la línea 12: Si no se encuentra un subespacio

adecuado para el paquete, se imprime un mensaje de error. Finalmente, en la línea 15: se retorna la lista *Contenedor* con los paquetes colocados.

En el contexto del llenado manual de contenedores, la propuesta de algoritmo presentado no es suficiente ya que no considera las restricciones propias de un llenado manual, por ejemplo un paquete más grande podría ser colocado encima de varios paquetes más pequeños. Por lo tanto se proponen adaptaciones al algoritmo DBLF para adaptarlo a las restricciones de un llenado manual de contenedores.

4.2.2. Unión de subespacios

El primer cambio a considerar es la posibilidad que un paquete pueda ser colocado encima de otros paquetes, en el algoritmo DBLF presentado, un paquete no podía ser colocado encima de otro paquete debido a que los subespacios superiores están separados, por lo tanto se propone una estrategia de unión de subespacios similares. Por ejemplo en la figura 16 se muestra como se unen dos subespacios superiores para permitir que un paquete de otro tipo pueda ser colocado encima de otros paquetes.

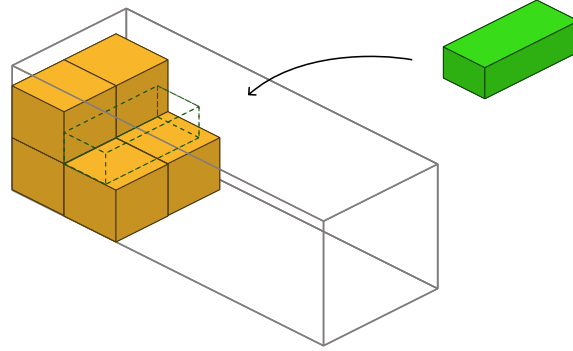


Figura 16: Ejemplo de unión de dos subespacios superiores para permitir que un paquete pueda ser colocado encima de otros paquetes.

Para realizar la unión de subespacios superiores, se propone un algoritmo de unión de subespacios que se detalla en el Algoritmo 2. Este algoritmo recibe como entrada la lista de subespacios disponibles y recorre dicha lista de atrás hacia adelante, buscando subespacios contiguos y similares para unirlos en un solo subespacio. El algoritmo se detiene cuando no se encuentran más subespacios para unir.

Algoritmo 2 Algoritmo de unión de subespacios

```

1: Subespacios  $\leftarrow$  lista de subespacios disponibles
2:  $i \leftarrow$  longitud de Subespacios - 1
3: while  $i > 0$  do
4:   if Subespacios[ $i$ ].esSimilar(Subespacios[ $i - 1$ ]) then
5:     Subespacios[ $i - 1$ ].unir(Subespacios[ $i$ ])
6:     Subespacios.remove(Subespacios[ $i$ ])
7:   end if
8:    $i \leftarrow i - 1$ 
9: end while
10: return Subespacios

```

En el Algoritmo 2, en la línea 1: se inicializa una lista llamada *Subespacios* que contiene los

subespacios disponibles en el contenedor. En la línea 2: se inicializa una variable i con la longitud de la lista de subespacios menos uno, esto para iterar siempre el último con el anterior. En la línea 3: Se inicia un ciclo while que recorre la lista de subespacios desde el último hasta el primero. En la línea 4: Se verifica si el subespacio actual y el subespacio anterior son similares, es decir si comparten ciertas características de posición en el contenedor y tamaño. En la línea 5: Si los subespacios son similares, se unen en un solo subespacio y se elimina el subespacio actual de la lista. En la línea 6: Se decrementa el valor de i en uno. En la línea 8: Se retorna la lista de subespacios actualizada.

4.2.3. Eliminación de subespacios inaccesibles

En el contexto del llenado manual un espacio se vuelve inaccesible cuando un operador no puede colocar un paquete en dicho espacio debido a que fue bloqueado por otro paquete, en la figura 17 se muestra un ejemplo de un subespacio inaccesible.

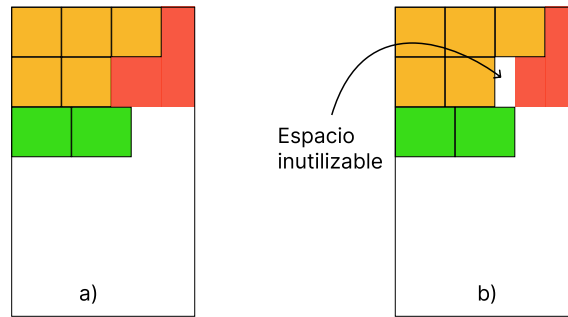


Figura 17: Ejemplo de un subespacio inaccesible.

La figura 17 muestra desde una perspectiva superior del contenedor, en a) espacios libres en rojo que ha sido bloqueado por un paquete verde colocado, en b) este espacio inaccesible no podrá ser utilizado en su totalidad y se partirá para que quede solo la parte accesible.

Para evitar que un subespacio inaccesible sea considerado en el proceso de llenado, se propone un algoritmo de eliminación de subespacios inaccesibles que se detalla en el Algoritmo 3. Este algoritmo recibe como entrada la lista de subespacios disponibles y recorre dicha lista desde el último hacia el primero, eliminando o recortando los subespacios inaccesibles.

Algoritmo 3 Algoritmo de eliminación de subespacios inaccesibles

```

1: Subespacios  $\leftarrow$  lista de subespacios disponibles
2:  $i \leftarrow$  longitud de Subespacios  $- 1$ 
3: while  $i > 0$  do
4:   if Subespacios[ $i$ ].esInaccesibleParcialmente() then
5:     Subespacios[ $i$ ].recortar()
6:   else if Subespacios[ $i$ ].esInaccesibleTotalmente() then
7:     Subespacios.remove(Subespacios[ $i$ ])
8:   end if
9:    $i \leftarrow i - 1$ 
10: end while
11: return Subespacios

```

En el Algoritmo 3, en la línea 1: se inicializa una lista llamada *Subespacios* que contiene los subespacios disponibles en el contenedor. En la línea 2: se inicializa una variable i con la longitud de la lista de subespacios menos uno, esto para iterar siempre desde el último ya que la lista podría

ser modificada durante la ejecución del bucle. En la línea 3: Se inicia un ciclo while que recorre la lista de subespacios desde el último hasta el primero. En la línea 4: Se verifica si el subespacio actual es inaccesible parcialmente, es decir si un paquete bloquea parcialmente el subespacio. En la línea 5: Si el subespacio es inaccesible parcialmente, se recorta el subespacio para eliminar la parte inaccesible. En la línea 6: Se verifica si el subespacio actual es inaccesible totalmente, es decir si un paquete bloquea completamente el subespacio. En la línea 7: Si el subespacio es inaccesible totalmente, se elimina el subespacio de la lista. En la línea 8: Se decrementa el valor de i en uno. En la línea 10: Se retorna la lista de subespacios actualizada.

4.2.4. Eliminación de subespacios profundos

Un espacio profundo se considera inaccesible cuando un operador no puede alcanzar dicho espacio usando sus brazos, en este caso la distancia máxima que una persona puede alcanzar con sus brazos es una constante a definir en el sistema ya que podría usarse un valor promedio que no resulte en un esfuerzo excesivo para el operador humano. Una estrategia para evitar que un espacio profundo sea considerado en el proceso de llenado es recortar la parte posterior del espacio, en la figura 18 se muestra un ejemplo de un subespacio profundo.

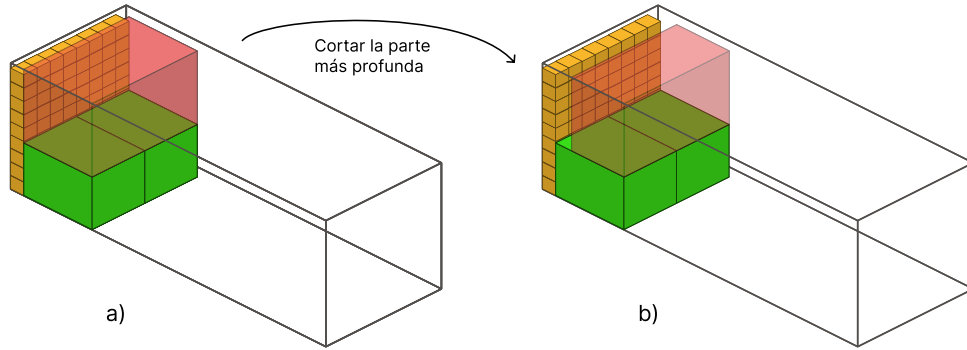


Figura 18: Ejemplo de un subespacio profundo.

La figura 18 muestra desde una perspectiva lateral del contenedor, en a) un espacio profundo en rojo, en b) este espacio profundo ha sido recortado para solo ser considerado la parte frontal accesible.

Para evitar que un subespacio profundo sea considerado en el proceso de llenado, se propone un algoritmo que se detalla en el Algoritmo 4. Este algoritmo recibe como entrada la posición de la caja más cercana a la puerta del contenedor para calcular la posición máxima que un operador puede alcanzar con sus manos, y también recibe la lista de subespacios disponibles, recorre dicha lista desde el último hacia el primero, eliminando o recortando los subespacios profundos.

Algoritmo 4 Algoritmo de eliminación de subespacios profundos

```
1: Subespacios  $\leftarrow$  lista de subespacios disponibles
2: PosicionCajaMasCercana  $\leftarrow$  posición de la caja más cercana a la puerta
3: PosicionMaxima  $\leftarrow$  posición máxima que un operador puede alcanzar
4: i  $\leftarrow$  longitud de Subespacios - 1
5: while i > 0 do
6:   if Subespacios[i].esProfundoParcialmente(PosicionMaxima) then
7:     Subespacios[i].recortar()
8:   else if Subespacios[i].esProfundoTotalmente(PosicionMaxima) then
9:     Subespacios.remove(Subespacios[i])
10:  end if
11:  i  $\leftarrow$  i - 1
12: end while
13: return Subespacios
```

En el Algoritmo 4, en la línea 1: se inicializa una lista llamada *Subespacios* que contiene los subespacios disponibles en el contenedor. En la línea 2: se inicializa una variable *PosicionCajaMasCercana* con la posición de la caja más cercana a la puerta del contenedor contando el largo de la caja, el cuál daría el punto más cercano a la puerta del contenedor. En la línea 3: se inicializa una variable *PosicionMaxima* con la posición máxima que un operador puede alcanzar con sus brazos, se calcula usando el punto más cercano a la puerta del contenedor menos la distancia establecida que los brazos de un operador puede alcanzar. En la línea 4: se inicializa una variable *i* con la longitud de la lista de subespacios menos uno, esto para iterar siempre desde el último ya que la lista podría ser modificada durante la ejecución del bucle. En la línea 5: Se inicia un ciclo while que recorre la lista de subespacios desde el último hasta el primero. En la línea 6: Se verifica si el subespacio actual es profundo parcialmente. En la línea 7: Si el subespacio es profundo parcialmente, se recorta el subespacio para eliminar la parte profunda y se mantiene la parte más frontal accesible. En la línea 8: Se verifica si el subespacio actual es profundo totalmente. En la línea 9: Si el subespacio es profundo totalmente, se elimina el subespacio completamente de la lista. En la línea 10: Se decrementa el valor de *i* en uno. En la línea 12: Se retorna la lista de subespacios actualizada.

4.2.5. Algoritmo de llenado manual adaptado

El Algoritmo 5 muestra el procedimiento de llenado manual de paquetes en un contenedor considerando que los paquetes se reciben por tipos además usando las estrategias de unión de subespacios, eliminación de subespacios inaccesibles y eliminación de subespacios profundos.

Algoritmo 5 Algoritmo de llenado manual de paquetes en un contenedor adaptado

```

1: Parámetros: Tipos  $\leftarrow$  lista de tipos de paquetes
2: Inicialización: dblf  $\leftarrow$  lista inicializada con el espacio total del contenedor
3: Contenedor  $\leftarrow$  lista vacía para almacenar los paquetes colocados
4: for tipo  $\in$  Tipos do
5:   for i  $\leftarrow$  1 to tipo.cantidad do
6:     SubespacioOptimo  $\leftarrow$  buscar el subespacio más adecuado en dblf
7:     if SubespacioOptimo  $\neq$  null then
8:       Contenedor.add(colocar(tipo, SubespacioOptimo))
9:       NuevosSubespacios  $\leftarrow$  dividir(SubespacioOptimo, tipo)
10:      dblf.remove(SubespacioOptimo)
11:      dblf.extend(NuevosSubespacios)
12:     else
13:       print "No se encontró espacio para el paquete."
14:     end if
15:   end for
16:   dblf  $\leftarrow$  unirSubespacios(dblf)
17:   dblf  $\leftarrow$  eliminarSubespaciosInaccesibles(dblf)
18:   dblf  $\leftarrow$  eliminarSubespaciosProfundos(dblf)
19: end for
20: return Contenedor

```

En el Algoritmo 5, en la línea 1: se recibe una lista llamada *Tipos* que contiene los tipos de paquetes cuya información incluye el tamaño, rotación y cantidad de paquetes. En la línea 2: Se inicializa una lista llamada *dblf*, que representa los subespacios libres en el *Contenedor*. Inicialmente, esta lista contiene un único subespacio que es el contenedor entero. En la línea 3: Se crea una lista vacía llamada *Contenedor* donde se almacenarán la posición y tamaños de los paquetes que se vayan colocando. Para la parte del proceso de llenado, en la línea 4: El ciclo for recorre cada tipo de paquete en la lista de tipos. En la línea 5: Se inicia un ciclo for que recorre la cantidad de paquetes por tipo. En la línea 6: Se busca en *dblf* el primer subespacio disponible que sea suficientemente grande para el paquete. La búsqueda tiene en cuenta que el subespacio debe ser el más profundo, más bajo y más a la izquierda posible donde el paquete pueda caber. En la línea 7: Si tiene una condicional por si se encuentra o no un subespacio adecuado. De continuar con el procedimiento En la línea 8: La función *colocar* ubica el *tipo* en la posición más profunda, más baja y más a la izquierda del *SubespacioOptimo* y devuelve la posición del paquete en el *Contenedor*, la función *add* agrega el valor devuelto a la lista *Contenedor*. En la línea 9: El subespacio original *SubespacioOptimo* donde se colocó el paquete se divide hasta en tres nuevos subespacios menores usando la función *dividir*. En la línea 10: se elimina el subespacio original *SubespacioOptimo* de *dblf* y en la línea 11: se agregan los nuevos subespacios a la lista *dblf*. En la línea 16: Se unen los subespacios similares en *dblf* usando la función *unirSubespacios*. En la línea 17: Se eliminan los subespacios inaccesibles en *dblf* usando la función *eliminarSubespaciosInaccesibles*. En la línea 18: Se eliminan los subespacios profundos en *dblf* usando la función *eliminarSubespaciosProfundos*. En la línea 20: Se retorna la lista *Contenedor* con los paquetes colocados.

4.2.6. Mejora adicional al algoritmo de llenado

Debido a la naturaleza altamente aleatoria propias de un algoritmo genético, es posible que se generen soluciones que se encuentren muy próximas de ser muy buenas pero que no evaluaron los espacios locales de manera adecuada, por lo que se propone un ajuste al algoritmo de llenado manual para que pueda aumentar la cantidad de cajas cuando sea posible de tal modo que se pueda aprovechar al máximo el espacio disponible.

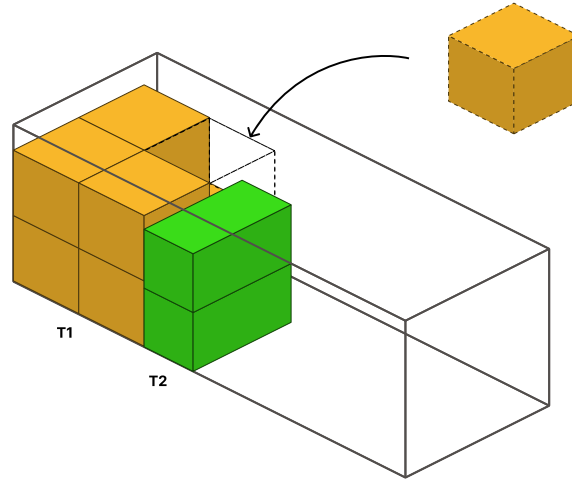


Figura 19: Ejemplo de llenado adicional de un paquete tipo T1 donde ya se ha llenado dos tipos T1 (7 Cajas) y T2 (2 Cajas).

Por ejemplo, en la Figura 19 se ilustra un caso donde se ha completado el llenado de un tipo T1 específico con 7 paquetes y un tipo T2 con 2 paquetes, pero aún queda un espacio no usado donde podría ser aprovechado para añadir un octavo paquete del mismo tipo T1, siempre y cuando aún dispongamos de paquetes adicionales de ese tipo.

Esta situación presenta dos posibles mejoras para la gestión de paquetes. La primera consiste en añadir un paquete justo después de completar la carga de paquetes del mismo tipo. Esta opción podría optimizar el uso del espacio disponible de forma más eficiente, aunque existe el riesgo de que pueda deteriorar la calidad de la solución, ya que podría inferir en la colocación de los siguientes tipos de cajas. La segunda mejora propuesta es añadir el paquete una vez que se hayan llenado completamente todos los paquetes de todos los tipos. Si bien esta estrategia asegura que la calidad de la solución no se vea comprometida, podría no siempre haber muchos espacios libres que puedan ser aprovechados.

La primera mejora es llamada *llenado adicional inmediato* y denominada M1, y la segunda mejora es llamada *llenado adicional final* y denominada M2. Ambas mejoras solo se pueden aplicar en circunstancias específicas durante la ejecución del algoritmo genético y su rendimiento será evaluado en la sección de experimentos.

A continuación se resume el algoritmo de llenado manual adaptado con las mejoras M1 en el Algoritmo 6.

Algoritmo 6 Algoritmo de llenado manual de paquetes en un contenedor adaptado con la mejora M1

```

1: Parámetros: Tipos  $\leftarrow$  lista de tipos de paquetes
2: Inicialización: dblf  $\leftarrow$  lista inicializada con el espacio total del contenedor
3: Contenedor  $\leftarrow$  lista vacía para almacenar los paquetes colocados
4: for tipo  $\in$  Tipos do
5:   for i  $\leftarrow$  1 to tipo.cantidad do
6:     ... ▷ Mismo procedimiento que en el Algoritmo 5
7:     if i = tipo.cantidad then
8:       dblfSuperior  $\leftarrow$  filtrar solo subespacios superiores en dblf
9:       SubespacioLibre  $\leftarrow$  buscar el subespacio más adecuado en dblfSuperior
10:      if SubespacioLibre  $\neq$  null then
11:        tipo.cantidad  $\leftarrow$  tipo.cantidad + 1
12:      end if
13:    end if
14:  end for
15:  ... ▷ Mismo procedimiento que en el Algoritmo 5
16: end for
17: return Contenedor

```

El Algoritmo 6 es similar al Algoritmo 5 con la diferencia de que se añade un paquete adicional del mismo tipo justo después de completar la carga de paquetes del mismo tipo. En la línea 7: Se verifica si el paquete actual es el último paquete del tipo, si es así, en la línea 8: se filtran solo los subespacios superiores en *dblf* y en la línea 9: se busca el subespacio más adecuado en *dblfSuperior*. En la línea 10: Si se encuentra un subespacio adecuado, en la línea 11: se incrementa la cantidad de paquetes del tipo en uno, esto para que el bucle no termina hasta que se haya colocado el paquete adicional. En las siguientes líneas se realiza el mismo procedimiento que en el Algoritmo 5. En la línea 17: Se retorna la lista *Contenedor* con los paquetes colocados.

Para el caso de la mejora M2 se sigue un procedimiento distinto, ya que se añade un paquete adicional una vez que se han llenado completamente todos los paquetes de todos los tipos. A continuación se resume el algoritmo de llenado manual adaptado con la mejora M2 en el Algoritmo 7.

Algoritmo 7 Algoritmo de llenado manual de paquetes en un contenedor adaptado con la mejora M2

```

1: Parámetros: Tipos  $\leftarrow$  lista de tipos de paquetes
2: Inicialización: dblf  $\leftarrow$  lista inicializada con el espacio total del contenedor
3: Contenedor  $\leftarrow$  lista vacía para almacenar los paquetes colocados
4: EspaciosSobrantes  $\leftarrow$  lista vacía para almacenar los espacios sobrantes
5: for tipo  $\in$  Tipos do
6:   for i  $\leftarrow$  1 to tipo.cantidad do
7:     ... ▷ Mismo procedimiento que en el Algoritmo 5
8:   end for
9:   if existeEspacioSobrante(dblf) then
10:    EspaciosSobrantes  $\leftarrow$  filtrar solo subespacios sobrantes en dblf
11:   end if
12: end for
13: ...
14: for tipo  $\in$  Tipos do
15:   for i  $\leftarrow$  tipo.cantidad to tipo.cantidadMaxima do
16:    SubespacioLibre  $\leftarrow$  buscar el subespacio más adecuado en EspaciosSobrantes
17:    if SubespacioLibre  $\neq$  null then
18:      Contenedor.add(colocar(tipo, SubespacioLibre))
19:      tipo.cantidad  $\leftarrow$  tipo.cantidad + 1
20:    end if
21:   end for
22: end for
23: ... ▷ Mismo procedimiento que en el Algoritmo 5
24: return Contenedor

```

El Algoritmo 7 también es similar al Algoritmo 5 con la diferencia de que se durante todo el procedimiento de llenado se van recopilando los espacios libres para que luego se añada un paquete adicional una vez que se han llenado completamente todos los paquetes de todos los tipos. En la línea 9: Se verifica si existe algún espacio sobrante en *dblf*, si es así, en la línea 10: se filtran solo los subespacios sobrantes en *dblf* y se almacenan en *EspaciosSobrantes*. En las siguientes líneas se realiza el mismo procedimiento que en el Algoritmo 5. En la línea 14: Se recorre nuevamente la lista de tipos de paquetes y en la línea 15: se recorre la cantidad de paquetes que faltan por añadir. En la línea 16: Se busca el subespacio más adecuado en *EspaciosSobrantes*. En la línea 17: Si se encuentra un subespacio adecuado, en la línea 18: se añade el paquete al contenedor y en la línea 19: se actualiza la cantidad de paquetes del tipo. En las siguientes líneas se realiza el mismo procedimiento que en el Algoritmo 5. En la línea 24: Se retorna la lista *Contenedor* con los paquetes colocados.

Como se ha podido observar en los algoritmos presentados, si bien ambos tipos de mejoras tienen la misma finalidad de añadir paquetes adicionales al contenedor, cada una tiene sus propias ventajas y desventajas, por lo que se propone evaluar el rendimiento de cada una de ellas en la sección de experimentos.

4.3. Población inicial

Para generar la población inicial se propone un mecanismo simple de generación de soluciones aleatorias. Dado un conjunto de tipos de paquetes $T = \{1, 2, \dots, n\}$, donde cada tipo de paquete $t_i \in T$ tiene un mínimo y un máximo de paquetes a insertar en el contenedor $t_i \in [t_i^{min}, t_i^{max}]$, se generan soluciones aleatorias de la siguiente manera:

1. Se selecciona aleatoriamente un tipo de paquete $t_i \in T$.

2. Se selecciona aleatoriamente un tipo de rotación $r_i \in R, R \in \{0, 1\}$.
3. Se selecciona aleatoriamente una cantidad de paquetes $q_i \in [t_i^{min}, t_i^{max}]$.
4. Se repiten los pasos 1 al 3 hasta que se hayan seleccionado todos los tipos de paquetes.
5. Se repiten los pasos 1 al 4 hasta que se hayan generado N soluciones.

El algoritmo de generación de la población inicial se muestra en el Algoritmo 8. En este algoritmo, T es el conjunto de tipos de paquetes, R es el conjunto de rotaciones, N es el tamaño de la población, t_i^{min} y t_i^{max} son el mínimo y el máximo de paquetes de cada tipo, respectivamente, y P es la población inicial.

Algoritmo 8 Generación de la población inicial

Require: T, R, N

Ensure: P

```

1:  $P \leftarrow \emptyset$ 
2: for  $i = 1$  to  $N$  do
3:    $S \leftarrow \emptyset$ 
4:   for each  $t_i \in T$  do
5:      $r \leftarrow$  rotación aleatoria de  $R$ 
6:      $q \leftarrow$  cantidad aleatoria de paquetes entre  $t_i^{min}$  y  $t_i^{max}$ 
7:      $S \leftarrow S \cup \{(t_i, r, q)\}$ 
8:   end for
9:    $P \leftarrow P \cup \{S\}$ 
10: end for
11: return  $P$ 

```

En el Algoritmo 8, en la línea 1: se inicializa una lista vacía P que contendrá la población inicial. En el ciclo for de la línea 2: se itera para generar N soluciones. En la línea 3: se inicializa una lista vacía S que representa una solución. En el ciclo for de la línea 4: se itera para seleccionar aleatoriamente un tipo de paquete del conjunto de tipos T . En las línea 5: se selecciona aleatoriamente una rotación del conjunto de rotaciones R . En la línea 6: se selecciona aleatoriamente una cantidad de paquetes de acuerdo al mínimo y máximo de paquetes de cada tipo. En la línea 7: se añade el tipo de paquete, la rotación y la cantidad de paquetes a la solución S . En la línea 9: se añade la solución a la población. En la línea 11: se retorna la población generada.

De esta manera, las soluciones aleatorias generadas respetan las restricciones específicas del problema. Esto asegura que la cantidad de cada tipo de paquete insertado en el contenedor cumple con los rangos establecidos, manteniéndose entre el mínimo y el máximo definidos. Adicionalmente, se establece que estas cantidades sean generadas como números enteros, manteniendo la integridad de las soluciones conforme a las necesidades del problema.

4.4. Selección

Para seleccionar las soluciones que se utilizarán en cada generación del algoritmo evolutivo, se propone un mecanismo de selección basado en el método de torneo usando un tamaño de torneo de 2, también conocido como torneo binario. Anand y otros [3] lo describen como uno de los más eficientes métodos de selección para algoritmos evolutivos. Para usar el método de torneo binario, se seleccionan aleatoriamente dos soluciones de la población y se selecciona la mejor solución de las dos. Este proceso se lleva a cabo junto con el operador de cruce para lo cual cada padre se selecciona mediante un torneo binario.

En el Algoritmo 9, P es la población, N es el tamaño de la población, $P1$ y $P2$ son los padres seleccionados.

Algoritmo 9 Selección de padres

Require: P, N

Ensure: $P1, P2$

```

1:  $P1 \leftarrow \emptyset$ 
2:  $P2 \leftarrow \emptyset$ 
3: for  $i = 1$  to  $N$  do
4:    $S1, S2 \leftarrow$  soluciones aleatorias de  $P$ 
5:   if  $S1 > P1$  then
6:      $P1 \leftarrow P1 \cup S1$ 
7:   else
8:      $P1 \leftarrow P1 \cup S2$ 
9:   end if
10:   $S3, S4 \leftarrow$  soluciones aleatorias de  $P$ 
11:  if  $S3 > P2$  then
12:     $P2 \leftarrow P2 \cup S3$ 
13:  else
14:     $P2 \leftarrow P2 \cup S4$ 
15:  end if
16: end for
17: return  $P1, P2$ 

```

En el Algoritmo 9, en la línea 1 y 2: se inicializan dos listas vacías $P1$ y $P2$ que contendrán los padres seleccionados. En el ciclo for de la línea 3: se itera para todo el tamaño de la población N . En las líneas 4 a la 9: se selecciona la mejor solución de las dos y se añade a la lista $P1$. En las líneas de la 10 a la 15: se vuelve a repetir el proceso para seleccionar el segundo padre y se añade a la lista $P2$. En la línea 17: se retornan los padres seleccionados.

De acuerdo a lo anterior, se generan dos listas de padres que se utilizarán en el operador de cruce para generar una nueva población. Este método de selección procura que siempre se tienda a seleccionar soluciones mejores aunque tampoco se descartan soluciones peores, lo que permite mantener la diversidad en la población.

4.5. Cruce

Como se vió en la selección de padres, se generaron dos listas $P1$ y $P2$ que se utilizarán en el operador de cruce para generar una nueva población. Para el operador de cruce se propone un mecanismo de cruce basado en el cruce de un punto. Según Umbarkar y Sheth (2015), para este tipo de operador, se selecciona un punto aleatorio en el vector solución y se intercambian las partes de las dos soluciones que están a la izquierda y a la derecha del punto. Este operador de cruce es simple y fácil de implementar, y se ha demostrado que es eficiente para resolver problemas de optimización combinatoria.

Es importante destacar que el operador de cruce se aplica a una fracción específica de la población, determinada por una tasa de cruce, P_{CROSS} . Esta tasa representa la probabilidad de que dos soluciones seleccionadas sean cruzadas. La implementación de este mecanismo promueve la diversidad dentro de la población y fomenta la generación de nuevas soluciones potencialmente superiores a las existentes.

En el Algoritmo 10, $P1$ y $P2$ son los padres seleccionados, $C1$ y $C2$ son los hijos generados, N es el

tamaño de la población y T es el conjunto de tipos de paquetes.

Algoritmo 10 Cruce de padres genérico

Require: $P1, P2, N, T$

Ensure: $C1, C2$

```

1:  $C1 \leftarrow \emptyset$ 
2:  $C2 \leftarrow \emptyset$ 
3:  $p \leftarrow$  punto aleatorio en la solución
4: for  $i = 1$  to  $N$  do
5:   if  $i \leq p$  then
6:      $C1 \leftarrow C1 \cup P1[i]$ 
7:      $C2 \leftarrow C2 \cup P2[i]$ 
8:   else
9:      $C1 \leftarrow C1 \cup P2[i]$ 
10:     $C2 \leftarrow C2 \cup P1[i]$ 
11:   end if
12: end for
13: return  $C1, C2$ 

```

Por lo tanto, al intentar aplicar el algoritmo sobre la Codificación de Soluciones propuesta en la sección 4.1, se puede representar en la Figura 20 el cruce de dos padres $P1$ y $P2$.

P1	Tipo	2	5	4	7	8	1	6	3
	Rotación	0	1	1	0	0	1	1	0
	Cantidad	27	10	18	13	17	31	14	9
P2	Tipo	4	8	2	6	1	3	5	7
	Rotación	1	1	0	1	0	0	1	0
	Cantidad	15	18	13	21	24	10	11	8

Figura 20: Cruce de dos padres $P1$ y $P2$

Como puede verse en la Figura 20, se selecciona un punto aleatorio en la solución (en este caso se encuentra en la quinta posición) y si se intercambian las partes de las dos soluciones que están a la izquierda y a la derecha del punto, las soluciones hijas resultantes tendrían una alta probabilidad de ser no factibles ya que se estarían repitiendo tipos de paquetes en el contenedor.

Si bien este algoritmo de cruce es ampliamente aplicable a diversos tipos de problemas, es particularmente adaptable al problema de llenado de contenedores, teniendo en cuenta la codificación propuesta en la sección 4.1. Se propone, por tanto, un mecanismo de cruce diseñado para asegurar la factibilidad de las soluciones generadas. Este método consiste en utilizar la primera partición de la solución de un progenitor y completarla con los tipos de paquetes aún no seleccionados de la misma partición. De esta manera, se asegura que cada solución generada, o hijo, contenga todos los tipos de paquetes, garantizando su viabilidad.

En la Figura 21 se muestra el cruce de dos padres $P1$ y $P2$ utilizando el algoritmo de cruce propuesto.

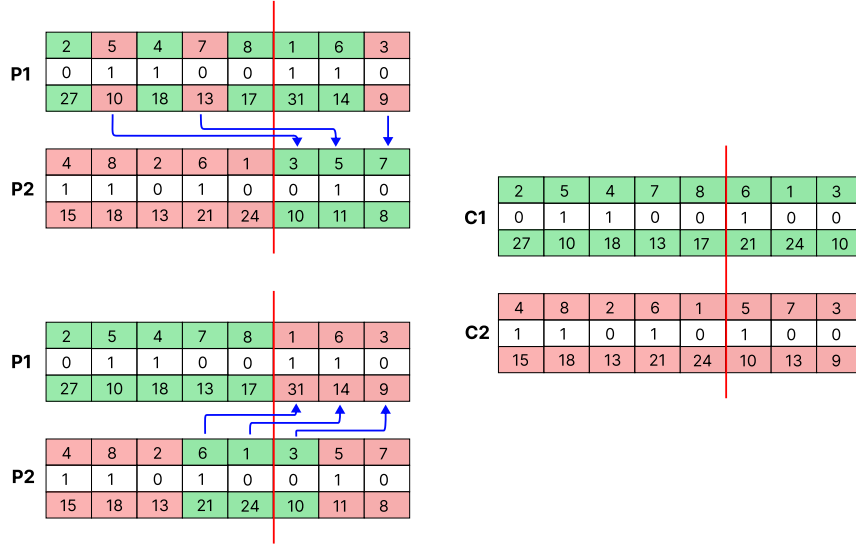


Figura 21: Cruce de dos padres $P1$ y $P2$ utilizando el algoritmo de cruce propuesto

Como se ilustra en la Figura 21, el proceso de cruce se detalla para los padres $P2$ y $P1$. A $P2$ le faltan los tipos de paquetes $T3$, $T5$ y $T7$; tras buscar en $P1$, se agregan en el orden encontrado: $T5$, $T7$, $T3$. En el caso de $P1$, le faltan los tipos $T1$, $T6$ y $T3$, que al buscarse en $P2$ se encuentran y se añaden en el orden de $T6$, $T1$ y $T3$. Este método asegura que las soluciones generadas, $C1$ y $C2$, sean factibles al completar los tipos de paquetes faltantes en cada padre, manteniendo así la viabilidad de las soluciones.

Por lo tanto, el algoritmo propuesto, adaptado a la codificación, para el cruce de padres se muestra en el Algoritmo 11.

Algoritmo 11 Cruce de padres propuesto

Require: $P1, P2, N, T$

Ensure: $C1, C2$

```

1:  $C1 \leftarrow \emptyset$ 
2:  $C2 \leftarrow \emptyset$ 
3:  $p \leftarrow$  punto aleatorio en la solución
4: for  $i = 1$  to  $N$  do
5:   if  $i \leq p$  then
6:      $C1 \leftarrow C1 \cup P1[i]$ 
7:      $C2 \leftarrow C2 \cup P2[i]$ 
8:   else
9:     for each  $t_i \in T$  do
10:      if  $C1$  no contiene  $P2[i]$  then
11:         $C1 \leftarrow C1 \cup P2[i]$ 
12:      end if
13:      if  $C2$  no contiene  $P1[i]$  then
14:         $C2 \leftarrow C2 \cup P1[i]$ 
15:      end if
16:    end for
17:   end if
18: end for
19: return  $C1, C2$ 

```

En el Algoritmo 11, en las líneas 1 y 2 se inicializan dos listas vacías $C1$ y $C2$ que contendrán los

hijos generados. En la línea 3 se selecciona un punto aleatorio p en la solución de tamaño N . En el ciclo for de la línea 4 se itera para todo el tamaño de la población N . En las líneas de la 5 a la 8 se conservan las primeras p posiciones de los padres $P1$ y $P2$ respectivamente. En la línea 9 se inicia un ciclo para cada tipo de paquete $t_i \in T$. En las líneas de la 10 a la 12 se verifica si el hijo $C1$ no contiene el tipo de paquete $P2[i]$ y si es así se añade a $C1$. En las líneas de la 13 a la 15 se verifica si el hijo $C2$ no contiene el tipo de paquete $P1[i]$ y si es así se añade a $C2$. En la línea 19 se retorna los dos hijos generados $C1$ y $C2$.

Consecuentemente, se asegura que las soluciones generadas no solo sean factibles y abarquen todos los tipos de paquetes, sino que también difieran de las soluciones padres, contribuyendo así a la preservación de la diversidad dentro de la población.

4.6. Mutación

Según Dockhorn y Lucas (2022), la mutación consiste en aplicar pequeños cambios a las soluciones candidatas para tratar de explorar nuevas regiones locales del espacio de búsqueda. Para el presente problema, ya que codificación propuesta consta de tres listas, se propone también tres operadores de mutación, uno para cada lista. El primer tipo de mutación intercambia dos tipos de paquetes en la secuencia de llenado junto a su rotación y cantidad específica, el segundo tipo de mutación solo intercambia la rotación de un tipo de paquete y el tercer tipo de mutación cambia la cantidad de paquetes de un tipo de paquete.

Cabe mencionar que el operador de mutación se aplica a una proporción muy pequeña de la población, dependiendo de una tasa de mutación P_{MUT} , que se define como la probabilidad de que una solución seleccionada mute y genere una nueva solución cercana a la inicial.

En la Figura 22 se muestra un ejemplo de mutación de un tipo de paquete en la secuencia de llenado, se elige dos tipos de paquetes aleatorios y se intercambian.

Tipo	2	5	4	7	8	1	6	3
P1 Rotación	0	1	1	0	0	1	1	0
Cantidad	27	10	18	13	17	31	14	9

Tipo	2	5	1	7	8	4	6	3
P1' Rotación	0	1	1	0	0	1	1	0
Cantidad	27	10	31	13	17	18	14	9

Figura 22: Mutación de un tipo de paquete en la secuencia de llenado

En la Figura 23 se muestra un ejemplo de mutación de la rotación de un tipo de paquete en la secuencia de llenado, se elige un tipo de paquete aleatorio y se cambia la rotación del estado 0 (no rotado) hacia el estado 1 (rotado).

Tipo	2	5	4	7	8	1	6	3
P1 Rotación	0	1	1	0	0	1	1	0
Cantidad	27	10	18	13	17	31	14	9

Tipo	2	5	4	7	8	1	6	3
P1' Rotación	0	1	1	0	1	1	1	0
Cantidad	27	10	18	13	17	31	14	9

Figura 23: Mutación de la rotación de un tipo de paquete en la secuencia de llenado

En la Figura 24 se muestra un ejemplo de mutación de la cantidad de paquetes de un tipo de paquete en la secuencia de llenado, se elige un tipo de paquete aleatorio y se cambia la cantidad de paquetes.

	Tipo	2	5	4	7	8	1	6	3
P1	Rotación	0	1	1	0	0	1	1	0
	Cantidad	27	10	18	13	17	31	14	9

	Tipo	2	5	4	7	8	1	6	3
P1'	Rotación	0	1	1	0	1	1	1	0
	Cantidad	27	10	20	13	17	31	14	9

Figura 24: Mutación de la cantidad de paquetes de un tipo de paquete en la secuencia de llenado

Aunque estos tres tipos de mutación no se pueden aplicar a la vez, se puede seleccionar el tipo de mutación aleatoriamente y aplicarla a la solución seleccionada. Por lo tanto, el algoritmo de mutación se muestra en el Algoritmo 12. En este algoritmo, S es la solución a mutar, T es el conjunto de tipos de paquetes, R es el conjunto de rotaciones, t_i^{\min} y t_i^{\max} son el mínimo y el máximo de paquetes de cada tipo, respectivamente, y S' es la solución mutada.

Algoritmo 12 Mutación de soluciones

Require: $S, T, R, t_i^{\min}, t_i^{\max}$

Ensure: S'

```

1:  $S' \leftarrow S$ 
2:  $m \leftarrow$  número aleatorio entre 1 y 3
3: if  $m = 1$  then
4:    $i, j \leftarrow$  dos números aleatorios entre 1 y  $|T|$ 
5:    $S'[i], S'[j] \leftarrow S[j], S[i]$ 
6: else if  $m = 2$  then
7:    $i \leftarrow$  número aleatorio entre 1 y  $|T|$ 
8:    $S'[i] \leftarrow$  intercambio de rotación de  $S[i]$ 
9: else if  $m = 3$  then
10:   $i \leftarrow$  número aleatorio entre 1 y  $|T|$ 
11:   $q \leftarrow$  variar la cantidad paquetes de  $S[i]$ 
12:   $S'[i] \leftarrow q$ 
13: end if
14: return  $S'$ 

```

En el Algoritmo 12, en la línea 1 se inicializa una solución mutada S' con la solución original S . En la línea 2 se selecciona aleatoriamente un número m entre 1 y 3 que representa el tipo de mutación a aplicar. En la línea 3 se verifica si el número aleatorio es igual a 1, si es así, en las líneas 4 y 5 se seleccionan dos números aleatorios i y j entre 1 y el tamaño de T y se intercambian los tipos de paquetes en las posiciones i y j de la solución S' . En la línea 6 se verifica si el número aleatorio es igual a 2, si es así, en la línea 7 se selecciona un número aleatorio i entre 1 y el tamaño de T y en la línea 8 se intercambia la rotación del tipo de paquete en la posición i de la solución S' . En la línea 9 se verifica si el número aleatorio es igual a 3, si es así, en las líneas 10 se selecciona un número aleatorio i entre 1 y el tamaño de T . En la línea 11 se varía la cantidad de paquetes del tipo de paquete en la posición i de la solución S' , esto implica que la cantidad puede incrementar o disminuir. En la línea 14 se retorna la solución mutada S' .

4.7. Elitismo

Según Hasni y Sabri (2013), el principio del elitismo consiste en mantener las mejores soluciones de una generación a la siguiente, lo que permite garantizar que siempre se mantenga la mejor solución encontrada hasta el momento, ya que los operadores de cruce y mutación pueden generar soluciones las cuales no sean mejores que las soluciones actuales, en dicho caso se reemplazan el mejor individuo de la generación anterior por el peor individuo de la generación actual.

Por lo tanto, se propone un mecanismo de elitismo que garantice que siempre se mantenga la mejor solución encontrada hasta el momento. Para ello, se selecciona la mejor solución de la población actual y se compara con la mejor solución élite encontrada hasta el momento. Si la mejor solución de la población actual es mejor que la mejor solución élite encontrada hasta el momento, se sustituye la mejor solución élite por la mejor solución de la población actual, de lo contrario, la solución élite reemplaza a la peor solución de la población actual.

El algoritmo de elitismo se muestra en el Algoritmo 13. En este algoritmo, P es la población actual, S es la mejor solución encontrada hasta el momento y S' es la mejor solución de la población actual.

Algoritmo 13 Elitismo

Require: P, S

Ensure: S'

- 1: $S' \leftarrow$ mejor solución de P
 - 2: **if** $S' > S$ **then**
 - 3: $S \leftarrow S'$
 - 4: **else**
 - 5: $P \leftarrow$ reemplazo de la peor solución de P por S
 - 6: **end if**
 - 7: **return** S
-

En el Algoritmo 13, en la línea 1 se selecciona la mejor solución de la población actual P y se almacena en S' . En la línea 2 se usa la función de evaluación para verificar si la mejor solución de la población actual S' es mejor que la mejor solución encontrada hasta el momento S , si es así, en la línea 3 se reemplaza la mejor solución encontrada hasta el momento S por la mejor solución de la población actual S' . Si la verificación de la línea 2 es falsa, en la línea 5 se reemplaza la peor solución de la población actual P por la mejor solución encontrada hasta el momento S . En la línea 6 se retorna la mejor solución S que se mantendrá en las generaciones futuras.

Como resultado, se asegura la preservación de la mejor solución identificada hasta el momento, evitando su pérdida en futuras generaciones.

4.8. Algoritmo genético

Con todos los operadores definidos, se puede definir el algoritmo genético completo. En el Algoritmo 14, T es el conjunto de tipos de paquetes, R es el conjunto de rotaciones, N es el tamaño de la población, P_{CROSS} es la tasa de cruce, P_{MUT} es la tasa de mutación y G es la condiciones de parada que podría ser una cantidad máxima de tiempo, una cantidad máxima de generaciones en que la mejor solución no ha mejorado o simplemente una cantidad máxima de generaciones en total.

Algoritmo 14 Algoritmo genético

Require: $T, R, N, P_{CROSS}, P_{MUT}, G$ **Ensure:** S

- 1: $P \leftarrow$ generación de la población inicial
 - 2: $S \leftarrow$ mejor solución de P
 - 3: **while** (no se cumpla la condición de parada G) **do**
 - 4: $P1, P2 \leftarrow$ Realizar selección en P
 - 5: $P \leftarrow$ Realizar cruzamiento en $P1$ y $P2$ con tasa P_{CROSS}
 - 6: $P \leftarrow$ Aplicar mutación a P con tasa P_{MUT}
 - 7: $S \leftarrow$ Realizar elitismo en P con S
 - 8: **end while**
 - 9: **return** S
-

En el Algoritmo 14, en la línea 1 se genera la población inicial P . En la línea 2 se usa la función de evaluación para seleccionar la mejor solución de la población inicial P y se almacena en S . En el ciclo while de las líneas 3 a 8 se itera mientras no se cumpla la condición de parada G . En las línea 4 se seleccionan los padres $P1$ y $P2$ de la futura generación y en la línea 5 se realiza el cruce de los padres seleccionados con una tasa de cruce P_{CROSS} . En la línea 6 se aplica la mutación a la población P (que ya paso por el operador de cruce) con una tasa de mutación P_{MUT} . En la línea 7 se realiza el elitismo en la población P (que ya paso por los operadores de cruce y mutación) con la mejor solución élite S . En la línea 9 se retorna la mejor solución S encontrada.

Al finalizar el algoritmo, se obtiene la mejor solución encontrada que representa la mejor forma de llenar el contenedor con los paquetes disponibles de modo que se maximice el valor total de los paquetes colocados y que se cumplan las restricciones prácticas del problema.

5. Estudio experimental

En este capítulo se presenta el diseño y los resultados de un experimento computacional que se realizó para evaluar el rendimiento de los algoritmos propuestos en el capítulo anterior. El objetivo principal de este experimento es comparar la eficacia de los algoritmos genéticos con y sin las mejoras propuestas, en términos de calidad de las soluciones obtenidas y tiempo de ejecución. Para ello, se utilizaron instancias de prueba generadas aleatoriamente. Se analizan los resultados obtenidos y se discuten las implicancias de los mismos en el contexto del problema de llenado de contenedor.

5.1. Generación de datos de prueba

En la literatura, se han propuesto diferentes conjuntos de instancias de prueba para evaluar algoritmos de llenado de contenedores, por ejemplo, Bischoff y Ratcliff (1995) establecieron un método de generación de instancias de prueba que se ha utilizado en varios estudios posteriores, con la finalidad de comparar los resultados obtenidos. Sin embargo, debido a las restricciones de llenado manual de contenedores, las instancias públicas de prueba no son adecuadas para ser usadas en un experimento computacional. La principal diferencia radica en que al usar el procedimiento manual de llenado definido en la sección 3, las soluciones obtenidas siempre serán menos óptimas que las generadas por otros algoritmos que no consideran restricciones reales de llenado. Por lo tanto, se decidió generar instancias de prueba aleatorias, siguiendo un enfoque más realista, que permita evaluar el rendimiento de los algoritmos propuestos en un escenario más cercano a la realidad.

Para generar las instancias de prueba, se consideraron las siguientes restricciones:

- Se define que las unidades de distancias usadas son los milímetros y siempre se redondean al entero más cercano.
- Se establece que se tiene un solo contenedor de tamaño fijo, cuyas dimensiones interiores son $L \times W \times H = 12010 \times 2330 \times 2380$.
- Cada instancias tiene un número fijo de tipos de cajas a llenar, que son 5, 10, 20 y 30, los cuales son denominados como 5T, 10T, 20T y 30T respectivamente.
- Las cajas de tipo 5T están contenidas también en las cajas de tipo 10T, las cajas de tipo 10T están contenidas en las cajas de tipo 20T y las cajas de tipo 20T están contenidas en las cajas de tipo 30T.
- Las dimensiones de las cajas a llenar son generadas aleatoriamente, siguiendo una distribución uniforme en el rango $[250, 750]$ para todas las dimensiones.
- Los valores de las cajas son generados aleatoriamente, siguiendo una distribución uniforme en el rango $[1, 100]$, es importante resaltar que los valores tienen dependencia del tamaño.

Para aumentar la diversidad de las instancias de prueba, se establece que en cada instancia se debe cumplir que la dimensiones de las cajas a generar no se repitan. En el Algoritmo 15 se detalla el procedimiento sencillo para evitar la repetición de las dimensiones de las cajas.

Algoritmo 15 Evitar repetir dimensiones**Require:** $|T|$, cantidad de cajas a generar.**Require:** *RangoDimensiones*, rango de dimensiones.**Ensure:** D , lista de dimensiones.

```

1:  $D \leftarrow \emptyset$ 
2: for  $i \leftarrow 1$  to  $|T|$  do
3:    $d \leftarrow \text{Aleatorio}(\text{RangoDimensiones})$ 
4:   while  $d \in D$  do
5:      $d \leftarrow \text{Aleatorio}(\text{RangoDimensiones})$ 
6:   end while
7:    $D \leftarrow D \cup d$ 
8: end for
9: return  $D$ 

```

En el Algoritmo 15, $|T|$ es la cantidad de cajas a generar, en la línea 1 se inicializa la lista D de dimensiones, en las líneas 2 a 8 se itera sobre la cantidad de cajas a generar, en la línea 3 se genera una dimensión aleatoria, en la línea 4 se verifica si la dimensión generada ya existe en la lista D . Si la dimensión generada ya existe, se vuelve a generar una dimensión aleatoria en la línea 5 y se vuelve a verificar si la dimensión generada ya existe en la lista D . Si la dimensión generada no existe en la lista D , se añade a la lista D en la línea 7. Finalmente, en la línea 9 se retorna la lista D con las dimensiones generadas.

Para determinar la cantidad máxima de cajas de cada tipo que se puede almacenar en un contenedor, se adopta el siguiente enfoque. Primero, se divide el volumen total del contenedor por el número de tipos de cajas especificadas, por ejemplo, cinco tipos de cajas resultarían en dividir el contenedor en cinco partes iguales ("5T"). Luego, se calcula cuántas cajas de cada tipo caben exactamente en la fracción del volumen asignada a ese tipo. Para añadir un nivel adicional de desafío y realismo, se incorpora un porcentaje aleatorio de cajas adicionales a cada tipo, complicando así la planificación del espacio. Este procedimiento se detalla en el Algoritmo 16, el cual ilustra paso a paso cómo generar las instancias de prueba.

Algoritmo 16 Generación de instancias de pruebas**Require:** V_{total} , volumen total del contenedor; T , tipos de cajas.**Ensure:** Q , lista de cantidades por tipo.

```

1:  $Q \leftarrow \emptyset$ 
2:  $v_{\text{fraccion}} \leftarrow V_{\text{total}}/|T|$  ▷ Dividir el volumen total en partes iguales
3: for  $t_i \in T$  do
4:    $v_i \leftarrow w_i * l_i * h_i$  ▷ Volumen de un tipo  $t_i$ 
5:    $q_i \leftarrow \lceil v_{\text{fraccion}}/v_i \rceil$ 
6:    $q_i^{\text{ajustado}} \leftarrow n_i + \text{Aleatorio}(\% \text{adicional})$  ▷ Añadir un porcentaje aleatorio
7:    $Q \leftarrow Q \cup q_i^{\text{ajustado}}$ 
8: end for
9: return  $Q$ 

```

En el Algoritmo 16, V_{total} es el volumen total del contenedor, T es el conjunto de tipos de cajas. En la línea 1, se inicializa la lista Q de cantidades por tipo. En la línea 2, se calcula la fracción del volumen total asignada a cada tipo de caja. En las líneas 3 se iteran sobre la cantidad de tipos de cajas, en la línea 4 se calcula el volumen de un tipo de caja, en la línea 5 se calcula la cantidad aproximada de cajas de un tipo que caben en la fracción del volumen asignada a ese tipo, en la línea 6 se ajusta la cantidad de cajas de un tipo añadiendo un porcentaje aleatorio, en la línea 7 se añade la cantidad ajustada a la lista Q . Finalmente, en la línea 8 se retorna la lista Q con las cantidades ajustadas por tipo.

De este modo se evita el riesgo de que las cantidades de cajas generadas sean muy pocas como para llenar un contenedor pero tampoco demasiadas como para que se necesiten varios contenedores. Además, se garantiza que las cajas generadas no se repitan, lo que aumenta la diversidad de las instancias de prueba.

Por lo tanto, el procedimiento para generar las instancias de prueba es el siguiente:

1. Se genera un conjunto de dimensiones únicas de cajas para cada tipo de caja según el Algoritmo 15.
2. Se asigna un valor aleatorio a cada caja.
3. Se genera un conjunto de cantidades de cajas para cada tipo de caja, según el Algoritmo 16.

El procedimiento anterior se puede resumir en el Algoritmo 17, el cual ilustra paso a paso cómo generar las instancias de prueba.

Algoritmo 17 Generación de instancias de pruebas

Require: V_{total} , volumen total del contenedor; C Cantidad de problemas por tipo T , tipos de cajas.

Ensure: P , lista de instancias de problemas.

```

1:  $P \leftarrow \emptyset$ 
2: for  $t_i \in T$  do
3:   for  $c \leftarrow 1$  to  $C$  do
4:      $D \leftarrow \text{EvitarRepetirDimensiones}(|T|, \text{RangoDimensiones})$ 
5:      $V \leftarrow \text{Aleatorio}(|T|, \text{RangoValores})$ 
6:      $Q \leftarrow \text{GenerarCantidades}(V_{\text{total}}, T)$ 
7:      $P \leftarrow P \cup (D, V, Q)$ 
8:   end for
9: end for
10: return  $P$ 

```

En el Algoritmo 17, V_{total} es el volumen total del contenedor, C es la cantidad de problemas por tipo de caja, T es el conjunto de tipos de cajas. En la línea 1, se inicializa la lista vacía P de instancias de problemas. En las líneas 2 a 8 se itera sobre la cantidad de tipos de cajas, en las líneas 3 a 7 se itera sobre la cantidad de problemas por tipo de caja, en la línea 4 se genera un conjunto de dimensiones únicas de cajas para cada tipo de caja, en la línea 5 se asigna un valor aleatorio a cada caja, en la línea 6 se genera un conjunto de cantidades de cajas para cada tipo de caja, en la línea 7 se añade la instancia de problema a la lista P . Finalmente, en la línea 9 se retorna la lista P con las instancias de problemas generadas. De esta manera, por ejemplo, si cuenta con 4 tipos de cajas (5T, 10T, 20T, 30T) y se desea generar 10 problemas por tipo, se obtendrán 40 instancias de problemas en total.

5.2. Diseño del experimento

Para evaluar el rendimiento de los algoritmos genéticos propuestos, se diseñó un experimento computacional que consiste en comparar los resultados obtenidos con y sin las mejoras propuestas. Se considera una versión sin mejora llamada M0 y otras cuatro mejoras, denominadas M1, M2, M3 y M4, que se describen a continuación:

- M0: Versión sin mejoras del algoritmo genético propuesto.

- M1: Se realiza la mejora de *llenado adicional inmediato* definido en el Algoritmo 6, que consiste en completar cuando sea posible espacios restantes durante el proceso de llenado.
- M2: Se implementa la mejora *llenado adicional final* definido en el Algoritmo 7, que consiste en completar los espacios restantes al final del proceso de llenado, para este caso se realiza la mejora en toda la población.
- M3: Esta mejora es una extensión de M2. Se dirige a la mitad superior de los individuos según su desempeño, optimizando así los recursos al concentrar los esfuerzos de mejora en aquellos que ya muestran un mayor potencial.
- M4: Esta mejora también sigue el principio de M2, pero se focaliza exclusivamente en el mejor individuo de la población. Al centrar todos los recursos de mejora en la solución más prometedora, se busca potenciar al máximo el rendimiento del líder del grupo, optimizando así el impacto general de la mejora.

Los algoritmos desarrollados se implementaron en Python v3.10 y están disponibles públicamente en un repositorio de GitHub¹.

Para su ejecución, se utilizó un equipo con un procesador Apple M1 de 8 núcleos, que cuenta con 4 núcleos de eficiencia y 4 núcleos de rendimiento, además de contar con 8 GB de memoria RAM. Con el objetivo de optimizar el uso de los recursos del procesador, se paralelizó la ejecución de los algoritmos genéticos para explotar específicamente los 4 núcleos de rendimiento. Para mejorar aún más el rendimiento de estos algoritmos, se empleó la librería Pypy como intérprete de Python.

En el marco de las pruebas, se crearon 25 instancias para cada tipo de caja, identificadas como 5T, 10T, 20T y 30T, sumando un total de 100 instancias de prueba. Cada instancia de prueba se ejecutó con las mejoras propuestas M0, M1, M2, M3 y M4, además de la versión sin mejora M0, para comparar los resultados obtenidos.

Cada algoritmo genético se configuró para operar con una misma población inicial de 100 individuos. Se estableció como condición de parada un límite de tiempo de ejecución de 5 minutos, garantizando así condiciones uniformes para la evaluación comparativa de cada mejora implementada en diversos escenarios con distintos tipos de cajas. La probabilidad de cruce, denotada como P_{CROSS} , se fijó en 0.05, mientras que la probabilidad de mutación, expresada como P_{MUT} , se definió en 0.01. Estos parámetros fueron seleccionados para promover un equilibrio entre exploración y explotación a lo largo del proceso evolutivo de los algoritmos.

5.3. Resultados y análisis

En esta sección se presentan y analizan los resultados obtenidos del experimento computacional. Los resultados se agrupan en función de las mejoras implementadas (M0, M1, M2, M3 y M4) y el tipo de instancia (5T, 10T, 20T, 30T). Se evalúan dos métricas principales: el valor aportado por las soluciones obtenidas y el tiempo de ejecución.

5.3.1. Valor aportado por las soluciones

Debido a que en las distintas instancias de problemas, los valores de las cajas tienen valores aleatorios independientes, no es posible comparar directamente las soluciones obtenidas, por ello se utiliza una métrica relativa que indica en cuanto por ciento ha mejorado una estrategia desde su población inicial. Para esto se toma el mejor valor obtenido al final y se calcula el porcentaje en base al

¹<https://github.com/josegustavo/lcp>

mejor valor de la población inicial, ya que todas las estrategias de mejora comienzan con la misma población inicial, esta métrica se considera justa para todas las estrategias.

Entonces se calcula el porcentaje de mejora por Tipo de Caja y Mejora propuesta, luego los resultados obtenidos se promedian sobre las 25 instancias y se muestran en la Tabla 1. Se presenta el promedio y la desviación estándar para cada mejora.

Cuadro 1: Valor aportado por las soluciones obtenidas

Tipo de Caja	M0		M1		M2		M3		M4	
	media	sd	media	sd	media	sd	media	sd	media	sd
T5	12.8 %	5.9	13.3 %	5.8	13.4 %	5.8	12.9 %	5.9	13.0 %	5.8
T10	17.1 %	8.2	19.9 %	8.7	19.5 %	8.5	17.5 %	8.2	18.2 %	8.0
T20	26.2 %	7.7	33.1 %	7.6	32.0 %	7.0	27.8 %	7.8	29.0 %	6.9
T30	28.4 %	9.3	39.0 %	8.8	38.5 %	8.8	36.8 %	9.0	31.8 %	9.0
T40	34.9 %	9.9	50.0 %	9.6	48.5 %	10.4	47.4 %	10.0	41.0 %	10.1

Como se observa en la Tabla 1, todas las mejoras propuestas (M1, M2, M3 y M4) superan a la versión sin mejoras (M0) en términos de valor aportado por las soluciones obtenidas. La mejoras M1 y M2, que se centran en ser aplicadas en toda la población, muestra los mejores resultados en todos los casos, aumentando la calidad de las soluciones en comparación con las demás versiones de mejora.

5.3.2. Tiempo de ejecución

El tiempo de ejecución es una métrica crucial para evaluar la eficiencia de los algoritmos genéticos. En este experimento, se midió el tiempo de ejecución de cada mejora y tipo de instancia, con el objetivo de evaluar el impacto de las mejoras en el rendimiento de los algoritmos.

Cada ejecución se realiza con un tipo de caja y una implementación específica (M0, M1, M2, M3 y M4). Como se estableció un límite de tiempo de 5 minutos para cada instancia, se registra el tiempo que cada evaluación tarda en encontrar su mejor solución, es decir, cuando el algoritmo ha llegado al mejor resultado dentro del tiempo permitido.

En la Tabla 2, se muestra el tiempo promedio que tardo en encontrar su mejor solución (en segundos), junto a la desviación estándar, para cada mejora y tipo de instancia.

Cuadro 2: Tiempo promedio en encontrar la mejor solución

Tipo de Caja	M0		M1		M2		M3		M4	
	media	sd	media	sd	media	sd	media	sd	media	sd
T5	71.4	65.4	25.6	48.5	50.4	55.8	48.4	41.9	56.6	65.9
T10	126.5	73.1	48.3	58.9	87.1	90.7	96.3	78.9	96.5	70.3
T20	217.7	71.8	127.6	64.9	148.5	77.7	213.8	71.6	191.1	74.4
T30	248.8	48.8	179.3	71.1	219.1	59.3	198.1	68.3	222.5	52.9
T40	267.6	23.2	218.7	73.7	230.3	49.2	236.0	61.5	234.3	48.4

La Tabla 2 muestra que las mejoras introducidas (M1, M2, M3 y M4) reducen ligeramente el tiempo

necesario para encontrar la mejor solución en comparación con M0. Esto se debe a que las mejoras permiten que el algoritmo genético converja más rápidamente. No obstante, también se observa una tendencia al aumento del tiempo de convergencia a medida que se incrementa el número de tipos de cajas a llenar. Esto también se puede observar en la Figura 25.

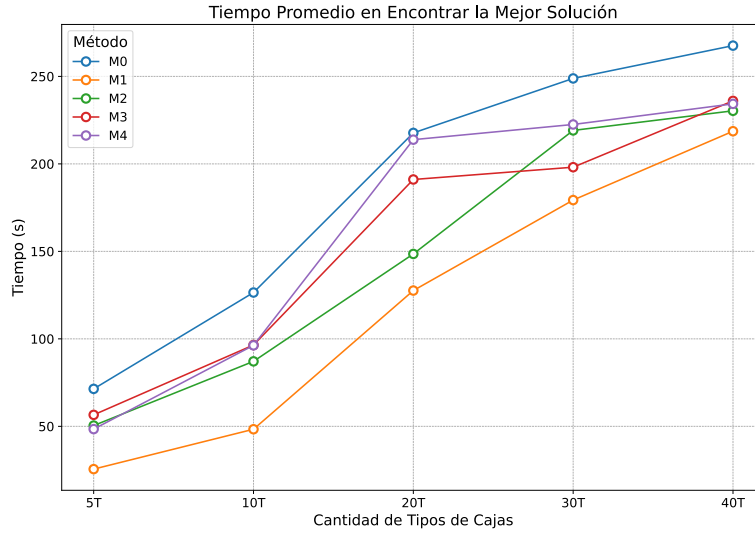


Figura 25: Tiempo promedio en encontrar la mejor solución

En la figura 25 se destaca que la mejora M1 presenta el menor tiempo de convergencia en comparación con los otros casos, seguida de M2 y M3 en algunas situaciones. La versión sin mejoras, M0, muestra un tiempo de convergencia significativamente mayor en comparación con las versiones mejoradas, es decir que las mejoras propuestas permiten que el algoritmo genético converja más rápidamente.

5.3.3. Rendimiento

Para proporcionar una visión clara del impacto de las mejoras, se presenta un análisis comparativo del rendimiento que puede ser definido como el incremento porcentual en la calidad de las soluciones por cada unidad tiempo que tarda en encontrar la mejor solución. La Figura 26 muestra el rendimiento de las mejoras propuestas en términos de calidad de las soluciones y tiempo de ejecución.

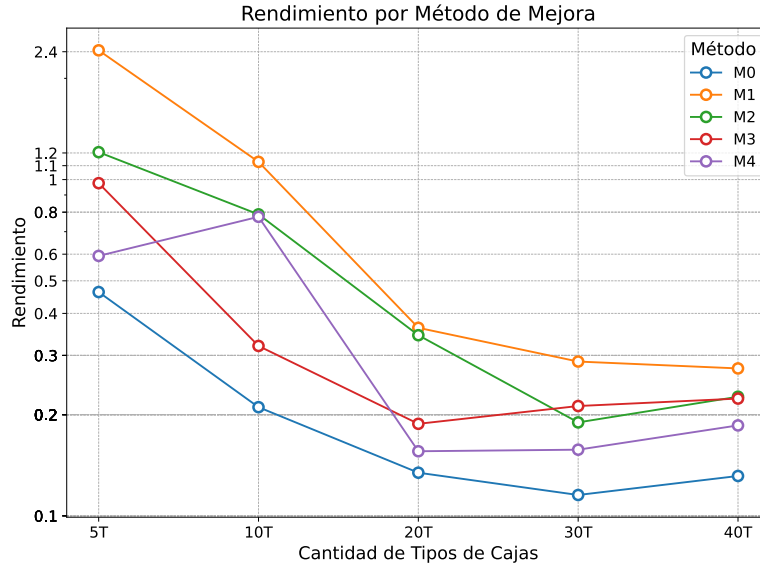


Figura 26: Rendimiento de las mejoras propuestas

La Figura 26 muestra que, la mejora M1 es la que un mayor incremento porcentual en la calidad de las soluciones por cada unidad de tiempo, seguida de M2 y M3 en algunas situaciones. La versión sin mejoras, M0, muestra un rendimiento significativamente menor en comparación con las versiones mejoradas. También es importante resaltar que cuando se usa 40 tipos de cajas (40T), los rendimientos tienden a aumentar en todas las estrategias de mejora,

También podemos evaluar el rendimiento según el tiempo que le toma a cada método de mejora en evaluar una generación, para ello se calcula el tiempo total de ejecución dividido por la cantidad de generaciones realizadas, esto nos da una idea de cuánto tiempo le toma a cada método de mejora evaluar cada generación. En la Figura 27 se muestra el tiempo promedio que le toma a cada método de mejora en evaluar una generación.

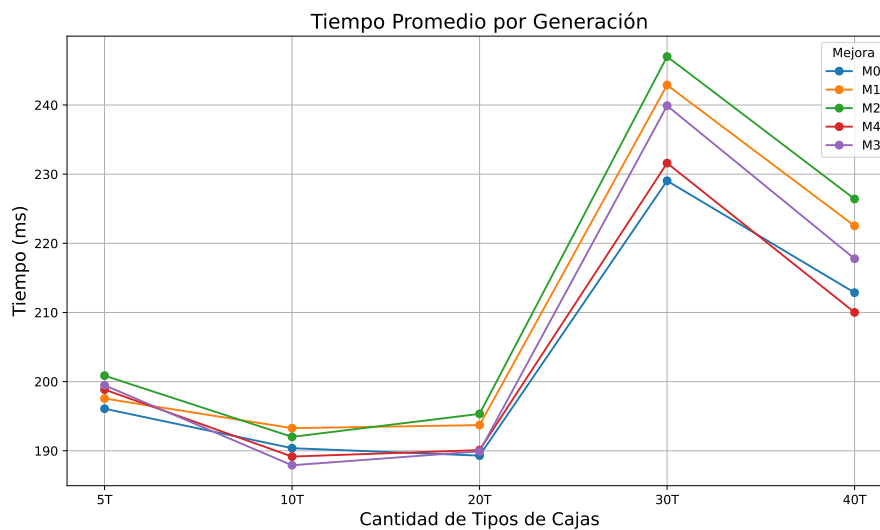


Figura 27: Tiempo promedio en evaluar una población

La Figura 27 muestra que la mejora M1, M2, M3 y M4 necesitan un mayor tiempo para evaluar una generación en comparación con la versión sin mejoras M0. Esto se debe a que las mejoras propuestas requieren un tiempo adicional para realizar las operaciones de mejora en cada generación. Sin embargo, como se ha podido ver en los gráficos anteriores, las mejoras propuestas también permiten que el algoritmo genético converja más rápidamente, lo que se traduce en un menor tiempo total de ejecución, a pesar de que cada generación requiera un tiempo adicional para ser evaluada.

Por último es importante resaltar que la mejora M1 es la que mejor rendimiento tiene en términos de calidad de las soluciones y tiempo de ejecución en la mayoría de evaluaciones con diferentes tipos de cajas. Luego M2 y M3 presentan un rendimiento similar en algunas situaciones, M2 es mejor en términos de calidad de las soluciones y M3 en términos de tiempo de ejecución. M4 es la que presenta el peor rendimiento en términos de calidad de las soluciones y tiempo de ejecución en la mayoría de evaluaciones con diferentes tipos de cajas.

6. Conclusiones

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetur adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

A. Appendix I

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

B. Appendix II

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Referencias

- Parreño, Francisco, Ramón Alvarez-Valdés, José Manuel Tamarit y José Fernando Oliveira (2008). «A Maximal-Space Algorithm for the Container Loading Problem». *INFORMS J. Comput.* **20**, páginas 412-422.
- Pisinger, David (2002). «Heuristics for the container loading problem». *European Journal of Operational Research* **141**, páginas 382-392. DOI: [https://doi.org/10.1016/S0377-2217\(02\)00132-7](https://doi.org/10.1016/S0377-2217(02)00132-7).
- Wäscher, Gerhard, Heike Haußner y Holger Schumann (2007). «An improved typology of cutting and packing problems». *European Journal of Operational Research* **183**, páginas 1109-1130. DOI: <https://doi.org/10.1016/j.ejor.2005.12.047>.
- Şafak, Özge y Güneş Erdoğan (2023). «A Large Neighbourhood Search Algorithm for Solving Container Loading Problems». *Computers & Operations Research* **154**, página 106199. DOI: <https://doi.org/10.1016/j.cor.2023.106199>.
- Barnett, S y GJ Kynch (1967). «Exact solution of a simple cutting problem». *Operations Research* **15**, páginas 1051-1056.
- George, J.A. y D.F. Robinson (1980). «A heuristic for packing boxes into a container». *Computers & Operations Research* **7**, páginas 147-156. DOI: [https://doi.org/10.1016/0305-0548\(80\)90001-5](https://doi.org/10.1016/0305-0548(80)90001-5).
- Alvarez-Valdes, Ramón, Maria Antónia Carravilla y José Fernando Oliveira (2018). «Cutting and Packing». En: *Handbook of Heuristics*. Editado por Rafael Martí, Panos M. Pardalos y Mauricio G. C. Resende. Cham: Springer International Publishing, páginas 931-977. DOI: [10.1007/978-3-319-07124-4_43](https://doi.org/10.1007/978-3-319-07124-4_43).
- de Queiroz, Thiago A., Flávio K. Miyazawa, Yoshiko Wakabayashi y Eduardo C. Xavier (2012). «Algorithms for 3D guillotine cutting problems: Unbounded knapsack, cutting stock and strip packing». *Computers & Operations Research* **39**, páginas 200-212. DOI: <https://doi.org/10.1016/j.cor.2011.03.011>.
- Scheithauer, Guntram (2017). *Introduction to cutting and packing optimization: Problems, modeling approaches, solution methods*. Volumen 263. Springer.
- Bortfeldt, Andreas y Gerhard Wäscher (2013). «Constraints in container loading-A state-of-the-art review». *European Journal of Operational Research* **229**. Cited by: 288, páginas 1-20. DOI: [10.1016/j.ejor.2012.12.006](https://doi.org/10.1016/j.ejor.2012.12.006).
- Ren, Jidong, Yajie Tian y Tetsuo Sawaragi (2011). «A priority-considering approach for the multiple container loading problem». *International Journal of Metaheuristics* **1**, páginas 298-316.
- Zhao, Xiaozhou, Julia A Bennell, Tolga Bektas y Kath Dowsland (2016). «A comparative review of 3D container loading algorithms». *International Transactions in Operational Research* **23**, páginas 287-320.
- Ramos, A. Galvão, José F. Oliveira, José F. Gonçalves y Manuel P. Lopes (2015). «Dynamic stability metrics for the container loading problem». *Transportation Research Part C: Emerging Technologies* **60**, páginas 480-497. DOI: <https://doi.org/10.1016/j.trc.2015.09.012>.
- Junqueira, Leonardo, Reinaldo Morabito y Denise Sato Yamashita (2012). «Three-dimensional container loading models with cargo stability and load bearing constraints». *Computers & Operations Research* **39**. Special Issue on Knapsack Problems and Applications, páginas 74-85. DOI: <https://doi.org/10.1016/j.cor.2010.07.017>.
- Nascimento, Oliviana Xavier do, Thiago Alves de Queiroz y Leonardo Junqueira (2021). «Practical constraints in the container loading problem: Comprehensive formulations and exact algorithm». *Computers & Operations Research* **128**, página 105186. DOI: <https://doi.org/10.1016/j.cor.2020.105186>.
- Bischoff, E.E. y M.S.W. Ratcliff (1995). «Issues in the development of approaches to container loading». *Omega* **23**, páginas 377-390. DOI: [https://doi.org/10.1016/0305-0483\(95\)00015-G](https://doi.org/10.1016/0305-0483(95)00015-G).
- Ranck Júnior, Rodolfo, Horacio Hideki Yanasse, Reinaldo Morabito y Leonardo Junqueira (2019). «A hybrid approach for a multi-compartment container loading problem». *Expert Systems with Applications* **137**, páginas 471-492. DOI: <https://doi.org/10.1016/j.eswa.2019.07.020>.

- Glover, Fred (1986). «Future paths for integer programming and links to artificial intelligence». *Computers & Operations Research* **13**. Applications of Integer Programming, páginas 533-549. DOI: [https://doi.org/10.1016/0305-0548\(86\)90048-1](https://doi.org/10.1016/0305-0548(86)90048-1).
- Sörensen, Kenneth y Fred W. Glover (2013). «Metaheuristics». En: *Encyclopedia of Operations Research and Management Science*. Editado por Saul I. Gass y Michael C. Fu. Boston, MA: Springer US, páginas 960-970. DOI: [10.1007/978-1-4419-1153-7_1167](https://doi.org/10.1007/978-1-4419-1153-7_1167).
- Martí, Rafael, Marc Sevaux y Kenneth Sörensen (2024). «50 years of metaheuristics». *European Journal of Operational Research*. DOI: <https://doi.org/10.1016/j.ejor.2024.04.004>.
- Goldberg, David E (2013). *Genetic algorithms*. pearson education India.
- Ramos, António G., Elsa Silva y José F. Oliveira (2018). «A new load balance methodology for container loading problem in road transportation». *European Journal of Operational Research* **266**, páginas 1140-1152. DOI: <https://doi.org/10.1016/j.ejor.2017.10.050>.
- Gonçalves, José Fernando y Mauricio G.C. Resende (2012). «A parallel multi-population biased random-key genetic algorithm for a container loading problem». *Computers & Operations Research* **39**, páginas 179-190. DOI: <https://doi.org/10.1016/j.cor.2011.03.009>.
- Kang, Kyungdaw, Ilkyeong Moon y Hongfeng Wang (2012a). «A hybrid genetic algorithm with a new packing strategy for the three-dimensional bin packing problem». *Applied Mathematics and Computation* **219**, páginas 1287-1299. DOI: <https://doi.org/10.1016/j.amc.2012.07.036>.
- Kuo, R.J., Pei-Cheng Ho y Ferani E. Zulvia (2023). «Application of metaheuristics algorithm on a multi-objective container loading problem considering container's utilization and vehicle's balance». *Applied Soft Computing* **143**, página 110417. DOI: <https://doi.org/10.1016/j.asoc.2023.110417>.
- Domingo, Bernd Matthias, SG Ponnambalam y Ganesan Kanagaraj (2012). «Particle Swarm Optimization for the single container loading problem». En: *2012 IEEE International Conference on Computational Intelligence and Computing Research*. IEEE, páginas 1-6.
- Cano, Isaac y Vicenç Torra (2010). «Particle Swarm Optimization for Container Loading of Nonorthogonal Objects». En: *Artificial Intelligence and Soft Computing*. Editado por Leszek Rutkowski, Rafał Scherer, Ryszard Tadeusiewicz, Lotfi A. Zadeh y Jacek M. Zurada. Berlin, Heidelberg: Springer Berlin Heidelberg, páginas 403-410.
- Zhao, Xiaozhou (2017). «The three-dimensional container loading problem». Tesis doctoral.
- Martínez-Franco, Juan, Edgar Céspedes-Sabogal y David Álvarez-Martínez (2020). «PackageCargo: A decision support tool for the container loading problem with stability». *SoftwareX* **12**, página 100601. DOI: <https://doi.org/10.1016/j.softx.2020.100601>.
- LoadCargo.in (2024). *LoadCargo.in: Software de Optimización de Carga para Contenedores y Camiones*. <https://loadcargo.in/>. Accedido el 10 de mayo de 2024.
- Gehring, H. y A. ortfeldt (1997). «A genetic algorithm for solving the container loading problem». *International Transactions in Operational Research* **4**, páginas 401-418. DOI: [https://doi.org/10.1016/S0969-6016\(97\)00033-6](https://doi.org/10.1016/S0969-6016(97)00033-6).
- Karabulut, Korhan y Mustafa Murat İnceoğlu (2005). «A Hybrid Genetic Algorithm for Packing in 3D with Deepest Bottom Left with Fill Method». En: *Advances in Information Systems*. Editado por Tatyana Yakhno. Berlin, Heidelberg: Springer Berlin Heidelberg, páginas 441-450.
- Wang, Hongfeng y Yanjie Chen (2010). «A hybrid genetic algorithm for 3D bin packing problems», páginas 703-707. DOI: [10.1109/BICTA.2010.5645211](https://doi.org/10.1109/BICTA.2010.5645211).
- Kang, Kyungdaw, Ilkyeong Moon y Hongfeng Wang (2012b). «A hybrid genetic algorithm with a new packing strategy for the three-dimensional bin packing problem». *Applied Mathematics and Computation* **219**, páginas 1287-1299.
- Umbarkar, Anant J y Pranali D Sheth (2015). «Crossover operators in genetic algorithms: a review.» *ICTACT journal on soft computing* **6**.
- Dockhorn, Alexander y Simon Lucas (2022). «Choosing Representation, Mutation, and Crossover in Genetic Algorithms». *IEEE Computational Intelligence Magazine* **17**, páginas 52-53. DOI: [10.1109/MCI.2022.3199626](https://doi.org/10.1109/MCI.2022.3199626).
- Hasni, Hamadi y Housseem Sabri (2013). «On a Hybrid Genetic Algorithm for Solving the Container Loading Problem with no Orientation Constraints». *Journal of Mathematical Modelling and Algorithms in Operations Research* **12**, páginas 67-84. DOI: [10.1007/s10852-012-9179-3](https://doi.org/10.1007/s10852-012-9179-3).