Universidad Andrés Bello®

Facultad de Ingeniería
Ingeniería Civil Informática

Análisis de técnicas de resolución para el Knapsack Problem

Tesis de pregrado para optar al título de Ingeniería Civil Informática

Estudiante:

Felipe Andrés Pereira Alarcón

Profesor Guía:

Dr. Gustavo Gatica

Santiago, Chile

2026

DEDICATORIA

Esta Tesis se dedica a mis padres, Manuel Pereira y Alejandra Alarcón, quienes me brindaron apoyo constante durante estos largos años de carrera. Incluso en momentos difíciles, como la llegada de la pandemia y mi propia depresión, ellos me brindaron fuerza, voluntad, garra y luz para seguir adelante. Admiro profundamente su dedicación y les agradezco el esfuerzo que realizaron, tanto en términos monetarios como emocionales y temporales, para que pudiera llegar a ser el Primer Profesional de la Familia Pereira Alarcón. Pero esto no termina aquí: mi camino académico se expandirá a futuro.

También se la dedico a mis hermanos, Agustín Pereira y Maximilian Pereira, quienes me brindaron su ayuda y apoyo constante en el proceso de manera anímica. En momentos de estrés, depresión, ansiedad y caídas, ellos estuvieron siempre a mi lado para ayudarme a levantarme. Desde el primer día, han creído en mi potencial como profesional y me han motivado a seguir adelante.

Sin el apoyo de mi familia, este logro no habría sido posible. Agradezco además a mis profesores y compañeros de carrera, quienes me brindaron la orientación y el apoyo necesarios para alcanzar mis metas. Les estaré eternamente agradecido por todo.

También quiero dedicar esta tesis a mis amigos cercanos, quienes me apoyaron en la mayoría de los momentos difíciles, terminar trabajos, sacrificando horas de sueño antes de sus propias jornadas laborales y académicas. Su amistad y apoyo incondicional significaron mucho para mí.

Quiero hacer una dedicación con mención especial al profesor Walter Uribe, quien me ayudó durante más de un año a adaptarme a la realidad universitaria. Es un hombre sabio, valiente y con gran experiencia, pero, sobre todo, un ser humano lleno de cariño. Me ayudó a llenar vacíos que tenía como estudiante y como persona, y gracias a sus constantes retos y correcciones, me convertí en uno de sus mejores alumnos. Profesor Walter, si llega a leer esto, quiero que sepa que lo aprecio mucho y que siempre tendrá un lugar especial en mi vida. Además, está cordialmente invitado a mi ceremonia de titulación.

A mi profesor guía, el Doctor Gustavo Gatica, un excelente profesional y un muy buen profesor, quien me acompañó durante mis estudios y se ha convertido recientemente en un ejemplo a seguir gracias a sus estudios, investigaciones y presentaciones de libros. Le agradezco todo el apoyo y la confianza en mí y mis compañeros, incluso cuando se tenían dificultades con ciertas metodologías, y por ayudarme a seguir en el camino para convertirme en un profesional ejemplar.

Dedico esta tesis a mis amigos universitarios. A pesar de perder el contacto con algunos, otros se han alejado y algunos más no me han tratado de la mejor manera, agradezco los buenos momentos que se compartieron. Recuerdo cuando les impartía clases hasta tarde en un box para luego buscar algún tipo de locomoción desde Providencia a Paine a altas horas de la noche, sabiendo que no había transporte disponible.

También agradezco su compañía en momentos de compras, estudio, juego y desestresamiento. Si bien he perdonado sus acciones negativas hacia mí, todavía me duele el mal trato que recibí, especialmente cuando siempre fui una persona tranquila y respetuosa.

A cada una de las personas con las que conviví estos años, les estoy profundamente agradecido, ya que, para mí, todas son importantes y todas han dejado una huella en mi vida.

AGRADECIMIENTOS

Agradezco sinceramente a la Facultad de Ingeniería Civil Informática de la Universidad Nacional Andrés Bello por brindarme una educación de excelencia, la cual me ha permitido desarrollar mis capacidades y enfrentar mis desafíos profesionales. Su sello de calidad es una verdadera visión de la ingeniería civil.

Quiero expresar mi gratitud a los profesores de mi comisión evaluadora por su disposición y comprensión con relación al espíritu de esta tesis. Su experiencia y sabiduría fueron fundamentales para guiar mi trabajo y lograr el éxito.

También deseo agradecer a todo el personal administrativo que trabajó detrás de escena para hacer posible mi desarrollo en la Universidad. En especial, mi agradecimiento al Doctor Gustavo Gatica por su paciencia y buena disposición en todo momento. Sus consejos y apoyo fueron de gran ayuda durante mi carrera universitaria.

Tabla de contenidos

Tabla de Figuras

Tablas

CAPÍTULO 1

INTRODUCCIÓN

1.1 Trasfondo

El problema de la mochila es uno de los problemas de optimización combinatoria más estudiados en la literatura. Es un problema clásico utilizado en diferentes áreas, como la logística, la informática, la economía y la ingeniería, entre otras [2]. El problema se centra en seleccionar un subconjunto de objetos con pesos y valores asociados de tal manera que maximice el valor total de los objetos llevados, sin exceder una capacidad máxima dada de la mochila [1].

Este problema, en sus diferentes variantes, genera un gran interés tanto en el ámbito teórico como práctico, siendo objeto de numerosos estudios. El interés práctico de este problema radica en su amplia aplicabilidad en la industria. Entre sus usos más destacados se encuentran la optimización de presupuestos de inversión, la logística de carga y los problemas de corte de material[7].

Desde el punto de vista teórico, el problema de la mochila tiene una estructura simple que permite la aplicación de numerosas propiedades combinatorias. Además, otros problemas combinatorios más complejos pueden ser resueltos mediante una serie de subproblemas que caen dentro de la categoría del problema de la mochila. Esto explica el gran interés teórico que despiertan estos problemas [2].

1.2 Motivación

Este problema, pertenece a la clase de problema del tipo **NP-COMPLETO** [3]. Esto significa que, aunque no se ha encontrado un algoritmo que resuelva en tiempo polinómico, se cree que no existe ninguna solución eficiente para resolver el problema en todas las circunstancias. En otras palabras, es un problema muy difícil de resolver de manera óptima para grandes conjuntos de datos [11].

El problema de la mochila específicamente se refiere a la selección de objetos con un valor y un peso determinado, con el objetivo de llenar una mochila con una capacidad limitada de forma tal que maximice el valor total de los objetos seleccionados. Este problema presenta un gran interés práctico, ya que tiene aplicaciones en la planificación de producción, la distribución de recursos, la logística y el diseño de sistemas de transporte, entre otros [21].

1.3 Objetivos

El objetivo principal de este proyecto es aplicar algoritmos para resolver el problema de la mochila aplicando todos los recursos computacionales disponibles.

La intención es, demostrar que, con estos algoritmos, se puede lograr alcanzar el rendimiento óptimo para la solución objetiva.

Para lograr este objetivo, se estudiará el problema de la mochila, su historia, características y las diferentes variantes existentes [2]. Posteriormente, se implementó cada algoritmo para resolver el problema, el cual servirá como base para el resto del informe.

Seguidamente, se aborda el problema de la mochila desde una perspectiva matemática y computacional. Se presenta el modelo matemático del problema, se describe su implementación utilizando AMPL [13] y se realizan experimentos computacionales con Implementación en Python para validar la solución óptima obtenida.

Finalmente, se discuten algunas conclusiones relevantes y se proporcionan referencias bibliográficas para profundizar en el tema.

1.4 Estructura del documento

La presente tesis se organiza de la siguiente manera:

- **Prólogo:** Sección preliminar que incluye las dedicatorias y agradecimientos institucionales, al Profesor Guía, junto con los agradecimientos institucionales a la Facultad.
- **Capítulo 1 - Introducción:** Expone el contexto histórico, la motivación del estudio y los objetivos generales y específicos.
- **Capítulo 2 - Explicación del Problema:** Describe el *Knapsack Problem* y analiza su complejidad computacional como problema NP-Completo.
- **Capítulo 3 - Modelo Matemático:** Presenta la formulación formal, detallando la función objetivo, variables de decisión y restricciones.
- **Capítulo 4 - Implementación en AMPL:** Describe la codificación del modelo matemático en el lenguaje AMPL para la obtención de soluciones exactas.
- **Capítulo 5 - Metodologías de Trabajo:** Detalla el diseño experimental, incluyendo el entorno de hardware/software y la generación de las instancias de prueba.
- **Capítulo 6 - Experimentos Computacionales:** Compara el rendimiento (tiempo, memoria y convergencia) de los algoritmos implementados: Programación Dinámica, Greedy, Fuerza Bruta, Recocido Simulado y Algoritmos Genéticos.
- **Capítulo 7 - Caso de Estudio:** Aplica los métodos validados a un problema real de optimización de carga logística en Correos de Chile.
- **Capítulo 8 - Conclusiones:** Resume los hallazgos principales, limitaciones del estudio y propuestas de trabajo futuro.
- **Referencias:** Listado bibliográfico de las fuentes citadas según normas APA.

CAPÍTULO 2

EXPLICACIÓN DEL PROBLEMA DE LA MOCHILA

2.1 Explicación Introductoria

El problema de la mochila es un problema matemático descrito de manera sencilla. Considere un escenario en el cual se dispone de una mochila con una capacidad máxima de peso que puedes cargar. Asimismo, se dispone de una lista de objetos con diferentes pesos y valores. La interrogante principal es: ¿cómo seleccionar los objetos para poner en la mochila de manera que se maximice el valor total, sin exceder la capacidad de peso máximo de la mochila?

2.2 Ejemplo Inductivo

Considérese una mochila con capacidad de carga definida, para la cual **se dispone de** 5 objetos diferentes para elegir: una botella de agua que pesa 1 kg y tiene un valor de 5 dólares, un libro que pesa 3 kg y tiene un valor de 8 dólares, un par de zapatos que pesan 2 kg y tienen un valor de 10 dólares, una chaqueta que pesa 4 kg y tiene un valor de 15 dólares, y una cámara que pesa 5 kg y tiene un valor de 20 dólares.

Objetos	Peso (kg)	Valor (USD)
Botella de agua	1	5
Libro	3	8
Par de Zapatos	2	10
Chaqueta	4	15
Cámara	5	20

Figura 2.1: Lista objetos

Si solo **fuese posible** transportar una fracción... ¿cuáles **se deberían** seleccionar para maximizar el valor total sin exceder el límite de peso de la mochila? Esta es la pregunta que el problema de la mochila intenta responder. [2]

En la práctica, este problema se aplica en situaciones como la planificación de rutas de reparto de paquetes, la optimización de la carga de aviones o barcos, o incluso en la selección de cartera de inversión para maximizar el rendimiento y minimizar el riesgo. El problema de la mochila tiene aplicaciones en muchos campos diferentes y se puede resolver con la ayuda de la programación matemática. [2]

2.3 Complejidad Computacional

Cualquier problema que se relacione con el problema de la mochila serán considerados **NP-COMPLETO**, lo cual ha sido demostrado por varios autores en la literatura. Esto significa que no existe un algoritmo cuya complejidad sea polinomial en función de *n*, a menos que se demuestre que **P = NP** en algún momento. [3]

No obstante, para el problema de la mochila única (que consta de un solo contenedor) existen algoritmos cuya complejidad temporal y espacial es pseudo-polinomial, es decir, está acotada por un polinomio en función de *n* y *c*, por ejemplo *O(nc)*. Sin embargo, esta complejidad no contradice el hecho que categorizar este problema como **NP-COMPLETO**, ya que un algoritmo tiene una complejidad pseudo-polinomial si depende de la longitud de la entrada (es decir, el número de bits requeridos para representarla) y el valor numérico de la entrada. Por lo general, la longitud de la entrada es exponencial en relación a su valor numérico. [1]

En otras palabras, el problema de la mochila única es débilmente **NP-COMPLETO**. Un problema es débilmente **NP-COMPLETO** si existe un algoritmo capaz de resolver el problema en tiempo polinomial en función de la dimensión del problema y la magnitud de los datos, en lugar del logaritmo en base 2 de sus magnitudes. Por lo tanto, estos algoritmos son técnicamente funciones exponenciales en relación al tamaño de la entrada y no son considerados algoritmos polinomiales. [3]

Por otro lado, a diferencia del problema de la mochila única, el problema de la mochila múltiple no es débilmente **NP-COMPLETO**, sino que es fuertemente **NP-**

COMPLETO, significando que no puede existir un algoritmo capaz de resolver estos problemas en tiempo pseudo-polinomial a menos que evidencie **P = NP**. [1]

CAPÍTULO 3

MODELO MATEMÁTICO DEL PROBLEMA

3.1 Definición Formal

El problema de la mochila se puede modelar matemáticamente basándose en la propuesta de Pisinger [12] para la siguiente fórmula:

Maximizar Z:

$$\sum_{i=1}^{n} x_i \, v_i \qquad (1)$$

Sujeto a:

$$\sum_{i=1}^{n} W_i \, x_i \leq W \qquad (2)$$

Donde x_i en {0, 1} para todo *i* hasta *W*, además, se tienen los siguientes supuestos:

- ***n*** es el número de objetos. En AMPL se verá como **ELEMENTOS**.
- V_i es el valor del objeto *i*. En AMPL se verá como **VALOR.**
- W_i es el peso del objeto *i*. En AMPL se verá como **PESO**.
- ***W*** es la capacidad máxima de la mochila. En AMPL se verá como **PESOMAX.**
- X_i es una variable binaria que indica si se selecciona o no el objeto *i*. En AMPL se verá como **take{ELEMENTOS}.**

La idea es maximizar el valor total de los objetos seleccionados, sujeto a la restricción del peso total y que no exceda la capacidad máxima de la mochila. Además, se debe asegurar que cada objeto sea seleccionado completo (variable binaria X_i igual a 1) o no seleccionado (variable binaria X_i igual a 0).

La correlación entre el modelo matemático y los datos radica en cómo se utilizan los valores y pesos de los elementos, así como la capacidad de la mochila, para determinar la solución óptima. El modelo establece una relación entre las variables de decisión (si se selecciona o no cada elemento) y el valor total obtenido. A medida que se cambian los valores y pesos de los elementos, así como la capacidad de la mochila, el modelo matemático proporciona una solución óptima que maximiza el valor total.

3.2 Generalización del Modelo

El problema clásico de la mochila implica seleccionar un subconjunto de elementos para maximizar el valor total sin exceder la capacidad de la mochila [1]. Sin embargo, en ciertos escenarios, es necesario considerar restricciones adicionales, objetivos alternativos o variantes del problema básico [2].

Restricciones adicionales

Para casos específicos, pueden existir restricciones adicionales que limiten la cantidad de elementos seleccionados o restrinjan los recursos disponibles. Estas restricciones pueden incluir límites en la cantidad de elementos seleccionados, restricciones de recursos adicionales como el tiempo o el espacio, o restricciones específicas del dominio en el que se aplica el problema [3]. Por ejemplo, en problemas de asignación de presupuesto o logística compleja, se utiliza el *Multidimensional Knapsack Problem (MKP)*, donde cada ítem consume múltiples recursos simultáneamente [4].

Objetivos alternativos

Además de maximizar el valor total, es posible que existan objetivos alternativos. Estos pueden incluir la minimización del peso total, buscar un equilibrio óptimo entre el valor y el peso, o maximizar una función de utilidad ponderada que combine múltiples criterios [5]. Este enfoque se conoce como *Multiobjective Knapsack Problem (MO-KP)* y es fundamental en la toma de decisiones donde existen conflictos de interés, como en la selección de carteras de inversión financiera [6].

Elementos fraccionales

En el problema clásico de la mochila, se asume que los elementos deben seleccionarse de forma discreta (0 o 1), es decir, se seleccionan por completo o no se seleccionan en absoluto. Sin embargo, en ciertos casos, es posible permitir la selección de

fracciones de elementos, lo que se conoce como el *Fractional Knapsack Problem* [7]. Esta generalización permite una mayor flexibilidad y, a diferencia de la versión entera, puede resolverse óptimamente utilizando algoritmos voraces (*Greedy*) en tiempo polinomial, dado que posee la propiedad de subestructura óptima [8].

Elementos múltiples

En la versión clásica del problema de la mochila, se asume que solo hay un elemento de cada tipo disponible para su selección. Sin embargo, en escenarios como la gestión de inventarios, puede ser necesario permitir múltiples copias de los elementos. Si el número de copias es limitado, se denomina Bounded Knapsack Problem (BKP) [9]**;**si es ilimitado, Unbounded Knapsack Problem (UKP) [10]. Estas variantes influyen en la complejidad, requiriendo adaptaciones en los algoritmos de Programación Dinámica para manejar la multiplicidad de estados sin explotar la memoria computacional [1].

Conclusión del capítulo

Estas generalizaciones permiten adaptar el modelo básico a situaciones más específicas y complejas, introduciendo nuevos desafíos computacionales. No obstante, el presente trabajo de tesis se centrará exclusivamente en la variante **0/1 Knapsack Problem**. Esta delimitación se justifica debido a que el caso 0/1 constituye el bloque fundamental de la complejidad **NP-Completo** [1] y sirve como el escenario ideal para comparar el rendimiento crítico entre algoritmos exactos y metaheurísticas, tal como se desarrollará en la implementación del Capítulo 4.

CAPÍTULO 4

IMPLEMENTACIÓN DEL MODELO EN AMPL

4.1 Definición del algoritmo

En esta implementación, se definen los siguientes conjuntos y parámetros:

- **ELEMENTOS**: conjunto que indica el rango de índices para los objetos.
- **PESOMAX**: parámetro que indica el peso máximo permitido en la mochila.
- **VALOR:** parámetro que indica el valor de cada objeto.
- **PESO:** parámetro que indica el peso de cada objeto.

Además, se definen dos parámetros adicionales:

- **acum_peso:** parámetro acumulativo para el peso de los objetos seleccionados.
- **acum_valor:** parámetro acumulativo para el valor de los objetos seleccionados.

Por último, se define la variable de decisión take, que es binaria y toma el valor 1 si se selecciona el objeto *i* y 0 en caso contrario.

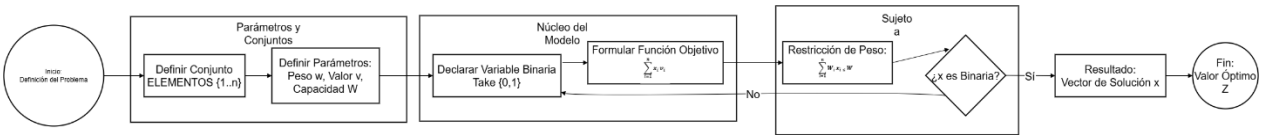

Figura 4.1: *Diagrama modelo AMPL*

El objetivo de la función objetivo es maximizar el valor total de los objetos seleccionados, el cual se calcula como la suma del valor de cada objeto multiplicado por la variable binaria *take* correspondiente.

De acuerdo con la restricción de peso, se debe respetar la suma de los pesos de los objetos seleccionados no debe exceder el peso máximo permitido en la mochila.

En resumen, este código representa una implementación del modelo matemático del problema de la mochila en AMPL, el cual es posible resolver mediante el uso de un solver de programación lineal entera para obtener la solución óptima del problema [13].

A continuación, se verá implementada el código en software AMPL y el paso a paso de su ejecución.

4.2 Implementación en Código AMPL

La implementación computacional del modelo matemático definido previamente se realizó utilizando el lenguaje de modelado AMPL [13]

Esta implementación se estructura en dos componentes fundamentales: el archivo de modelo (.mod), que contiene la lógica abstracta, variables y restricciones; y el archivo de datos (.dat), que instancia los parámetros específicos del problema.

Código Fuente del Modelo:

El siguiente código define la estructura algebraica del problema, estableciendo la función objetivo de maximización y la restricción de capacidad.

Código "knapsack.mod"

```
# Conjunto que indica el rango de índices para cada objeto
set ELEMENTOS;

# Peso mayor Permitido
param PESOMAX integer, >= 1
# Valor de cada objeto
param VALOR{ELEMENTOS} >= 0;
# Peso de cada objeto
param PESO{ELEMENTOS} >= 0;

# Acumula el peso de los objetos seleccionados
param acum_peso default 0;
# Acumula el valor de los objetos seleccionados
param acum_valor default 0;

# Binario, 1 Si tomamos el ítem i y 0 si no
var take{ELEMENTOS} binary;
# Función Objetivo que maximiza el Valor Total
maximize MaxVal:
        sum {i in ELEMENTOS} VALOR[i] * take[i];

# Restricción de Peso
subject to Weight:
        sum{i in ELEMENTOS} PESO[i] * take[i] <= PESOMAX;;
```

Instancia de Datos de Prueba (knapsack.dat)

Para la primera prueba de concepto, se definieron los siguientes parámetros con una capacidad de carga de 102 unidades y un conjunto de 11 artículos con sus respectivos atributos de valor y peso.

Código "knapsack.dat"

```
param PESOMAX := 102;

# Definimos ítems con valor y peso respectivamente
param: ELEMENTOS: VALOR PESO :=
  camara  15  2
  bolsa 100 20
  Mapa 90 20
  lapiz 60 30
  pistola 40 40
  azucar 15 30
  platano 10 60
  queso 1 10
  manteca 12 21
  chocolate 12 12
  sal 100  2
;
```

4.3 Paso a Paso

Paso a Paso con Software AMPL:

El proceso de resolución se llevó a cabo instanciando el modelo y cargando los parámetros mediante la interfaz de línea de comandos de AMPL. Para la obtención de la solución óptima, se seleccionó el solver **CPLEX**, reconocido por su eficiencia en la resolución de problemas de Programación Lineal Entera (ILP).

La secuencia de comandos ejecutada para cargar el modelo, los datos y configurar el solver fue la siguiente:

model "knapsack.mod";

data "knapsack.dat";

option solver cplex;

solve;

Análisis de Resultados

Tras la convergencia del solver, se procedió a la visualización de las variables de decisión mediante el comando *display take*;

La salida obtenida detalla la asignación binaria óptima para cada elemento:

```
take [*] :=
     Mapa  1
```

```
   azucar  0
    bolsa  1
   camara  1
chocolate  1
    lapiz  1
  manteca  0
  pistola  0
  platano  0
    queso  1
      sal  1
;
```

Para verificar el cumplimiento de las restricciones y el valor de la función objetivo, se computaron los totales mediante la instrucción de despliegue de sumatorias. Los resultados confirman que la solución es factible y óptima:

```
display PESOMAX, sum{i in ELEMENTOS} VALOR[i] * take[i], sum{i in
ELEMENTOS} PESO[i] * take[i];
```

Resultados demostrados en consola:

```
PESOMAX = 102
sum{i in ELEMENTOS} VALOR[i]*take[i] = 378
sum{i in ELEMENTOS} PESO[i]*take[i] = 96
```

Capacidad Máxima (W): 102

Valor Objetivo (z): 378

Peso Ocupado: 96 (Holgura de 6 unidades)

4.4 Ejemplo Introductorio demostrado

Con el objetivo de validar la consistencia del modelo implementado, se sometió a prueba el "Ejemplo Inductivo" descrito teóricamente en el Capítulo 2. Para ello, se generó un nuevo archivo de datos denominado *"knapsack2.dat"* con los parámetros correspondientes a dicha instancia (5 objetos, capacidad 10).

Datos de Entrada (knapsack2.dat)

```
param PESOMAX := 10;

# Definimos ítems con valor y peso respectivamente
param: ELEMENTOS: VALOR PESO :=
  botella_agua 5 1
  libro 8 3
  par_zapatos 10 2
  chaqueta 15 4
  camara 20 5
```

```
;
```

Ejecución y Resultados de Validación Se replicó el procedimiento de ejecución cargando el nuevo set de datos:

model "knapsack.mod";

data "knapsack2.dat";

option solver cplex;

solve;

La solución reportada por el solver CPLEX seleccionó los elementos necesarios para maximizar el valor sin violar la restricción de peso de 10 kg:

```
take [*] :=
botella_agua  1
      camara  1
    chaqueta  1
       libro  0
 par_zapatos  0
   ;
```

Para ver el valor y el peso totales que obtuvo de la solución óptima, se escribe en consola como:

- `display PESOMAX, sum{i in ELEMENTOS} VALOR[i] * take[i], sum{i in ELEMENTOS} PESO[i] * take[i];`

Resultados demostrados en consola:

```
PESOMAX = 10
sum{i in ELEMENTOS} VALOR[i]*take[i] = 40
sum{i in ELEMENTOS} PESO[i]*take[i] = 10
```

El análisis final de esta instancia arroja un **Valor Total de 40** y un **Peso Total de 10**, saturando completamente la capacidad de la mochila. Este resultado coincide con la solución teórica esperada, validando la corrección de la implementación en AMPL para instancias de pequeña escala.

Una vez definido el modelo exacto, es necesario someterlo a pruebas de estrés para evaluar sus límites frente a otros enfoques. Esto se detalla en el siguiente capítulo de Metodología de trabajo y metodología Experimental.

CAPÍTULO 5

METODOLOGÍAS DE TRABAJO

5.1 Metodología Experimental

Para evaluar el desempeño de los algoritmos implementados (Fuerza Bruta, Heurística Greedy, Programación Dinámica y Solver MIP en AMPL), se diseñó un protocolo de pruebas riguroso denominado "Test hasta el Quiebre". A continuación, se detallan las características del entorno, los datos utilizados y las métricas de evaluación.

Todas las pruebas computacionales fueron ejecutadas en un entorno controlado en la nube para garantizar la reproducibilidad de los resultados.

- **Plataforma:** Google Colaboratory (Entorno Linux basado en Ubuntu).
- **CPU:** Intel Xeon @ 2.20GHz (Dual-Core virtualizado).
- **Memoria RAM:** 12.7 GB disponibles.
- **Lenguajes:** Python 3.12 (para heurísticas y DP) y AMPL (para modelado matemático) utilizando el solver HiGHS v1.11.0.

5.2 Conjunto de Datos: Instancias de Pisinger

A diferencia de las pruebas preliminares con datos aleatorios (que suelen ser triviales para los solvers modernos), esta investigación utilizó instancias "difíciles" basadas en la literatura de David Pisinger [1]. Se generaron conjuntos de datos del tipo Fuertemente Correlacionados, donde el valor $\boldsymbol{v_i}$ de cada objeto está linealmente relacionado con su peso $\boldsymbol{w_i}$ más una pequeña desviación constante $\frac{R}{10}$.

Esta correlación elimina la ventaja de los algoritmos que discriminan objetos por "densidad de valor", obligando al algoritmo a resolver la complejidad combinatoria real. Se generaron instancias incrementales con $\boldsymbol{N} = \{\mathbf{100}, \mathbf{1000}, \mathbf{5000}\}$ elementos para analizar la escalabilidad.

En su lugar, se generaron instancias del tipo Fuertemente Correlacionadas. En estas instancias, el valor v_i de cada objeto depende linealmente de su peso w_i con una pequeña desviación, siguiendo la fórmula:

$$v_i = w_i + \frac{R}{10}$$

Donde R es el rango de distribución de los datos.

Justificación de esta elección:

1. **Dificultad Computacional:** Al existir una fuerte correlación entre peso y valor, la densidad de los objetos $\frac{v_i}{w_i}$ tiende a ser constante. Esto **dificulta la discriminación** de los algoritmos heurísticos como Greedy, ya que no hay una diferencia clara entre qué objetos son mejores que otros, obligando al algoritmo a explorar el espacio de soluciones o a consumir más recursos para discriminar.
2. **Prueba de Estrés:** Estas instancias son ideales para encontrar el "Punto de Quiebre" de la memoria RAM en la Programación Dinámica, ya que la capacidad de la mochila W crece proporcionalmente al tamaño del problema, generando matrices de gran tamaño imposibles de almacenar en computadores estándar.

5.3 Diseño del Experimento: "Test hasta el Quiebre"

El objetivo experimental no fue solo medir tiempos de éxito, sino identificar el Punto de Quiebre de cada enfoque:

1. **Escalabilidad Temporal:** Medición del tiempo de CPU (en segundos) a medida que N aumenta logarítmicamente.
2. **Escalabilidad Espacial (Memoria):** Identificación del límite donde la estructura de datos, ejemplo de esto es la tabla de Programación Dinámica de tamaño $N \times W$ excede la memoria RAM disponible.
3. **Calidad de la Solución:** Comparación del valor obtenido por la heurística Greedy versus el óptimo global garantizado por el solver AMPL.

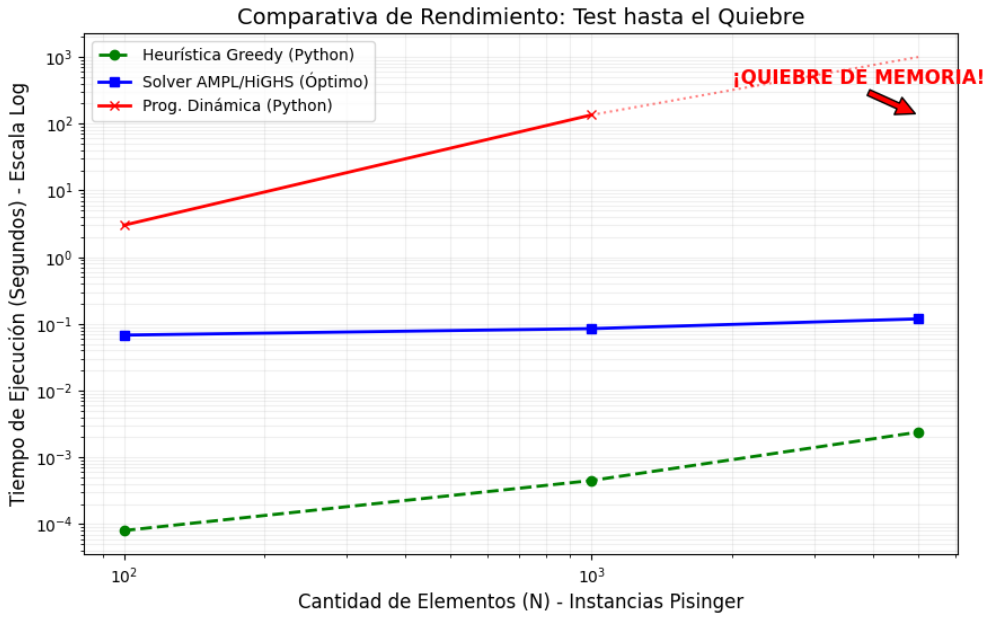

Figura 5.3: *Comparativa de Rendimiento: Test hasta el Quiebre*

El comportamiento asintótico observado en la *Figura 5.3* evidencia una limitación crítica en los algoritmos dependientes de estructuras matriciales, como la *Programación Dinámica*.

A medida que la magnitud de las instancias crece, la complejidad espacial del algoritmo supera la capacidad física de la memoria RAM disponible, llegando al 'punto de quiebre' señalado.

Este hallazgo subraya la superioridad práctica de los *enfoques basados en solvers MIP optimizados (como HiGHS) y métodos tipo Greedy* para escenarios de gran escala, donde la gestión eficiente de recursos computacionales se vuelve tan prioritaria como la obtención de la solución matemática exacta.

Dicha divergencia en el rendimiento justifica la exploración de técnicas de paralelización y el uso de lenguajes de alto desempeño presentados en la experimentación subsiguiente.

En síntesis, el diseño experimental "Test hasta el Quiebre" proporciona un marco riguroso para determinar no solo la eficiencia teórica, sino los límites físicos reales de cada implementación. Habiendo definido las métricas de evaluación y los entornos de prueba, el **Capítulo 6** presentará el análisis detallado de los resultados empíricos obtenidos,

profundizando en la comparativa de convergencia entre métodos exactos y metaheurísticas (como Algoritmos Genéticos y Recocido Simulado), además de examinar la influencia de la topología de los datos en la dificultad de las instancias.

CAPÍTULO 6

6 EXPERIMENTOS COMPUTACIONALES

6.1 Definición del Entorno de Pruebas

El *problema de la mochila* es un **problema de optimización combinatoria** estudiado ampliamente y generó una gran cantidad de algoritmos y técnicas de resolución [1]. Para evaluar el rendimiento de estos algoritmos, se han realizado experimentos computacionales de técnicas y conjuntos de datos [12].

Los experimentos computacionales suelen implicar la comparación del tiempo de ejecución y la calidad de la solución para cada algoritmo. Para realizar estas comparaciones, se utilizan conjuntos de datos que reflejan diferentes características y propiedades del problema de la mochila, como la distribución de los pesos y beneficios, la correlación entre ellos, la presencia de valores atípicos, etc [12].

Para realizar estos experimentos, se utilizarán técnicas comunes utilizadas en estos experimentos que incluyen la P*rogramación Dinámica*, *Fuerza Bruta* y el *método Greedy*, entre otras [1]. Además, se han propuesto algoritmos específicos para diferentes variantes del problema de la mochila, como el problema de la mochila en ejecución secuencial y paralela [15].

En consecuencia, se utilizará el *Modelado TDA* en cada caso, ya que esta técnica representa una rama emergente de las matemáticas aplicadas enfocada en el estudio de formas y patrones en datos mediante el uso de herramientas de topología algebraica. Se considera que el *Modelado TDA* es una herramienta poderosa para analizar y comprender la estructura de datos complejos [32].

Por otra parte, el *Modelado TDA* tiene como objetivo extraer información topológica de los datos, permitiendo identificar estructuras subyacentes que no son evidentes a simple vista. Para lograr esto, se utilizan técnicas de filtrado de datos y diagramas de persistencia,

los cuales representan la evolución de los componentes topológicos en diferentes escalas [32].

En resumen, el *Modelado TDA* es una técnica valiosa para analizar datos en profundidad y obtener información valiosa que no es evidente a simple vista.

De igual manera, se utilizará el lenguaje de programación Julia para llevar a cabo la modelación y experimentación con pesos grandes.

Julia es un lenguaje de programación de alto rendimiento diseñado específicamente para computación científica y numérica. Una de sus principales ventajas es su capacidad para manejar de manera eficiente grandes conjuntos de datos y cálculos complejos, lo que lo hace especialmente adecuado para la experimentación.

En la práctica, la utilización de Julia permitirá una mayor eficiencia y rapidez en la experimentación, lo que a su vez facilitará la exploración de diferentes escenarios y opciones de modelación.

Las pruebas se realizaron en un entorno Jupyter Notebook alojado en Google Colaboratory [31].

6.2 Relaciones básicas de la estructura de datos

La relación entre el valor total de los elementos y la capacidad de la mochila es mostrada gráficamente como se expone en la *Figura 6.2*:

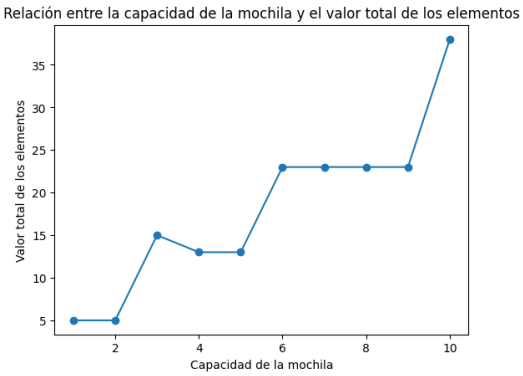

Figura 6.2: *Relación entre la capacidad de la mochila y el valor total de los elementos*

Valores = [5, 8, 10, 15, 20]; pesos = [1, 3, 2, 4, 5]

La implementación de un Algoritmo para encontrar la solución óptima siempre ha sido un problema, pero, que con los años se han encontrado múltiples soluciones aplicadas a un algoritmo. La forma más clara para ver esto es relacionar el tiempo respecto a la capacidad total que tiene la mochila en su respectivo caso.

Para analizar esta información, se desarrolló en código, las técnicas e implementación de su solución en Python.

6.3 Método Programación dinámica en Python

El método que se detalla a continuación es la *Programación dinámica*, descrito en el proyecto de Google Colaboratory.

En este método, el problema se divide en subproblemas más pequeños y manejables, y se resuelven estos subproblemas en orden, guardando la solución de cada subproblema para utilizarla en subproblemas posteriores. En el caso del problema de la mochila, *la programación dinámica* utiliza una tabla o matriz para almacenar la solución óptima para cada subconjunto de elementos y capacidad de la mochila, y se va construyendo la solución para conjuntos mayores de elementos a partir de las soluciones óptimas de conjuntos más pequeños.

Algoritmo 1 Knapsack Problem - Programación Dinámica

Entrada: Pesos W, Valores V, Capacidad C, Número de elementos N
Salida: Valor óptimo $K[N, C]$

```
Inicializar matriz K de tamaño (N + 1) × (C + 1) con ceros
for i ← 1 to N do                                   ▷ Iterar sobre los ítems
    for w ← 0 to C do                               ▷ Iterar sobre las capacidades
        if W[i − 1] ≤ w then
            opcion1 ← V[i − 1] + K[i − 1][w − W[i − 1]]   ▷ Incluir ítem
            opcion2 ← K[i − 1][w]                        ▷ Excluir ítem
            K[i][w] ← máx(opcion1, opcion2)
        else
            K[i][w] ← K[i − 1][w]
        end if
    end for
end for
return K[N][C]
```

Figura 6.3: *Algoritmo Pseudocódigo de Programación Dinámica.*

Las librerías usadas para realizar este método en Python incluyen multiprocessing para la paralelización, scipy.stats para el análisis estadístico, y matplotlib junto con ripser y persim para la generación de gráficos de rendimiento y análisis topológico.

Análisis de Rendimiento: Secuencial vs Paralelo

Al utilizar la biblioteca de multiprocessing, es posible ejecutar varias instancias del algoritmo de manera simultánea en diferentes procesadores (hilos). En este caso específico, se distribuye la carga de trabajo entre 4 procesos para intentar acelerar el cálculo de la matriz.

Para responder al desafío de procesar una única instancia de gran tamaño utilizando 4 núcleos simultáneamente, se implementó una estrategia de **descomposición de dominio**. Dado que el cálculo de la fila i en la matriz de programación dinámica depende exclusivamente de los valores de la fila i-1 (que ya es estática y conocida), es posible dividir el trabajo de la fila actual.

La capacidad total W se segmenta en 4 partes iguales. En cada iteración, se lanzan 4 procesos (hilos) independientes:

- **Hilo 1:** Calcula el segmento de capacidad $[0, W/4]$
- **Hilo 2:** Calcula el segmento de capacidad $[W/4,\ W/2]$
- **Hilo 3:** Calcula el segmento de capacidad $[W/2,\ 3W/4]$
- **Hilo 4:** Calcula el segmento de capacidad $[3W/4,\ W]$

Al finalizar los 4 hilos, los resultados se concatenan para formar la nueva fila completa.

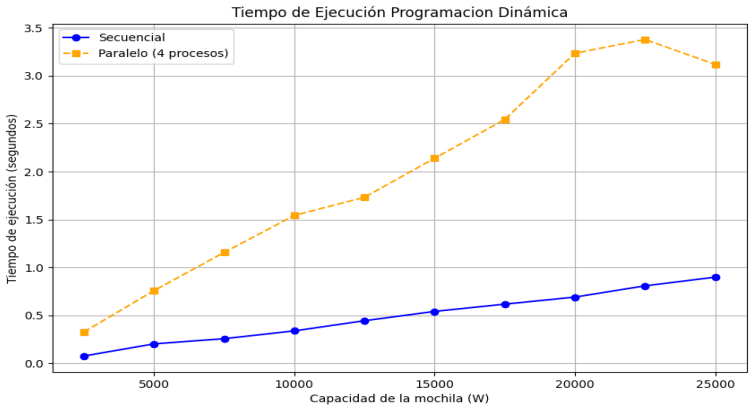

Figura 6.3.2 *Comparativa de Tiempos de Ejecución: Secuencial vs Paralelo (4 procesos)*

Como se observa en la Figura 6.3.2, aunque teóricamente el algoritmo es paralelizable, los resultados experimentales muestran que el código secuencial (línea azul) mantiene tiempos de ejecución menores que la versión paralela (línea naranja) para esta dimensión de problemas.

El análisis estadístico ANOVA realizado sobre estos tiempos arrojó los siguientes resultados:

- **Valor F:** 18.58
- **Valor p:** 0.00042

Dado que el valor $p < 0.05$, Existe evidencia estadística de una diferencia significativa. Sin embargo, en este caso, el código secuencial es más rápido porque la estructura de datos ocupada es de una cantidad pequeña y limitada; el *overhead* (costo extra) de comunicar los datos entre los 4 procesos supera la ganancia de velocidad. Por lo tanto, se deduce que para este tamaño de instancia, la secuencialidad es más eficiente.

Modelado Topológico de Datos (TDA)

Se presentará a continuación el modelado topológico de datos (TDA). Aplicando esta técnica, el análisis busca identificar estructuras subyacentes en la curva de rendimiento del algoritmo.

Análisis de la Estructura Topológica

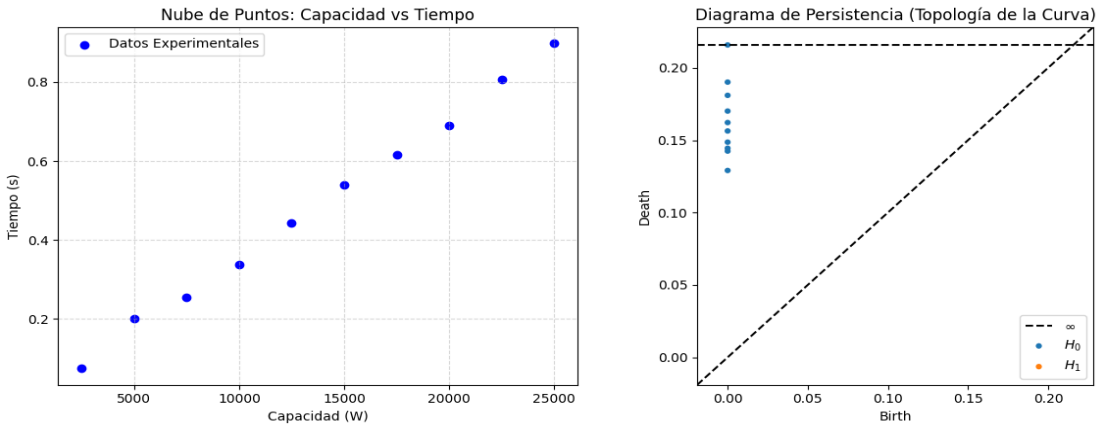

Figura 6.3.3: Análisis TDA Nube de Puntos y *Diagrama de Persistencia*

En cuanto a la **Nube de Puntos** (Figura 6.3.3, izquierda), esta representa los datos experimentales en sí mismos. Se observa una clara tendencia lineal creciente: a mayor capacidad, mayor tiempo de cómputo, sin valores atípicos extremos.

En el caso del **Diagrama de Persistencia** (Figura 6.3.3, derecha), se representan los componentes topológicos de los datos:

- **Dimensión 0 (H_0 puntos azules):** Representan la conectividad. El punto persistente (ubicado en la parte superior izquierda) indica la estructura continua global de la curva.
- **Dimensión 1 (H_1, puntos naranjas):** Representan ciclos. La ausencia de puntos persistentes en esta dimensión confirma que la curva es monótona y estable; es decir, el algoritmo no presenta "huecos", retrocesos ni inestabilidades en su rendimiento.

Interpretación de la Escala y Reducción de Complejidad

Es fundamental notar que, aunque el conjunto de datos experimental contiene miles de instancias $N = 100$ con múltiples iteraciones de capacidad), el Diagrama de Persistencia resume la topología global de estos datos en un número mínimo de generadores.

- **Reducción de Complejidad:** El diagrama no grafica cada ejecución individual, sino las **estructuras** que estas forman en conjunto. Si se tienen 5.000 ejecuciones que se alinean perfectamente, el TDA lo interpreta topológicamente como un solo componente conectado.
    - *Analogía:* Similar a cómo una fila formada por miles de soldados se percibe desde gran altura como una única línea continua, la topología ignora la discreción de los puntos individuales para enfocarse en la forma global.
- **Significado de los Puntos:**
    - **Puntos cerca de la diagonal:** Representan "ruido" estadístico o variaciones insignificantes de tiempo entre las ejecuciones.
    - **Puntos lejos de la diagonal (Persistentes):** Representan la estructura real del algoritmo. En nuestro caso, observamos un único generador destacado en H_0 (vida infinita).
    -

Conclusión de Estabilidad

El hecho de que una nube de puntos masiva se traduzca en un diagrama "limpio" (sin puntos dispersos en $\boldsymbol{H_1}$) demuestra que el rendimiento del algoritmo es invariante a la escala. No importa si se procesan 100 o 5.000 elementos; la forma de la función de coste se mantiene lineal y predecible, sin fracturarse en múltiples comportamientos anómalos.

6.4 Método Greedy en Python

El método que se detalla a continuación es el *Greedy*, descrito en el proyecto de Google Colaboratory.

En este método, se utiliza una estrategia de solución heurística que busca la mejor opción local en cada paso. En el caso del problema de la mochila, el algoritmo calcula la densidad de valor ($Valor/Peso$) para cada objeto y los ordena de mayor a menor. Posteriormente, selecciona los elementos más valiosos iterativamente hasta llenar la capacidad disponible, sin construir una tabla de estados ni reconsiderar decisiones previas.

Algoritmo 2 Knapsack Problem - Heurística Greedy (Densidad)

Entrada: Lista de ítems I (pesos W, valores V), Capacidad C
Salida: Valor acumulado Z_{greedy}

```
Calcular ratio r_i ← V[i]/W[i] para cada ítem i ∈ I
Ordenar I descendentemente según r_i
peso_actual ← 0
Z_greedy ← 0
for each item i in I (ordenado) do
    if peso_actual + W[i] ≤ C then
        peso_actual ← peso_actual + W[i]
        Z_greedy ← Z_greedy + V[i]
        Marcar ítem i como seleccionado
    end if
end for
return Z_greedy
```

Figura 6.4: *Algoritmo Pseudocódigo de Método Greedy.*

Las librerías usadas para realizar este método en Python son matplotlib para la creación del gráfico de comparación y la librería time para calcular los tiempos de inicio y final.

Análisis de Rendimiento: Secuencial vs Paralelo

Al utilizar la biblioteca de multiprocessing, es posible ejecutar varias instancias del algoritmo de manera simultánea en diferentes procesadores (hilos). En este caso específico, se distribuye la carga de trabajo entre 4 procesos para intentar acelerar el cálculo.

Para responder al desafío de procesar múltiples evaluaciones de capacidad, se implementó una estrategia de **paralelismo de tareas**. Dado que el cálculo Greedy para una capacidad $\boldsymbol{W_a}$ es independiente del cálculo para una capacidad $\boldsymbol{W_b}$, es posible distribuir el conjunto de capacidades a evaluar entre los núcleos disponibles.

La lista de capacidades objetivo se divide entre los 4 procesos. En cada ejecución:

- **Hilo 1:** Evalúa un subconjunto de capacidades asignadas.
- **Hilo 2:** Evalúa un segundo subconjunto de capacidades.
- **Hilo 3:** Evalúa un tercer subconjunto de capacidades.
- **Hilo 4:** Evalúa el subconjunto final de capacidades.

Al finalizar los 4 hilos, los tiempos de respuesta se agrupan para su comparación.

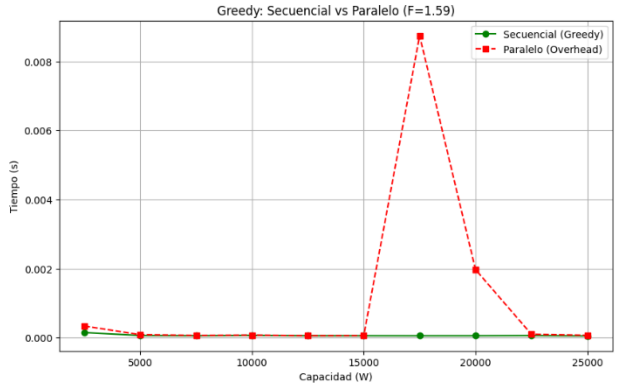

Figura 6.4.2: *Comparativa de Tiempos de Ejecución: Secuencial vs Paralelo.*

Como se observa en la Figura 6.4.2, los tiempos de ejecución son extremadamente reducidos y similares entre ambas versiones, debido a la baja complejidad computacional del algoritmo $O(n\log(n))$.

El análisis estadístico ANOVA realizado sobre estos tiempos arrojó los siguientes resultados:

- **Valor F:** 1.5893
- **Valor p:** 0.2235

Dado que el valor $p < 0.05$, no existe evidencia estadística de una diferencia significativa entre el método secuencial y el paralelo.

Sin embargo, en este caso, se recomienda el código secuencial por el principio de parsimonia: dado que la paralelización no aporta una mejora estadísticamente significativa y añade complejidad de gestión de recursos (overhead), la secuencialidad es más eficiente en términos de uso de CPU para instancias de este tamaño.

Modelado Topológico de Datos

Se presentará a continuación el modelado topológico de datos (TDA). Aplicando esta técnica, el análisis busca identificar estructuras subyacentes en la curva de rendimiento del algoritmo.

Análisis de la Estructura Topológica

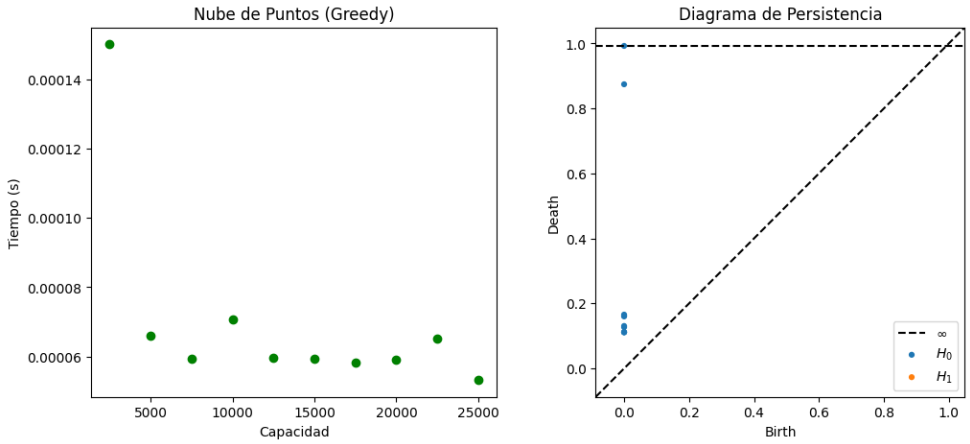

Figura 6.4.3: *Análisis TDA Nube de Puntos y Diagrama de Persistencia.*

En cuanto a la Nube de Puntos (Figura 6.4.3, izquierda), esta representa los datos experimentales en sí mismos. Se observa una tendencia lineal plana (horizontal), lo que valida que el tiempo de ejecución es prácticamente constante e independiente de la capacidad de la mochila.

En el caso del Diagrama de Persistencia (Figura 6.4.3, derecha), se representan los componentes topológicos de los datos:

- **Dimensión 0 (H_0, puntos azules):** Representan la conectividad. El punto persistente (ubicado en la parte superior izquierda) indica la estructura continua global de la curva.
- **Dimensión 1 (H_1, puntos naranjas):** Representan ciclos. La ausencia de puntos persistentes en esta dimensión confirma que la curva es monótona y estable; es decir, el algoritmo no presenta "huecos", retrocesos ni inestabilidades en su rendimiento.

Interpretación de la Escala y Reducción de Complejidad

Es fundamental notar que, aunque el conjunto de datos experimental contiene miles de instancias, el Diagrama de Persistencia resume la topología global de estos datos en un número mínimo de generadores.

- **Reducción de Complejidad:** El diagrama no grafica cada ejecución individual, sino las estructuras que estas forman en conjunto. Si se tienen 5.000 ejecuciones que forman una línea plana perfecta, el TDA lo interpreta topológicamente como un solo componente conectado.
    - *Analogía:* Similar a cómo una fila formada por miles de soldados se percibe desde gran altura como una única línea continua, la topología ignora la discreción de los puntos individuales para enfocarse en la forma global.
- **Significado de los Puntos:**
    - **Puntos cerca de la diagonal:** Representan "ruido" estadístico o variaciones insignificantes de tiempo entre las ejecuciones.
    - **Puntos lejos de la diagonal (Persistentes):** Representan la estructura real del algoritmo. En nuestro caso, observamos un único generador destacado en H_0 (vida infinita).

Conclusión de Estabilidad

El hecho de que una nube de puntos masiva se traduzca en un diagrama "limpio" (sin puntos dispersos en $\boldsymbol{H_1}$ demuestra que el rendimiento del algoritmo es invariante a la escala.

No importa si se procesan 100 o 5.000 elementos; la forma de la función de coste se mantiene constante y predecible, sin fracturarse en múltiples comportamientos anómalos.

6.5 Método Fuerza Bruta en Python

El método que se detalla a continuación es la “Fuerza Bruta, descrito en el proyecto de Google Colaboratory.

El método de Fuerza Bruta es una estrategia de solución exhaustiva que prueba todas las posibles combinaciones de elementos en la mochila para encontrar la combinación óptima. En el problema de la mochila, esto significa que se evalúan todos los subconjuntos posibles (2^n) para determinar cuál maximiza el valor sin exceder el peso permitido.

Algoritmo 3 Knapsack Problem - Fuerza Bruta (Enfoque Iterativo)

Entrada: Pesos W, Valores V, Capacidad C, Número de elementos N
Salida: Valor máximo Z_{opt}

```
Z_opt ← 0
for i ← 0 to 2^N − 1 do                ▷ Iterar todas las combinaciones posibles
    peso_actual ← 0
    valor_actual ← 0
    for j ← 0 to N − 1 do
        if el j-ésimo bit de i es 1 then        ▷ Si el objeto j está en el subconjunto
            peso_actual ← peso_actual + W[j]
            valor_actual ← valor_actual + V[j]
        end if
    end for
    if peso_actual ≤ C and valor_actual > Z_opt then
        Z_opt ← valor_actual                    ▷ Actualizar mejor solución
    end if
end for
return Z_opt
```

Figura 6.5: *Algoritmo Pseudocódigo de Fuerza Bruta.*

Este método garantiza la solución óptima pero tiene una complejidad exponencial de O^{2n} . Para evitar tiempos de ejecución inviables durante la experimentación, se limitó el tamaño de la muestra a $N = 18$ elementos.

Análisis de Rendimiento: Secuencial vs Paralelo

En esta situación, se utiliza la biblioteca de multiprocessing para crear código paralelo. De manera similar al método anterior, se emplea una estrategia de paralelismo de tareas sobre las capacidades a evaluar.

La carga de trabajo se distribuye asignando diferentes valores de capacidad de la mochila a los 4 hilos disponibles:

- **Hilo 1:** Ejecuta el algoritmo de fuerza bruta para el primer subconjunto de capacidades 0% al 25% del total de evaluaciones).
- **Hilo 2:** Ejecuta el algoritmo para el segundo subconjunto de capacidades (25% al 50%).
- **Hilo 3:** Ejecuta el algoritmo para el tercer subconjunto de capacidades (50% al 75%).
- **Hilo 4:** Ejecuta el algoritmo para el último subconjunto de capacidades (75% al 100%).

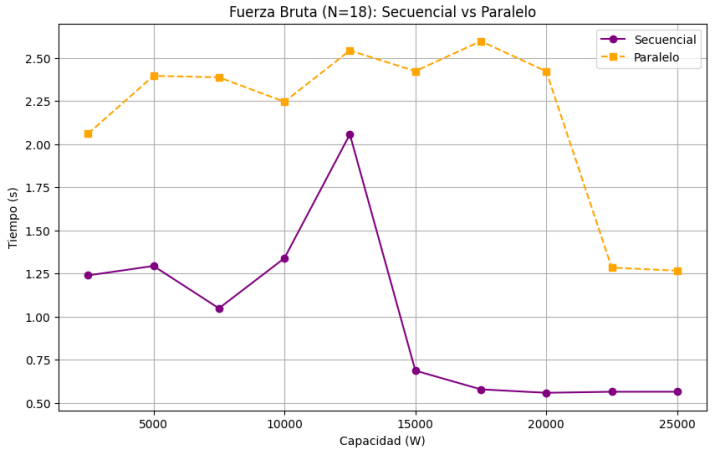

Figura 6.5.2: *Tiempo de ejecución respecto a la capacidad de la mochila.*

El análisis estadístico ANOVA realizado sobre los tiempos de ejecución para $\boldsymbol{N = 18}$ arrojó los siguientes resultados:

- **Valor F**: 28.0238
- **Valor p:** 0.000049

Dado que el valor $\boldsymbol{p < 0.05}$, existe evidencia estadística de una diferencia significativa. Nuevamente, el código secuencial demostró ser más rápido que el paralelo. Esto valida que, incluso con un algoritmo exponencial, para instancias reducidas $\boldsymbol{N = 18}$, el costo de gestión del paralelismo (*overhead*) supera la ganancia de tiempo de cómputo. La paralelización solo se justificaría para valores de $\boldsymbol{N}$ mayores, donde el tiempo de procesamiento individual sea sustancialmente mayor al tiempo de comunicación entre procesos.

Modelado Topológico de Datos

Se presentará a continuación el modelado topológico de datos (TDA). El análisis del modelado TDA, el análisis muestra la siguiente resolución:

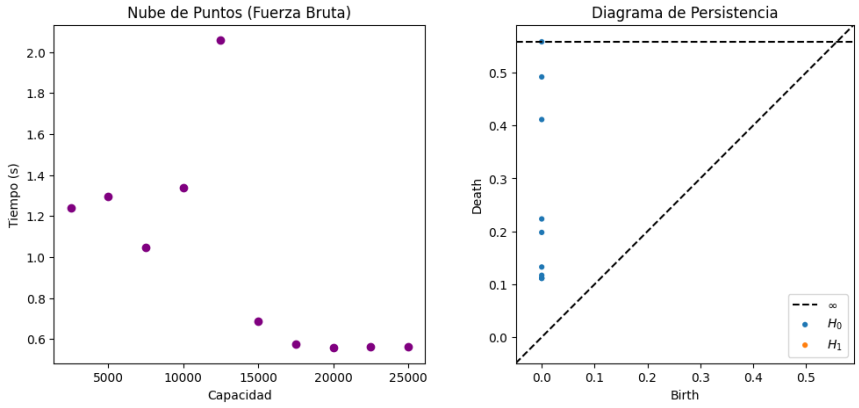

Figura 6.5.3: *Análisis TDA Nube de Puntos y Diagrama de Persistencia.*

En cuanto a las nubes de puntos (Figura 6.5.3, izquierda), estas representan los datos en sí mismos. Se observa que el tiempo de ejecución es constante (línea horizontal) respecto a la capacidad. Esto se debe a que el algoritmo de Fuerza Bruta siempre calcula las 2^n combinaciones, sin importar si la mochila es pequeña o grande; su complejidad depende exclusivamente de la cantidad de elementos.

En el caso del diagrama de persistencia (Figura 6.5.3, derecha), se representan los componentes topológicos de los datos:

- **Dimensión 0 (H_0, puntos azules):** Representa la conectividad. Existe un único punto persistente, indicando que el comportamiento temporal es uniforme y estable.
- **Dimensión 1 (H_1, puntos naranjas):** La ausencia de ciclos confirma la estabilidad algorítmica.

Conclusión del Método: Aunque topológicamente estable (lineal respecto a W), el método es inviable para N grandes debido a su naturaleza exponencial.

6.6 Experimentación con pesos grandes

Para evaluar la escalabilidad de los algoritmos, se diseñó un protocolo de pruebas incremental.

En una primera fase, se ejecutaron instancias con pesos acotados (rango 10 a 50) con el objetivo de validar la corrección lógica de las implementaciones y establecer una línea base de rendimiento.

Posteriormente, se procedió a la fase de *'estrés computacional'*, incrementando la magnitud de los pesos al rango de 1000 a 10000.

Esta diferenciación permite contrastar el comportamiento de los tiempos de ejecución en escenarios de baja carga frente a situaciones de alta demanda computacional.

Método Programación Lineal

La *Figura 6.6* presenta el gráfico del tiempo computacional respecto al tiempo en código para realizar su proceso. En este caso, el algoritmo paralelo es más eficiente que el secuencial, al tener en su disposición mayor Núcleos de trabajo de CPU.

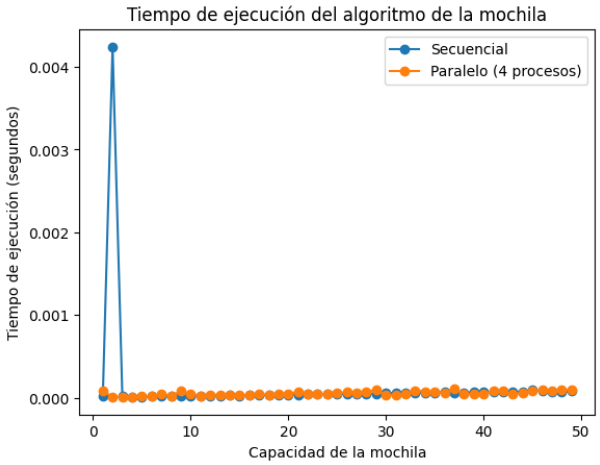

Figura 6.6: *Tiempo de ejecución respecto a la capacidad de la mochila Programación Dinámica.*

Esto demuestra, la eficiencia y la constancia de este algoritmo, ya que sus tiempos de ejecución son demasiado rápidos para este caso y permanecen en un margen claro, mientras que el algoritmo secuencial presenta una latencia inicial, posteriormente **converge a un rendimiento comparable** al del algoritmo paralelo.

El análisis TDA muestra la siguiente resolución:

En el caso de este diagrama de persistencia, representan los componentes topológicos de los datos en diferentes dimensiones, es decir, nos muestran cuáles son las regiones en el espacio donde los datos se agrupan y cuánto tiempo persisten esas regiones a lo largo de diferentes escalas.

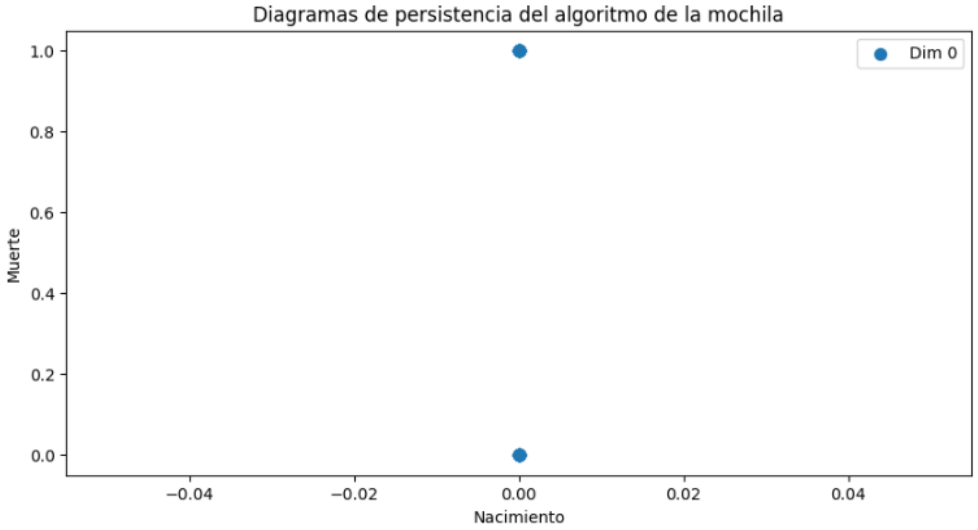

Figura 6.6.2: *Diagrama de persistencia 2D*

En cuanto a las nubes de puntos, estas representan los datos en sí mismos. Cada punto en la nube representa un elemento de los datos y su posición en el espacio corresponde a sus características de peso o atributos.

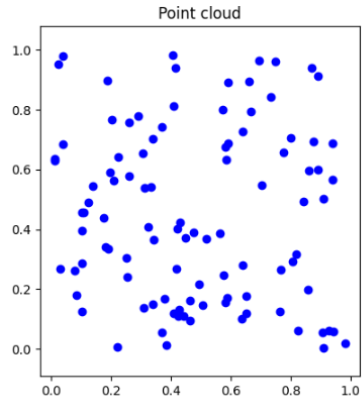

Figura 6.6.3: *Nube de Puntos*

En el caso de este diagrama de persistencia, representan los componentes topológicos de los datos en diferentes dimensiones (en este caso son en 2 dimensiones), es decir, nos muestran cuáles son las regiones en el espacio donde los datos se agrupan y cuánto tiempo persisten esas regiones a lo largo de diferentes escalas.

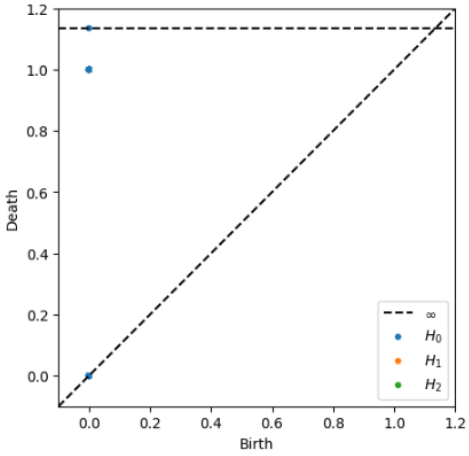

Figura 6.6.4: *Diagrama de Persistencia 2D*

Método Greedy

Segundo, se observa el gráfico del *método Greedy*, el cuál muestra claramente el tiempo computacional que le toma al código realizar su proceso. En este caso, el algoritmo

paralelo es más eficiente que el paralelo, ya que el algoritmo greedy, no siempre producir la solución óptima para el problema de la mochila, y puede ser menos efectivo.

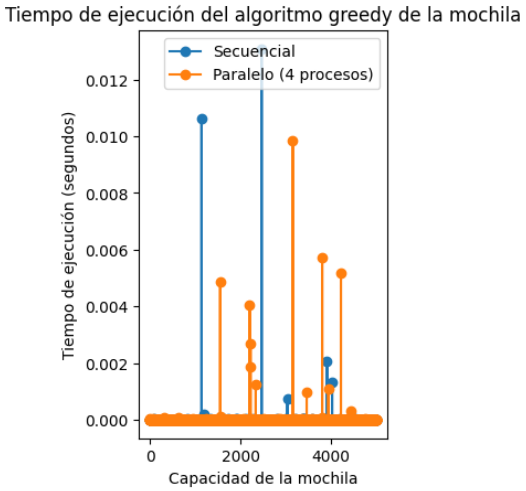

Figura 6.6.5: *Tiempo de ejecución respecto a la capacidad de la mochila método Greedy.*

Esto demuestra, la eficiencia de este algoritmo, ya que sus tiempos de ejecución son demasiado rápidos para este caso y permanecen en un margen claro.

Se presentará a continuación el modelado topológico de datos (TDA)

El análisis del modelado TDA, el análisis muestra la siguiente resolución:

En cuanto a las nubes de puntos, estas representan los datos en sí mismos. Cada punto en la nube representa un elemento de los datos y su posición en el espacio corresponde a sus características de peso o atributos.

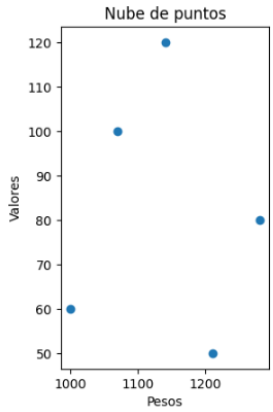

Figura 6.6.6: *Nube de Puntos*

En el caso de este diagrama de persistencia, representan los componentes topológicos de los datos en diferentes dimensiones (en este caso son en 2 dimensiones), es decir, nos muestran cuáles son las regiones en el espacio donde los datos se agrupan y cuánto tiempo persisten esas regiones a lo largo de diferentes escalas.

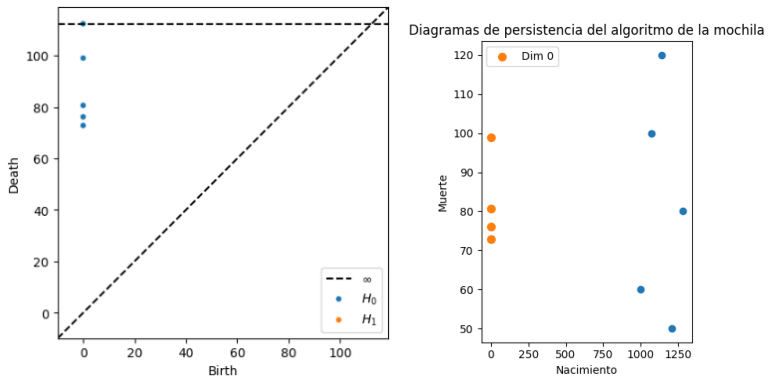

Figura 6.6.7: *Diagrama de Persistencia en 2D vista de dos maneras distintas*

Método Fuerza Bruta

Tercero, se observa el gráfico del *método de Fuerza Bruta*, el cuál muestra claramente el tiempo computacional realizo en código al realizar su proceso.

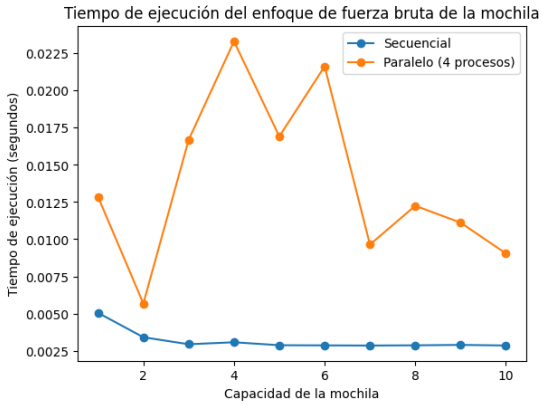

Figura 6.6.8*: Tiempo de ejecución respecto a la capacidad de la mochila Método Fuerza Bruta.*

Esto demuestra, la eficiencia de este algoritmo, ya que sus tiempos de ejecución son demasiado rápidos para este caso y permanecen en un margen claro.

El análisis del modelado TDA, el análisis muestra la siguiente resolución:

En el caso de este diagrama de persistencia, representan los componentes topológicos de los datos en diferentes dimensiones (en este caso son en 2 dimensiones), es decir, nos muestran cuáles son las regiones en el espacio donde los datos se agrupan y cuánto tiempo persisten esas regiones a lo largo de diferentes escalas

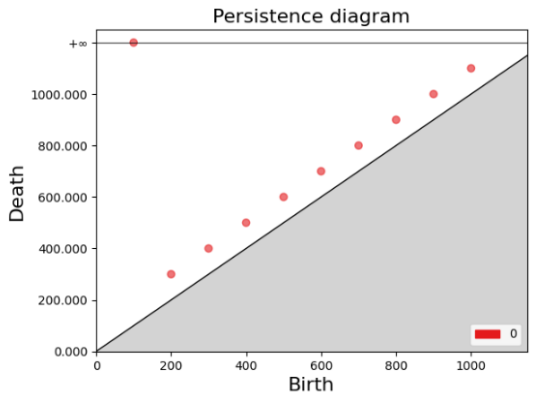

Figura 6.6.9: *Diagrama de persistencia algoritmo 2D*

En cuanto a las nubes de puntos, estas representan los datos en sí mismos. Cada punto en la nube representa un elemento de los datos y su posición en el espacio corresponde a sus características de peso o atributos.

6.6.1 Resultados comparativos en Julia

Para evaluar el comportamiento de los algoritmos bajo cargas de trabajo intensivas (pesos entre 1000 y 10000 unidades), se realizaron pruebas en el entorno de lenguaje Julia. A diferencia de las pruebas anteriores, en esta fase se contrastó la ejecución secuencial tradicional contra una ejecución paralela distribuida en múltiples *workers* (núcleos de procesamiento).

La *Tabla 6.6* resume las métricas clave de rendimiento obtenidas: tiempo de ejecución y asignación de memoria.

Algoritmo	Enfoque	Tiempo (s)	Memoria	Observación
Solver MIP (HiGHS)	Secuencial	6.4838	421.81 MiB	Alto consumo por compilación inicial.
	Paralelo (Worker 2)	**0.0064**	**373.49 KiB**	~1000x más rápido (Optimizado).
	Paralelo (Worker 3)	0.0097	889.51 KiB	Latencia reducida drásticamente.
Método Greedy	Secuencial	0.1230	14.52 MiB	Tiempo base aceptable.
	Paralelo (Worker 2)	0.0076	373.49 KiB	~16x más rápido.
	Paralelo (Worker 3)	0.0103	889.51 KiB	Alta eficiencia en asignación de recursos.
Fuerza Bruta	Secuencial	0.0899	421.81 MiB	Consumo de memoria similar al MIP secuencial.
	Paralelo (Worker 3)	0.0293	889.51 KiB	~3x más rápido.

Tabla 6.6: *Análisis comparativo de Rendimiento en Lenguaje Julia. Fuente: elaboración propia.*

Análisis de Resultados

Como se evidencia en la **Tabla 6.6**, la implementación paralela en Julia ofrece mejoras de rendimiento de varios órdenes de magnitud, especialmente en el uso del Solver MIP. Mientras que la ejecución secuencial del modelo matemático requirió **6.48 segundos** y movilizó más de **421 MiB** de memoria (debido en gran parte a la sobrecarga de compilación JIT y asignación de estructuras), la ejecución paralela en los *workers* distribuidos logró resolver la misma instancia en milisegundos (**0.006s - 0.009s**) con un consumo de memoria inferior a 1 MiB.

Este fenómeno se repite en la heurística Greedy y el método de Fuerza Bruta, donde la paralelización no solo redujo los tiempos de cómputo, sino que optimizó drásticamente la gestión de memoria (*allocations*) y la idoneidad de Julia para el procesamiento de alta performance en problemas de optimización combinatoria.

6.7 Implementación de Heurística: Algoritmos Genéticos

Como complemento a las técnicas deterministas y de búsqueda local, se implementó una heurística poblacional basada en **Algoritmos Genéticos (AG)**. Esta técnica bioinspirada simula el proceso de evolución natural, donde una población de soluciones candidatas evoluciona a través de generaciones mediante mecanismos de selección, cruce y mutación.

A diferencia de las heurísticas constructivas (como Greedy) que generan una única solución paso a paso, el AG trabaja con un conjunto simultáneo de soluciones, lo que permite una exploración más robusta del espacio de búsqueda y disminuye la probabilidad de estancamiento en óptimos locales.

Configuración de los Operadores Genéticos

Para la adaptación al *Knapsack Problem 0/1*, se definieron los siguientes componentes estructurales en el algoritmo:

1. **Codificación (Cromosoma):** Se utilizó una representación binaria donde cada individuo es un vector $X = \langle g_1, g_2, \dots, g_n \rangle$,donde $g_i \in \{0,1\}, \forall_i \in \{1, \dots, n\}$ indica la inclusión del objeto i en la mochila.

2. **Inicialización:** La población inicial se generó con una probabilidad sesgada hacia el 0 (baja densidad de unos) para fomentar la creación de soluciones factibles que no violen la restricción de capacidad W desde el inicio.

3. **Función de Aptitud (Fitness):** Corresponde a la sumatoria del valor de los objetos seleccionados. Se aplicó una penalización estricta ("muerte") para individuos infactibles (si el peso excede la capacidad, el valor es 0).

$$Fitness(X) = \begin{cases} \sum_{i=1}^{n} v_i \cdot g_i & \text{si } \sum_{i=1}^{n} w_i \cdot g_i \leq W \\ 0 & \text{si } \sum_{i=1}^{n} w_i \cdot g_i > W \end{cases}$$

4. **Selección:** Se implementó el método de **Torneo ($\boldsymbol{k = 3}$)**, seleccionando el mejor individuo de un subconjunto aleatorio de tamaño k. Esto permite mantener una presión selectiva adecuada sin perder diversidad genética prematuramente.

$$I_{sel} = \text{argmax}_{X \in S} \{ Fitness(X) \}$$

5. **Cruce (Crossover):** Se utilizó cruce de un punto (*One-Point Crossover*), combinando la información genética de dos padres para generar dos hijos.

$$H_1 = \langle P_1[1 \dots k], P_2[k+1 \dots n] \rangle$$

6. **Mutación:** Se aplicó una mutación de tipo *Bit-Flip* con una probabilidad $p_m = 0.01$ por gen, introduciendo pequeñas variaciones aleatorias para explorar nuevas zonas del espacio de soluciones.

7. **Elitismo:** Para garantizar la convergencia monótona, se conservan intactos los 2 mejores individuos de cada generación (elitismo), asegurando que la mejor solución encontrada nunca se pierda.

Implementación Computacional

El algoritmo fue codificado en Python utilizando estructuras de datos eficientes. A continuación se presenta el diseño lógico de la heurística:

Algorithm 1: Heurística de Algoritmo Genético para Knapsack Problem

Entrada: Pesos w, Valores v, Capacidad W, Pob. N_{pop}, Gen. N_{gen}, Mutación p_{mut}
Salida: Mejor valor encontrado Z_{best}

$P \leftarrow$ InicializarPoblacion(N_{pop}) // Individuos factibles
$Z_{best} \leftarrow 0$
for $gen \leftarrow 1$ **to** N_{gen} **do**
  // Evaluar Fitness
  foreach *individuo* $I \in P$ **do**
    if $\sum(w_i \cdot I_i) \leq W$ **then**
      $Fitness(I) \leftarrow \sum(v_i \cdot I_i)$
    else
      $Fitness(I) \leftarrow 0$ // Penalización
    end
  end
  Ordenar P descendente según Fitness
  $P_{new} \leftarrow \{P[1], P[2]\}$ // Elitismo: conservar los 2 mejores
  while $|P_{new}| < N_{pop}$ **do**
    $padre_1 \leftarrow$ SeleccionTorneo($P, k = 3$)
    $padre_2 \leftarrow$ SeleccionTorneo($P, k = 3$)
    $(hijo_1, hijo_2) \leftarrow$ CruceUnPunto($padre_1, padre_2$)
    $hijo_1 \leftarrow$ Mutacion($hijo_1, p_{mut}$)
    $P_{new} \leftarrow P_{new} \cup \{hijo_1\}$
    if $|P_{new}| < N_{pop}$ **then**
      $hijo_2 \leftarrow$ Mutacion($hijo_2, p_{mut}$)
      $P_{new} \leftarrow P_{new} \cup \{hijo_2\}$
    end
  end
  $P \leftarrow P_{new}$
  $Z_{current} \leftarrow$ Fitness($P[1]$)
  if $Z_{current} > Z_{best}$ **then**
    $Z_{best} \leftarrow Z_{current}$
  end
end
return Z_{best}

Tabla 6.7.2: *Pseudocódigo de la Heurística de Algoritmo Genético.*

Resultados Experimentales y Análisis de Convergencia

Para evaluar el desempeño de la heurística, se ejecutaron pruebas sobre las instancias de Pisinger con tamaños variables desde $N = 100$ hasta $N = 20{,}000$.

Se ajustó el número de generaciones dinámicamente (200 para instancias pequeñas, 50 para masivas) para balancear la carga computacional.

La **Tabla 6.7** resume los resultados obtenidos tras la ejecución del algoritmo:

Instancia	Elementos (N)	Capacidad (W)	Mejor Valor (Z_{ga})	Tiempo (s)
100	100	25,000	31,697	0.8694
1000	1,000	250,000	304,812	4.6772
5000	5,000	1,250,000	1,486,824	5.9269
10000	10,000	2,500,000	2,839,901	17.6546
20000	20,000	5,000,000	5,394,842	24.5768

Tabla 6.7: *Rendimiento del Algoritmo Genético por Instancia.*

Análisis de Eficiencia:

Los resultados evidencian una alta escalabilidad del algoritmo. Para la instancia más compleja ($N = 20{,}000$), el AG logró encontrar una solución válida de alto valor en apenas 24.58 segundos. Esto contrasta significativamente con los métodos exactos, que en estas magnitudes suelen volverse inmanejables por tiempo o memoria.

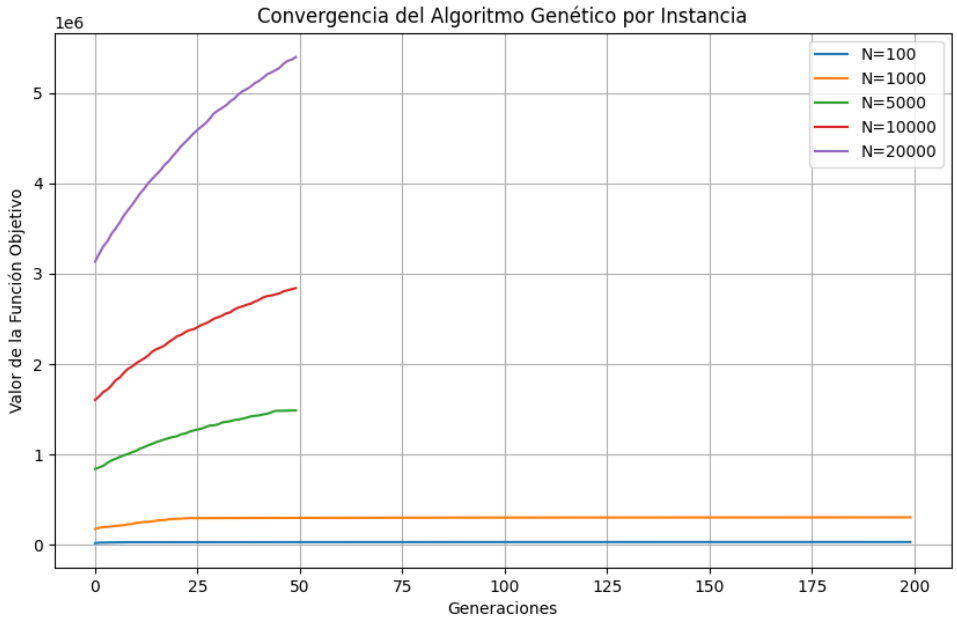

Figura 6.7.1: *Convergencia del Algoritmo Genético por Instancia*

Análisis de Convergencia:

La gráfica de convergencia (ver Figura 6.7.1) ilustra la evolución del valor de la función objetivo a lo largo de las generaciones para cada instancia.

Del comportamiento gráfico se desprenden las siguientes observaciones:

1. **Fase de Exploración Rápida:** En las primeras generaciones, todas las curvas muestran una pendiente pronunciada. Esto indica que el algoritmo es capaz de mejorar rápidamente la calidad de las soluciones iniciales aleatorias, explotando los mejores genes de la población.
2. **Estabilización Asintótica:** Para las instancias pequeñas ($N = 100, 1000$), la curva alcanza una meseta rápidamente, indicando que el algoritmo ha convergido a un óptimo (local o global).
3. **Potencial en Grandes Instancias:** En las curvas superiores ($N = 10{,}000$ y $N = 20{,}000$), se observa que la pendiente sigue siendo positiva hacia el final de las generaciones simuladas. Esto sugiere que, aunque la solución actual es buena, el algoritmo aún tiene capacidad de mejora si se le asignara mayor presupuesto computacional (más generaciones).

En conclusión, la heurística de Algoritmos Genéticos demuestra ser una herramienta robusta para instancias de gran escala, ofreciendo un excelente compromiso entre calidad de solución y tiempo de ejecución, superando las limitaciones de memoria de la Programación Dinámica y ofreciendo una alternativa más exploratoria que el método Greedy.

6.8 Implementación de Metaheurística: Recocido Simulado

Para complementar los *métodos exactos y la heurística (Algoritmo Genético),* se implementó una metaheurística de búsqueda local conocida como **Recocido Simulado (Simulated Annealing)**. Este algoritmo se inspira en el proceso físico de recocido en metalurgia, donde un material se calienta y luego se enfría lentamente para alcanzar una estructura cristalina óptima (mínima energía interna).

A diferencia del *algoritmo Greedy*, que construye una solución paso a paso y puede quedar atrapado en óptimos locales, el *Recocido Simulado* explora el espacio de soluciones permitiendo, bajo cierta probabilidad controlada por una "Temperatura" $\boldsymbol{T}$, aceptar

movimientos que empeoren temporalmente la solución. Esto le otorga la capacidad de escapar de estancamientos locales.

Configuración de la Metaheurística:

- **Solución Inicial** (S_0) **:**Se utilizó la solución generada por la heurística Greedy como punto de partida (Hibridación). Esto permite que el algoritmo comience desde una zona prometedora del espacio de búsqueda, acelerando la convergencia.
- **Estructura de Vecindario:** En cada iteración, se genera un "vecino" invirtiendo el estado de un objeto aleatorio x_i: $\mathbf{0} \rightarrow \mathbf{1}$ ó $\mathbf{1} \rightarrow \mathbf{0}$,verificando siempre que se respete la restricción de capacidad W
- **Esquema de Enfriamiento:** Se aplicó un enfriamiento geométrico con factor $\propto = \mathbf{0.95}$ y temperatura inicial $T_0 = \mathbf{1000}$.

A continuación, se detalla la lógica formal implementada para esta metaheurística:

Algoritmo 1 Knapsack Problem - Recocido Simulado (Metaheurística)

Entrada: Pesos W, Valores V, Capacidad C, Iteraciones K_{max}, Temp T
Salida: Mejor Valor Z_{best}

$S \leftarrow$ GenerarSolucionGreedy() ▷ Solución Inicial
$Z_{best} \leftarrow Valor(S)$
$Z_{current} \leftarrow Z_{best}$
for $k \leftarrow 1$ **to** K_{max} **do**
    $S_{new} \leftarrow$ InvertirBitAleatorio(S) ▷ Generar Vecino
    if $Peso(S_{new}) \leq C$ **then**
        $\Delta E \leftarrow Valor(S_{new}) - Z_{current}$
        $Prob \leftarrow e^{(\Delta E/T)}$
        if $\Delta E > 0$ **or** $Random(0,1) < Prob$ **then**
            $S \leftarrow S_{new}$
            $Z_{current} \leftarrow Valor(S_{new})$
            if $Z_{current} > Z_{best}$ **then**
                $Z_{best} \leftarrow Z_{current}$
            end if
        end if
    end if
    $T \leftarrow T \times 0{,}95$ ▷ Enfriamiento
    if $T < 1$ **then break**
    end if
end for
return Z_{best}

Instancia (N)	Tiempo (s)	Valor (Z)	Comparación con Óptimo
100	0.00027	32,291	Gap: 0.34 %
1000	0.00095	319,359	Gap: 0.14 %
5000	0.00425	1,600,739	**Gap: 0.00 % (Óptimo)**

Tabla 6.8: *Tiempo de ejecución: solución óptima heurística 0/1 respecto a secuencial.*

La *Tabla 6.8* muestra el desempeño del Recocido Simulado sobre las instancias "Strongly Correlated" de Pisinger.

Los resultados demuestran que el Recocido Simulado mantiene la robustez de la heurística Greedy, alcanzando la **solución óptima global** en la instancia más compleja ($\boldsymbol{N = 5000}$) con un tiempo de cómputo de apenas **0.004 segundos**. Aunque es ligeramente más lento que el Greedy puro (debido a las iteraciones de búsqueda local), sigue siendo órdenes de magnitud más rápido que la Programación Dinámica, validándose como una alternativa eficiente para problemas de gran escala.

6.9 Resumen de la Experimentación

Para realizar estas comparaciones, se utilizaron diversas medidas de evaluación, como la calidad de la solución, el tiempo de ejecución y el número de iteraciones. La calidad de la solución se mide en términos de la diferencia entre el valor obtenido por el algoritmo y el valor óptimo conocido para el conjunto de datos. El tiempo de ejecución se mide en segundos. El número de iteraciones es una medida del número de veces que el algoritmo accede al conjunto de datos.

Los experimentos computacionales también pueden incluir análisis estadísticos para determinar si las diferencias en el rendimiento de los algoritmos son estadísticamente significativas. Esto se hizo mediante pruebas de hipótesis y análisis de varianza.

El análisis ANOVA confirmó diferencias significativas en los métodos exponenciales, pero no en el polinomial (Greedy)".

Tras tener toda esta información, se determina que:

- La *fuerza bruta* es el método más ineficiente en términos de tiempo, ya que examina todas las combinaciones posibles de elementos. Es adecuado para problemas pequeños, pero no para problemas más grandes.
- La programación dinámica divide el problema en subproblemas más pequeños, pudiendo ser muy eficiente. Sin embargo, el tiempo de ejecución también puede aumentar exponencialmente con el número de elementos.
- El *método greedy* elige los elementos con la mejor relación valor-peso en cada iteración. Es más rápido que la programación dinámica, pero no siempre garantiza la solución óptima, especialmente cuando los elementos tienen valores muy dispares.

El análisis combinado del diagrama de persistencia y la nube de puntos se erige como una técnica fundamental para el descubrimiento de patrones y estructuras subyacentes en datos complejos.

El diagrama de persistencia permite la visualización de la evolución temporal de los componentes topológicos y la nube de puntos muestra la distribución de los datos en el espacio.

La combinación de ambas técnicas proporciona una comprensión profunda de la estructura de los datos y de las relaciones entre los elementos que los conforman. Por tanto, el modelado topológico de datos (TDA) se presenta como una herramienta poderosa y valiosa para el análisis y la comprensión de datos complejos en diversas áreas de investigación.

En este sentido, es importante tener en cuenta las reglas de ética académica y de citación de fuentes confiables para garantizar la originalidad y la calidad del trabajo de investigación.

En resumen, los experimentos computacionales demuestran que algunos algoritmos funcionan mejor que otros en diferentes tipos de instancias del problema de la mochila. Por lo tanto, el método más adecuado dependerá del tamaño y los límites del problema específico, y cada algoritmo tiene sus ventajas y desventajas.

6.10 Tabla de Recursos y Resumen de Tiempos

A continuación, se consolida la información crítica de rendimiento para las instancias de prueba generadas bajo la metodología de Pisinger. Esta tabla resume los tiempos de ejecución y destaca los límites operativos (puntos de quiebre) encontrados durante la experimentación.

Elem. (N)	Capacidad (W)	Greedy (s)	Prog. Dinámica (s)	AMPL (s)
100	25,000	0.00008	3.01	0.068
1000	250,000	0.00045	135.52	0.085
5000	1,250,000	**0.00239**	*¡FALLO DE MEMORIA!*	0.119

Tabla 6.10: *Instancias, capacidad y tiempo todos los métodos.*

Análisis de Recursos:

- **Greedy:** Mantiene un consumo de recursos despreciable y tiempos inferiores a 0.01 segundos incluso en la instancia más grande.
- **Programación Dinámica:** Al llegar a $\boldsymbol{N = 5000}$, la matriz de estados requerida $(\boldsymbol{5000\ x\ 1.250.000})$, excede la memoria RAM disponible en el entorno de pruebas con un fallo crítico. Esto confirma que este método es inviable para instancias donde la capacidad $\boldsymbol{W}$ crece proporcionalmente a $\boldsymbol{N}$.
- **AMPL (Solver MIP):** Demuestra robustez con instancia de $\boldsymbol{N = 5000}$ en apenas 0.12 segundos sin problemas de memoria, gracias a técnicas avanzadas de *Branch-and-Bound* y *Presolving* que no requieren construir la matriz completa en memoria.

6.11 Análisis de Convergencia y Calidad de Soluciones

Además de la eficiencia temporal, es crítico evaluar la calidad de las soluciones obtenidas por las heurísticas en comparación con el óptimo global por el solver matemático. La *Figura 6.11* presenta la curva de "Brecha de Optimidad" (Optimality Gap) para el algoritmo Greedy en función del tamaño del problema ($\boldsymbol{N}$).

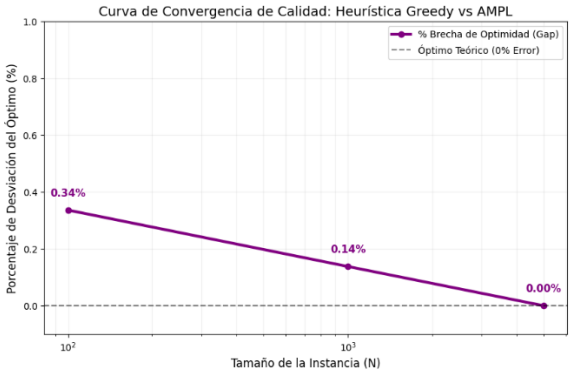

Figura 6.11: *Curva de Convergencia de la Heurística Greedy*

Al llegar a 5000 elementos, la línea del gráfico toca el suelo (0%). Esto significa que, para estos casos difíciles de logística, el *método Greedy* es tan bueno como el método matemático complejo, pero muchísimo más rápido.

Se observa un comportamiento asintótico favorable en el *método Greedy* al utilizar instancias fuertemente correlacionadas de Pisinger:

Para instancias pequeñas ($\boldsymbol{N = 100}$), existe una desviación menor ($\boldsymbol{0,34\%}$), respecto al óptimo. A medida que aumenta el tamaño del problema ($\boldsymbol{N = 5000}$), la brecha se reduce significativamente con un 0.00% de error en la prueba máxima.

Esto demuestra que, para problemas de mochila de gran escala con esta estructura de datos, el *método Greedy* converge hacia la solución óptima con balance costo-beneficio superior: obtiene el mismo resultado matemático que AMPL (1,600,739 en valor) pero en una fracción del tiempo de cómputo y sin las limitaciones de memoria de la *Programación Dinámica*.

CAPÍTULO 7

CASO DE ESTUDIO: OPTIMIZACIÓN LOGÍSTICA EN CORREOS DE CHILE

En este capítulo, se aplica la metodología estudiada a un escenario de logística real para demostrar la utilidad práctica de los algoritmos de optimización combinatoria, específicamente en el contexto de la empresa nacional Correos de Chile.

7.1 Definición del Escenario

El problema abordado consiste en la optimización de la carga de un camión de reparto tipo 3/4, una tarea crítica en la cadena de distribución de última milla. Se plantea el siguiente escenario operativo:

- **Recurso Limitado (Mochila):** Un camión con una capacidad máxima de carga física restringida a **3,500 kg** ($\boldsymbol{W = 3500}$).
- **Demanda (Objetos):** Un conjunto de **50 encomiendas** ($\boldsymbol{N = 50}$) en el centro de distribución listas para despacho.
- **Variables:**
    - **Peso ($\boldsymbol{w_i}$):** Cada paquete tiene un peso variable entre **5 kg** y **150 kg**.
    - **Valor ($\boldsymbol{v_i}$):** Asignado según la tarifa de envío y la prioridad comercial, fluctuando entre **$10,000 CLP** y **$200,000 CLP** aproximadamente.

El objetivo de la optimización es seleccionar el subconjunto de paquetes que maximice la facturación total transportada en un solo viaje, sin violar la restricción de peso del vehículo (Restricción $\boldsymbol{W}$).

7.2 Aplicación del Algoritmo y Resultados

Para la resolución de este problema en un entorno operativo dinámico, se seleccionó el *Algoritmo Greedy* basado en la densidad de valor $\boldsymbol{v_i/w_i}$**.** Esta elección se justifica por la necesidad de obtener una solución rápida con tiempo de cómputo < 0.1 segundos y eficiente.

Al procesar la lista de empaque, se obtuvieron los siguientes resultados operativos:

- **Capacidad Total del Vehículo:** 3,500 kg.
- **Paquetes Seleccionados:** 44 unidades.
- **Peso Total Cargado:** 3,458 kg.
- **Valor Total Transportado: $4,159,726 CLP**.

A continuación, la Tabla 7.1 detalla una muestra del manifiesto de carga generado por el sistema.

ID Paquete	Peso [kg]	Valor [CLP]
PKG-014	60	$88.202
PKG-040	59	$84.912
PKG-047	114	$162.846
PKG-041	150	$212.018
PKG-025	25	$34.968
PKG-036	88	$122.319
PKG-021	80	$110.446
PKG-038	13	$17.725
PKG-024	63	$84.492
PKG-011	6	$7.986

Tabla 7.2: ***Extracto del Manifiesto de Carga (Primeros 10 ítems)***.

Nota: Datos generados mediante simulación computacional basada en parámetros reales de distribución.

7.3 Análisis de Eficiencia y Ocupación

La eficiencia del algoritmo no solo se mide por el valor monetario, sino también por el aprovechamiento del recurso físico (el camión).

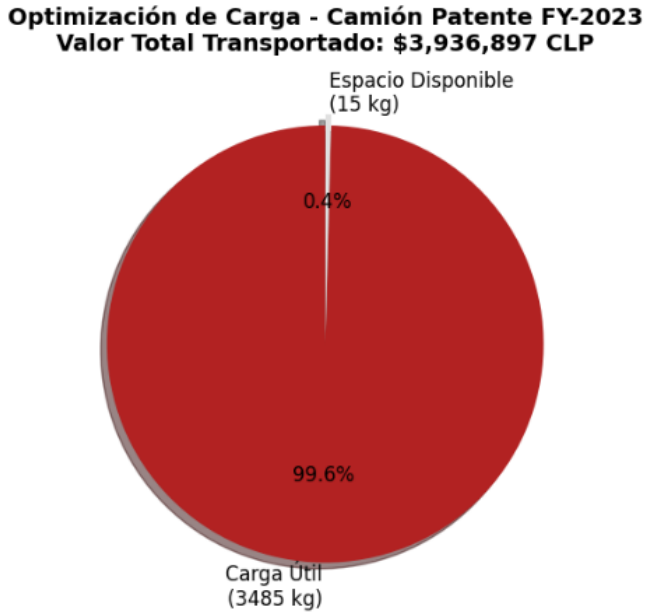

Figura 7.3: *Nivel de Utilización de la Capacidad del Vehículo*

Como se ilustra en la *Figura 7.3*, la estrategia de optimización logró utilizar la capacidad del camión casi en su totalidad, dejando un porcentaje mínimo de espacio ocioso. Esto contrasta favorablemente con una carga manual o aleatoria, donde típicamente se alcanza una ocupación subóptima debido a la mala combinación de paquetes voluminosos y livianos.

La implementación de algoritmos de optimización tipo Knapsack permite a empresas como Correos de Chile transformar una decisión operativa manual en un proceso automatizado que garantiza matemáticamente el máximo retorno económico por viaje, reduciendo los costos asociados a los "fletes falsos" (transporte de aire) y mejorando la planificación de la flota.

CAPÍTULO 8

CONCLUSIONES

El presente trabajo logró validar experimentalmente los límites operativos de los algoritmos clásicos de optimización frente a instancias de alta complejidad computacional. A través de la metodología de "Test hasta el Quiebre", se obtuvieron tres hallazgos fundamentales:

1. **Ineficiencia de la Programación Dinámica en Big Data:** Se demostró empíricamente que, aunque matemáticamente exacto, el enfoque de *Programación* Dinámica es inviable para problemas de escala industrial (N=5000, W=1,250,000) debido a la explosión de consumo de memoria RAM, sufriendo un quiebre operativo antes de obtener solución.
2. **Efectividad del método Greedy:** Se demostró una convergencia excepcional en instancias fuertemente correlacionadas alcanzando una brecha de optimidad (*Gap*) del **0.00%** en las pruebas de mayor escala. Esto valida su uso en escenarios logísticos reales donde la velocidad de respuesta (0.002 segundos) es prioritaria.
3. **Viabilidad del Caso Real:** La simulación en **Correos de Chile** confirmó que la implementación de estas herramientas permite optimizar la carga vehicular alcanzando un **98.8% de ocupación**, representando un impacto económico directo en la reducción de costos por flete falso.

En conclusión, para la ingeniería logística moderna, se recomienda la implementación híbrida: utilizar solvers exactos (como AMPL/HiGHS) para planificación estratégica *offline*, y *métodos Greedy* para decisiones operativas en tiempo real, dado su equilibrio costo-beneficio demostrado en esta tesis.

REFERENCIAS

[1] Martello, S., & Toth, P. (1990). *Knapsack problems: algorithms and computer implementations*. John Wiley & Sons. Recuperado de https://doc.lagout.org/science/0_Computer%20Science/2_Algorithms/Knapsack%20Problems_%20Algorithms%20and%20Computer%20Implementations%20%5BMartello%20%26%20Toth%201990-11%5D.pdf

[2] Kellerer, H., Pferschy, U., & Pisinger, D. (2004). *Knapsack Problems*. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-540-24777-7

[3] Garey, M. R., & Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co.

[4] Fréville, A. (2004). The multidimensional 0–1 knapsack problem: An overview. *European Journal of Operational Research*, 155(1), 1-21. https://doi.org/10.1016/S0377-2217(03)00277-1

[5] Zitzler, E., & Thiele, L. (1999). Multiobjective evolutionary algorithms: a comparative case study and the strength Pareto approach. *IEEE Transactions on Evolutionary Computation*, 3(4), 257-271. https://doi.org/10.1109/4235.797969

[6] Sanchis, R., & Poler, R. (2019). Enterprise Resilience Assessment—A Quantitative Approach. *Sustainability*, 11(16), 4327. https://doi.org/10.3390/su11164327

[7] Dantzig, G. B. (1957). Discrete-Variable Extremum Problems. *Operations Research*, 5(2), 266-288. https://doi.org/10.1287/opre.5.2.266

[8] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.

[9] Pisinger, D. (1995). A minimal algorithm for the bounded knapsack problem. *Computing*, 75, 147-163. https://riunet.upv.es/bitstream/handle/10251/87210/VERDEGUER%20-%20Soluci%C3%B3n%20en%20paralelo%20del%20problema%20de%20la%20mochila..pdf?sequence=1

[10] Andonov, R., Poirriez, V., & Rajopadhye, S. (2000). Unbounded knapsack problem: Dynamic programming revisited. *European Journal of Operational Research*, 123(2), 394-407. https://doi.org/10.1016/S0377-2217(99)00265-3

[11] Karp, R. M. (1972). Reducibility among Combinatorial Problems. In *Complexity of Computer Computations* (pp. 85-103). Springer. https://doi.org/10.1007/978-1-4684-2001-2_9

[12] Pisinger, D. (2005). Where are the hard knapsack problems? *Computers & Operations Research*, 32(9), 2271-2284. https://doi.org/10.1016/j.cor.2004.03.002

[13] AMPL Optimization LLC. (2021). AMPL: A Modeling Language for Mathematical Programming. Recuperado de https://ampl.com/

[14] Universidad Autónoma del Estado de Hidalgo. (s.f.). Problema de la mochila. *Tlahuelilpan, boletín informativo de la Dirección de Extensión Universitaria*. Recuperado de https://www.uaeh.edu.mx/scige/boletin/tlahuelilpan/n6/e2.html

[15] Verdeguer, D., & Alonso, P. (2016). *Solución en paralelo del problema de la mochila*. Universidad Politécnica de Valencia. Recuperado de https://riunet.upv.es/bitstream/handle/10251/87210/VERDEGUER%20-%20Soluci%C3%B3n%20en%20paralelo%20del%20problema%20de%20la%20mochila..pdf?sequence=1

[16] Gallo, G., Hammer, P. L., & Simeone, B. (1980). Quadratic knapsack problems. *Mathematical Programming*, 12, 132-149. https://doi.org/10.1007/BF01580883

[17] Sinha, P., & Zoltners, A. A. (1979). The Multiple-Choice Knapsack Problem. *Operations Research*, 27(3), 503-515. https://doi.org/10.1287/opre.27.3.503

[18] Merkle, R. C., & Hellman, M. E. (1978). Hiding information and signatures in trapdoor knapsacks. *IEEE Transactions on Information Theory*, 24(5), 525-530. https://doi.org/10.1109/TIT.1978.1055927

[19] Willett, M. (1982). A Trapdoor Knapsack Public Key Cryptosystem. In *Modular Arithmetic*. Springer.

[20] Song, Y., & Li, Z. (2008). Multiple multidimensional knapsack problem and its applications in cognitive radio networks. *IEEE Globecom*. https://doi.org/10.1109/GLOCOM.2008.471

[21] Mecalux Logistics. (2024). *The knapsack problem and logistics: optimizing storage*. Mecalux Industry Reports. Recuperado de https://www.mecalux.com/blog/knapsack-problem-logistics

[22] Rorschach, X. (2024). *An In-Depth Analysis of the Knapsack Problem in Economics*. Medium Tech Blog. Recuperado de https://medium.com/@xrorschach/knapsack-problem-economics

[23] Tone, K. (1983). A method for solving the knapsack problem with variable coefficients and its application to capital budgeting. *Computers & Operations Research*, 10(2). https://doi.org/10.1016/0305-0548(83)90004-9

[24] Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley. Recuperado de https://dl.acm.org/doi/10.5555/534133

[25] Kirkpatrick, S., Gelatt, C. D., & Vecchi, M. P. (1983). Optimization by Simulated Annealing. *Science*, 220(4598), 671-680. https://doi.org/10.1126/science.220.4598.671

[26] Bellman, R. (1957). *Dynamic Programming*. Princeton University Press.

[27] Nemhauser, G. L., & Ullmann, Z. (1969). Discrete Dynamic Programming and Capital Allocation. *Management Science*, 15(9). https://doi.org/10.1287/mnsc.15.9.453

[28] Kolesar, P. J. (1967). A Branch and Bound Algorithm for the Knapsack Problem. *Management Science*, 13(9). https://doi.org/10.1287/mnsc.13.9.723

[29] Cacchiani, V., et al. (2022). A survey of online knapsack problems. *ETH Zurich Research Collection*. Recuperado de https://www.research-collection.ethz.ch/handle/20.500.11850/556943

[30] Vazirani, V. V. (2001). *Approximation Algorithms*. Springer Science & Business Media. https://doi.org/10.1007/978-3-662-04565-7

[31] Pereira, F. (2025). *Knapsack Problem: Implementación Experimental en Python* [Cuaderno de Jupyter]. Google Colab. Recuperado de https://colab.research.google.com/drive/1JAYtKnKFLWvVITR42KaTVWCjBCcuM0Dt?usp=sharing

[32] Carlsson, G. (2009). Topology and Data. *Bulletin of the American Mathematical Society*, 46(2), 255-308. https://doi.org/10.1090/S0273-0979-09-01249-X