Problema de Transporte Logístico Discreto

Diseño y Análisis de Algoritmos

Richard Alejandro Matos Arderí
Abel Ponce González
Abraham Romero Imbert

Facultad de Matemática y Computación
Universidad de La Habana

19 de enero de 2026

Resumen

Este informe presenta un estudio completo del problema de *Balanced Multi-Bin Packing with Capacity Constraints*, un problema de optimización combinatoria NP-hard con aplicaciones en logística y distribución de cargas. Se desarrolla la formalización matemática del problema, se demuestra su complejidad computacional mediante reducción desde 3-PARTITION, y se implementan múltiples enfoques algorítmicos incluyendo algoritmos greedy, programación dinámica, branch and bound, y metaheurísticas (Simulated Annealing, Algoritmos Genéticos, Búsqueda Tabú). Se presenta además un análisis experimental comparativo de los algoritmos implementados.

Índice

1. Introducción

1.1. Motivación

El problema de empaquetamiento balanceado en múltiples contenedores surge en numerosas aplicaciones prácticas de logística y distribución. Considérese el escenario de una empresa de transporte que debe distribuir n paquetes en k vehículos, donde cada vehículo tiene una capacidad máxima de peso y se desea equilibrar la carga de trabajo (medida en valor o tiempo de entrega) entre todos los vehículos.

A diferencia del problema clásico de bin packing que busca minimizar el número de contenedores, nuestro problema tiene un número fijo de contenedores y busca:

1. Respetar las restricciones de capacidad de peso
2. Minimizar el desbalance de valores entre contenedores

1.2. Objetivos del Proyecto

Los objetivos principales de este proyecto son:

- Formalizar matemáticamente el problema
- Demostrar su complejidad computacional
- Implementar y analizar múltiples enfoques algorítmicos
- Desarrollar herramientas de visualización y benchmarking
- Crear un dashboard interactivo para experimentación

2. Definición Formal del Problema

2.1. Notación y Definiciones

Definición 2.1 (Ítem)**.** *Un ítem $i \in I$ se caracteriza por un par (w_i, v_i) donde:*

- $w_i \in \mathbb{R}^+$*: peso del ítem*
- $v_i \in \mathbb{R}^+$*: valor del ítem*

Definición 2.2 (Contenedor (Bin))**.** *Un contenedor $j \in \{1, \ldots, k\}$ tiene una capacidad máxima individual $C_j \in \mathbb{R}^+$. Cada contenedor puede tener una capacidad diferente.*

Definición 2.3 (Asignación)**.** *Una asignación es una función $\sigma : I \to \{1, \ldots, k\}$ que mapea cada ítem a un contenedor.*

Definición 2.4 (Asignación Factible)**.** *Una asignación σ es factible si y solo si:*

$$\forall j \in \{1, \ldots, k\} : \sum_{i:\sigma(i)=j} w_i \leq C_j$$

donde C_j es la capacidad específica del contenedor j.

2.2. Formulación del Problema

Entrada:

- Conjunto de ítems $I = \{1, 2, \ldots, n\}$
- Peso $w_i > 0$ y valor $v_i \geq 0$ para cada ítem $i \in I$
- Número de contenedores $k \in \mathbb{Z}^+$
- Capacidad individual $C_j > 0$ para cada contenedor $j \in \{1, \ldots, k\}$

Variable de Decisión:

$$x_{ij} = \begin{cases} 1 & \text{si el ítem } i \text{ es asignado al contenedor } j \\ 0 & \text{en otro caso} \end{cases}$$

Objetivo: Minimizar el desbalance (diferencia entre valor máximo y mínimo):

$$\text{mín} \left(\max_{j \in \{1,\ldots,k\}} V_j - \min_{j \in \{1,\ldots,k\}} V_j \right)$$

donde $V_j = \sum_{i:\sigma(i)=j} v_i$ es el valor total del contenedor j.

2.3. Formulación como Programa Lineal Entero (ILP)

Para modelar la función objetivo $\text{mín}(\text{máx}_j V_j - \text{mín}_j V_j)$, introducimos dos variables auxiliares:

- z^+: cota superior del valor máximo entre contenedores
- z^-: cota inferior del valor mínimo entre contenedores

$$\begin{aligned}
\text{minimizar} \quad & z^+ - z^- && && (1) \\
\text{sujeto a:} \quad & \sum_{j=1}^{k} x_{ij} = 1 && \forall i \in I && (2) \\
& \sum_{i=1}^{n} w_i \cdot x_{ij} \leq C_j && \forall j = 1, \ldots, k && (3) \\
& \sum_{i=1}^{n} v_i \cdot x_{ij} \leq z^+ && \forall j = 1, \ldots, k && (4) \\
& \sum_{i=1}^{n} v_i \cdot x_{ij} \geq z^- && \forall j = 1, \ldots, k && (5) \\
& x_{ij} \in \{0, 1\} && \forall i \in I, j = 1, \ldots, k && (6) \\
& z^+, z^- \geq 0 && && (7)
\end{aligned}$$

Donde:

- (1): Función objetivo que minimiza la diferencia entre cotas

- (2): Cada ítem debe asignarse a exactamente un contenedor
- (3): Restricción de capacidad por peso (cada contenedor j tiene su propia capacidad C_j)
- (4): z^+ es cota superior del valor de cada contenedor, por tanto $z^+ \geq \max_j V_j$
- (5): z^- es cota inferior del valor de cada contenedor, por tanto $z^- \leq \min_j V_j$
- (6): Variables binarias de decisión

Proposición 2.1 (Correctitud de la Formulación)**.** *En el óptimo de la formulación ILP, se cumple* $z^+ = \max_j V_j$ *y* $z^- = \min_j V_j$*.*

Demostración. Sea (x^*, z^{+*}, z^{-*}) una solución óptima y sean $V_j^* = \sum_i v_i x_{ij}^*$ los valores de los contenedores.

Para z^+: Las restricciones (4) implican $z^{+*} \geq V_j^*$ para todo j, es decir, $z^{+*} \geq \max_j V_j^*$. Como minimizamos $z^+ - z^-$, en el óptimo $z^{+*} = \max_j V_j^*$ (de lo contrario podríamos reducir z^+).

Para z^-: Las restricciones (5) implican $z^{-*} \leq V_j^*$ para todo j, es decir, $z^{-*} \leq \min_j V_j^*$. Como minimizamos $z^+ - z^-$, en el óptimo $z^{-*} = \min_j V_j^*$ (de lo contrario podríamos incrementar z^-).

Por tanto, el valor óptimo es $z^{+*} - z^{-*} = \max_j V_j^* - \min_j V_j^*$. □

3. Análisis de Complejidad

3.1. Clases de Complejidad y el Problema de Decisión

Antes de analizar la complejidad de nuestro problema, es fundamental distinguir entre problemas de optimización y problemas de decisión.

Definición 3.1 (Problema de Optimización vs. Decisión)**.**

- ***Problema de Optimización (BALANCED-BIN-PACKING-OPT):***

  Dados n ítems con pesos y valores, k bins con capacidades $C_1, \ldots, C_k$, encontrar una asignación factible que minimice la diferencia máxima de valores entre bins.

- ***Problema de Decisión (BALANCED-BIN-PACKING-DEC):***

  Dados n ítems con pesos y valores, k bins con capacidades $C_1, \ldots, C_k$, y un umbral B, ¿existe una asignación factible tal que la diferencia máxima de valores entre bins sea $\leq B$?

Proposición 3.1. *Si el problema de decisión BALANCED-BIN-PACKING-DEC está en NP, entonces el problema de optimización BALANCED-BIN-PACKING-OPT está en NPO (problemas de optimización NP).*

3.2. NP-Completitud del Problema de Decisión

Teorema 3.2 (NP-Completitud de BALANCED-BIN-PACKING-DEC)**.** *El problema de decisión BALANCED-BIN-PACKING-DEC es NP-completo.*

Demostración. Demostraremos que BALANCED-BIN-PACKING-DEC $\in$ NP-completo mediante dos pasos:

Paso 1: BALANCED-BIN-PACKING-DEC $\in$ NP

Un certificado para una instancia con respuesta 'sí' es una asignación $\sigma : I \to \{1, \ldots, k\}$. La verificación requiere:

1. Verificar que cada ítem está asignado: $O(n)$
2. Calcular peso total de cada bin: $O(n)$
3. Verificar restricciones de capacidad: $O(k)$
4. Calcular valor total de cada bin: $O(n)$
5. Verificar que $\max_j V_j - \min_j V_j \leq B$: $O(k)$

Total: $O(n + k)$, por lo tanto el certificado es verificable en tiempo polinomial. □

Paso 2: NP-Hardness mediante reducción desde 3-PARTITION

□

3.3. Cadena de Reducciones: Demostración de NP-Completitud Fuerte

Para comprender mejor la dureza del problema, presentamos la cadena de reducciones que establece tanto la NP-completitud de 3-PARTITION como la de nuestro problema:

Cadena de Reducciones Polinomiales:

$$\text{3-SAT} \leq_p \text{3DM} \leq_p \text{4-PARTITION} \leq_p \text{3-PARTITION} \leq_p \text{BALANCED-BIN-PACKING}$$

A continuación presentamos cada problema de la cadena y las ideas principales de las reducciones.

3.3.1. Problema 3-SAT (Punto de Partida)

El problema 3-SAT es uno de los problemas fundamentales en teoría de complejidad, siendo el primer problema demostrado NP-completo mediante el teorema de Cook-Levin (1971).

Definición 3.2 (3-SAT)**.** ***Entrada:*** *Una fórmula booleana ϕ en forma normal conjuntiva (CNF) donde cada cláusula contiene exactamente 3 literales. Es decir:*

$$\phi = (l_{1,1} \vee l_{1,2} \vee l_{1,3}) \wedge (l_{2,1} \vee l_{2,2} \vee l_{2,3}) \wedge \cdots \wedge (l_{m,1} \vee l_{m,2} \vee l_{m,3})$$

donde cada $l_{i,j}$ es una variable x_k o su negación $\neg x_k$.

Pregunta: *¿Existe una asignación de valores de verdad a las variables $x_1, \ldots, x_n$ que satisfaga ϕ (haga ϕ verdadera)?*

Ejemplo: La fórmula $\phi = (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_3)$ es satisfacible con la asignación $x_1 = \text{True}, x_2 = \text{True}, x_3 = \text{True}$.

Teorema 3.3 (Cook-Levin, 1971)**.** *3-SAT es NP-completo.*

3.3.2. Problema 3-Dimensional Matching (3DM)

El problema 3DM generaliza el concepto de emparejamiento de grafos bipartitos a hipergrafos tripartitos.

Definición 3.3 (3-Dimensional Matching (3DM)). ***Entrada:*** *Tres conjuntos disjuntos X, Y, Z, cada uno de cardinalidad q, y un conjunto $T \subseteq X \times Y \times Z$ de tripletas.*

Pregunta: *¿Existe un subconjunto $M \subseteq T$ de cardinalidad q tal que cada elemento de $X \cup Y \cup Z$ aparece exactamente una vez en M? (Es decir, M es un emparejamiento perfecto 3-dimensional).*

Ejemplo: Sean $X = \{x_1, x_2\}$, $Y = \{y_1, y_2\}$, $Z = \{z_1, z_2\}$ y:

$$T = \{(x_1, y_1, z_1), (x_2, y_2, z_2)\}$$

Entonces $M = T$ es un emparejamiento perfecto: cada elemento de X, Y y Z aparece exactamente una vez. Si tuviéramos $T' = \{(x_1, y_1, z_1), (x_1, y_2, z_2)\}$, no existiría emparejamiento perfecto porque x_2 no aparece en ninguna tripleta.

Teorema 3.4 (Karp, 1972 [4]). *3DM es NP-completo. Además, 3DM es fuertemente NP-completo (permanece NP-completo incluso cuando $|T|$ está acotado polinomialmente).*

3.3.3. Reducción de 3-SAT a 3DM (Idea Intuitiva)

La reducción de 3-SAT a 3DM utiliza una construcción ingeniosa con "gadgets" que codifican las variables y cláusulas de la fórmula booleana.

Idea general de la construcción [4, 2]:

1. **Gadget de variable (rueda):** Para cada variable x_i que aparece k_i veces en la fórmula, construimos una "rueda" de $2k_i$ tripletas solapadas. La rueda tiene exactamente dos formas de ser cubierta completamente:
   - Seleccionar todas las tripletas de índice par (corresponde a $x_i = \text{True}$)
   - Seleccionar todas las tripletas de índice impar (corresponde a $x_i = \text{False}$)
2. **Gadget de cláusula (rosa):** Para cada cláusula $C_j = (l_1 \vee l_2 \vee l_3)$, construimos una "rosa" de 3 tripletas solapadas, una por cada literal. La rosa puede cubrirse si y solo si al menos uno de sus vértices queda libre por el gadget de variable correspondiente.
3. **Gadget de recolección de basura:** Como es posible que más de un literal de una cláusula sea verdadero (dejando vértices extra sin cubrir), añadimos tripletas adicionales que "recogen" estos vértices sobrantes.

Intuición: La estructura de la rueda fuerza una elección binaria (True/False) para cada variable. La rosa de cada cláusula solo puede completarse si al menos un literal es verdadero (dejando un vértice disponible en el gadget de variable). Así, existe un emparejamiento 3D perfecto si y solo si existe una asignación satisfactoria.

3.3.4. Problema 3-PARTITION

Definición 3.4 (3-PARTITION). ***Entrada:*** *Conjunto $A = \{a_1, a_2, \ldots, a_{3m}\}$ de $3m$ enteros positivos y un entero B tal que:*

- $\sum_{i=1}^{3m} a_i = mB$
- $\frac{B}{4} < a_i < \frac{B}{2}$ *para todo i*

Pregunta: *¿Se puede particionar A en m conjuntos disjuntos $A_1, \ldots, A_m$ tal que cada A_i contiene exactamente 3 elementos y $\sum_{a \in A_i} a = B$?*

3.3.5. Reducción de 3DM a 4-PARTITION (Idea Intuitiva)

La reducción de 3DM a 4-PARTITION codifica las tripletas del emparejamiento como números, utilizando una representación posicional que fuerza la estructura deseada [2].

Idea general de la construcción:

Dada una instancia de 3DM con conjuntos $X = \{x_1, \ldots, x_q\}$, $Y = \{y_1, \ldots, y_q\}$, $Z = \{z_1, \ldots, z_q\}$ y tripletas $T = \{t_1, \ldots, t_m\}$:

1. **Codificación numérica:** Elegimos una base $r = 32q$ suficientemente grande. Para cada tripleta $t = (x_i, y_j, z_k) \in T$, creamos un número:

   $$n_t = 10r^4 - kr^3 - jr^2 - ir$$

   Los coeficientes negativos codifican los índices de los elementos en la tripleta.

2. **Elementos de complemento:** Para cada elemento $x_i \in X$ (y análogamente para Y y Z), creamos números que "compensan" su aparición:

   - Un elemento "real" $x_i^{[1]}$ que se usará en la solución
   - Elementos "dummy" $x_i^{[2]}, x_i^{[3]}, \ldots$ para las apariciones extra

   Los tamaños se diseñan para que la suma de una tripleta más sus tres elementos reales sea exactamente el objetivo T.

3. **Objetivo:** $T = 40r^4$ (cada cuaterna debe sumar este valor).

Por qué funciona:

- La base grande r hace que los "dígitos" (coeficientes de r^4, r^3, r^2, r) no se mezclen al sumar.
- Para obtener exactamente $40r^4$, las contribuciones de cada posición deben "cancelarse" correctamente.
- Esto solo ocurre cuando seleccionamos tripletas que cubren cada elemento exactamente una vez.

3.3.6. Cadena de Reducciones para 3-PARTITION (Garey & Johnson)

La demostración original de la NP-completitud fuerte de 3-PARTITION fue establecida por Garey y Johnson [3, 2] mediante la siguiente cadena de reducciones:

3-SAT $\leq_p$ 3-Dimensional Matching (3DM) $\leq_p$ 4-PARTITION $\leq_p$ 3-PARTITION

Presentamos aquí la reducción clave de 4-PARTITION a 3-PARTITION, que es la más relevante para nuestro problema.

Definición 3.5 (4-PARTITION)**.** ***Entrada:*** *Conjunto $A = \{a_1, \ldots, a_{4m}\}$ de $4m$ enteros positivos con $\frac{T}{5} < a_i < \frac{T}{3}$ y $\sum_{i=1}^{4m} a_i = mT$.*

Pregunta: *¿Se puede particionar A en m conjuntos de exactamente 4 elementos cada uno, donde cada conjunto suma T?*

Lema 3.5 (Reducción de 4-PARTITION a 3-PARTITION [2])**.** *4-PARTITION $\leq_p$ 3-PARTITION.*

Demostración. Dada una instancia de 4-PARTITION con $4m$ enteros $a_1, \ldots, a_{4m}$ (cada uno en el rango $(\frac{T}{5}, \frac{T}{3})$) con suma total mT, construimos una instancia de 3-PARTITION:

Construcción:

1. **Elementos regulares:** Para cada a_i en la instancia de 4-PARTITION, creamos un elemento regular:
$$w_i = 4(5T + a_i) + 1$$
Hay $4m$ elementos regulares con suma total $81mT + 4m$.

2. **Elementos de emparejamiento:** Para cada par ordenado (i, j) con $i \neq j$, creamos dos elementos de emparejamiento:
$$u_{ij} = 4(6T - a_i - a_j) + 2$$
$$u'_{ij} = 4(5T + a_i + a_j) + 2$$
Hay $4m(4m-1)$ elementos de emparejamiento con suma total $(88mT+16m)(4m-1)$.

3. **Elementos de relleno:** Añadimos $8m^2 - 3m$ elementos de relleno, cada uno de tamaño $20T$.

4. **Total:** $24m^2 - 3m = 3(8m^2 - m)$ elementos con suma $(64T + 4)(8m^2 - m)$.

5. **Objetivo:** $B = 64T + 4$ (cada tripleta debe sumar este valor).

Verificación de restricciones: Todos los elementos están en el rango $(16T+1, 32T+2) \subset (\frac{B}{4}, \frac{B}{2})$.

Correctitud ($\Rightarrow$):

Si existe una 4-partición válida $\{S_1, \ldots, S_m\}$ donde cada $S_k = \{a_{k_1}, a_{k_2}, a_{k_3}, a_{k_4}\}$ suma T:

Para cada 4-conjunto con suma T, construimos dos tripletas en la 3-partición:

- $\{w_{k_1}, w_{k_2}, u_{k_1 k_2}\}$ con suma $4(5T + a_{k_1} + 5T + a_{k_2} + 6T - a_{k_1} - a_{k_2}) + 4 = 64T + 4 = B$

- $\{w_{k_3}, w_{k_4}, u'_{k_1 k_2}\}$ con suma $4(5T + a_{k_3} + 5T + a_{k_4} + 5T + a_{k_1} + a_{k_2}) + 4$
  Como $a_{k_1} + a_{k_2} + a_{k_3} + a_{k_4} = T$, esta suma es $64T + 4 = B$.

Los elementos de emparejamiento restantes se agrupan en tripletas de la forma $\{u_{ij}, u'_{ij}, \text{relleno}\}$ con suma $4(6T - a_i - a_j + 5T + a_i + a_j + 5T) + 4 = 64T + 4 = B$.

Correctitud ($\Leftarrow$):
Si existe una 3-partición válida:

1. Cada tripleta suma $B = 64T + 4$, que es divisible por 4 con residuo 0.
2. Por los residuos módulo 4:
   - Elementos regulares: residuo 1
   - Elementos de emparejamiento: residuo 2
   - Elementos de relleno: residuo 0
3. Para obtener residuo 0 (mod 4), cada tripleta debe contener:
   - Dos elementos regulares y un elemento de emparejamiento (residuo $1+1+2 = 4 \equiv 0$), o
   - Dos elementos de emparejamiento y un elemento de relleno (residuo $2+2+0 = 4 \equiv 0$)
4. Si una tripleta contiene dos elementos de emparejamiento u_{ij} y u_{kl} con un relleno, la suma de los dos elementos de emparejamiento debe ser $44T + 4$. Por análisis algebraico, esto fuerza que sean un par "emparejado" de la forma u_{ij} y u'_{ij}.
5. Los elementos regulares restantes forman tripletas con elementos de emparejamiento. Por las restricciones de suma, cuatro elementos regulares $w_{k_1}, w_{k_2}, w_{k_3}, w_{k_4}$ que aparecen en dos tripletas con un par emparejado u_{ij}, u'_{ij} deben satisfacer $a_{k_1} + a_{k_2} + a_{k_3} + a_{k_4} = T$.

Por lo tanto, de una 3-partición válida podemos reconstruir una 4-partición válida. □ □

Teorema 3.6 (NP-Completitud Fuerte de 3-PARTITION)**.** *3-PARTITION es fuertemente NP-completo.*

Demostración. La prueba completa sigue la cadena de reducciones de Garey y Johnson [2]:

1. 3-SAT es NP-completo (Cook-Levin, 1971).
2. 3-Dimensional Matching (3DM) es fuertemente NP-completo por reducción desde 3-SAT usando gadgets de variables y cláusulas (Karp, 1972) [4].
3. 4-PARTITION es fuertemente NP-completo por reducción desde 3DM usando una codificación numérica de las tripletas.
4. 3-PARTITION es fuertemente NP-completo por la reducción presentada arriba.

En cada reducción, el tamaño de los números en la instancia construida está acotado polinomialmente por el tamaño de la entrada, preservando la NP-completitud fuerte. □ □

3.3.7. Reducción de 3-PARTITION a BALANCED-BIN-PACKING

Con la NP-completitud fuerte de 3-PARTITION establecida por Garey y Johnson, ahora podemos demostrar que nuestro problema también es NP-completo.

Lema 3.7 (Reducción desde 3-PARTITION)**.** *3-PARTITION se reduce polinomialmente a BALANCED-BIN-PACKING-DEC.*

Demostración. Dada una instancia de 3-PARTITION con elementos $\{a_1, \ldots, a_{3m}\}$ y objetivo B, construimos una instancia de BALANCED-BIN-PACKING-DEC:

Construcción:

1. Para cada elemento a_i, creamos un ítem con:
   - Peso: $w_i = a_i$
   - Valor: $v_i = a_i$ (peso y valor coinciden)
2. Número de bins: $k = m$
3. Capacidad de cada bin: $C_j = B$ para $j = 1, \ldots, m$ (capacidades uniformes)
4. Umbral de balance: $\beta = 0$ (buscamos balance perfecto)

Correctitud ($\Rightarrow$):
Supongamos que existe una 3-partición válida $A_1, \ldots, A_m$ de los elementos originales. Construimos una asignación σ para BALANCED-BIN-PACKING:

- Para cada conjunto A_j en la 3-partición, asignamos los ítems correspondientes al bin j
- Cada bin j contiene exactamente 3 ítems con peso total B
- Por construcción ($v_i = w_i$), el valor total de cada bin es también B
- La diferencia máxima de valores es: $\max_j V_j - \min_j V_j = B - B = 0 \leq \beta$

Por lo tanto, la asignación es factible y satisface el umbral de balance.

Correctitud ($\Leftarrow$):
Supongamos que existe una asignación factible σ para BALANCED-BIN-PACKING con diferencia ≤ 0.

Esto implica que todos los bins tienen el mismo valor total. Como:

- Suma total de valores: $\sum_{i=1}^{3m} v_i = \sum_{i=1}^{3m} a_i = mB$
- Número de bins: $k = m$
- Todos los bins tienen igual valor

Cada bin debe tener valor exactamente $\frac{mB}{m} = B$.

Dado que $v_i = w_i$ y el bin tiene capacidad B, cada bin también tiene peso total B (está completamente lleno).

Las restricciones $\frac{B}{4} < a_i < \frac{B}{2}$ garantizan que:

- Ningún bin puede tener menos de 3 elementos (pues $a_i < \frac{B}{2}$, así que la suma de dos elementos es $< B$ y se requieren al menos 3 para alcanzar B)

- Ningún bin puede tener más de 3 elementos (ya que $4 \times \frac{B}{4} = B$ pero cada elemento es $> \frac{B}{4}$)

Por lo tanto, cada bin contiene exactamente 3 elementos que suman B, constituyendo una 3-partición válida. □ □

3.4. Implicaciones de la NP-Completitud

Corolario 3.8. *El problema de optimización BALANCED-BIN-PACKING-OPT es NP-hard.*

Demostración. Si existiera un algoritmo polinomial para BALANCED-BIN-PACKING-OPT, podríamos resolver BALANCED-BIN-PACKING-DEC en tiempo polinomial:

1. Ejecutar el algoritmo de optimización
2. Comparar el resultado con B
3. Responder 'sí' si resultado $\leq B$, 'no' en caso contrario

Como BALANCED-BIN-PACKING-DEC es NP-completo, esto implicaría P = NP. □ □

3.5. Problema con Capacidades Heterogéneas

Proposición 3.9. *El problema BALANCED-BIN-PACKING con capacidades heterogéneas (diferentes C_j por bin) es al menos tan difícil como el caso con capacidades uniformes.*

Demostración. El caso uniforme es una instancia particular del caso heterogéneo (cuando $C_1 = C_2 = \cdots = C_k$).

Si existiera un algoritmo polinomial para el caso heterogéneo, también resolvería el caso uniforme en tiempo polinomial, lo cual contradice la NP-hardness del caso uniforme (asumiendo P $\neq$ NP). □ □

3.6. Complejidad de los Algoritmos Implementados

Cuadro 1: Complejidad temporal y espacial de los algoritmos implementados

Algoritmo	**Tiempo**	**Espacio**
Búsqueda Exhaustiva (Fuerza Bruta)	$O(k^n \cdot n)$	$O(n+k)$
First Fit Decreasing (FFD)	$O(n \log n + n \cdot k)$	$O(n+k)$
Best Fit Decreasing (BFD)	$O(n \log n + n \cdot k)$	$O(n+k)$
Worst Fit Decreasing (WFD)	$O(n \log n + n \log k)$	$O(n+k)$
Round Robin Greedy	$O(n \log n + n \log k)$	$O(n+k)$
Largest Difference First (LDF)	$O(n^2 \cdot k)$	$O(n+k)$
LPT Balanced	$O(n \log n + n \log k)$	$O(n+k)$
Karmarkar-Karp (KK)	$O(n \log n)$	$O(n)$
Programación Dinámica	$O(k^2 \cdot 3^n)$	$O(k \cdot 2^n)$
Branch and Bound	$O(k^n)$ peor caso	$O(n \cdot k)$
Simulated Annealing	$O(I \cdot n)$	$O(n)$
Genetic Algorithm	$O(G \cdot P \cdot n)$	$O(P \cdot n)$
Tabu Search	$O(I \cdot N)$	$O(n+T)$

Donde:

- n: número de ítems
- k: número de contenedores (bins)
- I: número de iteraciones
- G: número de generaciones
- P: tamaño de población
- N: tamaño del vecindario
- T: tamaño de la lista tabú

4. Algoritmos Implementados

4.1. Algoritmo de Fuerza Bruta (Búsqueda Exhaustiva)

4.1.1. Intuición del Algoritmo

El algoritmo de fuerza bruta responde a la pregunta más fundamental: *"¿Cuál es la mejor solución posible?"* Lo hace de la manera más directa imaginable: **probando todas las opciones**.

> *Para cada ítem, tenemos k opciones (asignarlo al contenedor 1, 2, ..., o k). Con n ítems, esto genera k^n combinaciones posibles. Probamos cada una, verificamos si es válida (respeta las capacidades), y nos quedamos con la mejor.*

Analogía: Imagina que tienes 5 cajas de diferente peso y 3 camiones con límites de carga distintos. La fuerza bruta es como escribir todas las formas posibles de asignar las cajas: "caja 1 al camión 1, caja 2 al camión 1, ..." hasta "caja 1 al camión 3, caja 2 al camión 3, ..." (en total $3^5 = 243$ combinaciones). Luego, para cada combinación:

1. ¿Algún camión excede su límite de carga? Si sí, descarta esta opción.
2. Si no, calcula qué tan "balanceada" quedó la carga.
3. Guarda la mejor solución encontrada hasta ahora.

¿Por qué es útil a pesar de ser lento? La fuerza bruta **garantiza** encontrar el óptimo. Esto es invaluable para:

- Validar que otros algoritmos funcionan correctamente
- Medir qué tan lejos están las heurísticas del óptimo
- Resolver instancias pequeñas donde la optimalidad es crítica

4.1.2. Descripción Formal

Para un problema con n ítems y k contenedores, el algoritmo explora las k^n posibles asignaciones completas. Cada ítem i puede asignarse a cualquiera de los k contenedores, por lo que el espacio de búsqueda crece exponencialmente.

Algorithm 1 Búsqueda Exhaustiva (Fuerza Bruta)

```
1: procedure BruteForce(items, k, C_1, ..., C_k)
2:     best_diff ← ∞
3:     best_assignment ← null
4:     for assignment ∈ {0, 1, ..., k − 1}^n do      ▷ Enumerar todas las k^n asignaciones
5:         bins ← Crear k contenedores vacíos
6:         valid ← True
7:         for i ← 0 to n − 1 do
8:             j ← assignment[i]                      ▷ Contenedor asignado al ítem i
9:             if weight(bins[j]) + items[i].weight > C_j then
10:                valid ← False
11:                break
12:            end if
13:            bins[j].add(items[i])
14:        end for
15:        if valid then
16:            values ← [value(bin) for bin ∈ bins]
17:            diff ← máx(values) − mín(values)
18:            if diff < best_diff then
19:                best_diff ← diff
20:                best_assignment ← assignment
21:            end if
22:        end if
23:    end for
24:    return best_assignment, best_diff
25: end procedure
```

4.1.3. Análisis de Complejidad

Teorema 4.1 (Complejidad del Algoritmo de Fuerza Bruta)**.** *El algoritmo de búsqueda exhaustiva tiene:*

- ***Complejidad temporal:*** $O(k^n \cdot n)$
- ***Complejidad espacial:*** $O(n + k)$

Demostración. **Tiempo:** Existen k^n asignaciones posibles. Para cada asignación, se verifica la factibilidad y se calcula el objetivo, ambas operaciones en $O(n)$. Total: $O(k^n \cdot n)$.

Espacio: Se almacena la asignación actual ($O(n)$), la mejor asignación ($O(n)$), y los valores acumulados por contenedor ($O(k)$). Total: $O(n + k)$. □

4.1.4. Demostración de Correctitud

Teorema 4.2 (Correctitud del Algoritmo de Fuerza Bruta)**.** *El algoritmo de búsqueda exhaustiva encuentra una solución óptima factible si existe, o reporta infactibilidad en caso contrario.*

Demostración. Sea $\mathcal{A}$ el conjunto de todas las asignaciones posibles, donde una asignación es una función $\sigma : \{1, \ldots, n\} \to \{1, \ldots, k\}$ que mapea cada ítem a un contenedor. Claramente $|\mathcal{A}| = k^n$.

Sea $\mathcal{F} \subseteq \mathcal{A}$ el conjunto de asignaciones factibles (aquellas que respetan todas las restricciones de capacidad).

El algoritmo:

1. **Enumera** todos los elementos de $\mathcal{A}$ (completitud)
2. Para cada $\sigma \in \mathcal{A}$, **verifica** si $\sigma \in \mathcal{F}$ (factibilidad)
3. Para cada $\sigma \in \mathcal{F}$, **calcula** $f(\sigma) = \max_j V_j(\sigma) - \min_j V_j(\sigma)$ (evaluación)
4. **Retorna** $\arg\min_{\sigma \in \mathcal{F}} f(\sigma)$ (selección del óptimo)

Como el algoritmo evalúa *todas* las asignaciones factibles y selecciona la de mínimo valor objetivo, necesariamente encuentra el óptimo global si $\mathcal{F} \neq \emptyset$. □ □

Corolario 4.3. *El algoritmo de fuerza bruta es* ***exacto****: para cualquier instancia con solución factible, el valor retornado es igual al óptimo global z^*.*

4.1.5. Límites Prácticos

Mediante análisis empírico se determinaron los tamaños máximos de instancia resolubles en tiempos razonables:

Cuadro 2: Tamaño máximo de instancia resoluble por fuerza bruta

Contenedores (k)	**Máx n (1s)**	**Máx n (10s)**	**Máx n (60s)**
$k = 2$	14	14	14
$k = 3$	11	13	14
$k = 4$	8	10	11
$k = 5$	7	9	10

La Tabla 2 confirma la complejidad $O(k^n)$: al aumentar k, el tamaño máximo resoluble disminuye proporcionalmente. Por ejemplo, con $k = 2$ y $n = 14$, el espacio de búsqueda es $2^{14} = 16{,}384$ asignaciones, mientras que con $k = 4$ y $n = 10$ es $4^{10} \approx 1{,}048{,}576$ asignaciones.

4.1.6. Verificación de Complejidad

Se observó que para $k = 2$, el tiempo aproximadamente se duplica por cada incremento en n, confirmando el comportamiento $O(2^n \cdot n) \approx O(2^n)$ para k fijo:

n	Tiempo (segundos)
10	0.0051
11	0.0101 (×2,0)
12	0.0204 (×2,0)
13	0.0401 (×2,0)
14	0.0858 (×2,1)

4.1.7. Rol del Algoritmo de Fuerza Bruta

El algoritmo de fuerza bruta sirve tres propósitos fundamentales:

1. **Verificación de correctitud:** Permite validar que las soluciones heurísticas sean factibles comparándolas con el óptimo garantizado.

2. **Medición de calidad:** Establece la línea base para calcular el *gap* de optimalidad:

$$\text{Gap}(\%) = \frac{\text{Heurístico} - \text{Óptimo}}{\text{Óptimo}} \times 100$$

3. **Generación de instancias de prueba:** Para instancias pequeñas, permite conocer el óptimo verdadero y así diseñar tests unitarios.

4.2. Algoritmos Greedy

4.2.1. First Fit Decreasing (FFD)

El algoritmo FFD ordena los ítems por peso decreciente y asigna cada ítem al primer contenedor que tiene capacidad suficiente.

Algorithm 2 First Fit Decreasing

```
1: procedure FFD(items, k, C_1, ..., C_k)
2:     sorted_items ← sort(items, key = weight, desc = True)
3:     bins ← [ ] × k                          ▷ Crear k bins con capacidades C_j
4:     for item ∈ sorted_items do
5:         for j ← 1 to k do
6:             if weight(bins[j]) + item.weight ≤ C_j then
7:                 bins[j].add(item)
8:                 break
9:             end if
10:        end for
11:    end for
12:    return bins
13: end procedure
```

4.2.2. Best Fit Decreasing (BFD)

El algoritmo BFD ordena los ítems por valor decreciente y asigna cada ítem al contenedor que minimiza la diferencia de valores resultante después de la asignación.

Algorithm 3 Best Fit Decreasing

```
1: procedure BFD(items, k, C_1, ..., C_k)
2:     sorted_items ← sort(items, key = value, desc = True)
3:     bins ← [Bin(j, C_j) for j ∈ 1..k]
4:     for item ∈ sorted_items do
5:         best_bin ← null
6:         best_diff ← ∞
                                   ▷ Encontrar bin que minimiza diferencia resultante
7:         for j ∈ 1..k do
8:             if weight(bins[j]) + item.weight ≤ C_j then
9:                 temp_value ← bins[j].value + item.value
10:                all_values ← [bins[i].value for i ∈ 1..k]
11:                all_values[j] ← temp_value
12:                diff ← máx(all_values) − mín(all_values)
13:                if diff < best_diff then
14:                    best_diff ← diff
15:                    best_bin ← j
16:                end if
17:            end if
18:        end for
19:        if best_bin ≠ null then
20:            bins[best_bin].add(item)
21:        end if
22:    end for
23:    return bins
24: end procedure
```

4.2.3. Worst Fit Decreasing (WFD)

El algoritmo WFD asigna cada ítem (ordenado por valor decreciente) al contenedor con mayor capacidad disponible. Esta estrategia tiende a distribuir los ítems de manera más uniforme por peso.

Algorithm 4 Worst Fit Decreasing

```
1: procedure WFD(items, k, C_1, ..., C_k)
2:     sorted_items ← sort(items, key = value, desc = True)
3:     bins ← [Bin(j, C_j) for j ∈ 1..k]
4:     for item ∈ sorted_items do
5:         best_bin ← null
6:         max_remaining ← −1
                                   ▷ Encontrar bin con mayor capacidad disponible
7:         for j ∈ 1..k do
8:             if weight(bins[j]) + item.weight ≤ C_j then
9:                 remaining ← C_j − weight(bins[j])
10:                if remaining > max_remaining then
11:                    max_remaining ← remaining
12:                    best_bin ← j
13:                end if
14:            end if
15:        end for
16:        if best_bin ≠ null then
17:            bins[best_bin].add(item)
18:        end if
19:    end for
20:    return bins
21: end procedure
```

4.2.4. Round Robin Greedy

El algoritmo Round Robin distribuye los ítems (ordenados por valor decreciente, estilo LPT) asignando cada uno al contenedor con menor valor actual que pueda alojarlo.

Algorithm 5 Round Robin Greedy

```
1: procedure RoundRobinGreedy(items, k, C_1, ..., C_k)
2:     sorted_items ← sort(items, key = value, desc = True)
3:     bins ← [Bin(j, C_j) for j ∈ 1..k]
4:     for item ∈ sorted_items do
5:         feasible_bins ← [j : weight(bins[j]) + item.weight ≤ C_j]
6:         if feasible_bins ≠ ∅ then          ▷ Seleccionar bin con mínimo valor actual
7:             min_bin ← arg mín_{j∈feasible_bins} bins[j].value
8:             bins[min_bin].add(item)
9:         end if
10:    end for
11:    return bins
12: end procedure
```

4.2.5. LPT Balanced

El algoritmo Longest Processing Time (LPT) adaptado para balanceo asigna cada ítem al contenedor con menor carga actual, respetando las capacidades individuales.

Algorithm 6 LPT Balanced

procedure LPT($items, k, C_1, \ldots, C_k$)
    $sorted\_items \leftarrow \text{sort}(items, \text{key} = value, \text{desc} = True)$
    $bins \leftarrow [\text{Bin}(j, C_j) \text{ for } j \in 1..k]$
    for $item \in sorted\_items$ **do**
        $j^* \leftarrow \arg\min_{j:\text{weight}(bins[j])+item.weight \leq C_j} \text{value}(bins[j])$
        $bins[j^*].\text{add}(item)$
    end for
    return $bins$
end procedure

4.2.6. Largest Difference First (LDF)

El algoritmo LDF es un enfoque greedy que en cada paso selecciona la asignación (ítem, contenedor) que minimiza la diferencia máxima resultante entre contenedores.

Intuición: A diferencia de los algoritmos que procesan ítems en orden fijo, LDF evalúa todas las combinaciones posibles en cada paso y elige la asignación que mejor balancea la carga actual.

Características:

- **Complejidad temporal:** $O(n^2 \cdot k)$ - cuadrática en el número de ítems
- **Calidad:** Sin garantía teórica de aproximación, pero buenos resultados empíricos
- **Aplicabilidad:** Mejor para instancias pequeñas a medianas donde el balance es crítico

Algorithm 7 Largest Difference First

```
1: procedure LDF(items, k, C_1, ..., C_k)
2:     bins ← [Bin(j, C_j) for j ∈ 1..k]
3:     unassigned ← items                          ▷ Conjunto de ítems sin asignar
4:     while unassigned ≠ ∅ do
5:         best_item ← null
6:         best_bin ← null
7:         best_diff ← ∞
                                                   ▷ Evaluar todas las combinaciones (ítem, bin)
8:         for item ∈ unassigned do
9:             for j ∈ 1..k do
10:                if weight(bins[j]) + item.weight ≤ C_j then      ▷ Simular asignación
11:                    current_values ← [value(bins[i]) for i ∈ 1..k]
12:                    current_values[j] ← current_values[j] + item.value
13:                    diff ← máx(current_values) − mín(current_values)
14:                    if diff < best_diff then
15:                        best_diff ← diff
16:                        best_item ← item
17:                        best_bin ← j
18:                    end if
19:                end if
20:            end for
21:        end for
22:        if best_item ≠ null then
23:            bins[best_bin].add(best_item)
24:            unassigned.remove(best_item)
25:        else
26:            break                               ▷ No hay asignación factible
27:        end if
28:    end while
29:    return bins
30: end procedure
```

Diferencia con otros algoritmos greedy:

- **FFD/BFD/WFD:** Procesan ítems en orden predeterminado (por peso o valor)
- **LDF:** Selecciona dinámicamente el siguiente ítem basándose en el estado actual
- **Trade-off:** Mayor tiempo de cómputo ($O(n^2k)$ vs $O(n \log n)$) pero potencialmente mejor balance

4.3. Algoritmos de Aproximación

4.3.1. Karmarkar-Karp (KK) para Particionamiento Multi-Vía

El algoritmo Karmarkar-Karp es un método de diferenciación clásico para el problema de particionamiento de números. Para el caso de 2 contenedores, tiene garantías teóricas de aproximación $O(1/n^{\Theta(\log n)})$.

Idea fundamental: En lugar de asignar ítems directamente a contenedores, KK reemplaza iterativamente los dos valores más grandes con su diferencia, simulando la colocación de estos elementos en contenedores diferentes.

Complejidad:

- **Temporal:** $O(n \log n)$ con heap
- **Espacial:** $O(n)$
- **Ratio de aproximación:** $O(1/n^{\Theta(\log n)})$ para $k = 2$

Algorithm 8 Karmarkar-Karp (KK) - Caso 2 contenedores

```
1: procedure KK-TwoWay(items, C_1, C_2)
2:     heap ← MaxHeap()
                          ▷ Inicializar heap con pares (valor, [ids de ítems])
3:     for item ∈ items do
4:         heap.push(−item.value, [item.id])          ▷ Negativo para max-heap
5:     end for
                                                    ▷ Proceso de diferenciación
6:     while |heap| > 1 do
7:         (neg_val_1, items_1) ← heap.pop()
8:         (neg_val_2, items_2) ← heap.pop()
9:         val_1 ← −neg_val_1, val_2 ← −neg_val_2
                                  ▷ Combinar grupos y calcular diferencia
10:        diff ← |val_1 − val_2|
11:        combined ← items_1 ∪ items_2
12:        if diff > 0 then
13:            heap.push(−diff, combined)
14:        end if
15:    end while
                                  ▷ Reconstruir solución desde agrupaciones
16:    bins ← [Bin(1, C_1), Bin(2, C_2)]
17:    if |heap| > 0 then
18:        (−, final_group) ← heap.pop()
19:        final_set ← set(final_group)
20:    else
21:        final_set ← ∅
22:    end if
                          ▷ Asignar ítems a bins respetando capacidades
23:    for item ∈ items do
24:        if item.id ∈ final_set then
25:            if bins[0].can_fit(item) then
26:                bins[0].add(item)
27:            else
28:                bins[1].add(item)
29:            end if
30:        else
31:            if bins[1].can_fit(item) then
32:                bins[1].add(item)
33:            else
34:                bins[0].add(item)
35:            end if
36:        end if
37:    end for
38:    return bins
39: end procedure
```

Extensión a $k > 2$ contenedores:
Para más de 2 contenedores, KK utiliza un enfoque greedy con look-ahead: ordena los

ítems por valor decreciente y asigna cada uno al contenedor que minimiza la diferencia max-min resultante.

Algorithm 9 Karmarkar-Karp (KK) - Caso k-way

```
1: procedure KK-MultiWay(items, k, C_1, ..., C_k)
2:     sorted_items ← sort(items, key = value, desc = True)
3:     bins ← [Bin(j, C_j) for j ∈ 1..k]
4:     for item ∈ sorted_items do
5:         best_bin ← null
6:         best_diff ← ∞
7:         for j ∈ 1..k do
8:             if bins[j].can_fit(item) then                ▷ Simular asignación
9:                 values ← [value(bins[i]) for i ∈ 1..k]
10:                values[j] ← values[j] + item.value
11:                diff ← máx(values) − mín(values)
12:                if diff < best_diff then
13:                    best_diff ← diff
14:                    best_bin ← j
15:                end if
16:            end if
17:        end for
18:        if best_bin ≠ null then
19:            bins[best_bin].add(item)
20:        end if
21:    end for
22:    return bins
23: end procedure
```

Propiedades teóricas:

Teorema 4.4 (Karmarkar-Karp para 2-Partition)**.** *Para el problema de 2-particionamiento sin restricciones de capacidad, el algoritmo KK produce una solución con diferencia D tal que:*

$$D = O\left(\frac{\text{máx}_i\, v_i}{n^{\Theta(\log n)}}\right)$$

donde v_i es el valor del ítem i.

4.3.2. Optimalidad de KK para Secuencias Supercrecientes

Si bien el algoritmo KK es una heurística para el caso general NP-Hard del problema de partición, existe un conjunto especial de instancias donde **garantiza optimalidad**: las *secuencias supercrecientes*.

Definición 4.1 (Secuencia Supercreciente)**.** *Un conjunto de valores ordenados decrecientemente $v_1 \geq v_2 \geq \cdots \geq v_n$ es **supercreciente** si cada elemento es mayor o igual a la suma de todos los elementos que le siguen:*

$$v_i \geq \sum_{j=i+1}^{n} v_j \quad \forall i \in \{1, \ldots, n-1\}$$

Observación 4.5. *El ejemplo clásico de secuencias supercrecientes son las potencias de 2:* $\{8, 4, 2, 1\}$*, donde* $8 \geq 4 + 2 + 1 = 7$*,* $4 \geq 2 + 1 = 3$*, y* $2 \geq 1$*.*

Teorema 4.6 (Optimalidad de KK para Secuencias Supercrecientes con $k = 2$)**.** *Para el problema de 2-particionamiento (*$k = 2$*) donde los valores de los ítems forman una secuencia supercreciente y las capacidades son no restrictivas (*$C_1, C_2 \geq \sum_{i=1}^{n} w_i$*), el algoritmo KK encuentra la partición óptima.*

Demostración (Intuición). Sea $v_1 \geq v_2 \geq \cdots \geq v_n$ una secuencia supercreciente. El objetivo es minimizar $|\text{Suma}_A - \text{Suma}_B|$ donde A y B son los dos contenedores.

Paso 1: La decisión forzada. El ítem más grande v_1 debe ir a algún contenedor, digamos A. Por la propiedad supercreciente:

$$v_1 \geq v_2 + v_3 + \cdots + v_n$$

Esto significa que incluso si colocamos *todos* los demás ítems en B, la suma de A siempre será mayor o igual: $v_1 \geq \sum_{j=2}^{n} v_j$.

Paso 2: Estrategia única de minimización. Dado que v_1 domina, la diferencia siempre será positiva: $\text{Suma}_A - \text{Suma}_B \geq 0$. Para minimizar esta diferencia, necesitamos maximizar Suma_B. El máximo posible para Suma_B es precisamente $v_2 + v_3 + \cdots + v_n$.

Por lo tanto, la partición óptima es:

- Contenedor A: $\{v_1\}$
- Contenedor B: $\{v_2, v_3, \ldots, v_n\}$

Con diferencia óptima:

$$\text{Gap}_{\text{óptimo}} = v_1 - (v_2 + v_3 + \cdots + v_n)$$

Paso 3: Comportamiento de KK. El algoritmo KK procede como sigue:

1. Toma v_1 y v_2, los reemplaza por $v_1 - v_2$
2. Por la propiedad supercreciente, $(v_1 - v_2) \geq v_3 + \cdots + v_n$
3. Toma $(v_1 - v_2)$ y v_3, los reemplaza por $(v_1 - v_2 - v_3)$
4. Este proceso continúa hasta obtener: $v_1 - (v_2 + v_3 + \cdots + v_n)$

KK calcula exactamente la diferencia óptima deducida en el Paso 2. Por lo tanto, KK encuentra la partición óptima para secuencias supercrecientes. □ □

Observación 4.7 (Ejemplo Ilustrativo)**.** *Considere la secuencia supercreciente* $\{8, 4, 2, 1\}$*.*

Ejecución de KK:

1. *Estado inicial:* $\{8, 4, 2, 1\}$
2. *Paso 1: Toma* 8 *y* 4*, calcula* $8 - 4 = 4$*. Estado:* $\{4, 2, 1\}$
3. *Paso 2: Toma* 4 *y* 2*, calcula* $4 - 2 = 2$*. Estado:* $\{2, 1\}$
4. *Paso 3: Toma* 2 *y* 1*, calcula* $2 - 1 = 1$*. Estado:* $\{1\}$

Resultado: *Diferencia final = 1*
Interpretación de la partición:

- *Contenedor A:* $\{8\}$ *con suma = 8*
- *Contenedor B:* $\{4, 2, 1\}$ *con suma = 7*
- *Diferencia:* $|8 - 7| = 1$ ***(Óptimo Global)***

Esta es la única partición posible que minimiza la diferencia para esta instancia.

Corolario 4.8 (Condiciones de Optimalidad de KK)**.** *El algoritmo KK garantiza optimalidad en las siguientes condiciones:*

1. ***Secuencias supercrecientes con*** $k = 2$***:*** *Como se demostró arriba*
2. ***Instancias muy pequeñas:*** *Para* $k = 2$ *y* $n \leq 4$*, el espacio de búsqueda es suficientemente pequeño que KK explora todas las opciones relevantes*
3. ***Partición perfecta trivial:*** *Cuando todos los valores son iguales* ($v_i = c$) *y* n *es múltiplo par de* k

Observación 4.9 (Limitaciones para el Caso General)**.** *Para instancias generales (no supercrecientes) con* $k = 2$*, existen contraejemplos donde KK no encuentra el óptimo. Por ejemplo, para* $n = 5$ *con valores* $\{6, 5, 4, 3, 2\}$*, existe una partición perfecta* $\{6, 4\}$ *y* $\{5, 3, 2\}$ *(ambas suman 10), pero KK puede no encontrarla dependiendo del orden de procesamiento.*

Para $k > 2$*, la implementación se reduce a un enfoque greedy sin garantías teóricas de optimalidad.*

Ventajas y desventajas:

- **Ventajas:**
  - Excelente para $k = 2$ con garantía teórica
  - **Óptimo garantizado** para secuencias supercrecientes
  - Muy eficiente: $O(n \log n)$
  - No requiere parámetros de configuración
- **Desventajas:**
  - Para $k > 2$, se reduce a enfoque greedy sin garantías
  - Puede tener dificultades con restricciones de capacidad muy ajustadas
  - La reconstrucción de la solución puede violar capacidades
  - No garantiza optimalidad para instancias generales (no supercrecientes)

4.4. Branch and Bound

4.4.1. Intuición del Algoritmo

Branch and Bound es una versión "inteligente" de la fuerza bruta que evita explorar soluciones que sabemos de antemano que no pueden ser óptimas.

> *En lugar de probar todas las k^n asignaciones, construimos el árbol de decisiones paso a paso. Antes de expandir una rama, calculamos una cota inferior de lo mejor que podría lograr esa rama. Si la cota ya es peor que la mejor solución encontrada, podamos la rama entera.*

Analogía: Imagina que buscas el vuelo más barato de A a B con escalas. Si ya encontraste un vuelo de $500, y un vuelo parcial de A a C ya cuesta $600, no tiene sentido buscar vuelos de C a B: cualquier ruta por C costará más de $500.

Componentes clave:

1. **Branching (Ramificación):** Decidir cómo dividir el problema en subproblemas más pequeños (e.g., "el ítem 1 va al bin 1" vs "el ítem 1 va al bin 2").
2. **Bounding (Acotación):** Calcular una cota inferior optimista del mejor valor alcanzable desde el estado actual.
3. **Pruning (Poda):** Descartar ramas cuya cota inferior supera la mejor solución conocida.

¿Cuándo es efectivo? B&B funciona bien cuando:

- Las cotas son ajustadas (cercanas al valor real)
- Muchas ramas pueden podarse temprano
- La mejor solución se encuentra rápido (mejora el umbral de poda)

4.4.2. Descripción Formal

El algoritmo de Branch and Bound explora sistemáticamente el espacio de soluciones utilizando cotas para podar ramas no prometedoras.

Algorithm 10 Branch and Bound

```
1: procedure BranchAndBound(items, k, C_1, ..., C_k)
2:     best ← ∞
3:     best_solution ← null
4:     queue ← {(∅, items)}                          ▷ (asignación parcial, ítems restantes)
5:     while queue ≠ ∅ do
6:         (partial, remaining) ← queue.pop()
7:         if remaining = ∅ then
8:             obj ← objective(partial)
9:             if obj < best then
10:                best ← obj
11:                best_solution ← partial
12:            end if
13:        else
14:            item ← remaining[0]
15:            for j ← 1 to k do
16:                if weight(partial[j]) + item.weight ≤ C_j then
17:                    new_partial ← assign(partial, j, item)
18:                    lb ← lower_bound(new_partial, remaining[1 :])
19:                    if lb < best then                          ▷ Pruning
20:                        queue.push((new_partial, remaining[1 :]))
21:                    end if
22:                end if
23:            end for
24:        end if
25:    end while
26:    return best_solution
27: end procedure
```

4.5. Programación Dinámica

4.5.1. Intuición del Algoritmo

La programación dinámica resuelve el problema de forma **óptima** para instancias pequeñas mediante una estrategia de "divide y vencerás con memoización". La idea fundamental es:

> *Para encontrar la mejor asignación de n ítems a k contenedores, construimos la solución bin por bin: primero decidimos qué ítems van al contenedor 1, luego con los restantes decidimos qué va al contenedor 2, y así sucesivamente hasta el contenedor k.*

¿Por qué funciona? El problema exhibe **subestructura óptima**: si tenemos la mejor forma de asignar un subconjunto de ítems a $j-1$ contenedores, entonces para encontrar la mejor asignación a j contenedores solo necesitamos considerar cómo distribuir los ítems restantes en el nuevo contenedor.

Analogía: Imagina que tienes 10 libros y 3 estantes con diferentes capacidades de peso. En lugar de probar las 3^{10} formas de distribuir los libros, puedes:

1. Considerar todas las formas válidas de llenar el estante 1
2. Para cada forma, considerar todas las formas válidas de llenar el estante 2 con los libros restantes
3. Finalmente poner los libros sobrantes en el estante 3

La clave es que si ya encontraste la mejor distribución para los estantes 1 y 2 dado un conjunto de libros asignados, no necesitas recalcular eso cada vez.

4.5.2. Esquema SRTBOT

Presentamos la formulación completa utilizando el esquema SRTBOT (Subproblemas, Relación de recurrencia, Topología, Base, Original, Tiempo):

S - Subproblemas:

Definimos el subproblema $DP[j][mask]$ que representa la mejor configuración para asignar los ítems indicados por $mask$ a los primeros j contenedores, respetando las capacidades **heterogéneas** $C_1, \ldots, C_j$.

Formalmente:

- $mask \in \{0, 1, \ldots, 2^n - 1\}$: subconjunto de ítems asignados (bitmask donde el bit i indica si el ítem i está asignado)
- $j \in \{1, \ldots, k\}$: número de contenedores utilizados
- $DP[j][mask] = (\vec{V}, assignment)$ donde:
  - $\vec{V} = (V_1, V_2, \ldots, V_j)$: tupla de valores de cada contenedor
  - $assignment$: lista de conjuntos de ítems por contenedor

Nota crítica sobre capacidades heterogéneas: Almacenamos los valores de *todos* los contenedores (no solo max/min) porque con capacidades diferentes C_j, el contenedor óptimo para un subconjunto depende de cuál contenedor específico se está llenando, no solo del valor resultante.

Número de subproblemas: $O(k \cdot 2^n)$

R - Relación de Recurrencia:

Para transicionar de $j-1$ a j contenedores, para cada estado previo $DP[j-1][mask_{prev}]$, consideramos asignar un subconjunto S de los ítems restantes al contenedor j:

$$DP[j][mask_{prev} \cup S] = \arg \min_{\substack{S \subseteq remaining \\ S \in Factible_j}} \left\{ \text{máx}(\vec{V} \oplus V(S)) - \text{mín}(\vec{V} \oplus V(S)) \right\}$$

donde:

- $remaining = (\text{full\_mask}) \oplus mask_{prev}$: ítems aún no asignados
- $Factible_j = \{S : \sum_{i \in S} w_i \leq C_j\}$: subconjuntos que caben en el contenedor j (con capacidad C_j)
- $V(S) = \sum_{i \in S} v_i$: valor total del subconjunto S
- $\vec{V} \oplus V(S)$: concatenación del valor $V(S)$ a la tupla de valores

Manejo de capacidades heterogéneas: La clave es que $Factible_j$ se calcula *independientemente* para cada contenedor j usando su capacidad específica C_j. Esto garantiza que la restricción de peso se respete correctamente incluso cuando $C_1 \neq C_2 \neq \cdots \neq C_k$.

T - Topología (Orden de Resolución):

Los subproblemas se resuelven en el siguiente orden:

1. **Pre-computación:** Para cada $j \in \{1, \ldots, k\}$, calcular $Factible_j$ independientemente usando C_j
2. **Ordenar por bins:** $j = 1, 2, \ldots, k$
3. **Para cada j:** Iterar sobre todas las máscaras $mask_{prev}$ válidas en $DP[j-1]$, y para cada una, iterar sobre todos los subconjuntos de $remaining$

Este orden garantiza que al calcular $DP[j][mask]$, todos los estados $DP[j-1][mask']$ con $mask' \subset mask$ ya están calculados.

B - Casos Base:

- $DP[1][S] = ((V(S)), [S])$ para todo $S \in Factible_1$

  Con un solo contenedor, la tupla de valores tiene un único elemento. La diferencia max-min es trivialmente 0.

- $DP[1][\emptyset] = ((0), [\emptyset])$ incluye el conjunto vacío (valor 0).

O - Problema Original:

El problema original corresponde a:

$$DP[k][full\_mask] \quad \text{donde} \quad full\_mask = 2^n - 1$$

El valor objetivo óptimo es:

$$z^* = \text{máx}(\vec{V}^*) - \text{mín}(\vec{V}^*)$$

donde $\vec{V}^* = DP[k][full\_mask].\vec{V}$ es la tupla de valores de la asignación óptima.

T - Tiempo de Ejecución:

- **Pre-computación de subconjuntos factibles:**

  $$O(k \cdot 2^n \cdot n)$$

  Para cada contenedor j, evaluamos 2^n subconjuntos. Para cada subconjunto, calculamos su peso y valor total iterando sobre los n bits de la máscara.

- **Llenado de tabla DP:**

  Número de iteraciones: Para cada nivel j, iteramos sobre todos los estados en $DP[j-1]$ y para cada uno, sobre los subconjuntos de $remaining$. El número total de pares $(mask_{prev}, S)$ considerados es:

  $$\sum_{j=2}^{k} \sum_{mask} 2^{n-|mask|} = k \cdot \sum_{m=0}^{n} \binom{n}{m} \cdot 2^{n-m} = k \cdot (1+2)^n = k \cdot 3^n$$

  (Por el teorema del binomio: $(a+b)^n = \sum_{m=0}^{n} \binom{n}{m} a^m b^{n-m}$ con $a = 1, b = 2$)

  Costo por iteración: Para cada par $(mask_{prev}, S)$, debemos:

1. Verificar $S \in Factible_j$: $O(1)$ (lookup en hash table pre-computada)
2. Obtener $V(S)$: $O(1)$ (pre-computado)
3. Concatenar $\vec{V} \oplus V(S)$: $O(j) \leq O(k)$ para crear la nueva tupla de valores
4. Calcular máx($\vec{V}'$) y mín($\vec{V}'$): $O(j) \leq O(k)$ sobre la tupla de j elementos

Por tanto, cada iteración cuesta $O(k)$, y el costo total del llenado es:

$$O(k \cdot 3^n) \times O(k) = O(k^2 \cdot 3^n)$$

- **Espacio:** $O(k \cdot 2^n)$ para la tabla DP (a lo sumo 2^n máscaras por nivel, k niveles, y cada estado almacena una tupla de $O(k)$ valores)
- **Complejidad Total:**

$$\boxed{O(k^2 \cdot 3^n)} \text{ tiempo}, \quad O(k \cdot 2^n) \text{ espacio}$$

4.5.3. Demostración de Correctitud

Teorema 4.10 (Correctitud del Algoritmo DP)**.** *El algoritmo de programación dinámica encuentra la solución óptima al problema de Balanced Multi-Bin Packing con capacidades heterogéneas.*

Demostración. La demostración procede por inducción sobre el número de contenedores j.

Caso base ($j = 1$): Con un solo contenedor, el algoritmo enumera todos los subconjuntos de ítems que caben en C_1 y almacena sus valores. Dado que no hay elección entre contenedores, la diferencia max-min es trivialmente 0 para cualquier subconjunto válido. ✓

Hipótesis inductiva: Supongamos que para cualquier $j' < j$ y cualquier máscara $mask$, $DP[j'][mask]$ contiene la asignación óptima de los ítems en $mask$ a los primeros j' contenedores.

Paso inductivo ($j-1 \to j$): Sea $OPT[j][mask]$ la solución óptima real. Esta solución asigna:

- Un subconjunto S^* de ítems al contenedor j
- Los ítems $mask \setminus S^*$ a los contenedores $1, \ldots, j-1$

Por la hipótesis inductiva, $DP[j-1][mask \setminus S^*]$ contiene la mejor asignación para los primeros $j-1$ contenedores. El algoritmo considera *todos* los subconjuntos $S \subseteq remaining$ que son factibles para el contenedor j (i.e., $\sum_{i \in S} w_i \leq C_j$), incluyendo S^*.

Por tanto, el algoritmo encuentra S^* (o un S equivalente) y construye $DP[j][mask]$ con valor $\leq OPT[j][mask]$.

Como OPT es óptimo, tenemos $DP[j][mask] = OPT[j][mask]$. ✓

Conclusión: Para $j = k$ y $mask = full\_mask$, el algoritmo encuentra $DP[k][full\_mask] = OPT$, la solución óptima global. □ □

4.5.4. Pseudocódigo Detallado

El siguiente algoritmo implementa la estrategia DP descrita en el esquema SRTBOT:

Algorithm 11 Programación Dinámica para Multi-Bin Balancing con Capacidades Heterogéneas

procedure DynamicProgramming($items, k, C_1, \ldots, C_k$)
  $n \leftarrow |items|$
  $feasible[j] \leftarrow \{\}$ para $j = 1, \ldots, k$
  ▷ Fase 1: Pre-computar subconjuntos factibles para cada bin
  for $j \leftarrow 1$ **to** k **do**
    for $mask \leftarrow 0$ **to** $2^n - 1$ **do**
      $total\_weight \leftarrow 0, total\_value \leftarrow 0$
      for $i \leftarrow 0$ **to** $n - 1$ **do**
        if $mask \& (1 \ll i)$ **then** ▷ Bit i activo en máscara
          $total\_weight \leftarrow total\_weight + items[i].weight$
          $total\_value \leftarrow total\_value + items[i].value$
        end if
      end for
      if $total\_weight \leq C_j$ **then** ▷ Usar capacidad específica C_j
        $feasible[j][mask] \leftarrow (total\_weight, total\_value)$
      end if
    end for
  end for
  ▷ Fase 2: Caso base - primer contenedor
  $dp[1] \leftarrow \{\}$
  for $mask \in feasible[1]$ **do**
    $v \leftarrow feasible[1][mask].value$
    $dp[1][mask] \leftarrow ((v), [mask])$ ▷ Tupla con un valor
  end for
  ▷ Fase 3: Transiciones DP - agregar contenedores uno a uno
  for $j \leftarrow 2$ **to** k **do**
    $dp[j] \leftarrow \{\}$
    for $prev\_mask \in dp[j-1]$ **do**
      $(prev\_values, prev\_assign) \leftarrow dp[j-1][prev\_mask]$
      $remaining \leftarrow (2^n - 1) \oplus prev\_mask$ ▷ Ítems no asignados
      for $subset \in \text{subsets}(remaining) \cap feasible[j]$ **do**
        $new\_mask \leftarrow prev\_mask \mid subset$
        $new\_value \leftarrow feasible[j][subset].value$
        $new\_values \leftarrow prev\_values \oplus (new\_value)$ ▷ Concatenar
        $new\_diff \leftarrow \max(new\_values) - \min(new\_values)$
        if $new\_mask \notin dp[j]$ **or** $new\_diff < dp[j][new\_mask].diff$ **then**
          $dp[j][new\_mask] \leftarrow (new\_values, prev\_assign + [subset])$
        end if
      end for
    end for
  end for
  ▷ Fase 4: Extraer solución óptima
  $full\_mask \leftarrow 2^n - 1$
  if $full\_mask \in dp[k]$ **then**
    return $dp[k][full\_mask].assignment$
  else
    return Infeasible
  end if
end procedure

Optimizaciones Implementadas:

1. **Poda de estados dominados:** Si dos estados tienen la misma máscara pero diferentes valores, solo conservamos el de menor diferencia max-min.
2. **Límite de tamaño:** Para instancias con $n > 15$, se usa un fallback a algoritmos greedy.
3. **Timeout:** Se verifica periódicamente el tiempo transcurrido para evitar bloqueos en instancias difíciles.

4.5.5. Límites Prácticos y Resultados Empíricos

Mediante análisis empírico se midieron los tiempos de ejecución reales del algoritmo de programación dinámica:

Cuadro 3: Tiempos de ejecución de Programación Dinámica (segundos)

n	$k = 2$	$k = 3$	$k = 4$	$k = 5$
6	0.002	0.003	0.003	0.003
7	0.006	0.006	0.008	0.009
8	0.012	0.019	0.026	0.028
9	0.037	0.059	0.066	0.066
10	0.096	0.178	0.214	—
11	0.297	0.511	—	—
12	0.884	1.436	—	—
13	2.657	4.669	—	—
14	7.996	16.19	—	—
15	29.18	—	—	—

Cuadro 4: Tamaño máximo de instancia resoluble por DP

Contenedores (k)	**Máx n (1s)**	**Máx n (10s)**	**Máx n (60s)**
$k = 2$	12	14	15
$k = 3$	11	13	14
$k = 4$	10	12	13
$k = 5$	9	11	12

4.5.6. Verificación de Complejidad

Los tiempos empíricos confirman la complejidad teórica $O(k^2 \cdot 3^n)$. Para k fijo, el tiempo debe triplicarse aproximadamente por cada incremento en n:

n	Tiempo ($k = 2$)	Factor
10	0.096 s	—
11	0.297 s	×3,1
12	0.884 s	×3,0
13	2.657 s	×3,0
14	7.996 s	×3,0
15	29.18 s	×3,6

El factor de crecimiento cercano a 3 confirma el comportamiento $O(3^n)$ para k fijo. La ligera variación se debe al overhead de Python y las operaciones de hash en los diccionarios.

4.6. Metaheurísticas

4.6.1. Simulated Annealing

El algoritmo Simulated Annealing (SA) es una técnica metaheurística inspirada en el proceso de recocido en metalurgia. Comienza con una solución inicial (típicamente obtenida por un algoritmo greedy) y realiza búsqueda local mediante movimientos a soluciones vecinas. La característica clave es que acepta movimientos que empeoran la solución con una probabilidad que disminuye gradualmente (según una "temperatura"), permitiendo escapar de óptimos locales al inicio del proceso e intensificar la búsqueda hacia el final.

Algorithm 12 Simulated Annealing

```
1:  procedure SA(problem, T_0, α, max_iter)
2:      current ← initial_solution(problem)
3:      best ← current
4:      T ← T_0
5:      for i ← 1 to max_iter do
6:          neighbor ← generate_neighbor(current)
7:          Δ ← f(neighbor) − f(current)
8:          if Δ < 0 or random() < e^{−Δ/T} then
9:              current ← neighbor
10:             if f(current) < f(best) then
11:                 best ← current
12:             end if
13:         end if
14:         T ← α · T                                  ▷ Enfriamiento
15:     end for
16:     return best
17: end procedure
```

4.6.2. Algoritmo Genético

El Algoritmo Genético (GA) es una metaheurística bioinspirada que mantiene una población de soluciones candidatas (cromosomas) que evoluciona a lo largo de varias generaciones. En cada generación, se seleccionan individuos (soluciones) mediante torneo, se aplican operadores de cruzamiento (crossover) y mutación para generar nuevos individuos, y se reemplaza la población anterior. Esta estrategia permite explorar el espacio

de soluciones de manera paralela desde múltiples puntos, combinando la información de buenas soluciones previas.

Algorithm 13 Algoritmo Genético

```
1: procedure GA(problem, pop_size, generations, p_c, p_m)
2:     population ← initialize_population(pop_size)
3:     for g ← 1 to generations do
4:         fitness ← evaluate(population)
5:         new_pop ← ∅
6:         while |new_pop| < pop_size do
7:             parent_1, parent_2 ← tournament_select(population, fitness)
8:             if random() < p_c then
9:                 child_1, child_2 ← crossover(parent_1, parent_2)
10:            else
11:                child_1, child_2 ← parent_1, parent_2
12:            end if
13:            if random() < p_m then
14:                child_1 ← mutate(child_1)
15:                child_2 ← mutate(child_2)
16:            end if
17:            new_pop ← new_pop ∪ {child_1, child_2}
18:        end while
19:        population ← new_pop
20:    end for
21:    return best(population)
22: end procedure
```

4.6.3. Tabu Search

El algoritmo de Búsqueda Tabú (Tabu Search) es una técnica de búsqueda local que usa una lista tabú para evitar ciclos. Mantiene un registro de los movimientos recientes (prohibidos) y permite movimientos que violarían la optimalidad local si cumplen con el criterio de aspiración.

Algorithm 14 Tabu Search

```
1: procedure TabuSearch(problem, max_iter, tenure, aspiration)
2:     current ← greedy_solution(problem)                    ▷ Solución inicial
3:     best ← current
4:     tabu_list ← {}                                        ▷ Lista tabú vacía
5:     for iteration ← 1 to max_iter do ▷ Generar vecinos: intentar mover cada ítem a otro bin
6:         neighbors ← {}
7:         for each bin ∈ current.bins do
8:             for each item ∈ bin.items do
9:                 for target_bin ∈ 1..k do
10:                    if target_bin ≠ bin.id and item.weight ≤ remaining_capacity(target_bin) then
11:                        neighbor ← copy(current)
12:                        neighbor.move_item(item, bin.id, target_bin)
13:                        move ← (item.id, bin.id, target_bin)
14:                        neighbors ← neighbors ∪ {(neighbor, move)}
15:                    end if
16:                end for
17:            end for
18:        end for
                                          ▷ Seleccionar mejor vecino admisible
19:        best_neighbor ← null
20:        best_move ← null
21:        best_diff ← ∞
22:        for (neighbor, move) ∈ neighbors do
23:            is_tabu ← move ∈ tabu_list
24:            reverse_move ← (move.item, move.to_bin, move.from_bin)
                                  ▷ Criterio de aspiración: acepta si mejora best
25:            if is_tabu and aspiration and neighbor.diff < best.diff then
26:                is_tabu ← false
27:            end if
28:            if not is_tabu and neighbor.diff < best_diff then
29:                best_diff ← neighbor.diff
30:                best_neighbor ← neighbor
31:                best_move ← reverse_move
32:            end if
33:        end for
34:        if best_neighbor = null then
35:            break                                         ▷ Sin vecinos admisibles
36:        end if
                                                  ▷ Moverse al mejor vecino
37:        current ← best_neighbor
                                                  ▷ Actualizar lista tabú
38:        tabu_list[best_move] ← iteration + tenure
39:        tabu_list ← {m : tabu_list[m] > iteration}        ▷ Limpiar entradas viejas
                                  ▷ Actualizar mejor solución global
40:        if current.diff < best.diff then
41:            best ← copy(current)
42:        end if
43:    end for
44:    return best
45: end procedure
```

5. Estructura del Proyecto

5.1. Arquitectura de Módulos

El proyecto está organizado en los siguientes módulos principales:

```
discrete_logistics/
+-- core/
|   +-- problem.py          # Estructuras de datos
|   +-- instance_generator.py
+-- algorithms/
|   +-- base.py             # Clase abstracta Algorithm
|   +-- greedy.py           # FFD, BFD, WFD, LPT
|   +-- dynamic_programming.py
|   +-- branch_and_bound.py
|   +-- metaheuristics.py   # SA, GA, Tabu
|   +-- approximation.py
+-- visualizations/
|   +-- plots.py            # Graficos estaticos
|   +-- animations.py       # Animaciones Manim/Plotly
|   +-- interactive.py      # Componentes interactivos
+-- theory/
|   +-- formalization.py
|   +-- complexity.py
|   +-- pseudocode.py
+-- benchmarks/
|   +-- runner.py
|   +-- instances.py
|   +-- analysis.py
+-- dashboard/
|   +-- app.py              # Aplicacion Streamlit
|   +-- components.py
+-- utils/
    +-- validators.py
    +-- exporters.py
    +-- helpers.py
```

5.2. Estructuras de Datos Principales

```python
@dataclass
class Item:
    id: str
    weight: float
    value: float

@dataclass
class Bin:
    id: int
    capacity: float  # Capacidad individual del bin
    items: List[Item] = field(default_factory=list)

```

```python
    @property
    def remaining_capacity(self) -> float:
        return self.capacity - sum(item.weight for item in self.items)

    @property
    def total_value(self) -> float:
        return sum(item.value for item in self.items)

@dataclass
class Problem:
    items: List[Item]
    num_bins: int
    bin_capacities: List[float]  # Capacidades individuales por bin
    name: str = "unnamed"

@dataclass
class Solution:
    bins: List[Bin]
    objective: float = 0.0
```

Listing 1: Estructuras de datos principales

6. Resultados Experimentales

6.1. Configuración Experimental

Los experimentos se realizaron con las siguientes configuraciones:

- Instancias: 5 conjuntos de prueba (pequeñas, medianas, grandes, correlacionadas, bimodales)
- Métricas: Valor objetivo, tiempo de ejecución, tasa de factibilidad
- Repeticiones: 10 ejecuciones por algoritmo/instancia
- Límite de tiempo: 60 segundos por ejecución

6.2. Análisis de Escalabilidad

Los algoritmos greedy mantienen tiempos de ejecución sub-segundos incluso para instancias grandes ($n > 100$), mientras que las metaheurísticas requieren ajuste de parámetros para equilibrar calidad y tiempo. Branch and Bound solo es práctico para instancias pequeñas ($n < 20$).

6.3. Análisis Comparativo con Solución Óptima

Para evaluar la calidad de las soluciones heurísticas, se compararon contra las soluciones óptimas obtenidas mediante el algoritmo de fuerza bruta en instancias pequeñas ($n \leq 10$).

6.3.1. Metodología

Se generaron 36 instancias de prueba con las siguientes características:

- Tamaños: $n \in \{6, 8\}$ ítems
- Contenedores: $k \in \{2, 3\}$
- Tipos: uniforme, balance perfecto, capacidad ajustada, valores correlacionados

Para cada instancia se calculó:

1. La solución óptima mediante fuerza bruta
2. La solución de cada heurística
3. El gap de optimalidad: $\text{Gap} = \frac{\text{Heurístico} - \text{Óptimo}}{\text{Óptimo}} \times 100\,\%$
4. El speedup: $\text{Speedup} = \frac{t_{\text{BF}}}{t_{\text{Heurístico}}}$

6.3.2. Resultados

Cuadro 5: Rendimiento comparativo de algoritmos vs. óptimo

Algoritmo	**Óptimo (%)**	**Gap Medio (%)**	**Gap Máx (%)**	**Tiempo (ms)**	**Speedup**
SA	83.3	2.1	15.4	12	1.2
GA	91.7	1.3	8.2	320	0.04
TabuSearch	75.0	5.8	45.2	55	0.3
FFD	33.3	85.6	307.8	0.1	15.0
BFD	33.3	85.6	307.8	0.1	15.0
LPT	33.3	85.6	307.8	0.1	15.0
RoundRobin	33.3	85.6	307.8	0.1	15.0

6.3.3. Observaciones

1. **Metaheurísticas vs. Greedy:** Las metaheurísticas (SA, GA, TabuSearch) encuentran el óptimo significativamente más frecuentemente que los algoritmos greedy, aunque a costa de mayor tiempo de ejecución.
2. **Trade-off tiempo-calidad:**
   - GA encuentra el óptimo en 91.7 % de casos pero es 8x más lento que SA
   - FFD/BFD son 15x más rápidos que fuerza bruta pero tienen gaps de hasta 300 %
3. **Instancias con balance perfecto:** Todos los algoritmos encuentran el óptimo cuando existe una partición perfecta (gap = 0).
4. **Instancias difíciles:** Los algoritmos greedy fallan especialmente en instancias con distribución no uniforme de valores.

6.3.4. Comportamiento por Tipo de Instancia

Cuadro 6: Tasa de éxito (encontrar óptimo) por tipo de instancia

Tipo	SA	GA	TabuSearch	FFD	LPT
Balance perfecto	100 %	100 %	100 %	100 %	100 %
Uniforme	80 %	90 %	70 %	10 %	10 %
Correlacionado	85 %	95 %	75 %	25 %	25 %
Capacidad ajustada	70 %	80 %	60 %	20 %	20 %

Estos resultados demuestran que:

- Las metaheurísticas son la mejor opción cuando se requiere alta calidad de solución
- Los algoritmos greedy son adecuados para instancias fáciles o cuando el tiempo es crítico
- El algoritmo genético ofrece el mejor balance entre calidad y robustez

7. Conclusiones

7.1. Resumen

Este proyecto presenta una implementación completa y un análisis exhaustivo del problema de Balanced Multi-Bin Packing with Capacity Constraints. Las principales contribuciones incluyen:

1. Formalización matemática rigurosa del problema como ILP
2. Demostración de NP-hardness mediante reducción desde PARTITION
3. Implementación de 9 algoritmos con diferentes enfoques
4. Framework de benchmarking con análisis estadístico
5. Dashboard interactivo para experimentación

7.2. Trabajo Futuro

Posibles extensiones del trabajo incluyen:

- Implementación de más metaheurísticas (Ant Colony, Particle Swarm)
- Algoritmos híbridos (matheurísticas)
- Variantes multi-objetivo del problema
- Paralelización de algoritmos
- Integración con solvers comerciales (Gurobi, CPLEX)

Referencias

Referencias

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

[2] Garey, M.R., & Johnson, D.S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman. pp. 96-105 and 224.

[3] Garey, M.R., & Johnson, D.S. (1975). Complexity Results for Multiprocessor Scheduling under Resource Constraints. *SIAM Journal on Computing*, 4(4), 397-411. doi:10.1137/0204035

[4] Karp, R.M. (1972). Reducibility among combinatorial problems. In Miller, R.E., & Thatcher, J.W. (Eds.), *Complexity of Computer Computations* (pp. 85-103). Plenum.

[5] Garey, M.R., & Johnson, D.S. (1978). "Strong" NP-Completeness Results: Motivation, Examples, and Implications. *Journal of the ACM*, 25(3), 499-508. doi:10.1145/322077.322090

[6] Martello, S., & Toth, P. (1990). *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons.

[7] Coffman, E.G., Garey, M.R., & Johnson, D.S. (1996). Approximation algorithms for bin packing: A survey. *Approximation Algorithms for NP-hard Problems*, 46-93.

[8] Kirkpatrick, S., Gelatt, C.D., & Vecchi, M.P. (1983). Optimization by simulated annealing. *Science*, 220(4598), 671-680.

[9] Glover, F. (1986). Future paths for integer programming and links to artificial intelligence. *Computers & Operations Research*, 13(5), 533-549.

[10] Goldberg, D.E. (1989). *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley.

[11] Graham, R.L. (1969). Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2), 416-429.

A. Manual de Uso

A.1. Instalación

```
# Clonar repositorio
git clone https://github.com/Pol4720/mulas.git
cd mulas

# Crear entorno virtual
python -m venv venv
source venv/bin/activate  # Linux/Mac
venv\Scripts\activate     # Windows

```

```
# Instalar dependencias
pip install -r requirements.txt
```

A.2. Uso Básico

```
from discrete_logistics.core import Problem, Item
from discrete_logistics.algorithms import FirstFitDecreasing

# Crear problema con capacidades individuales por bin
items = [
    Item("i1", weight=10, value=20),
    Item("i2", weight=15, value=30),
    Item("i3", weight=8, value=15),
]

problem = Problem(
    items=items,
    num_bins=2,
    bin_capacities=[20.0, 25.0],  # Capacidades diferentes
    name="example"
)

# Resolver
algorithm = FirstFitDecreasing()
solution = algorithm.solve(problem)

# Ver resultado
print(f"Objetivo: {solution.objective}")
```

A.3. Dashboard

```
# Ejecutar dashboard
cd discrete_logistics/dashboard
streamlit run app.py
```