

Balanced Multi-Bin Packing with Capacity Constraints

Problema de Transporte Logístico Discreto

Diseño y Análisis de Algoritmos

Proyecto DAA

`discrete_logistics`

6 de diciembre de 2025

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 (NP-hardness)
- 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 $j \in \{1, \dots, k\}$ 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 $\sigma : I \rightarrow \{1, \dots, k\}$ que mapea cada ítem a un contenedor.*

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

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

donde C_j es la capacidad específica del contenedor j .

2.2. Formulación del Problema

Entrada:

- Conjunto de ítems $I = \{1, 2, \dots, n\}$
- 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 $j \in \{1, \dots, k\}$

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 *makespan* (valor máximo entre contenedores):

$$\text{mín } z = \max_{j=1}^k \sum_{i=1}^n v_i \cdot x_{ij}$$

O equivalentemente, minimizar la diferencia máxima:

$$\text{mín} \left(\max_j V_j - \min_j V_j \right)$$

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)

$$\text{minimizar } z \tag{1}$$

$$\text{sujeto a: } \sum_{j=1}^k x_{ij} = 1 \quad \forall i \in I \tag{2}$$

$$\sum_{i=1}^n w_i \cdot x_{ij} \leq C_j \quad \forall j = 1, \dots, k \tag{3}$$

$$\sum_{i=1}^n v_i \cdot x_{ij} \leq z \quad \forall j = 1, \dots, k \tag{4}$$

$$x_{ij} \in \{0, 1\} \quad \forall i \in I, j = 1, \dots, k \tag{5}$$

$$z \geq 0 \tag{6}$$

Donde:

- (??): Cada ítem debe asignarse a exactamente un contenedor
- (??): Restricción de capacidad por peso (cada contenedor j tiene capacidad C_j)
- (??): Definición del makespan
- (??): Variables binarias de decisión

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, \dots, 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, \dots, 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-Complejidad del Problema de Decisión

Teorema 3.2 (NP-Complejidad 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 $\sigma : I \rightarrow \{1, \dots, k\}$. 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. \square

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

\square

3.3. Cadena de Reducciones: De PARTITION a Nuestro Problema

Para comprender mejor la dureza del problema, presentamos la cadena de reducciones desde problemas fundamentales:

PARTITION

3-PARTITION

BIN PACKING

BALANCED-BIN-PACKING

3.3.1. Problema PARTITION (Punto de Partida)

Definición 3.2 (PARTITION). **Entrada:** Conjunto $S = \{a_1, a_2, \dots, a_n\}$ de enteros positivos.

Pregunta: ¿Existe un subconjunto $S' \subseteq S$ tal que $\sum_{a_i \in S'} a_i = \sum_{a_i \in S \setminus S'} a_i = \frac{1}{2} \sum_{a_i \in S} a_i$?

PARTITION es uno de los 21 problemas originales de Karp (1972) demostrados NP-completos.

3.3.2. Reducción 1: PARTITION \leq_p 3-PARTITION

Definición 3.3 (3-PARTITION). **Entrada:** Conjunto $A = \{a_1, a_2, \dots, a_{3m}\}$ 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, \dots, A_m tal que cada A_i contiene exactamente 3 elementos y $\sum_{a \in A_i} a = B$?

Importancia: 3-PARTITION es fuertemente NP-completo, lo que significa que permanece NP-completo incluso si los números se representan en unario (Garey & Johnson, 1979).

3.3.3. Reducción 2: 3-PARTITION \leq_p BALANCED-BIN-PACKING

Lema 3.3 (Reducción desde 3-PARTITION). *3-PARTITION se reduce polinomialmente a BALANCED-BIN-PACKING-DEC.*

Demostración. Dada una instancia de 3-PARTITION con elementos $\{a_1, \dots, a_{3m}\}$ 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 $j = 1, \dots, m$ (capacidades uniformes)
4. Umbral de balance: $\beta = 0$ (buscamos balance perfecto)

Correctitud (\Rightarrow):

Supongamos que existe una 3-partición válida A_1, \dots, 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 (ya que $3 \times \frac{B}{4} > \frac{3B}{4}$ pero necesitamos llegar a 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. \square

3.4. Implicaciones de la NP-Compleitud

Corolario 3.4. *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 $P = NP$. □

Corolario 3.5 (Inaproximabilidad). *Bajo la hipótesis $P \neq NP$, no existe un esquema de aproximación en tiempo polinomial (PTAS) para BALANCED-BIN-PACKING a menos que se cumplan condiciones adicionales sobre la estructura de las instancias.*

3.5. Problema con Capacidades Heterogéneas

Proposición 3.6. *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 $C_1 = C_2 = \dots = C_k$).

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
First Fit Decreasing (FFD)	$O(n \log n)$	$O(n)$
Best Fit Decreasing (BFD)	$O(n^2)$	$O(n)$
Worst Fit Decreasing (WFD)	$O(n \log n)$	$O(n)$
LPT Balanced	$O(n \log n)$	$O(n)$
Programación Dinámica	$O(n \cdot C^k)$	$O(C^k)$
Branch and Bound	$O(k^n)$ peor caso	$O(n)$
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
- C : capacidad de los contenedores
- 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. Algoritmos Greedy

4.1.1. 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 1 First Fit Decreasing

```

1: procedure FFD( $items, k, C_1, \dots, C_k$ )
2:    $sorted\_items \leftarrow \text{sort}(items, \text{key} = \text{weight}, \text{desc} = \text{True})$ 
3:    $bins \leftarrow [ ] \times k$  ▷ Crear  $k$  bins con capacidades  $C_j$ 
4:   for  $item \in sorted\_items$  do
5:     for  $j \leftarrow 1$  to  $k$  do
6:       if  $\text{weight}(bins[j]) + item.\text{weight} \leq C_j$  then
7:          $bins[j].\text{add}(item)$ 
8:         break
9:       end if
10:    end for
11:  end for
12:  return  $bins$ 
13: end procedure

```

4.1.2. 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 2 LPT Balanced

```
1: procedure LPT(items, k,  $C_1, \dots, C_k$ )
2:   sorted_items  $\leftarrow$  sort(items, key = value, desc = True)
3:   bins  $\leftarrow$  [ ]  $\times k$  ▷ Bins con capacidades individuales
4:   for item  $\in$  sorted_items do
5:      $j^* \leftarrow \arg \min_{j: \text{weight}(\text{bins}[j]) + \text{item.weight} \leq C_j} \text{value}(\text{bins}[j])$ 
6:     bins[ $j^*$ ].add(item)
7:   end for
8:   return bins
9: end procedure
```

4.2. Branch and Bound

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

Algorithm 3 Branch and Bound

```
1: procedure BRANCHANDBOUND(items, k,  $C_1, \dots, C_k$ )
2:   best  $\leftarrow \infty$ 
3:   best_solution  $\leftarrow$  null
4:   queue  $\leftarrow$  {( $\emptyset$ , items)} ▷ (asignación parcial, ítems restantes)
5:   while queue  $\neq \emptyset$  do
6:     (partial, remaining)  $\leftarrow$  queue.pop()
7:     if remaining =  $\emptyset$  then
8:       obj  $\leftarrow$  objective(partial)
9:       if obj < best then
10:        best  $\leftarrow$  obj
11:        best_solution  $\leftarrow$  partial
12:      end if
13:     else
14:       item  $\leftarrow$  remaining[0]
15:       for  $j \leftarrow 1$  to k do
16:         if weight(partial[j]) + item.weight  $\leq C_j$  then
17:           new_partial  $\leftarrow$  assign(partial, j, item)
18:           lb  $\leftarrow$  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
```

4.3. Programación Dinámica

La programación dinámica resuelve el problema de forma óptima para instancias pequeñas, construyendo soluciones incrementalmente usando memoización.

4.3.1. Enfoque con Capacidades Heterogéneas

Para el caso con capacidades individuales por contenedor, utilizamos un enfoque de DP basado en subconjuntos factibles por bin.

Idea Principal:

1. Pre-computar todos los subconjuntos factibles de ítems para cada contenedor
2. Usar DP para encontrar la mejor k-partición de ítems
3. Estado: $DP[j][mask]$ = mejor solución usando j bins y asignando ítems en $mask$

Algorithm 4 Programación Dinámica para Multi-Bin Balancing

```
1: procedure DYNAMICPROGRAMMING(items, k,  $C_1, \dots, C_k$ )
2:    $n \leftarrow |\text{items}|$ 
3:    $\text{feasible}[j] \leftarrow \{\}$  para  $j = 1, \dots, k$ 
                                      $\triangleright$  Pre-computar subconjuntos factibles para cada bin
4:   for  $j \leftarrow 1$  to  $k$  do
5:     for  $\text{mask} \leftarrow 0$  to  $2^n - 1$  do
6:        $\text{total\_weight} \leftarrow 0, \text{total\_value} \leftarrow 0$ 
7:       for  $i \leftarrow 0$  to  $n - 1$  do
8:         if  $\text{mask} \& (1 \ll i)$  then
9:            $\text{total\_weight} \leftarrow \text{total\_weight} + \text{items}[i].\text{weight}$ 
10:           $\text{total\_value} \leftarrow \text{total\_value} + \text{items}[i].\text{value}$ 
11:        end if
12:      end for
13:      if  $\text{total\_weight} \leq C_j$  then
14:         $\text{feasible}[j][\text{mask}] \leftarrow (\text{total\_weight}, \text{total\_value})$ 
15:      end if
16:    end for
17:  end for
                                      $\triangleright$  DP:  $\text{dp}[j][\text{mask}] = (\text{max\_val}, \text{min\_val}, \text{assignment})$ 
18:   $\text{dp}[1] \leftarrow \{\}$ 
19:  for  $\text{mask} \in \text{feasible}[1]$  do
20:     $v \leftarrow \text{feasible}[1][\text{mask}].\text{value}$ 
21:     $\text{dp}[1][\text{mask}] \leftarrow (v, v, [\text{mask}])$ 
22:  end for
                                      $\triangleright$  Llenar tabla DP
23:  for  $j \leftarrow 2$  to  $k$  do
24:     $\text{dp}[j] \leftarrow \{\}$ 
25:    for  $\text{prev\_mask} \in \text{dp}[j - 1]$  do
26:       $(\text{prev\_max}, \text{prev\_min}, \text{prev\_assign}) \leftarrow \text{dp}[j - 1][\text{prev\_mask}]$ 
27:       $\text{remaining} \leftarrow \text{full\_mask} \oplus \text{prev\_mask}$ 
28:      for  $\text{subset} \in \text{subsets}(\text{remaining}) \cap \text{feasible}[j]$  do
29:         $\text{new\_mask} \leftarrow \text{prev\_mask} \mid \text{subset}$ 
30:         $\text{new\_value} \leftarrow \text{feasible}[j][\text{subset}].\text{value}$ 
31:         $\text{new\_max} \leftarrow \text{máx}(\text{prev\_max}, \text{new\_value})$ 
32:         $\text{new\_min} \leftarrow \text{mín}(\text{prev\_min}, \text{new\_value})$ 
33:         $\text{new\_diff} \leftarrow \text{new\_max} - \text{new\_min}$ 
34:        if  $\text{new\_mask} \notin \text{dp}[j]$  or  $\text{new\_diff} < \text{dp}[j][\text{new\_mask}].\text{diff}$  then
35:           $\text{dp}[j][\text{new\_mask}] \leftarrow (\text{new\_max}, \text{new\_min}, \text{prev\_assign} + [\text{subset}])$ 
36:        end if
37:      end for
38:    end for
39:  end for
40:   $\text{full\_mask} \leftarrow 2^n - 1$ 
41:  return  $\text{dp}[k][\text{full\_mask}].\text{assignment}$ 
42: end procedure
```

Complejidad:

- Pre-computación: $O(k \cdot 2^n \cdot n)$
- DP principal: $O(k \cdot 3^n)$ (iterar sobre particiones)
- Espacio: $O(k \cdot 2^n)$

La complejidad $O(3^n)$ surge del hecho de que para cada máscara, debemos iterar sobre todos sus subconjuntos, lo cual es $\sum_{|S|=m} \binom{n}{m} 2^m = 3^n$ por el teorema del binomio.

Optimizaciones Implementadas:

1. Poda de estados dominados
2. Límite en el tamaño de instancia ($n \leq 20$)
3. Fallback a algoritmo greedy para instancias grandes

4.4. Metaheurísticas

4.4.1. Simulated Annealing

Algorithm 5 Simulated Annealing

```

1: procedure SA(problem,  $T_0$ ,  $\alpha$ , max_iter)
2:   current  $\leftarrow$  initial_solution(problem)
3:   best  $\leftarrow$  current
4:    $T \leftarrow T_0$ 
5:   for  $i \leftarrow 1$  to max_iter do
6:     neighbor  $\leftarrow$  generate_neighbor(current)
7:      $\Delta \leftarrow f(\textit{neighbor}) - f(\textit{current})$ 
8:     if  $\Delta < 0$  or random()  $< e^{-\Delta/T}$  then
9:       current  $\leftarrow$  neighbor
10:      if  $f(\textit{current}) < f(\textit{best})$  then
11:        best  $\leftarrow$  current
12:      end if
13:    end if
14:     $T \leftarrow \alpha \cdot T$  ▷ Enfriamiento
15:  end for
16:  return best
17: end procedure

```

4.4.2. Algoritmo Genético

Algorithm 6 Algoritmo Genético

```
1: procedure GA(problem, pop_size, generations, pc, pm)
2:   population  $\leftarrow$  initialize_population(pop_size)
3:   for g  $\leftarrow$  1 to generations do
4:     fitness  $\leftarrow$  evaluate(population)
5:     new_pop  $\leftarrow$   $\emptyset$ 
6:     while  $|new\_pop| < pop\_size$  do
7:       parent1, parent2  $\leftarrow$  tournament_select(population, fitness)
8:       if random()  $< p_c$  then
9:         child1, child2  $\leftarrow$  crossover(parent1, parent2)
10:      else
11:        child1, child2  $\leftarrow$  parent1, parent2
12:      end if
13:      if random()  $< p_m$  then
14:        child1  $\leftarrow$  mutate(child1)
15:        child2  $\leftarrow$  mutate(child2)
16:      end if
17:      new_pop  $\leftarrow$  new_pop  $\cup$  {child1, child2}
18:    end while
19:    population  $\leftarrow$  new_pop
20:  end for
21:  return best(population)
22: 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

```

1 @dataclass
2 class Item:
3     id: str
4     weight: float
5     value: float
6
7 @dataclass
8 class Bin:
9     id: int
10    capacity: float # Capacidad individual del bin
11    items: List[Item] = field(default_factory=list)
12
13    @property
14    def remaining_capacity(self) -> float:
15        return self.capacity - sum(item.weight for item in self.items)
16
17    @property
18    def total_value(self) -> float:
19        return sum(item.value for item in self.items)
20
21 @dataclass
22 class Problem:
23     items: List[Item]
24     num_bins: int
25     bin_capacities: List[float] # Capacidades individuales por bin
26     name: str = "unnamed"
27
28 @dataclass
29 class Solution:
30     bins: List[Bin]
31     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. Resultados Comparativos

Cuadro 2: Comparación de algoritmos en instancias medianas ($n=30$, $k=5$)

Algoritmo	Obj. Medio	Tiempo (s)	Factible %
FFD	15.23	0.001	100
BFD	14.87	0.002	100
WFD	12.45	0.001	100
LPT	8.32	0.001	100
Simulated Annealing	5.67	2.34	100
Genetic Algorithm	4.89	5.67	100
Tabu Search	5.12	1.89	100
Branch & Bound	3.45	45.2	85

6.3. 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 ($n < 20$).

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 3-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] Martello, S., & Toth, P. (1990). *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons.
- [3] 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.
- [4] Kirkpatrick, S., Gelatt, C.D., & Vecchi, M.P. (1983). Optimization by simulated annealing. *Science*, 220(4598), 671-680.
- [5] Glover, F. (1986). Future paths for integer programming and links to artificial intelligence. *Computers & Operations Research*, 13(5), 533-549.
- [6] Goldberg, D.E. (1989). *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley.
- [7] 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

```

1 # Clonar repositorio
2 git clone <repository-url>
3 cd mulas
4
5 # Crear entorno virtual
6 python -m venv venv
7 source venv/bin/activate # Linux/Mac
8 venv\Scripts\activate    # Windows
9
10 # Instalar dependencias
11 pip install -r requirements.txt

```

A.2. Uso Básico

```

1 from discrete_logistics.core import Problem, Item
2 from discrete_logistics.algorithms import FirstFitDecreasing
3
4 # Crear problema con capacidades individuales por bin
5 items = [
6     Item("i1", weight=10, value=20),
7     Item("i2", weight=15, value=30),
8     Item("i3", weight=8, value=15),
9 ]
10
11 problem = Problem(
12     items=items,
13     num_bins=2,
14     bin_capacities=[20.0, 25.0], # Capacidades diferentes
15     name="example"
16 )
17
18 # Resolver
19 algorithm = FirstFitDecreasing()
20 solution = algorithm.solve(problem)
21
22 # Ver resultado
23 print(f"Objetivo: {solution.objective}")

```

A.3. Dashboard

```

1 # Ejecutar dashboard
2 cd discrete_logistics/dashboard
3 streamlit run app.py

```