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 $j \in \{1, \ldots, k\}$ 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 $I = \{1, 2, \dots, n\}$
- 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):

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

O equivalentemente, minimizar la diferencia máxima:

$$\min \left(\max_j V_j - \min_j V_j \right)$$

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)

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

Donde:

- (2): Cada ítem debe asignarse a exactamente un contenedor
- (3): Restricción de capacidad por peso (cada contenedor j tiene capacidad C_j)
- (4): Definición del makespan
- (5): 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 $C_1, \ldots, C_k$, 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: De PARTITION a Nuestro Problema

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

Cadena de Reducciones Polinomiales:

PARTITION $\leq_p$ 3-PARTITION $\leq_p$ BIN PACKING $\leq_p$ BALANCED-BIN-PACKING

(Cada flecha representa una reducción en tiempo polinomial)

3.3.1. Problema PARTITION (Punto de Partida)

Definición 3.2 (PARTITION)**.** ***Entrada:*** *Conjunto* $S = \{a_1, a_2, \ldots, a_n\}$ *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* $A = \{a_1, a_2, \ldots, a_{3m}\}$ *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* $\sum_{a \in A_i} a = B$*?*

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 $\{a_1, \ldots, a_{3m}\}$ 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 (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. □ □

3.4. Implicaciones de la NP-Completitud

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 $C_1 = C_2 = \cdots = C_k$).

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**
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

procedure FFD($items, k, C_1, \ldots, C_k$)
  $sorted\_items \leftarrow \text{sort}(items, \text{key} = weight, \text{desc} = True)$
  $bins \leftarrow [\,] \times k$ ▷ Crear k bins con capacidades C_j
  for $item \in sorted\_items$ **do**
    for $j \leftarrow 1$ **to** k **do**
      if $\text{weight}(bins[j]) + item.weight \leq C_j$ **then**
        $bins[j].\text{add}(item)$
        break
      end if
    end for
  end for
  return $bins$
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

procedure LPT($items, k, C_1, \ldots, C_k$)
  $sorted\_items \leftarrow \text{sort}(items, \text{key} = value, \text{desc} = True)$
  $bins \leftarrow [\,] \times k$ ▷ Bins con capacidades individuales
  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. 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

```
procedure BranchAndBound(items, k, C_1, ..., C_k)
    best ← ∞
    best_solution ← null
    queue ← {(∅, items)}                          ▷ (asignación parcial, ítems restantes)
    while queue ≠ ∅ do
        (partial, remaining) ← queue.pop()
        if remaining = ∅ then
            obj ← objective(partial)
            if obj < best then
                best ← obj
                best_solution ← partial
            end if
        else
            item ← remaining[0]
            for j ← 1 to k do
                if weight(partial[j]) + item.weight ≤ C_j then
                    new_partial ← assign(partial, j, item)
                    lb ← lower_bound(new_partial, remaining[1 :])
                    if lb < best then                 ▷ Pruning
                        queue.push((new_partial, remaining[1 :]))
                    end if
                end if
            end for
        end if
    end while
    return best_solution
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. Esquema SRTBOT

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

S - Subproblemas:

Definimos el subproblema $DP[j][mask]$ como la mejor solución (mínima diferencia entre valores máximo y mínimo) para asignar los ítems indicados por la máscara de bits $mask$ a los primeros j contenedores, respetando las capacidades $C_1, \ldots, C_j$.

Formalmente:

- $mask \in \{0, 1, \ldots, 2^n - 1\}$: subconjunto de ítems asignados (representado como máscara de bits)
- $j \in \{1, \ldots, k\}$: número de contenedores utilizados

- $DP[j][mask] = (V_{max}, V_{min}, assignment)$ donde:
  - V_{max}: valor máximo entre los j contenedores
  - V_{min}: valor mínimo entre los j contenedores
  - $assignment$: lista de máscaras indicando asignación por contenedor

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

R - Relación de Recurrencia:

Para cada subproblema $DP[j][mask]$, consideramos todos los subconjuntos S de los ítems restantes que son factibles para el contenedor j:

$$DP[j][mask \cup S] = \min_{\substack{S \subseteq remaining \\ S \in Factible(j)}} \{\max(DP[j-1][mask].V_{max}, V(S)) - \min(DP[j-1][mask].V_{min}, V(S))$$

donde:

- $remaining = (\text{full\_mask}) \oplus mask$: ítems aún no asignados
- $Factible(j) = \{S : \sum_{i \in S} w_i \leq C_j\}$: subconjuntos que caben en contenedor j
- $V(S) = \sum_{i \in S} v_i$: valor total del subconjunto S

T - Topología (Orden de Resolución):

Los subproblemas se resuelven en el siguiente orden:

1. Pre-computación: calcular $Factible(j)$ para cada $j \in \{1, \dots, k\}$
2. Ordenar por número de contenedores: $j = 1, 2, \dots, k$
3. Para cada j, iterar sobre máscaras $mask$ en orden creciente de cardinalidad (número de bits en 1)

Este orden garantiza que cuando calculamos $DP[j][mask]$, ya tenemos calculados todos los valores $DP[j-1][mask']$ necesarios donde $mask' \subset mask$.

B - Casos Base:

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

  Es decir, con un solo contenedor, la diferencia es 0 (solo hay un valor) y el estado almacena el valor del único contenedor.

- $DP[j][\emptyset] = (\infty, 0, [])$ representa el estado inicial sin ítems asignados (no válido para solución final)

O - Problema Original:

El problema original corresponde a:

$$DP[k][full\_mask]$$

donde $full\_mask = 2^n - 1$ representa el conjunto de todos los ítems.

El valor objetivo óptimo es:

$$z^* = DP[k][full\_mask].V_{max} - DP[k][full\_mask].V_{min}$$

T - Tiempo de Ejecución:

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

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

Para cada contenedor, evaluamos 2^n subconjuntos, y cada evaluación de peso/valor toma $O(n)$.

- **Llenado de tabla DP:**

Para cada estado $(j, mask)$, debemos iterar sobre todos los subconjuntos de $remaining = full\_mask \oplus mask$. El número total de operaciones es:

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

Multiplicando por k contenedores: $O(k \cdot 3^n)$

- **Espacio:** $O(k \cdot 2^n)$ para almacenar la tabla DP

- **Complejidad Total:** $O(k \cdot 3^n)$ tiempo, $O(k \cdot 2^n)$ espacio

4.3.2. 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

```
procedure DynamicProgramming(items, k, C_1, ..., C_k)
    n ← |items|
    feasible[j] ← {} para j = 1, ..., k
                                    ▷ Pre-computar subconjuntos factibles para cada bin
    for j ← 1 to k do
        for mask ← 0 to 2^n − 1 do
            total_weight ← 0, total_value ← 0
            for i ← 0 to n − 1 do
                if mask&(1 ≪ i) then
                    total_weight ← total_weight + items[i].weight
                    total_value ← total_value + items[i].value
                end if
            end for
            if total_weight ≤ C_j then
                feasible[j][mask] ← (total_weight, total_value)
            end if
        end for
    end for
                                    ▷ DP: dp[j][mask] = (max_val, min_val, assignment)
    dp[1] ← {}
    for mask ∈ feasible[1] do
        v ← feasible[1][mask].value
        dp[1][mask] ← (v, v, [mask])
    end for
                                    ▷ Llenar tabla DP
    for j ← 2 to k do
        dp[j] ← {}
        for prev_mask ∈ dp[j − 1] do
            (prev_max, prev_min, prev_assign) ← dp[j − 1][prev_mask]
            remaining ← full_mask ⊕ prev_mask
            for subset ∈ subsets(remaining) ∩ feasible[j] do
                new_mask ← prev_mask | subset
                new_value ← feasible[j][subset].value
                new_max ← máx(prev_max, new_value)
                new_min ← mín(prev_min, new_value)
                new_diff ← new_max − new_min
                if new_mask ∉ dp[j] or new_diff < dp[j][new_mask].diff then
                    dp[j][new_mask] ← (new_max, new_min, prev_assign + [subset])
                end if
            end for
        end for
    end for
    full_mask ← 2^n − 1
    return dp[k][full_mask].assignment
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

```
procedure SA(problem, T_0, α, max_iter)
    current ← initial_solution(problem)
    best ← current
    T ← T_0
    for i ← 1 to max_iter do
        neighbor ← generate_neighbor(current)
        Δ ← f(neighbor) − f(current)
        if Δ < 0 or random() < e^{−Δ/T} then
            current ← neighbor
            if f(current) < f(best) then
                best ← current
            end if
        end if
        T ← α · T                                  ▷ Enfriamiento
    end for
    return best
end procedure
```

4.4.2. Algoritmo Genético

Algorithm 6 Algoritmo Genético

```
procedure GA(problem, pop_size, generations, p_c, p_m)
    population ← initialize_population(pop_size)
    for g ← 1 to generations do
        fitness ← evaluate(population)
        new_pop ← ∅
        while |new_pop| < pop_size do
            parent_1, parent_2 ← tournament_select(population, fitness)
            if random() < p_c then
                child_1, child_2 ← crossover(parent_1, parent_2)
            else
                child_1, child_2 ← parent_1, parent_2
            end if
            if random() < p_m then
                child_1 ← mutate(child_1)
                child_2 ← mutate(child_2)
            end if
            new_pop ← new_pop ∪ {child_1, child_2}
        end while
        population ← new_pop
    end for
    return best(population)
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)

    @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. 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

```bash
# Clonar repositorio
git clone <repository-url>
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

```python
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

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