

Proyecto de Diseño y Análisis de Algoritmos: Problema de Logística Discreta (DLP)

Salma Fonseca Curbelo C-412, José Ernesto Morales Lazo C-412

6 de enero de 2026

1. Formalización del Modelo de Logística Discreta

1.1. Estructuras de Datos de Entrada

- **Mulas (M):** Un arreglo `mulas[]` de tamaño n , donde cada posición i contiene la capacidad c_i .
 - $c_i \in \mathbb{R}^+$: Capacidad de carga de la mula m_i .
- **Artículos (A):** Un arreglo `articulos[]` de tamaño k , donde cada posición j contiene una tupla $\langle w_j, v_j \rangle$.
 - $w_j \in \mathbb{R}^+$: Peso del artículo a_j .
 - $v_j \in \mathbb{R}^+$: Valor del artículo a_j .

1.2. Variables de Decisión

Definimos la matriz binaria de asignación:

$$x_{ij} = \begin{cases} 1 & \text{si el artículo } j \text{ se asigna a la mula } i \\ 0 & \text{en otro caso} \end{cases}$$

$\forall i \in \{1, \dots, n\}, \forall j \in \{1, \dots, k\}$.

1.3. Modelo Matemático (Linealizado)

Para que el modelo sea procesable por un computador, linealizamos la función objetivo utilizando dos variables auxiliares: V_{max} (valor de la mula más cargada) y V_{min} (valor de la mula menos cargada).

Función Objetivo:

$$\text{Minimizar } Z = V_{max} - V_{min}$$

Sujeto a (Restricciones):

1. **Cálculo de Valores por Mula:** El valor total V_i transportado por la mula i se define como:

$$V_i = \sum_{j=1}^k v_j x_{ij} \quad \forall i \in \{1, \dots, n\}$$

Y debe cumplirse que:

$$V_{min} \leq V_i \leq V_{max} \quad \forall i \in \{1, \dots, n\}$$

2. **Asignación Única (Partición):** Cada artículo j debe asignarse exactamente a una mula.

$$\sum_{i=1}^n x_{ij} = 1 \quad \forall j \in \{1, \dots, k\}$$

3. **Capacidad de Peso:** La carga de la mula i no debe exceder su capacidad individual c_i .

$$\sum_{j=1}^k w_j x_{ij} \leq c_i \quad \forall i \in \{1, \dots, n\}$$

4. **Integridad de Variables:**

$$x_{ij} \in \{0, 1\}, \quad V_{max}, V_{min} \geq 0$$

1.4. Propiedades de la Salida

- **Factibilidad:** Si no existe ninguna combinación de x_{ij} que satisfaga simultáneamente la capacidad de peso de todas las mulas y la asignación de todos los artículos, el sistema debe informar que la instancia es Irresoluble.
- **Optimidad:** La solución devuelta debe garantizar que no existe otra configuración donde la brecha de riesgo ($V_{max} - V_{min}$) sea menor.
- **Formato de Salida:** Una lista de conjuntos $S = \{M_1, M_2, \dots, M_n\}$, donde cada M_i contiene los índices j de los artículos tales que $x_{ij} = 1$.

2. Análisis de Complejidad Computacional

2.1. Demostración de que el Problema de la Partición es NP Completo

El Problema de la Partición: Dado un multiconjunto S de enteros positivos , ¿es posible dividir S en dos subconjuntos S_1 y S_2 tales que la suma de los elementos en S_1 sea igual a la suma de los elementos en S_2 ?

2.1.1. Demostración de que el problema de la partición es NP:

Un problema está en la clase NP si una solución propuesta (un "certificado") puede ser verificada en tiempo polinomial.

1. **Certificado:** Una lista de los elementos que pertenecen al subconjunto S_1 .
2. **Verificación:** Un algoritmo simplemente suma los elementos de S_1 y suma los elementos de $S \setminus S_1$ (que sería S_2). Luego compara si las sumas son iguales.
3. **Tiempo:** La suma y la comparación son operaciones lineales. Por lo tanto, el problema de la partición es NP.

2.1.2. Demostración de que el problema de la partición es NP-Completo:

Se usará el problema de la Suma de Subconjuntos (Subset-Sum), el cual se vió en clase que es NP-Completo.

Subset-Sum: Dado un conjunto de enteros A y un entero objetivo t , ¿existe un subconjunto $A' \subseteq A$ tal que la suma de sus elementos sea t ?

Reducción Subset-Sum \leq_p Partición

Se debe transformar cualquier instancia de Subset-Sum en una instancia del problema de la Partición en tiempo polinomial.

1. **Construcción:** Sea una instancia de Subset-Sum con conjunto A y objetivo t . Se calcula la suma total de A , denotada como $\Sigma = \sum_{i=1}^n a_i$. Se construye un nuevo conjunto S para el problema partición agregando dos elementos adicionales especiales, J y K , al conjunto A .

- $J = 2\Sigma - t$
- $K = \Sigma + t$
- $S = A \cup \{J, K\}$

2. **Equivalencia:** La suma total de los elementos en el nuevo conjunto es:

$$\text{Sum}(S) = \Sigma + (2\Sigma - t) + (\Sigma + t) = 4\Sigma$$

Para que exista una partición válida en S , se debe dividir S en dos subconjuntos, cada uno sumando exactamente la mitad del total, es decir, 2Σ .

Cómo se puede llegar a 2Σ :

- Los elementos J y K no pueden estar en el mismo subconjunto porque $J+K = 3\Sigma$, lo cual excede el objetivo 2Σ .
- Por lo tanto, J debe estar en un subconjunto (S_1) y K en el otro (S_2).
- Para que S_1 sume 2Σ , debe contener a J más algunos elementos de A (A_{sub}) tal que:

$$J + \Sigma A_{sub} = 2\Sigma$$

$$(2\Sigma - t) + \Sigma A_{sub} = 2\Sigma$$

$$\Sigma A_{sub} = t$$

Conclusión: Existe una partición válida en S si y solo si existe un subconjunto en A que sume exactamente t . Como la transformación es polinomial, el problema de la Partición es NP-Completo.

2.2. Demostración de que el Problema de la DLP es NP Completo

2.2.1. Demostración de que DLP es NP:

1. **Certificado:** Una asignación propuesta de cada artículo a una mula específica.

2. **Verificación:** Un algoritmo verificador calcula la suma de pesos de cada una de las m mulas y comprueba si excede el límite de peso C . Luego calcula la suma de valores V_j para cada mula y compara para cada par de mulas V_a y V_b y verifica si $|V_a - V_b| \leq K$.
3. **Tiempo:** Las sumas son operaciones lineales $O(n)$ y como hay $m(m - 1)/2$ pares de mulas y $m \leq n$, la compración es $O(n^2)$, por lo que se puede verificar en tiempo polinomial.

Como la verificación se realiza en tiempo polinomial, DLP es NP.

2.2.2. Demostración que DLP es NP-difícil

Se usará el problema de la Partición, el cual se demostró anteriormente que es NP-Completo.

Reducción Partición \leq_p DLP

Se tomará una instancia arbitraria de Partición con el conjunto S . Sea $W_{total} = \sum_{i=1}^n x_i$. Se construye una instancia de DLP con los siguientes parámetros específicos:

1. **Artículos:** Para cada número $x_i \in S$, se crea un artículo i .
2. **Pesos y Valores:** Se define tanto el peso como el valor igual al número original: $w_i = x_i, v_i = x_i$.
3. **Número de Mulas (m):** Se fija $m = 2$.
4. **Diferencia permitida (K):** Se fija $K = 0$.
5. **Capacidad (C):** Se fija $C = W_{total}/2$. (Si W_{total} es impar, no hay solución para Partición ni para este DLP, así que se asume par).

Demostración de Equivalencia:

Si Partición tiene solución: Existen S_1 y S_2 tales que $\sum S_1 = \sum S_2 = W_{total}/2$. Se asignan los artículos correspondientes a S_1 a la Mula 1 y los de S_2 a la Mula 2. Verificación de Peso: La Mula 1 lleva peso $W_{total}/2$, que es $= C$. La Mula 2 igual. (Cumple). Verificación de Balance: El valor en la Mula 1 es $W_{total}/2$. El valor en la Mula 2 es $W_{total}/2$. La diferencia es $|W_{total}/2 - W_{total}/2| = 0$. Como $0 \leq K$ (donde $K = 0$), cumple la condición. Por tanto, DLP devuelve SÍ.

Si DLP devuelve SÍ: Significa que existe una distribución en 2 mulas tal que la diferencia de valores es ≤ 0 . Como el valor absoluto no puede ser negativo, la diferencia debe ser exactamente 0.

$$\sum_{i \in m1} v_i = \sum_{i \in m2} v_i$$

Dado que se definió $v_i = x_i$, esto implica:

$$\sum_{i \in m1} x_i = \sum_{i \in m2} x_i$$

Esto constituye una partición válida del conjunto original en dos sumas iguales.

Conclusión: Se ha demostrado que el problema Partición es un caso particular del problema DLP (específicamente cuando $m = 2, w_i = v_i$ y $K = 0$). Dado que el caso particular es NP-Completo, el caso general es al menos igual de difícil.

Por lo tanto, el Problema de Logística Discreta (DLP) es NP-Completo.

3. Diseño de Soluciones Algorítmicas

Dado que el Problema de Logística Discreta (DLP) ha sido demostrado como NP-Completo en la sección anterior, es computacionalmente intratable resolverlo de manera exacta para instancias grandes en tiempo polinomial. Por ello, en esta sección se exploran soluciones algorítmicas prácticas. Primero, se presenta un algoritmo de fuerza bruta que garantiza la solución óptima, útil como línea base para comparar resultados en instancias pequeñas. Posteriormente, se propone una solución eficiente basada en heurísticas, combinando un enfoque voraz (greedy) con búsqueda local reforzada, para obtener soluciones aproximadas de alta calidad en tiempo razonable.

3.1. Enfoque Exacto: Programación Dinámica con Memoización

Para obtener la solución óptima del problema de transporte discreto, se implementó un algoritmo basado en Programación Dinámica con enfoque *Top-Down*. Este método explora el espacio de soluciones de manera sistemática, asegurando el cumplimiento estricto de las restricciones de peso (similar al problema *Bin Packing*) mientras minimiza la función objetivo de balanceo ($V_{max} - V_{min}$).

A diferencia de una fuerza bruta pura, la inclusión de una tabla de memoización evita el recálculo de subproblemas idénticos que surgen de diferentes permutaciones de asignación que resultan en el mismo estado de carga acumulada.

3.1.1. Definición Formal

El problema se modela mediante la función recursiva $DP(i, \vec{S})$, definida por las siguientes características:

- **Subproblema:** Determinar la asignación óptima para el conjunto de artículos restantes $\{i, \dots, N-1\}$, dado un estado actual de las mulas \vec{S} . El estado \vec{S} se representa como una tupla que contiene pares (v_k, w_k) indicando el valor y peso acumulado de cada mula k .
- **Relación de Recurrencia:** Para el artículo i , se exploran M ramas de decisión (donde M es el número de mulas). La función elige la opción que minimiza la diferencia final:

$$DP(i, \vec{S}) = \min_{k=1}^M \{DP(i+1, \vec{S}'_k)\} \quad (1)$$

Donde \vec{S}'_k es el nuevo estado resultante de asignar el artículo i a la mula k . Esta transición es válida si y solo si se cumple la restricción de capacidad:

$$w_k + w_{articulo} \leq Capacidad_k$$

- **Casos Base:** Cuando $i = N$ (todos los artículos han sido procesados), se evalúa la función objetivo sobre el estado final:

$$f(\vec{S}) = \max_k(v_k) - \min_k(v_k) \quad (2)$$

- **Problema Original:** La ejecución comienza con la llamada $DP(0, \vec{S}_{vacía})$, donde todas las mulas inician con valor y peso cero.

3.1.2. Complejidad Computacional

La complejidad temporal está determinada por el tamaño del espacio de estados únicos visitados. Si bien la memoización reduce drásticamente el tiempo comparado con $O(M^N)$, la complejidad sigue siendo **pseudo-polinomial** y exponencial respecto al número de mulas:

$$T(N, M) \approx O(N \cdot M \cdot |\mathcal{S}|)$$

Donde $|\mathcal{S}|$ representa el número de combinaciones posibles de sumas de valores y pesos acumulados válidos en las M mulas.

3.1.3. Implementación y Pseudocódigo

A continuación se presenta la implementación del algoritmo en Python. La variable `idx` denota el artículo actual (orden topológico $0 \rightarrow N$) y `memo` almacena las soluciones óptimas para evitar recomputar ramas del árbol.

```
def resolver_transporte_discreto(articulos, mulas_capacidades):
    num_articulos = len(articulos)
    num_mulas = len(mulas_capacidades)
    # Tabla de memoización para almacenar subproblemas resueltos
    memo = {}

    def solve(idx, estados_mulas):
        # Identificador único del subproblema (Estado)
        estado_actual = (idx, estados_mulas)

        # 1. Verificación en tabla de memoización (Poda por redundancia)
        if estado_actual in memo:
            return memo[estado_actual]

        # 2. Caso Base: Todos los artículos asignados
        if idx == num_articulos:
            valores = [m[0] for m in estados_mulas]
            # Retorna la diferencia (objetivo) y el estado final
            return max(valores) - min(valores), estados_mulas

        mejor_dif = float('inf')
        mejor_config = None
        valor_art, peso_art = articulos[idx]

        # 3. Transición: Intentar asignar el artículo actual a cada mula
        for i in range(num_mulas):
            v_curr, p_curr = estados_mulas[i]

            # Verificación de restricción de capacidad (Poda por factibilidad)
            if p_curr + peso_art <= mulas_capacidades[i]:
                # Crear nuevo estado (tuplas inmutables para hashing)
                nuevas_mulas = list(estados_mulas)
                nuevas_mulas[i] = (v_curr + valor_art, p_curr + peso_art)

                mejor_dif = min(mejor_dif, solve(idx + 1, nuevas_mulas))

        memo[estado_actual] = mejor_dif
        return mejor_dif, estados_mulas
```

```

# Llamada Recursiva (Paso inductivo)
dif, config = solve(idx + 1, tuple(nuevas_mulas))

# Minimización de la diferencia en el retorno
if dif < mejor_dif:
    mejor_dif = dif
    mejor_config = config

# Almacenar el mejor resultado encontrado para este estado
memo[estado_actual] = (mejor_dif, mejor_config)
return mejor_dif, mejor_config

# Estado Inicial: idx=0, mulas vacías
estado_inicial_mulas = tuple((0, 0) for _ in range(num_mulas))
resultado, configuracion_final = solve(0, estado_inicial_mulas)

return resultado, configuracion_final

```

3.2. Solución Eficiente Basada en Heurísticas y Metaheurísticas

Para instancias mayores, se propone una heurística híbrida que combina kernelización previa, un algoritmo voraz inspirado en Longest Processing Time (LPT) adaptado al balance de valores, y una búsqueda local reforzada con movimientos simples e intercambios. Esta aproximación no garantiza optimalidad, pero ofrece una garantía práctica de rendimiento cercana al óptimo en muchos casos, con complejidad temporal polinomial ($O(k \log k + kn + I \cdot k^2)$), donde I es el número de iteraciones de la búsqueda local (típicamente pequeño).

3.2.1. Kernelización Previa

Se realiza un preprocessamiento para reducir la instancia, rechazando artículos que excedan la capacidad máxima de cualquier mula.

3.2.2. Algoritmo Voraz (Greedy LPT Adaptado)

- Ordenar los artículos por valor descendente para priorizar el balance de riesgo.
- Asignar cada artículo a la mula con el menor valor acumulado actual que tenga espacio suficiente en peso.
- Si un artículo no cabe en ninguna mula, se marca como no asignado.

3.2.3. Búsqueda Local Reforzada

- Iterativamente, identificar la mula más rica” (mayor valor) y la más ”pobre” (menor valor).
- Intentar movimientos: Transferir un artículo de la rica a la pobre si mejora la diferencia y respeta el peso.

- Si no, intentar intercambios (swaps): Cambiar un artículo caro por uno barato entre rica y pobre, verificando pesos y mejora en la diferencia.
- Repetir hasta que no se encuentren mejoras.

Esta metaheurística mejora localmente la solución greedy, escapando de óptimos locales simples.

3.2.4. Pseudocódigo

```
def solucion_profesional_mejorada(articulos, mulas_capacidades):
    # 1. Kernelización
    items_procesables, rechazo_inicial = kernelizacion_previa(articulos, mulas_capacidades)

    # 2. Greedy LPT Adaptado
    items = sorted([(v, p, i) for i, (v, p) in enumerate(items_procesables)],
                  key=lambda x: x[0], reverse=True)
    num_mulas = len(mulas_capacidades)
    mulas = [{'valor': 0, 'peso': 0, 'capacidad': cap, 'items': []}
              for cap in mulas_capacidades]
    no_asignados = []
    for item in items:
        mulas_ordenadas = sorted(range(num_mulas), key=lambda i: mulas[i]['valor'])
        asignado = False
        for i in mulas_ordenadas:
            if mulas[i]['peso'] + item[1] <= mulas[i]['capacidad']:
                mulas[i]['valor'] += item[0]
                mulas[i]['peso'] += item[1]
                mulas[i]['items'].append(item)
                asignado = True
                break
        if not asignado: no_asignados.append(item)

    # 3. Búsqueda Local Reforzada
    mejorado = True
    while mejorado:
        mejorado = False
        rica = max(range(num_mulas), key=lambda i: mulas[i]['valor'])
        pobre = min(range(num_mulas), key=lambda i: mulas[i]['valor'])
        dif_actual = mulas[rica]['valor'] - mulas[pobre]['valor']

        # Movimiento Simple
        for i, item_r in enumerate(mulas[rica]['items']):
            if mulas[pobre]['peso'] + item_r[1] <= mulas[pobre]['capacidad']:
                nueva_dif = abs((mulas[rica]['valor'] - item_r[0]) -
                                (mulas[pobre]['valor'] + item_r[0]))
                if nueva_dif < dif_actual:
                    mulas[pobre]['items'].append(mulas[rica]['items'].pop(i))
                    mulas[rica]['valor'] -= item_r[0]; mulas[rica]['peso'] -= item_r[1]
                    mejorado = True
```

```

        mulas[pobre]['valor'] += item_r[0]; mulas[pobre]['peso'] += item_r[1]
        mejorado = True; break
    if mejorado: continue

    # Intercambio (Swap)
    for i, item_r in enumerate(mulas[rica]['items']):
        for j, item_p in enumerate(mulas[pobre]['items']):
            peso_rica_tras = mulas[rica]['peso'] - item_r[1] + item_p[1]
            peso_pobre_tras = mulas[pobre]['peso'] - item_p[1] + item_r[1]
            if (peso_rica_tras <= mulas[rica]['capacidad'] and
                peso_pobre_tras <= mulas[pobre]['capacidad']):
                nueva_dif = abs((mulas[rica]['valor'] - item_r[0] + item_p[0]) -
                                (mulas[pobre]['valor'] - item_p[0] + item_r[0]))
                if nueva_dif < dif_actual:
                    mulas[rica]['items'][i], mulas[pobre]['items'][j] = \
                        mulas[pobre]['items'][j], mulas[rica]['items'][i]
                    mulas[rica]['valor'] += (item_p[0] - item_r[0])
                    mulas[rica]['peso'] += (item_p[1] - item_r[1])
                    mulas[pobre]['valor'] += (item_r[0] - item_p[0])
                    mulas[pobre]['peso'] += (item_r[1] - item_p[1])
                    mejorado = True; break
            if mejorado: break

    return (max(m['valor'] for m in mulas) - min(m['valor'] for m in mulas)), mulas, :

```