Proyecto de Diseño y Análisis de Algoritmos: Problema del Negocio del Transporte Discreto (NTD)

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

1. Formalización del Modelo del Transporte Discreto

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:

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

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

1.3. Modelo Matemático (Linealizado)

Para que el modelo sea computable, 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:
$$V_i = \sum_{j=1}^{k} v_j x_{ij} \quad \forall i \in \{1, \ldots, n\}$$
Y debe cumplirse que:
$$V_{min} \leq V_i \leq V_{max} \quad \forall i \in \{1, \ldots, n\}$$

2. **Asignación Única (Partición):** Cada artículo j debe asignarse exactamente a una mula.
$$\sum_{i=1}^{n} x_{ij} = 1 \quad \forall j \in \{1, \ldots, k\}$$

3. **Capacidad de Peso:** La carga de la mula i no debe exceder su capacidad individual c_i.
$$\sum_{j=1}^{k} w_j x_{ij} \leq c_i \quad \forall i \in \{1, \ldots, n\}$$

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 $S = \{M_1, M_2, \ldots, M_n\}$, 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 çertificado") 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-Hard:

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 un conjunto $A = \{a_1, a_2, \ldots, a_n\}$ y un objetivo t. Sea $\Sigma = \sum_{a_i \in A} a_i$. Sin pérdida de generalidad, asumimos $t \leq \Sigma$ (de lo contrario la solución es trivialmente No). Construimos el conjunto $S = A \cup \{J, K\}$ donde:
   - $J = 2\Sigma - t$
   - $K = \Sigma + t$

   La suma total del conjunto S es $\text{Sum}(S) = \Sigma + (2\Sigma - t) + (\Sigma + t) = 4\Sigma$. Para que S tenga una partición, debe existir un subconjunto $S_1 \subset S$ tal que $\text{Sum}(S_1) = 2\Sigma$.
2. **Lema de Separación (Demostración de que J y K están en subconjuntos distintos):** Para que exista una partición válida S_1, S_2, los elementos J y K no pueden pertenecer al mismo subconjunto.

   Prueba por contradicción: Supongamos que $\{J, K\} \subseteq S_1$. Entonces, la suma de S_1 sería:
   $$\text{Sum}(S_1) \geq J + K = (2\Sigma - t) + (\Sigma + t) = 3\Sigma$$
   Como para una partición válida se requiere que $\text{Sum}(S_1) = 2\Sigma$, y dado que $\Sigma > 0$, tenemos que $3\Sigma > 2\Sigma$. Esto contradice la definición de partición. Por lo tanto, J y K deben estar en subconjuntos distintos.
3. **Demostración de la Equivalencia (A tiene subconjunto que suma $t \iff S$ tiene partición):**
   - **($\implies$):** Supongamos que existe $A' \subseteq A$ tal que $\sum_{a \in A'} a = t$. Definimos $S_1 = A' \cup \{J\}$. La suma de este subconjunto es:
     $$\text{Sum}(S_1) = \text{Sum}(A') + J = t + (2\Sigma - t) = 2\Sigma$$
     Como $\text{Sum}(S_1) = \frac{1}{2}\text{Sum}(S)$, el conjunto S admite una partición.

- **($\Longleftarrow$):** Supongamos que S admite una partición S_1, S_2 tal que Sum$(S_1) =$ Sum$(S_2) = 2\Sigma$. Por el *Lema de Separación*, sabemos que J y K están en conjuntos distintos. Sin pérdida de generalidad, supongamos que $J \in S_1$ y $K \in S_2$.

  Sea $A_{sub} = S_1 \cap A$ (el subconjunto de elementos originales de A que acompañan a J en S_1). Dado que $S_1 = A_{sub} \cup \{J\}$, su suma es:

  $$\text{Sum}(A_{sub}) + J = 2\Sigma$$

  Sustituyendo el valor de J:

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

  $$\text{Sum}(A_{sub}) = t$$

  Como $A_{sub} \subseteq A$, hemos encontrado un subconjunto de los elementos originales que suma exactamente t.

Conclusión: La instancia de Subset-Sum tiene solución si y solo si la instancia construida de Partición la tiene. La construcción requiere calcular Σ y añadir dos elementos, lo cual se realiza en tiempo $O(n)$. Por tanto, el problema de la Partición es NP y NP-hard por lo que queda demostrado que es NP-Completo.

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

2.2.1. Demostración de que NTD 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, NTD es NP.

2.2.2. Demostración que NTD es NP-hard

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

Reducción Partición $\leq_p$ NTD Se tomará una instancia arbitraria de Partición con el conjunto S. Sea $W_{total} = \Sigma_{i=1}^{n} x_i$. La reducción verifica en tiempo $O(1)$ si W_{total} es impar. De ser así, se genera una instancia trivial de NTD sin solución factible. Si es par, se construye la instancia de NTD:

1. **Artículos:** Para cada $x_i \in S$, se crea un artículo con $w_i = x_i, v_i = x_i$.

2. **Parámetros:** $m = 2$, $K = 0$, y Capacidad $C = W_{total}/2$.

Demostración de Equivalencia:

Si Partición tiene solución: Existen S_1 y S_2 tales que $\Sigma S_1 = \Sigma S_2 = W_{total}/2$. 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, NTD devuelve SÍ.

Si NTD 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.

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

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

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

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 NTD (específicamente cuando $m = 2, w_i = v_i$ y $K = 0$). Dado que el caso particular es NP Y NP-hard por lo que es NP-Completo, el caso general es al menos igual de difícil.

Por lo tanto, el Problema del Transporte Discreto (NTD) es NP-Completo.

3. Diseño de Soluciones Algorítmicas

Dado que el Problema del Transporte Discreto (NTD) 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.

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

- **Subproblema:** Determinar la asignación óptima para el conjunto de artículos restantes $\{i, \ldots, N-1\}$, 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:

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

  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:

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

- **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

```python
def solve_discrete_transport(articulos, mulas_capacidades):
    num_articulos = len(articulos)
    num_mulas = len(mulas_capacidades)
    memo = {}

    def solve(idx, estados_mulas):
        estado_actual = (idx, estados_mulas)
        if estado_actual in memo:
            return memo[estado_actual]

        if idx == num_articulos:
            valores = [m[0] for m in estados_mulas]
            return max(valores) - min(valores), estados_mulas
```

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

        for i in range(num_mulas):
            v_curr, p_curr = estados_mulas[i]
            if p_curr + peso_art <= mulas_capacidades[i]:
                nuevas_mulas = list(estados_mulas)
                nuevas_mulas[i] = (v_curr + valor_art, p_curr +
peso_art)
                dif, config = solve(idx + 1, tuple(nuevas_mulas))
                if dif < mejor_dif:
                    mejor_dif = dif
                    mejor_config = config

        memo[estado_actual] = (mejor_dif, mejor_config)
        return mejor_dif, mejor_config

    estado_inicial_mulas = tuple((0, 0) for _ in range(num_mulas))
    return solve(0, estado_inicial_mulas)
```

Listing 1: Implementación de Programación Dinámica (Exacto)

3.2. Solución Eficiente Basada en Heurísticas

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 preprocesamiento 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.

```
def efficient_solution(articulos, mulas_capacidades):
    # 1. Kernelizacion (Pre-procesamiento)
    cap_maxima = max(mulas_capacidades)
    items_procesables = [a for a in articulos if a[1] <= cap_maxima]
    no_asignados = [a for a in articulos if a[1] > cap_maxima]

    # 2. Greedy LPT Adaptado
    # Ordenar por valor descendente
    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]

    for item in items:
        # Priorizar mulas con menor valor acumulado (Balanceo)
        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. Busqueda Local Reforzada (Movimientos e Intercambios)
    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']

        # Intento de Movimiento Simple (Transferencia)
        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]
                    mulas[pobre]['valor'] += item_r[0]; mulas[pobre]['peso'] += item_r[1]
                    mejorado = True; break
        if mejorado: continue

        # Intento de Intercambio (Swap entre dos articulos)
        for i, item_r in enumerate(mulas[rica]['items']):
            for j, item_p in enumerate(mulas[pobre]['items']):
                p_rica_t = mulas[rica]['peso'] - item_r[1] + item_p[1]
                p_pobre_t = mulas[pobre]['peso'] - item_p[1] + item_r[1]

                if (p_rica_t <= mulas[rica]['capacidad'] and
                    p_pobre_t <= 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

    res = max(m['valor'] for m in mulas) - min(m['valor'] for m in mulas)
    return res, mulas, no_asignados
```

Listing 2: Solucion Eficiente Completa (Kernel + Greedy + B. Local Reforzada)

4. Análisis Experimental de Resultados

Para validar el desempeño de los algoritmos propuestos, se diseñó e implementó un marco de experimentación exhaustivo. El objetivo de esta fase es comparar la *calidad* de las soluciones (qué tan cerca están del óptimo) y la *eficiencia* temporal (cuánto tardan en ejecutarse) bajo diferentes escenarios.

4.1. Metodología de Pruebas

Se utilizó un generador para crear un conjunto de más de 100 instancias de prueba aleatorias, variando el número de artículos (N) entre 5 y 22 para las comparativas directas, y hasta $N = 45$ para las pruebas de carga máxima. Se clasificaron las instancias en cuatro tipos para evaluar el comportamiento en los límites del problema:

- **Normal:** Pesos y valores distribuidos uniformemente. Representa un escenario estándar de operación.
- **Capacidad Limitada (Tight):** Artículos muy pesados (30-60 % de la capacidad de la mula). Pone a prueba la capacidad de aprovechamiento de espacio.
- **Artículos Pequeños (Tiny):** Muchos artículos de poco tamaño y poco valor. Este escenario suele ser el más difícil para la Programación Dinámica (PD) debido a la profundidad que alcanza el árbol de recursión.
- **Casos Complejos (Greedy Trap):** Instancias diseñadas específicamente para dificultar la tarea a los algoritmos voraces, colocando valores altos al principio que desequilibran la carga final.

4.2. Análisis de Escalamiento y Tiempo de Ejecución

La Figura muestra la comparación de tiempos de ejecución en escala logarítmica. Se observa una diferencia notable entre ambos enfoques:

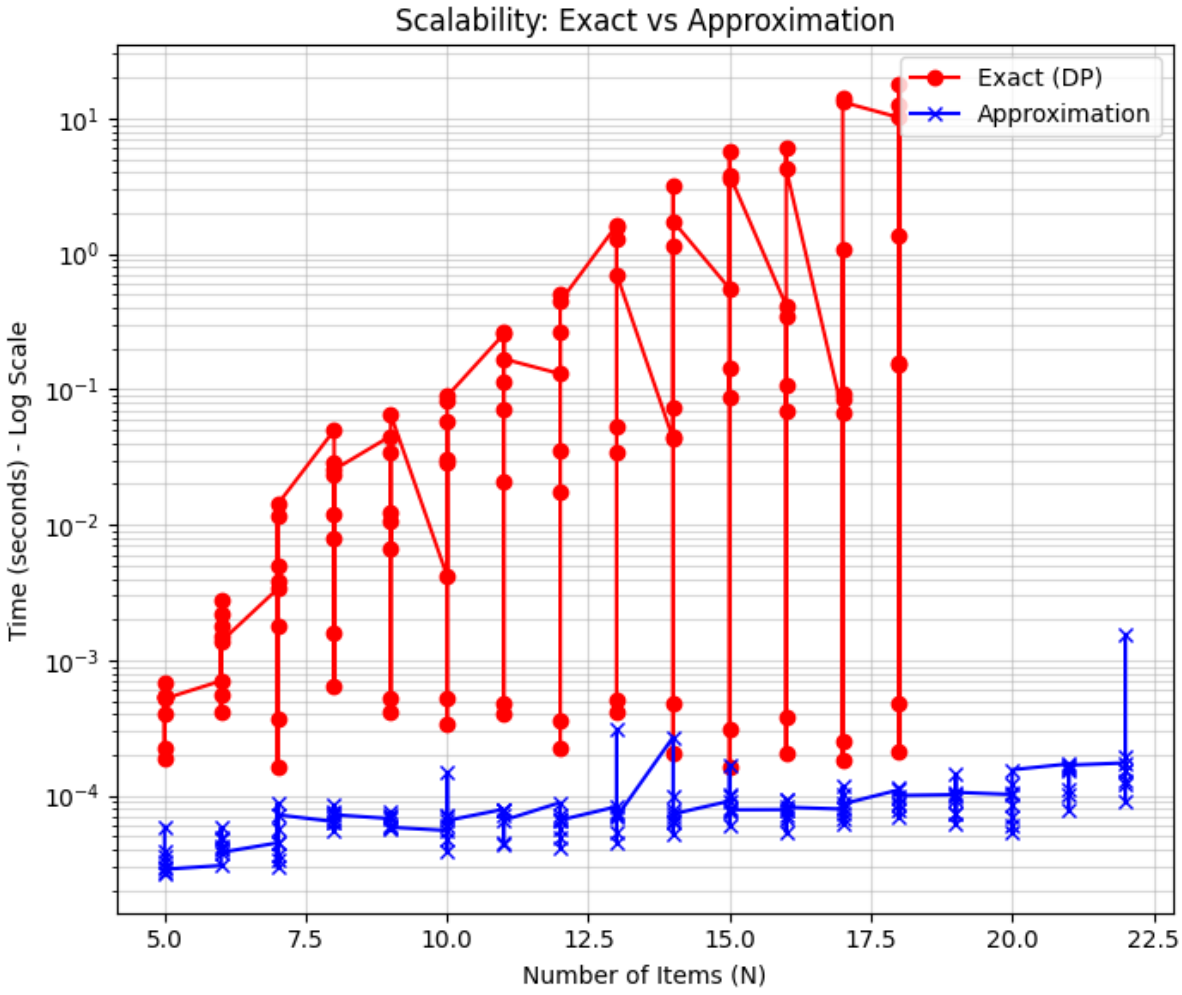

Figura 1: Comparativa de Escalabilidad

- **Algoritmo Exacto (PD):** Muestra una curva de crecimiento muy acelerada. Aunque para $N < 10$ responde en milisegundos, el tiempo aumenta drásticamente de forma impredecible dependiendo de la dificultad de la instancia.
- **Algoritmo Eficiente (Aproximación):** Mantiene un tiempo de ejecución prácticamente constante y lineal (cercano a 10^{-4} segundos) incluso cuando N aumenta.

Una observación importante es que en las instancias con *Artículos Pequeños*. Mientras que la aproximación las resolvió instantáneamente, el algoritmo exacto sufrió retardos significativos (ej. 14.1 segundos para $N = 17$) debido a que los artículos pequeños dificultan el descarte de opciones, obligando al algoritmo a explorar casi todas las combinaciones posibles.

4.3. Evaluación de la Calidad (Diferencia respecto al Óptimo)

Se analizó el porcentaje de diferencia de la solución aproximada respecto a la óptima como se observa en la siguiente figura.

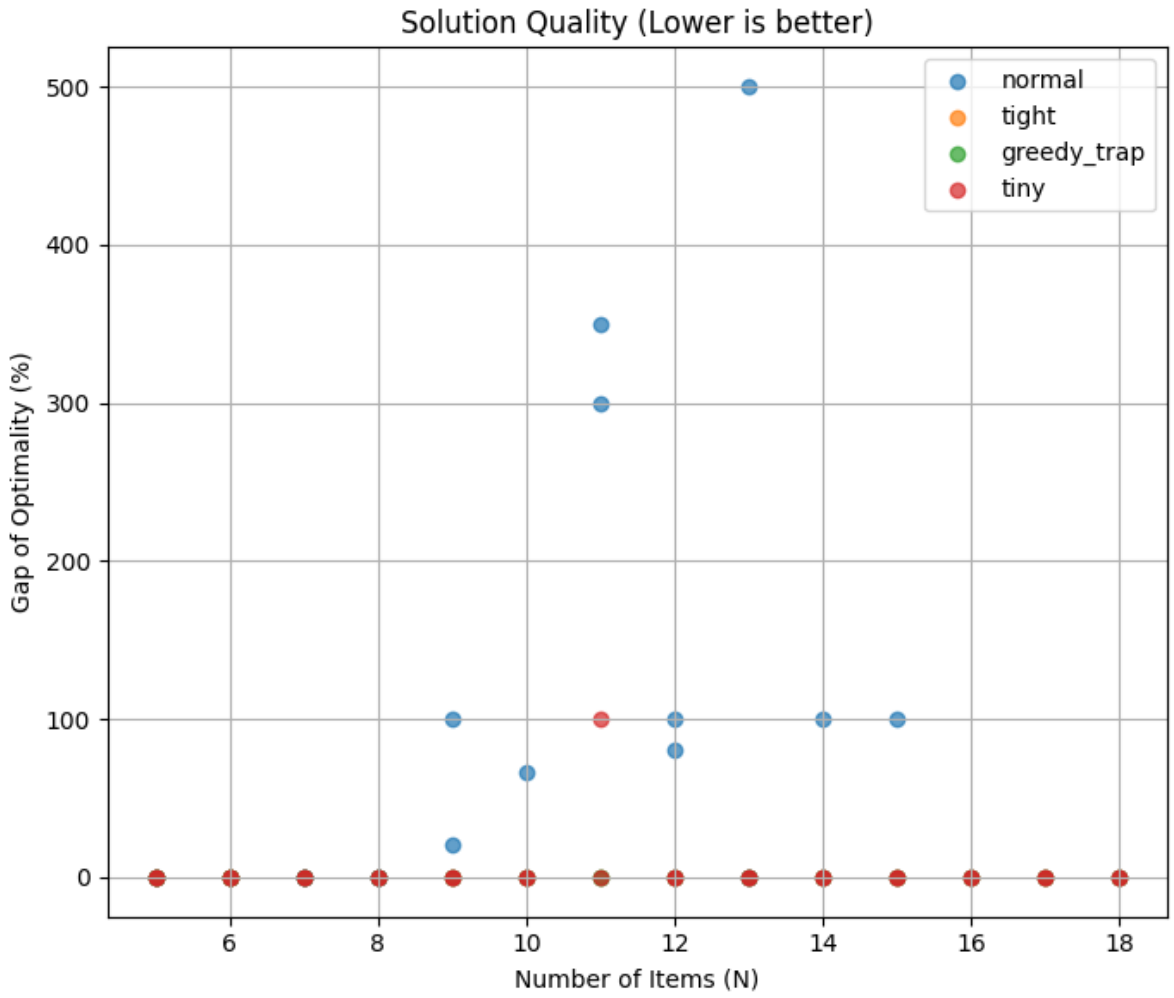

Figura 2: Calidad de las Soluciones

1. **Eficacia en Casos Complejos:** Sorprendentemente, la heurística propuesta logró igualar al óptimo en la mayoría de las instancias de *Casos Complejos* y *Artículos Pequeños*. Esto sugiere que la fase de mejora local es altamente efectiva para corregir las decisiones iniciales.

2. **Robustez en Saturación:** En casos donde las mulas se llenan rápidamente, el algoritmo exacto a menudo indicó que el problema no tenía solución al no poder encajar todos los artículos obligatorios. En contraste, el algoritmo eficiente demostró mayor flexibilidad: en lugar de fallar, descartó inteligentemente los artículos menos valiosos para entregar una solución válida con la carga máxima posible.

3. **Variabilidad:** En instancias normales, se observaron diferencias ocasionales mayores. Esto ocurre en escenarios matemáticamente sensibles donde existe una combinación perfecta difícil de encontrar, aunque en promedio la solución se mantuvo adecuada y muy cercana a la ideal.

4.4. Límite Computacional de la Búsqueda Exhaustiva

Mediante una prueba de carga incremental con un límite de tiempo de 60 segundos, se determinó el límite operativo del enfoque exacto.

- **Zona Estable ($N \leq 30$):** El algoritmo resuelve la mayoría de casos en menos de 5 segundos.
- **Zona Inestable ($30 < N < 40$):** Los tiempos oscilan drásticamente. Se registraron casos de $N = 33$ tomando 47 segundos, seguidos de $N = 37$ tomando 3 segundos, evidenciando la dependencia de la estructura de los datos.
- **Punto de Quiebre ($N \approx 43$):** Al superar los 43 artículos con 3 mulas, el algoritmo consistentemente superó el minuto de ejecución o agotó la memoria disponible.

4.5. Conclusiones Finales

Los resultados experimentales confirman las predicciones teóricas de complejidad. La siguiente tabla resume la aplicabilidad recomendada para cada algoritmo en el contexto del negocio:

Característica	Algoritmo Exacto (DP)	Solución Eficiente
Precisión	Óptima garantizada.	Alta en promedio; excelente en casos con muchos artículos pequeños.
Velocidad	Lenta (Exponencial). Inviable para $N > 40$.	Extremadamente rápida (Tiempo Real).
Robustez	Se detiene si la instancia es imposible de resolver perfectamente.	Se adapta descartando excedentes para mantener la operación.

Cuadro 1: Resumen comparativo de los enfoques implementados.