Teoría de Algoritmos
(TB024) Curso Buchwald - Genender

Trabajo Práctico 3
Problemas NP-Completos para la defensa de la Tribu del Agua

16 de noviembre de 2025

Alen Davies
107.084

Hugo Huzan
67.910

Joaquin Vedoya
110.282

Franco Altieri Lamas
105.400

Trabajo Práctico 3:
Problemas NP-Completos para la defensa de la Tribu del Agua

1. Introducción

Es el año 95 DG. La Nación del Fuego sigue su ataque, esta vez hacia la Tribu del Agua, luego de una humillante derrota a manos del Reino de la Tierra, gracias a nuestra ayuda. La tribu debe defenderse del ataque.

El maestro Pakku ha recomendado hacer lo siguiente: Separar a todos los Maestros Agua en k grupos $(S_1, S_2, \cdots, S_k)$. Primero atacará el primer grupo. A medida que el primer grupo se vaya cansando entrará el segundo grupo. Luego entrará el tercero, y de esta manera se busca generar un ataque constante, que sumado a la ventaja del agua por sobre el fuego, buscará lograr la victoria.

En función de esto, lo más conveniente es que los grupos estén parejos para que, justamente, ese ataque se mantenga constante.

Conocemos la fuerza/maestría/habilidad de cada uno de los maestros agua, la cual podemos cuantificar diciendo que para el maestro i ese valor es x_i, y tenemos todos los valores $x_1, x_2, \cdots, x_n$ (todos valores positivos).

Para que los grupos estén parejos, lo que buscaremos es minimizar la adición de los cuadrados de las sumas de las fuerzas de los grupos. Es decir:

$$\text{mín} \sum_{i=1}^{k} \left(\sum_{x_j \in S_i} x_j \right)^2$$

El Maestro Pakku nos dice que esta es una tarea difícil, pero que con tiempo y paciencia podemos obtener el resultado ideal.

Resolución

2. El problema es NP

Se demostrará que el problema pertenece a la clase de complejidad **NP**, presentando un *certificador eficiente* capaz de verificar un *certificado* en tiempo polinomial.

Certificado

Un vector $C = (S_1, \ldots, S_k)$, donde cada S_j es un subconjunto de índices que indica a qué grupo pertenece cada elemento x_i.

Algoritmo del certificador

El certificador debe verificar las siguientes condiciones:

- Que la cantidad de subconjuntos S_j sea exactamente k.
- Que cada elemento x_i pertenezca a algún subconjunto S_j.
- Que ningún elemento x_i aparezca en más de un subconjunto.
- Que la suma de los cuadrados de las sumas de los elementos de cada S_j sea menor o igual a B.

La implementación del certificador se encuentra en `certificador.py`

```python
def es_particion_valida(maestros, k, B, particion):
    if len(particion) != k:
        return False

    vistos = set()
    for grupo in particion:
        for m in grupo:
            if m in vistos:
                return False
            vistos.add(m)

    if len(vistos) != len(maestros):
        return False

    suma_total = 0
    for grupo in particion:
        suma_grupo = 0
        for maestro in grupo:
            suma_grupo += maestros[maestro]
        suma_total += suma_grupo**2

    if suma_total > B:
        return False

    return True
```

Justificación de la complejidad

Sean:

- n: la cantidad de elementos (maestros) a agrupar,
- k: la cantidad de grupos permitidos,
- B: el valor máximo permitido para la suma de los cuadrados.

El certificador realiza las siguientes verificaciones:

1. Que la cantidad de grupos sea exactamente k.
2. Que ningún elemento se repita en más de un grupo.
3. Que todos los elementos estén asignados a algún grupo.
4. Que la suma total de los cuadrados de las sumas de cada grupo se calcule correctamente.
5. Que el resultado obtenido se compare con el valor límite B.

El análisis de complejidad de cada paso es el siguiente:

Paso	**Descripción**	**Complejidad**
1	Comparar la cantidad de grupos con k	$O(1)$
2	Recorrer todos los elementos para verificar repeticiones	$O(n)$
3	Verificar que no falte ningún elemento	$O(1)$
4	Calcular la suma total de los cuadrados	$O(n)$
5	Comparar el resultado con B	$O(1)$

Por lo tanto, la complejidad total del certificador es:

$$O(1) + O(n) + O(1) + O(n) + O(1) = O(n)$$

Es decir, el certificador se ejecuta en tiempo lineal con respecto a la cantidad de elementos n. Dado que la verificación de una solución candidata puede realizarse en tiempo polinomial respecto del tamaño de la entrada, el **Problema de la Tribu del Agua pertenece a la clase NP**.

3. El problema es NP-Completo

Habiendo demostrado que el Problema de la Tribu del Agua pertenece a NP, vamos a probar que es NP-Completo mediante una reducción desde el problema Partition.

Primero demostramos que Partition es NP-Completo, usando una reducción desde Subset Sum (que sabemos que es NP-Completo).

Problema de decisión Partition

Dado un conjunto de enteros positivos

$$A = \{a_1, a_2, \ldots, a_n\},$$

queremos saber si es posible dividir ese conjunto en dos subconjuntos disjuntos y que la suma de los elementos de cada subconjunto sea igual.

En otras palabras:

¿Existe una forma de separar los elementos en dos grupos de manera que ambas sumas sean iguales?

El problema Partition es NP-Completo

Partition pertenece a NP

Para verificar una solución, podrían darnos dos subconjuntos, A' y su complemento $A \setminus A'$.

Podemos chequear en tiempo polinomial:

- Que los dos subconjuntos no comparten elementos y juntos contienen a todos.
- Que la suma de los elementos del primer subconjunto es igual a la del segundo.

Hacer estas verificaciones sólo requiere recorrer los elementos y sumar, lo cual lleva tiempo lineal. Por eso, Partition pertenece a NP.

Reducción desde Subset Sum a Partition

Recordemos el problema Subset Sum:

- **Instancia**: un conjunto de enteros positivos $A = \{a_1, \ldots, a_n\}$ y un entero objetivo T.
- **Pregunta**: ¿existe un subconjunto $A' \subseteq A$ tal que

$$\sum_{a_i \in A'} a_i = T\,?$$

Sabemos que Subset Sum es NP-Completo.

Dada una instancia (A, T) de Subset Sum, construimos una instancia de Partition.

Normalización de la instancia

Sea

$$S = \sum_{i=1}^{n} a_i.$$

Podemos tratar algunos casos en tiempo polinomial:

- Si $T > S$, claramente no puede haber subconjunto que sume T. Devolvemos una instancia fija de Partition sin solución.
- Si $T = 0$ o $T = S$, la respuesta de Subset Sum es trivialmente *sí* y podemos devolver una instancia fija de Partition con solución.
- Si $T > S/2$, observamos que existe un subconjunto que suma T si y sólo si existe un subconjunto que suma $S - T$ (su complemento). En ese caso, reemplazamos T por $T' = S - T$ y obtenemos una instancia equivalente.

Luego de este preprocesamiento, podemos suponer que

$$0 < T < \frac{S}{2},$$

lo que implica en particular que $S - 2T > 0$.

Construcción de la nueva instancia

Partimos entonces de una instancia *normalizada* de Subset Sum: un conjunto de números $A = \{a_1, \ldots, a_n\}$ y un valor objetivo T con $0 < T < S/2$.

Transformamos esta instancia en una del problema Partition agregando un único número nuevo al conjunto. Una partición válida del nuevo conjunto A' debe dar una solución del Subset Sum original. Por eso, el número extra que agregamos no puede ser cualquier valor, sino uno que garantice que una partición correcta sólo exista si había un subconjunto que sumaba exactamente T.

En Subset Sum buscamos un subconjunto que sume T. En Partition queremos dividir en dos grupos de igual suma.

Para relacionar ambos problemas, necesitamos que:

- uno de los lados de la partición represente al subconjunto que suma T,
- y el otro lado represente al complemento que suma $S - T$.

Pero en Partition ambas mitades deben sumar lo mismo. Entonces necesitamos modificar el conjunto para que:

$$T + (\text{algo}) = S - T.$$

Ese “algo” es el número que debemos agregar.

Definir el número a agregar

A partir de la ecuación anterior:

$$\text{algo} = S - 2T,$$

definimos ese número como:

$$b = S - 2T.$$

Como $0 < T < S/2$, tenemos $S - 2T > 0$, luego b es un entero positivo, consistente con la definición de Partition.

Agregamos b para ajustar la suma del subconjunto U y llevarla al valor necesario para que sea una mitad válida.

Definir el nuevo conjunto A'

$$A' = A \cup \{b\}.$$

Agregamos sólo un número para no alterar demasiado la estructura del conjunto original. Esta construcción es claramente polinomial en el tamaño de la instancia (sólo sumamos los elementos y agregamos un valor).

Calcular la nueva suma total S'

La suma de A' es:

$$S' = S + b.$$

Al reemplazar b, obtenemos:

$$S' = S + (S - 2T) = 2(S - T).$$

Esto confirma que la mitad de S' es:

$$\frac{S'}{2} = S - T.$$

Queremos que si existe un subconjunto U con suma T, entonces:

- el complemento $A \setminus U$ suma $S - T$,
- y $U \cup \{b\}$ también suma $S - T$.

De esta manera, la partición construida a partir de una solución de Subset Sum coincide con dos mitades iguales en Partition.

La nueva pregunta de Partition

La pregunta para la instancia transformada es:

$$\text{¿Se puede dividir } A' \text{ en dos partes que sumen } \frac{S'}{2} = S - T \text{ cada una?}$$

Demostración de la reducción

Ida $\Rightarrow$

Si Subset Sum tiene solución, entonces Partition tiene solución.

Si hay solución de Subset Sum, significa que existe un subconjunto $U \subseteq A$ que suma exactamente T:

$$\sum_{a_i \in U} a_i = T.$$

En la instancia de Partition ya construida, la mitad deseada es $S - T$. El complemento $A \setminus U$ ya suma ese valor, así que es directamente uno de los dos grupos de la partición:

$$S_1 = A \setminus U.$$

Para formar el otro grupo, tomamos el subconjunto U y le agregamos el número especial b que incorporamos en la construcción. Ese número fue elegido de manera tal que:

$$T + b = T + (S - 2T) = S - T.$$

Entonces definimos:

$$S_2 = U \cup \{b\}.$$

Así, los dos grupos S_1 y S_2 tienen la misma suma $S - T = S'/2$, por lo que (S_1, S_2) es una partición válida de A'.

Entonces, toda solución del Subset Sum se convierte en una solución válida de Partition.

Vuelta $\Leftarrow$

Si Partition tiene solución, entonces Subset Sum tiene solución.

Si hay solución de Partition significa que se pueden dividir los elementos de A' en dos grupos que suman ambos $S - T = S'/2$.

En esa partición, el número agregado b debe encontrarse en exactamente uno de los dos grupos (porque sólo aparece una vez). Sin pérdida de generalidad, supongamos que $b \in S_2$.

Entonces:

$$\sum_{x \in S_2} x = (S - T),$$

y esta suma puede escribirse como:

$$\sum_{x \in S_2} x = b + \sum_{a_i \in S_2 \cap A} a_i = (S - 2T) + \sum_{a_i \in S_2 \cap A} a_i.$$

Igualando ambas expresiones:

$$(S - 2T) + \sum_{a_i \in S_2 \cap A} a_i = S - T \implies \sum_{a_i \in S_2 \cap A} a_i = T.$$

Es decir, el conjunto

$$U = S_2 \cap A$$

es un subconjunto formado únicamente por elementos de A que suman exactamente T.

Y eso es precisamente una solución de la instancia original de Subset Sum.

Por lo tanto, toda solución de Partition nos permite reconstruir una solución de Subset Sum.

Conclusión sobre Partition

Hemos construido una reducción polinomial desde Subset Sum a Partition y probado que:

$$(A, T) \text{ tiene solución de Subset Sum} \iff A' \text{ tiene solución de Partition.}$$

Como Subset Sum es NP-Completo y Partition pertenece a NP, concluimos que:

Partition es NP-Completo.

Problema de decisión de la Tribu del Agua

Dado un conjunto de n valores positivos $x_1, x_2, \ldots, x_n$ que representan las habilidades de los maestros agua, un número entero k y un valor entero B, el problema de decisión de la Tribu del Agua pregunta:

¿Existe una partición de los n elementos en k grupos disjuntos $S_1, S_2, \ldots, S_k$ tal que

$$\sum_{i=1}^{k} \left(\sum_{x_j \in S_i} x_j \right)^2 \leq B?$$

Cada elemento x_i debe pertenecer a exactamente un grupo.

Reducción desde Partition al Problema de la Tribu del Agua

Ahora reducimos Partition al Problema de la Tribu del Agua para probar que éste es NP-Completo.

Elección de los parámetros de la nueva instancia

Sea una instancia de Partition dada por un conjunto

$$A = \{a_1, a_2, \ldots, a_n\},$$

y definamos

$$S = \sum_{i=1}^{n} a_i.$$

Si S es impar, la respuesta de Partition es *no* (no se puede dividir en dos partes de igual suma entera). En ese caso, podemos devolver una instancia fija del Problema de la Tribu del Agua que no tenga solución. Por lo tanto, podemos suponer que S es par.

- **Valores x_i:** Tomamos los mismos valores que en Partition:

  $$x_i = a_i \quad \text{para todo } i.$$

- **Número de grupos k:** Elegimos $k = 2$ porque Partition pregunta justamente si se pueden separar los elementos en dos subconjuntos de suma igual.

- **Construcción del parámetro B:**

  En el Problema de la Tribu del Agua, la condición a satisfacer es:

  $$\sum_{i=1}^{k} \left(\sum_{x_j \in S_i} x_j \right)^2 \leq B.$$

  Como fijamos $k = 2$, esto se convierte en:

  $$\left(\sum_{x_j \in S_1} x_j \right)^2 + \left(\sum_{x_j \in S_2} x_j \right)^2 \leq B.$$

  Si llamamos x a la suma del primer grupo, entonces la del segundo es $S - x$. La expresión total depende únicamente de x:

$$f(x) = x^2 + (S - x)^2.$$

Expandimos:

$$f(x) = x^2 + S^2 - 2Sx + x^2 = 2x^2 - 2Sx + S^2.$$

Derivamos:

$$f'(x) = 4x - 2S.$$

Igualamos a cero para hallar el punto crítico:

$$4x - 2S = 0 \quad \Longrightarrow \quad x = \frac{S}{2}.$$

La segunda derivada es:

$$f''(x) = 4 > 0,$$

así que el punto crítico es un mínimo local y, como f es una función cuadrática con coeficiente principal positivo, es el mínimo global.

Evaluamos el valor mínimo:

$$f\left(\frac{S}{2}\right) = \left(\frac{S}{2}\right)^2 + \left(S - \frac{S}{2}\right)^2 = \frac{S^2}{4} + \frac{S^2}{4} = \frac{S^2}{2}.$$

Para todo x se cumple

$$f(x) \geq f\left(\frac{S}{2}\right) = \frac{S^2}{2},$$

y la igualdad ocurre sólo cuando $x = \frac{S}{2}$.

Por lo tanto, definimos:

$$B = \frac{S^2}{2}.$$

Éste es el valor mínimo posible de la expresión. Cualquier partición donde la suma de uno de los grupos sea distinta de $\frac{S}{2}$ produce un valor estrictamente mayor que B, por lo que no puede satisfacer la desigualdad.

Con esta construcción, la pregunta del Problema de la Tribu del Agua queda:

$$\text{¿Existe una partición } (S_1, S_2) \text{ tal que } \left(\sum_{x_j \in S_1} x_j\right)^2 + \left(\sum_{x_j \in S_2} x_j\right)^2 \leq B?$$

Demostración de la reducción

Debemos demostrar que:

$$\text{Hay solución de Partition} \iff \text{Hay solución de la Tribu del Agua con } k = 2.$$

Ida $\Rightarrow$
Si hay solución del problema Partition, entonces hay solución del Problema de la Tribu del Agua.

Si existe un subconjunto $A' \subseteq A$ tal que:

$$\sum_{a_i \in A'} a_i = \sum_{a_i \in A \setminus A'} a_i = \frac{S}{2},$$

tomamos la partición:

$$S_1 = A', \qquad S_2 = A \setminus A'.$$

En esta partición, las sumas de los grupos son:

$$\sum_{x_j \in S_1} x_j = \frac{S}{2}, \qquad \sum_{x_j \in S_2} x_j = \frac{S}{2}.$$

La expresión del Problema de la Tribu del Agua con $k = 2$ es:

$$\left(\sum_{x_j \in S_1} x_j\right)^2 + \left(\sum_{x_j \in S_2} x_j\right)^2 = \left(\frac{S}{2}\right)^2 + \left(\frac{S}{2}\right)^2 = \frac{S^2}{2} = B.$$

Por lo tanto, la partición (S_1, S_2) cumple la condición:

$$\left(\sum_{x_j \in S_1} x_j\right)^2 + \left(\sum_{x_j \in S_2} x_j\right)^2 \leq B.$$

Entonces, si existe una solución de Partition, existe una solución para la instancia construida del Problema de la Tribu del Agua con $k = 2$.

Vuelta $\Leftarrow$
Si hay solución del Problema de la Tribu del Agua, entonces hay solución de Partition.

Supongamos que hay solución del Problema de la Tribu del Agua para la instancia construida. Es decir, existe una partición (S_1, S_2) con $k = 2$ tal que:

$$\left(\sum_{x_j \in S_1} x_j\right)^2 + \left(\sum_{x_j \in S_2} x_j\right)^2 \leq B,$$

donde

$$B = \frac{S^2}{2}.$$

Sea

$$x = \sum_{x_j \in S_1} x_j, \qquad \sum_{x_j \in S_2} x_j = S - x.$$

La expresión total es:

$$f(x) = x^2 + (S - x)^2.$$

Sabemos que

$$f(x) = 2x^2 - 2Sx + S^2,$$

y que

$$f'(x) = 4x - 2S, \qquad f''(x) = 4 > 0.$$

Como $f''(x) > 0$, el único punto crítico $x = S/2$ es su mínimo global. Además, ya calculamos que

$$f\left(\frac{S}{2}\right) = \frac{S^2}{2} = B.$$

Para todo $x \neq S/2$ se verifica

$$f(x) > f\left(\frac{S}{2}\right) = B.$$

Pero en nuestra instancia de la Tribu del Agua se cumple

$$f(x) \leq B.$$

La única forma de que una función tome un valor menor o igual a su mínimo global es que esté exactamente en el punto de mínimo. Por lo tanto, necesariamente

$$x = \frac{S}{2} \quad \Longrightarrow \quad S - x = \frac{S}{2}.$$

Es decir:

$$\sum_{x_j \in S_1} x_j = \sum_{x_j \in S_2} x_j = \frac{S}{2}.$$

Por definición de $x_i = a_i$, esto significa que (S_1, S_2) define una partición del conjunto A en dos subconjuntos de igual suma, lo cual es una solución del problema Partition.

Conclusión

Habiendo demostrado que:

- Partition es NP-Completo.
- Existe una reducción polinomial desde Partition al Problema de la Tribu del Agua.
- La instancia de Partition tiene solución si y sólo si la instancia construida del Problema de la Tribu del Agua tiene solución.

y dado que el Problema de la Tribu del Agua pertenece a NP, concluimos que el **Problema de la Tribu del Agua es NP-Completo**.

4. Backtracking

TODO: Completar Intro

Podas Implementadas:

- Poda por Cota Superior: `if sum_cuad_actual >= mejor_valor: Podar`
- Poda de simetría básica para evitar permutaciones de grupos vacíos. `ya_uso_vacio = False ....`

Mejoras sobre el algoritmo backtracking básico:

- Ordenamiento previo de habilidades en forma descendente; tiende a encontrar mejores soluciones temprano y permitir mayor poda.
- Actualización incremental de `suma_cuad_actual`. Evita recalcular toda la suma en cada nodo.

4.1. Código Python

```python
import math


def backtracking(i, habilidades, k, sumas, sum_cuad_actual, particion, mejor_valor, mejor_particion):

    if sum_cuad_actual >= mejor_valor:
        return mejor_valor, mejor_particion

    if i == len(habilidades):
        if sum_cuad_actual < mejor_valor:
            mejor_valor = sum_cuad_actual
            mejor_particion = [list(g) for g in particion]
        return mejor_valor, mejor_particion

    valor, indice = habilidades[i]

    ya_uso_vacio = False  # para evitar algunas permutaciones de los conjuntos
    for g in range(k):
        if sumas[g] == 0:
            if ya_uso_vacio:
                continue
            ya_uso_vacio = True

        suma_anterior = sumas[g]

        incremento = (suma_anterior + valor) ** 2 - (suma_anterior ** 2)

        particion[g].append(indice)
        sumas[g] += valor
        sum_cuad_actual += incremento

        mejor_valor, mejor_particion = backtracking(
            i + 1, habilidades, k, sumas, sum_cuad_actual, particion, mejor_valor, mejor_particion)

        sumas[g] -= valor
        sum_cuad_actual -= incremento
        particion[g].pop()

    return mejor_valor, mejor_particion


def resolver(k, habilidades, ordenar=True):

    tuplas = sorted([(v, i)
```

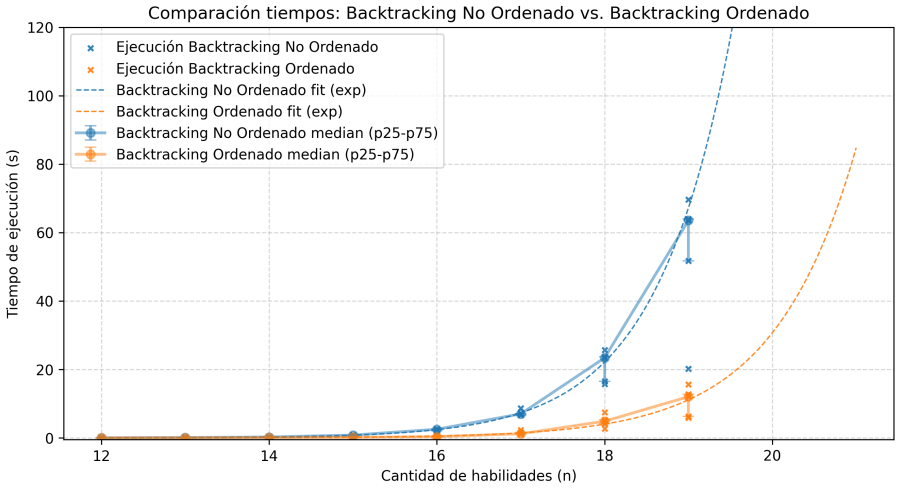
```
                    for i, v in enumerate(habilidades)], reverse=True) if ordenar
    else [(v, i) for i, v in enumerate(habilidades)]

    mejor_valor, mejor_particion = math.inf, None

    sumas = [0] * k
    sum_cuad_actual = 0
    particion = [[] for _ in range(k)]

    return backtracking(0, tuplas, k, sumas, sum_cuad_actual, particion,
    mejor_valor, mejor_particion)
```

4.2. Mediciones

TODO: Agregar contexto

Con o Sin Ordenamiento Previo

Se muestra el efecto de realizar o no el Ordenamiento previo descendente de las habilidades.



Se observa que ordenando los datos, en forma descendente, antes del realizar el backtracking, los tiempos de ejecución se reducen a aproximadamente un octavo.

TODO: Ampliar

5. Programación Lineal

En esta sección se planteará un modelo de Programación Lineal para la resolución de este problema. Primero se presentará la idea original, basada en una función no lineal con variables binarias, y luego se describirá un modelo lineal que aproxima la solución óptima.

Lo que se buscará al resolverlo con programación lineal en este informe es que, al recibir una secuencia de n maestros y un número k, se devuelva la menor diferencia posible entre el grupo de mayor suma y el de menor suma, además de los grupos asignados.

La idea central consiste en introducir una variable binaria y_{ij} que indique si el maestro j pertenece al grupo i:

$$y_{ij} = \begin{cases} 1 & \text{si el maestro } j \text{ pertenece al grupo } i, \\ 0 & \text{en otro caso.} \end{cases}$$

Con esta definición, la función original a minimizar es:

$$\text{mín} \sum_{i=1}^{k} \left(\sum_{j=1}^{n} x_j y_{ij} \right)^2 .$$

Dado que esta expresión no es lineal, se optará por formular un modelo lineal que proporcione una aproximación razonable.

5.1. Variables

- $y_{ij} \in \{0, 1\}$: indica si el maestro j pertenece al grupo i.
- $s_i \in \mathbb{R}$: suma total de fuerzas asignadas al grupo i.
- $Z_{\text{máx}}, Z_{\text{mín}} \in \mathbb{R}$: valor máximo y mínimo, respectivamente, entre todas las sumas s_i.

5.2. Restricciones

Las restricciones del modelo aseguran la correcta asignación de maestros a grupos y la relación entre s_i, $Z_{\text{máx}}$ y $Z_{\text{mín}}$.

- **Asignación de maestros:** Lo principal será limitar a cuántos grupos i puede pertenecer cada maestro j. En este caso se busca que cada maestro esté asignado a un solo grupo, por lo tanto, tendremos una restricción tal que:

$$\sum_{i=1}^{k} y_{ij} = 1 \quad \forall j.$$

  Esto garantiza que ningún maestro quede sin asignar ni aparezca en más de un grupo.

- **Definición de las sumas s_i:** Para cada grupo i, la suma de fuerzas es igual a la suma de los valores x_j de los maestros asignados a él:

$$s_i = \sum_{j=1}^{n} x_j y_{ij} \quad \forall i.$$

  Se sumarán todas las fuerzas de cada maestro j solo si pertenece al grupo i ($y_{ij} = 1$).

- **Relación con $Z_{\text{máx}}$ y $Z_{\text{mín}}$:** Para que $Z_{\text{máx}}$ y $Z_{\text{mín}}$ representen efectivamente el mayor y el menor valor entre los s_i, se imponen:

$$s_i \leq Z_{\text{máx}} \quad \forall i,$$

$$s_i \geq Z_{\text{mín}} \quad \forall i.$$

Se tendría un total de $n + 3k$ inecuaciones.

5.3. Función Objetivo

La aproximación lineal consiste en minimizar la diferencia entre la suma más grande y la suma más pequeña:

$$\text{mín}\left(Z_{\text{máx}} - Z_{\text{mín}}\right).$$

Este criterio es razonable porque minimizar la diferencia entre los grupos tiende a equilibrar sus sumas, lo cual aproxima el objetivo original basado en la minimización de los cuadrados.

5.4. Implementación en Python con PuLP

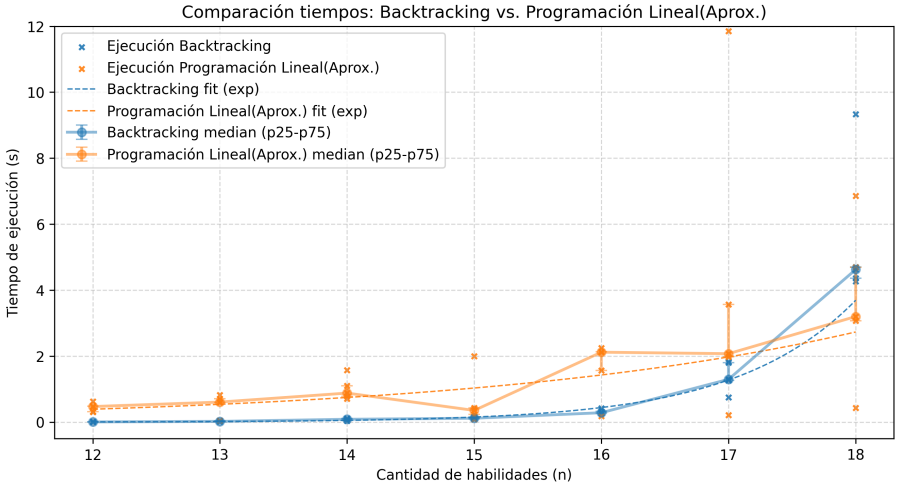
```python
import pulp
from pulp import PULP_CBC_CMD

def obtener_particiones(n, k, y):
    particion = [[] for _ in range(k)]
    for i in range(k):
        for j in range(n):
            if y[i][j].value() == 1:
                particion[i].append(j)

    return particion


def prog_lineal(k, habilidades):
    n = len(habilidades)

    modelo = pulp.LpProblem("Tribu_del_Agua", pulp.LpMinimize)

    # Variables
    y = pulp.LpVariable.dicts("y", (range(k), range(n)), cat="Binary")
    s = pulp.LpVariable.dicts("s", range(k))
    zmax = pulp.LpVariable("Zmax")
    zmin = pulp.LpVariable("Zmin")

    # Restricciones

    # Asignacion de maestros
    for j in range(n):
        modelo += pulp.lpSum(y[i][j] for i in range(k)) == 1

    # Definicion de sumas, zmax y zmin
    for i in range(k):
        modelo += s[i] == pulp.lpSum(habilidades[j] * y[i][j]
                                     for j in range(n))
        modelo += s[i] <= zmax
        modelo += s[i] >= zmin

    modelo += zmax - zmin

    modelo.solve(PULP_CBC_CMD(msg=False))

```

```
    particion = obtener_particiones(n, k, y)
    suma = sum((s[i].value())**2 for i in range(k))

    valor_objetivo = (zmax.value() - zmin.value())

    return suma, particion, valor_objetivo
```

5.5. Mediciones

TODO: Agregar contexto

Tiempos de Ejecución



TODO: Agregar Interpretación

Ratio de la Aproximación

TODO: Agregar contexto

TODO: Agregar Gráfico

TODO: Agregar Interpretación

6. Greedy - Algoritmo de Pakku

Se implementó el algoritmo propuesto por el maestro Pakku.

Para determinar el "grupo con menos habilidad hasta ahora", se consideró la suma de habilidades del grupo, por corresponder el mínimo de esta suma al mínimo del cuadrado de la suma.

6.1. Código Python

```python
def greedy_pakku(k, habilidades):
    ordenadas = sorted([(v, i)
                        for i, v in enumerate(habilidades)], reverse=True)
    sumas = [0]*k
    particion = [[] for _ in range(k)]
    for valor, indice in ordenadas:
        j = min(range(k), key=lambda x: sumas[x])
        particion[j].append(indice)
        sumas[j] += valor
    return sum(s*s for s in sumas), particion
```

6.2. Análisis de Complejidad

Complejidad Temporal

El algoritmo ordena la lista de habilidades de tamaño n, con costo $\mathcal{O}(n \log n)$.

Luego, para cada uno de los n elementos, selecciona el grupo con menor suma actual mediante una búsqueda lineal entre los k grupos, lo que cuesta $\mathcal{O}(k)$ por iteración.

El costo total de esta fase es $\mathcal{O}(nk)$.

Por lo tanto, la complejidad temporal total es

$$T(n, k) = \mathcal{O}(n \log n + nk).$$

Complejidad Espacial

El algoritmo usa las siguientes estructuras:

(i) la lista ordenada: $\mathcal{O}(n)$, (ii) el arreglo de sumas: $\mathcal{O}(k)$, (iii) las particiones: $\mathcal{O}(n)$.

En consecuencia, el uso total de memoria es

$$S(n, k) = \mathcal{O}(n + k).$$

6.3. Calidad de la Aproximación

El objetivo del problema consiste en particionar los n maestros en k grupos $S_1, \ldots, S_k$ de forma tal que, si denotamos por L_j a la suma de habilidades del grupo j, se minimice la función

$$f(S_1, \ldots, S_k) = \sum_{j=1}^{k} L_j^2.$$

Para una instancia I, denotamos por $OPT(I)$ al valor óptimo de esta función y por $A(I)$ al valor obtenido por el algoritmo Greedy de Pakku.

Cota inferior para el óptimo

Sea $S = \sum_{i=1}^{n} x_i$ la suma total de habilidades. Para cualquier partición de los maestros en k grupos, se cumple

$$\sum_{j=1}^{k} L_j = S.$$

Usando que la función x^2 es convexa, obtenemos la siguiente cota inferior válida para *toda* partición:

$$\sum_{j=1}^{k} L_j^2 \;\geq\; k \left(\frac{1}{k} \sum_{j=1}^{k} L_j \right)^2 = k \left(\frac{S}{k} \right)^2 = \frac{S^2}{k}.$$

Por lo tanto, para toda instancia I se cumple

$$OPT(I) \;\geq\; \frac{S^2}{k}.$$

Cota superior para el algoritmo Greedy

Analicemos ahora el comportamiento del algoritmo de Pakku.

- ordena las habilidades de mayor a menor,
- y asigna cada maestro al grupo con menor suma actual.

Sea p la habilidad máxima (es decir, $p = \text{máx}_i\, x_i$). En el momento en que se asigna el maestro de habilidad p, la suma de habilidades ya asignadas es $S - p$, con lo cual la carga promedio de los grupos en ese instante es

$$\frac{S-p}{k}.$$

Como el algoritmo elige siempre el grupo con menor suma, la carga de ese grupo antes de asignar p es, a lo sumo,

$$L_{\text{antes}} \;\leq\; \frac{S-p}{k}.$$

Luego de agregar p, la carga de ese grupo queda acotada por

$$L_{\text{después}} \;\leq\; \frac{S-p}{k} + p = \frac{S}{k} + \left(1 - \frac{1}{k}\right) p.$$

En consecuencia, si denotamos por M_G a la carga máxima en la solución Greedy, tenemos

$$M_G \;\leq\; \frac{S}{k} + \left(1 - \frac{1}{k}\right) p.$$

Por otro lado, como la suma de las cargas es S, podemos acotar el valor de la función objetivo de la solución Greedy como

$$A(I) = \sum_{j=1}^{k} L_j^2 \;\leq\; M_G \sum_{j=1}^{k} L_j = M_G \cdot S \;\leq\; \left(\frac{S}{k} + \left(1 - \frac{1}{k}\right) p \right) S.$$

Factor de aproximación

Combinando la cota inferior del óptimo con la cota superior del algoritmo, se obtiene

$$\frac{A(I)}{OPT(I)} \;\leq\; \frac{\left(\frac{S}{k} + \left(1 - \frac{1}{k}\right) p\right) S}{\frac{S^2}{k}} = 1 + (k-1)\frac{p}{S}.$$

Como $p \leq S$ (la habilidad máxima no puede exceder la suma total), se tiene

$$\frac{p}{S} \leq 1 \quad \Rightarrow \quad \frac{A(I)}{OPT(I)} \;\leq\; 1 + (k-1) \;=\; k.$$

Es decir, el algoritmo de Pakku es un algoritmo de *aproximación k-aproximado*: para cualquier instancia I con k grupos, el valor de la solución Greedy satisface

$$A(I) \;\leq\; k \cdot OPT(I).$$

Además, la cota más fina

$$\frac{A(I)}{OPT(I)} \;\leq\; 1 + (k-1)\frac{p}{S}$$

muestra que, cuando ningún maestro domina fuertemente la suma total (es decir, cuando el cociente p/S es pequeño), el factor de aproximación real es mucho más cercano a 1 que a k. En las mediciones experimentales presentadas más adelante se observa justamente que, para las instancias consideradas, el algoritmo Greedy produce soluciones muy cercanas al óptimo.

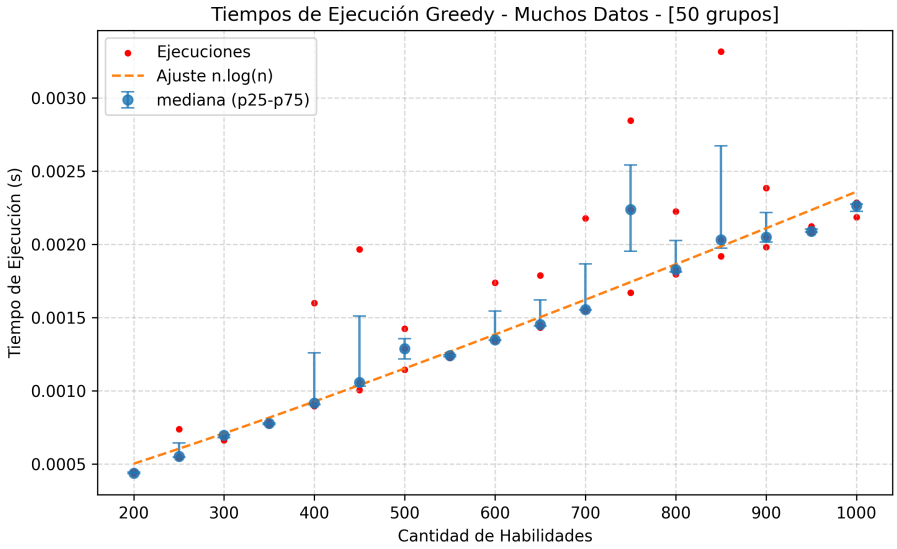
6.4. Mediciones

Tiempos de Ejecución - Datasets Grandes

Para contrastar la estimación teórica de la complejidad temporal con observaciones experimentales, se generaron diversos conjuntos de datos aleatorios de tamaños variables. En total se realizaron mediciones para $17 * 5 = 85$ conjuntos distintos de habilidades, cada uno con entre 200 y 1000 elementos.:

- Se generó un conjunto de datos con 85 sets de tamaño comprendido entre 200 y 1000 elementos (habilidades).
- Cada habilidad con un valor comprendido entre 10 y 1000.
- Se definieron 50 grupos ($k = 50$)
- Para cada set de ese conjunto de datos, se midió el tiempo de procesamiento.

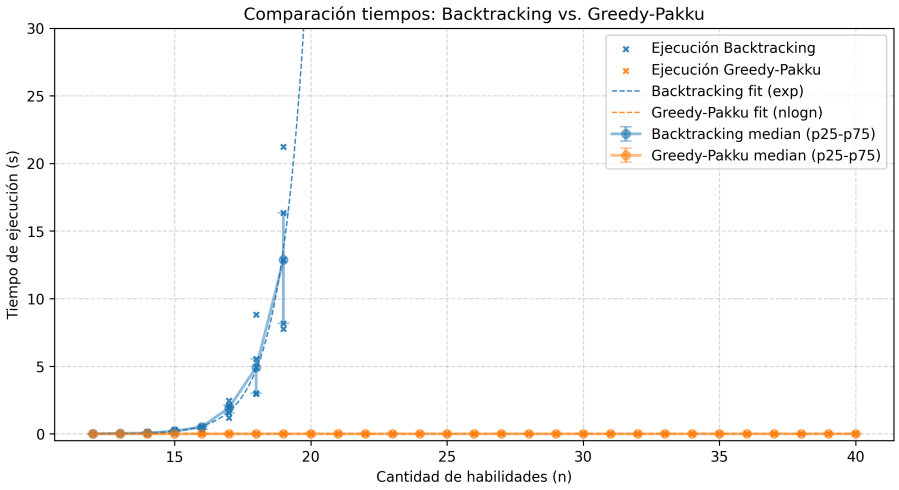
Finalmente, se graficaron los tiempos medidos junto con la curva de ajuste correspondiente a la complejidad teórica estimada.



Se observa que, si bien hay elevada dispersión en los resultados, los valores medidos se ajustan bien al estimado teórico $n.log(n)$.

Comparación Tiempos - Backtraking vs. Greedy-Pakku

TODO: Agregar Intro

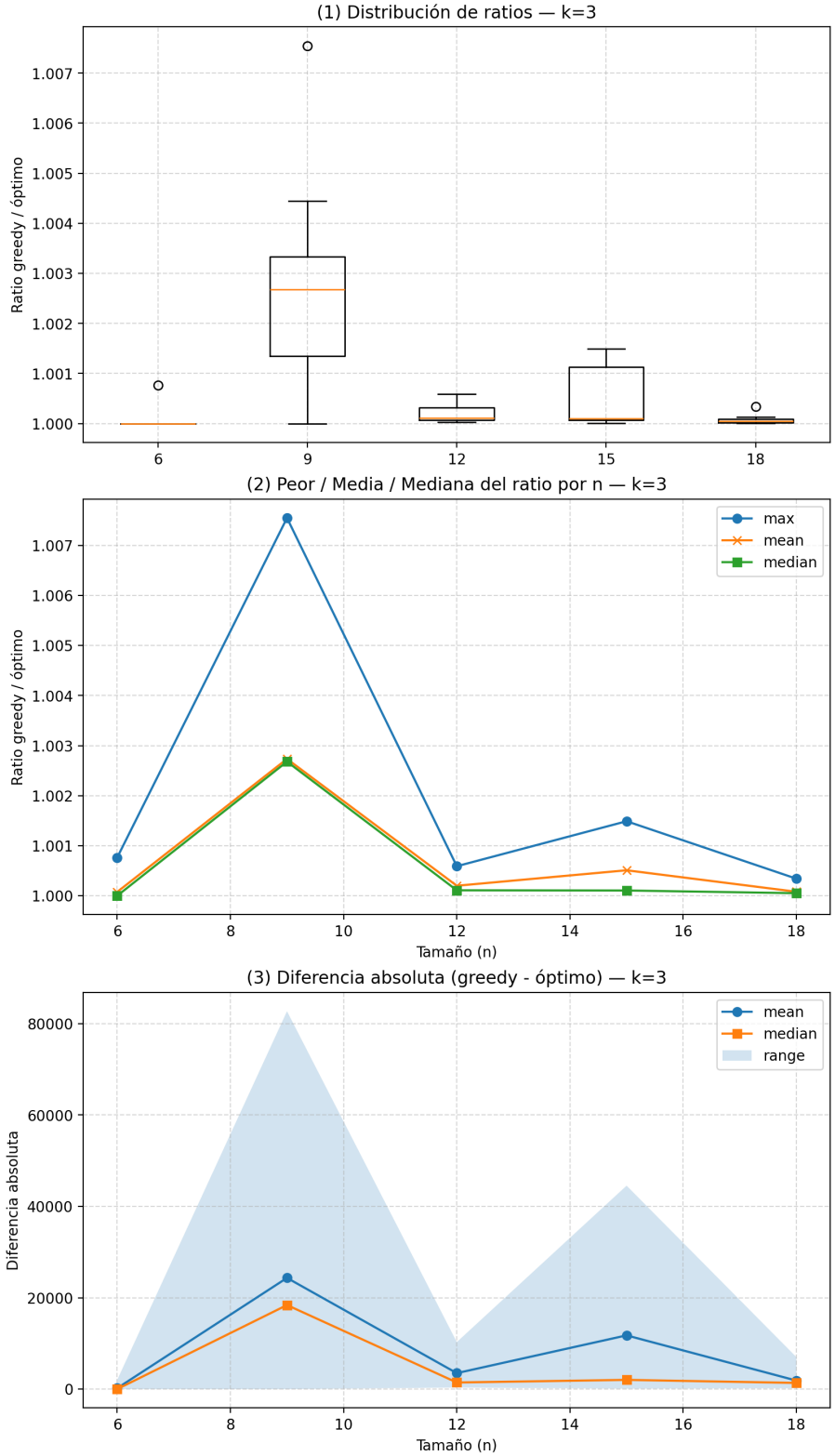


Nota: si bien los tiempos para el algoritmo Greedy parecen constantes, eso es debido al pequeño intervalo considerado y a la gran diferencia de escala con los tiempos de Backtraking. Como se vio en gráfico anterior, la solución Greedy tiene un orden $n.log(n)$

TODO: Ampliar

Ratio de la Aproximación

TODO: Agregar contexto



TODO: EL GRÁFICO ES MUY REDUNDANTE (además de feo)

TODO: Agregar Interpretación

7. Greedy - FFD

Se implementó .....

7.1. Código Python

```python
def greedy_ffd(k, habilidades):
    ordenadas = sorted([(v, i) for i, v in enumerate(habilidades)], reverse=True)

    sumas = [0] * k
    particion = [[] for _ in range(k)]

    for valor, indice in ordenadas:
        mejor_g = None
        mejor_aumento = float('inf')

        for g in range(k):
            aumento = (sumas[g] + valor)**2 - (sumas[g]**2)
            if aumento < mejor_aumento:
                mejor_aumento = aumento
                mejor_g = g

        particion[mejor_g].append(indice)
        sumas[mejor_g] += valor

    return sum(s*s for s in sumas), particion
```

7.2. Análisis de Complejidad

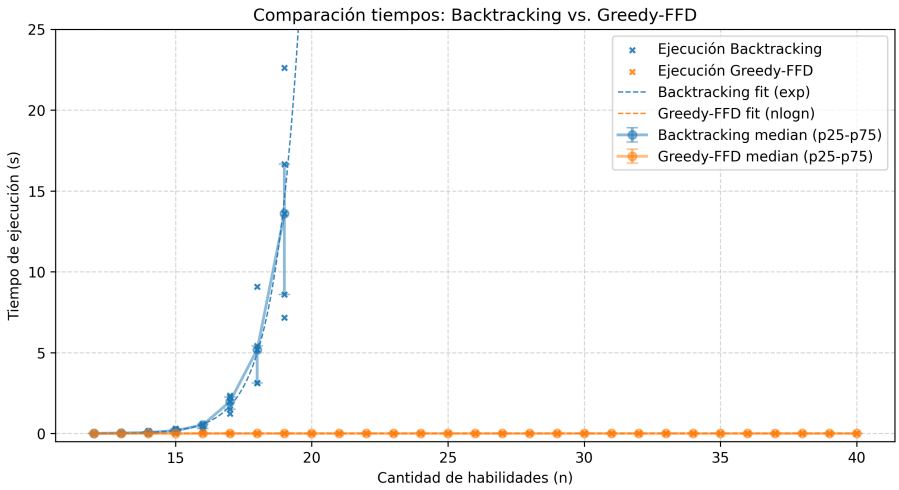
Complejidad Temporal

Complejidad Espacial

7.3. Mediciones

Comparación Tiempos - Backtraking vs. Greedy-Pakku

TODO: Agregar Intro

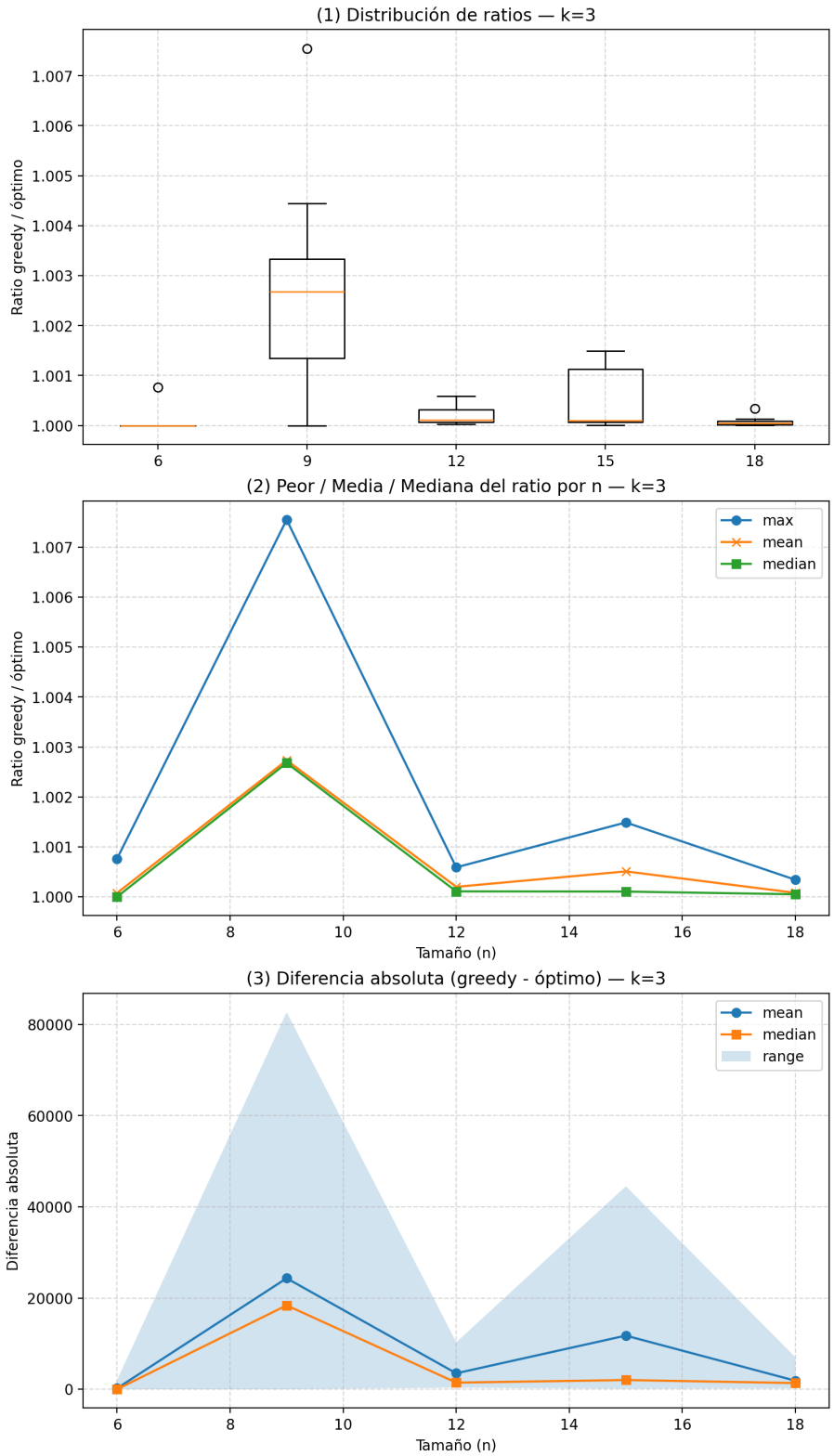


Nota: si bien los tiempos para el algoritmo Greedy parecen constantes, eso es debido al pequeño intervalo considerado y a la gran diferencia de escala con los tiempos de Backtraking. Como se vio en gráfico anterior, la solución Greedy tiene un orden $n.log(n)$

TODO: Ampliar

Ratio de la Aproximación

TODO: Agregar contexto



TODO: EL GRÁFICO ES MUY REDUNDANTE (además de feo)

TODO: Agregar Interpretación

8. Conclusiones