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 pertenece a NP, vamos a demostrar que el Problema de la Tribu del Agua es NP-Completo mediante una reducción desde el problema Subset Sum, que es un problema 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 Subset Sum al Problema de la Tribu del Agua

Una instancia de Subset Sum está dada por un conjunto de números positivos $A = \{a_1, a_2, \ldots, a_n\}$ y una suma objetivo T.

El problema de decisión pregunta si existe un subconjunto $A' \subseteq A$ cuyos elementos sumen exactamente T.

Elección de los parámetros de la nueva instancia

- **Valores x_i:** Tomamos los mismos valores que en Subset Sum:

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

- **Número de grupos k:** Elegimos $k = 2$ porque Subset Sum consiste justamente en separar los elementos en dos subconjuntos: uno que suma T y el otro que suma lo que queda.
- **La suma total S:** Definimos

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

  Si un subconjunto suma T, su complemento suma necesariamente $S - T$.

- **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 depende únicamente de x:

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

Para garantizar que una partición sea aceptada únicamente cuando uno de los grupos suma exactamente T, definimos:

$$B = T^2 + (S - T)^2.$$

Este valor es el mínimo posible de toda la expresión. Cualquier partición donde la suma de un grupo sea distinta de T produce un valor estrictamente mayor, por lo que no puede satisfacer la desigualdad.

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

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

Demostración de la reducción

Debemos demostrar que:

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

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

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

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

En nuestra reducción fijamos $k = 2$. Esto implica que cualquier solución del Problema de la Tribu del Agua debe consistir en dividir los elementos en dos grupos. Por lo tanto, podemos usar el subconjunto solución de Subset Sum para formar uno de los grupos, y su complemento para formar el otro:

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

En esta partición, las sumas de los grupos son:

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

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

Dado que el Problema de la Tribu del Agua con $k = 2$ evalúa:

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

sustituimos las sumas obtenidas:

$$T^2 + (S - T)^2.$$

Y en la construcción de la instancia definimos:

$$B = T^2 + (S - T)^2.$$

Esto significa que esta partición satisface la desigualdad:

$$(\sum S_1)^2 + (\sum S_2)^2 = B.$$

Por lo tanto, la partición (S_1, S_2) cumple la condición del Problema de la Tribu del Agua:

$$(\sum S_1)^2 + (\sum S_2)^2 \leq B.$$

Entonces, si existe una solución de Subset Sum, 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 Subset Sum.

Si hay solución del Problema de la Tribu del Agua, entonces existe una partición (S_1, S_2) con $k = 2$ tal que:

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

recordemos que:

$$B = T^2 + (S - T)^2.$$

Al partir los elementos en dos grupos, si uno suma x, el otro suma $S - x$. La expresión total depende sólo de x:

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

Elegimos B justamente como el valor de esta expresión cuando $x = T$, que es el valor mínimo posible. Si la partición encontrada cumple la desigualdad, entonces necesariamente debe tener:

$$\sum_{x_j \in S_1} x_j = T.$$

Es decir, uno de los grupos suma exactamente T, lo cual constituye una solución de Subset Sum.

Conclusión

Habiendo demostrado que la instancia de Subset Sum tiene solución si y sólo si la instancia construida del Problema de la Tribu del Agua tiene solución, y dado que la reducción es polinomial, 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)
```

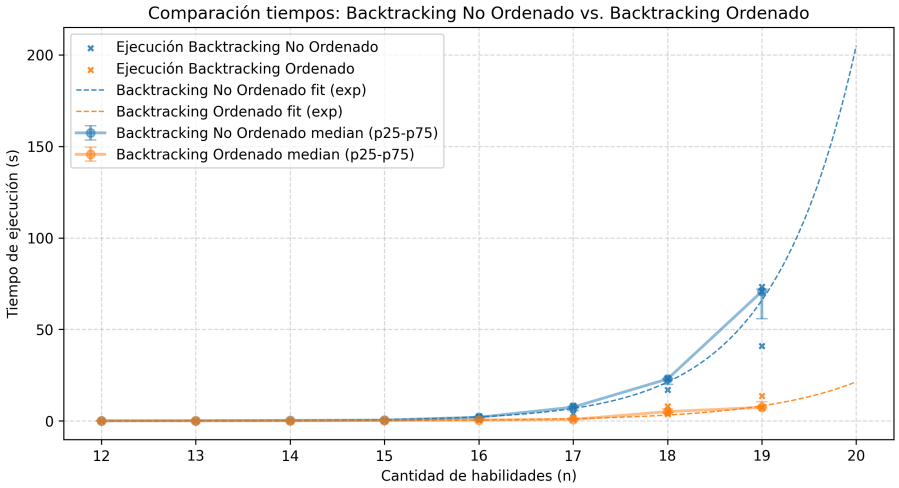
```
45                      for i, v in enumerate(habilidades)], reverse=True) if ordenar
    else [(v, i) for i, v in enumerate(habilidades)]
46
47      mejor_valor, mejor_particion = math.inf, None
48
49      sumas = [0] * k
50      sum_cuad_actual = 0
51      particion = [[] for _ in range(k)]
52
53      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. 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.

5.1. 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
```

5.2. 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

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

5.3. Calidad de la Aproximación

TBD

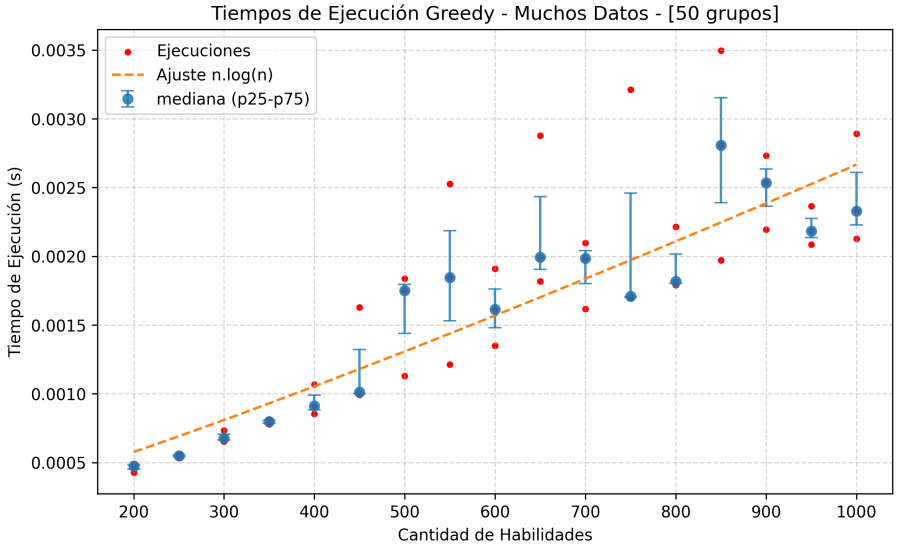
5.4. Mediciones

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 $17 * 3 = 51$ conjuntos distintos de habilidades, cada uno con entre 200 y 1000 elementos.:

- Se generó un conjunto de datos con sets de tamaño variable con 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.

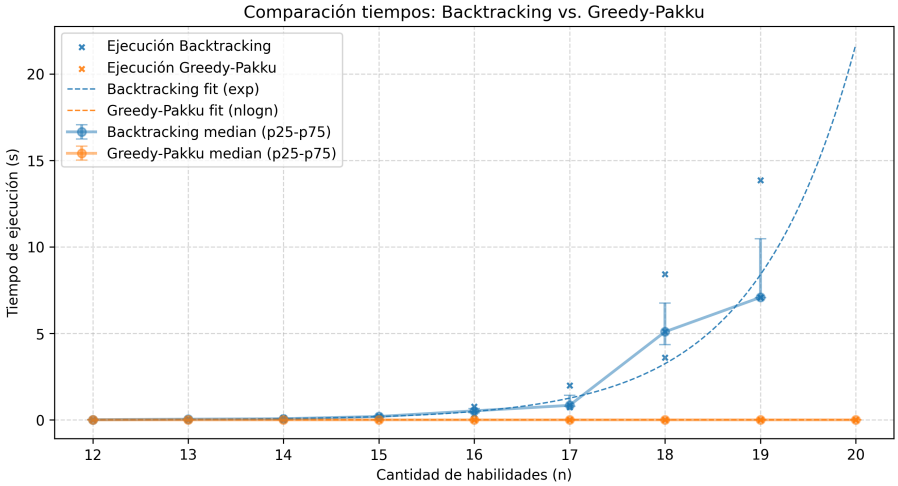
Finalmente, se graficaron los tiempos medidos junto con la curva de ajuste correspondiente a la complejidad teórica estimada.



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

Comparación Tiempos - Backtraking vs. Greedy-Pakku

TODO: Agregar Intro



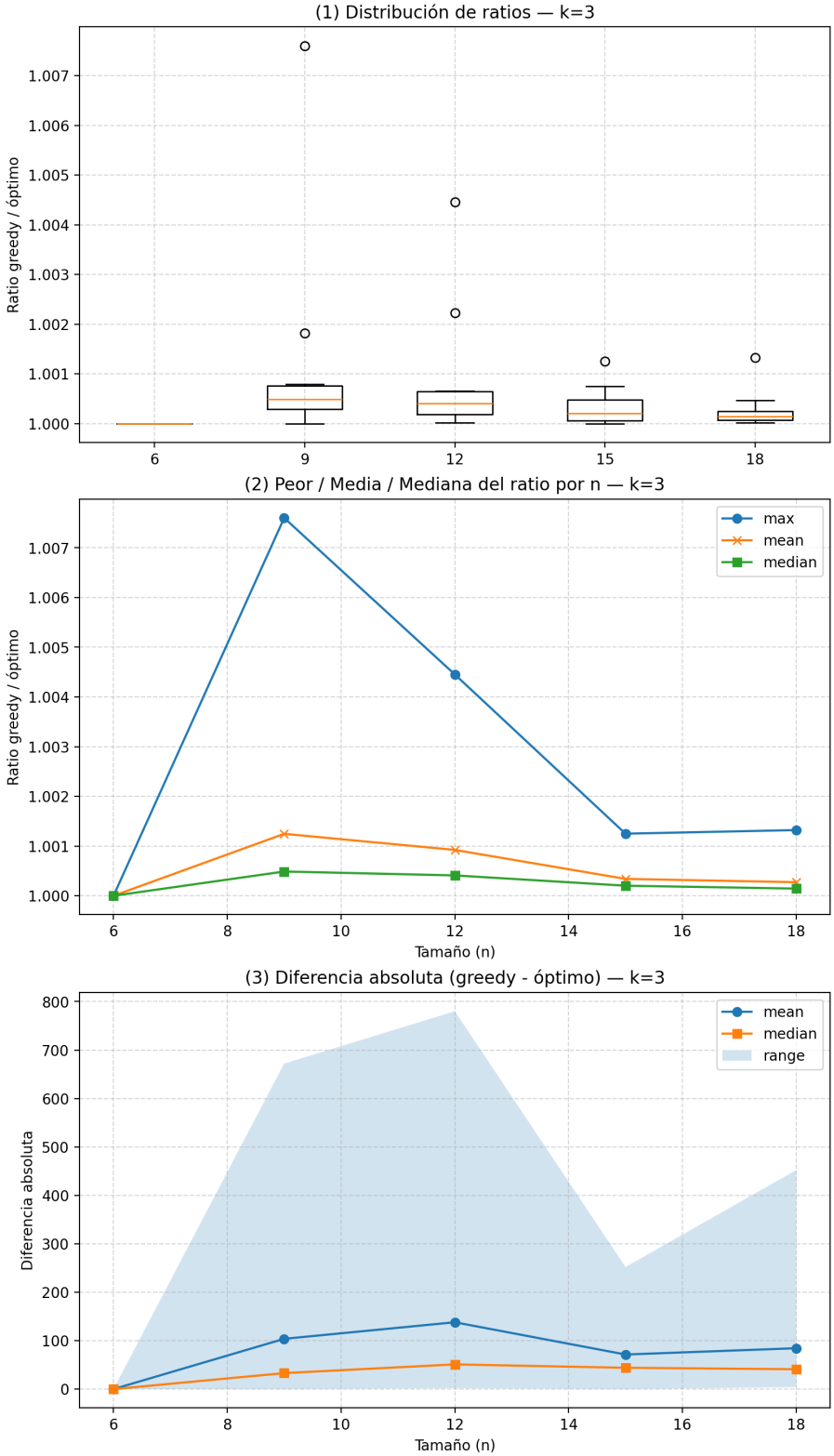
Nota: si bien los tiempos para el algoritmo Greedy parecen constantes, eso es debido al pequeño

intervalo considerado y a la gran diferencia de escala con los tiempos de Backtraking. Como se vio en gráfico anterior, siguen 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

6. 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.

6.1. 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.

6.2. 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.

6.3. 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.

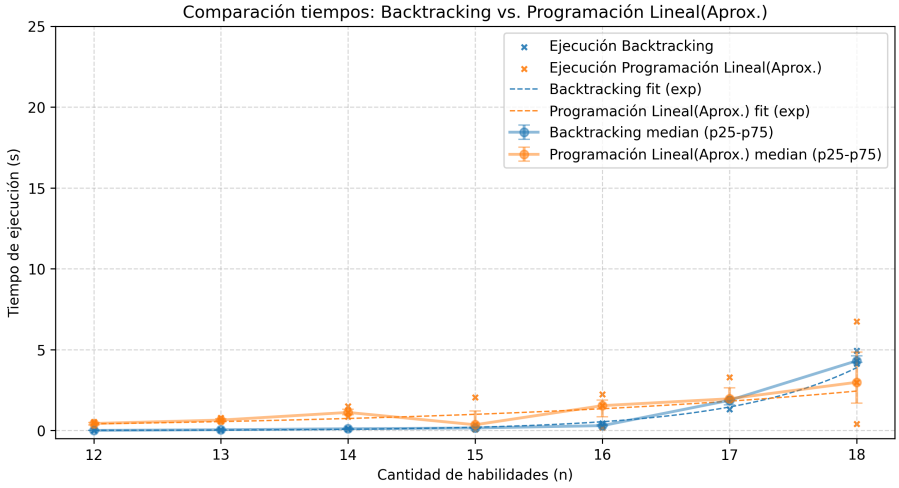
6.4. Implementación en Python con PuLP

```python
import pulp
from pulp import PULP_CBC_CMD


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

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

    return valor_objetivo, particion
```

6.5. Mediciones

TODO: Agregar contexto

Tiempos de Ejecución



TODO: Agregar Interpretación

Ratio de la Aproximación

TODO: Agregar contexto

TODO: Agregar Gráfico

TODO: Agregar Interpretación

7. Ejemplo de Ejecuciones

8. Conclusiones