Teoría de Algoritmos

(75.29) Curso Buchwald - Genender

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

18 de noviembre de 2025

Tomas Goncalves Rei
111405

Valentín Abaca
112088

Alexis Maximiliano Torrez Vargas
111449

Índice

1. Introducción

1.1. Enunciado

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

$$S_1, S_2, \ldots, S_k.$$

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 cuál podemos cuantificar diciendo que para el maestro i ese valor es x_1, y tenemos todos los valores $x_1, x_2 \ldots, x_n$ (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.

La versión de decisión del problema solicita determinar, dados los valores $x_1, \ldots, x_n$, un número de grupos k y una cota B, si existe una partición tal que

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

respetando que cada maestro sea asignado exactamente a un único grupo.

1.2. Planteo y Análisis del Trabajo Práctico

El problema descrito combina elementos de optimización combinatoria, análisis de complejidad y técnicas de diseño algorítmico avanzadas. La estructura de la función objetivo —una suma de cuadrados de sumas parciales— introduce dependencias no lineales entre los grupos, lo que lo convierte en un problema difícil de resolver mediante enfoques clásicos de programación lineal o greedy. Esta complejidad se ve reflejada en que la versión de decisión del problema es NP-Completa.

El objetivo central de este trabajo práctico es estudiar distintas estrategias para abordar el problema:

- **Diseñar e implementar un algoritmo exacto de Backtracking**, incorporando técnicas de poda y heurísticas que permitan reducir el espacio de búsqueda. Este algoritmo debe ser capaz de obtener la solución óptima respecto de la función cuadrática original.
- **Formular un modelo de Programación Lineal** que actúe como una aproximación. Dado que la función objetivo original no es lineal, se utilizará una formulación alternativa basada en minimizar la diferencia entre la carga máxima y mínima de los grupos, permitiendo así obtener soluciones balanceadas mediante un solver estándar.

- **Comparar empíricamente ambos enfoques**, midiendo tiempos de ejecución, calidad de solución y escalabilidad. El algoritmo exacto permite obtener un óptimo verdadero, mientras que el modelo lineal ofrece soluciones aproximadas en tiempos razonables para instancias mayores.
- **Analizar la dificultad computacional del problema**, justificando su pertenencia a la clase NP y demostrando su NP-Compleción mediante reducciones apropiadas.

A través de estos componentes, el trabajo permite estudiar en profundidad tanto una técnica exacta de búsqueda exhaustiva con poda, como una técnica aproximada mediante optimización matemática, explorando las diferencias prácticas entre exactitud y eficiencia computacional.

2. Desarrollo

2.1. Problema de la Tribu del Agua en NP

Versión de decisión del Problema de la Tribu del Agua. La entrada está dada por una secuencia de n fuerzas/habilidades positivas de maestros agua $x_1, \ldots, x_n$, un entero positivo k (cantidad de subgrupos) y un entero positivo B. La pregunta es:

¿Existe una partición de los maestros en k subgrupos $S_1, \ldots, S_k$ tal que

$$\sum_{i=1}^{k} \left(\sum_{x_j \in S_i} x_j \right)^2 \leq B$$ y cada maestro pertenece a exactamente un subgrupo?

Posible solución. Una solución candidata consiste en una lista de k listas, donde cada lista representa un subconjunto S_i de maestros asignados al grupo i.

Verificación. Dado una posible solución, definimos un verificador polinomial que ejecuta los siguientes pasos:

1. Comprueba que la cantidad de subgrupos entregada sea exactamente k.
2. Recorre cada subgrupo y agrega los maestros encontrados a un conjunto auxiliar M; si algún maestro aparece más de una vez, rechaza.
3. Verifica que $|M| = n$ y que todo maestro de la instancia aparece exactamente una vez en los subgrupos.
4. Calcula para cada subgrupo S_i la suma $s_i = \sum_{x_j \in S_i} x_j$ y computa

$$F = \sum_{i=1}^{k} s_i^2.$$

   Si $F \leq B$, acepta; en caso contrario, rechaza.

Este procedimiento corresponde directamente al siguiente verificador implementado en Python:

```python
def verificador(maestros, fuerzas, k, B, subgrupos):
    if len(subgrupos) != k:
        return False

    maestros_auxiliar = set()
    for subgrupo in subgrupos:
        for maestro in subgrupo:
            if maestro in maestros_auxiliar:
                return False
            maestros_auxiliar.add(maestro)

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

    for maestro in maestros:
        if maestro not in maestros_auxiliar:
            return False

    fuerza_total = 0
    for subgrupo in subgrupos:
        fuerza_grupal = 0
        for maestro in subgrupo:
            fuerza_grupal += fuerzas[maestro]
        fuerza_total += fuerza_grupal * fuerza_grupal
    if fuerza_total > B:
        return False
    return True
```

Complejidad. El verificador recorre cada maestro a lo sumo una vez en los pasos 2, 3 y 4. Las operaciones sobre conjuntos y diccionarios en Python tienen tiempo amortizado $O(1)$.

- Recorrer todos los subgrupos y maestros: $O(n)$.
- Verificar que la cantidad de maestros asignados coincida con los de la instancia: $O(1)$.
- Comprobar que todos los maestros aparecen exactamente una vez: $O(n)$.
- Calcular la suma de fuerzas y la suma de cuadrados: $O(n)$.

Por lo tanto, la complejidad total del verificador es:

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

que es polinomial en el tamaño de la entrada.

Conclusión. Como existe un verificador polinomial para revisar la validez de cualquier posible solución, el Problema de la Tribu del Agua pertenece a la clase NP.

2.2. Problema de la Tribu del Agua en NP-Completo

Problema Partition (versión de decisión). La entrada está dada por una secuencia de n enteros positivos $a_1, \ldots, a_n$. Denotando

$$T \;=\; \sum_{i=1}^{n} a_i,$$

la pregunta es:

¿Existe una partición del conjunto $\{a_1, \ldots, a_n\}$ en dos subconjuntos A_1 y A_2 tal que

$$\sum_{a_i \in A_1} a_i \;=\; \sum_{a_i \in A_2} a_i \;=\; T/2\,?$$

Se sabe que Partition es un problema NP-Completo.

Reducción de Partition al Problema de la Tribu del Agua. Dada una instancia de Partition formada por los números $a_1, \ldots, a_n$, construimos en tiempo polinomial la siguiente instancia del Problema de la Tribu del Agua (versión de decisión):

- Tomamos las fuerzas de los maestros como $x_i = a_i$ para todo i.
- Fijamos $k = 2$ subgrupos.
- Sea $T = \sum_{i=1}^{n} a_i$ y definimos
$$B \;=\; \frac{T^2}{2}.$$

Es claro que esta transformación se puede realizar en tiempo $O(n)$: simplemente copiamos los números, calculamos la suma total T y luego $B = T^2/2$.

La instancia resultante del Problema de la Tribu del Agua pregunta si existe una partición de los maestros en dos subgrupos S_1 y S_2 tal que cada maestro pertenece a exactamente un subgrupo y se cumple

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

Denotemos $s_1 = \sum_{x_j \in S_1} x_j$ y $s_2 = \sum_{x_j \in S_2} x_j$. Siempre se cumple $s_1 + s_2 = T$ porque los subgrupos forman una partición.

($\Rightarrow$) Supongamos que la instancia de Partition responde True. Entonces existe una partición $\{A_1, A_2\}$ tal que $\sum_{a_i \in A_1} a_i = \sum_{a_i \in A_2} a_i = T/2$. Tomamos $S_1 = A_1$ y $S_2 = A_2$ como partición de maestros.

En ese caso,

$$s_1 = s_2 = \frac{T}{2} \quad \Longrightarrow \quad s_1^2 + s_2^2 = \left(\frac{T}{2}\right)^2 + \left(\frac{T}{2}\right)^2 = \frac{T^2}{2} = B.$$

Luego, la instancia del Problema de la Tribu del Agua también responde True.

($\Leftarrow$) Supongamos que la instancia del Problema de la Tribu del Agua responde True. Entonces existe una partición en dos subgrupos S_1 y S_2 tal que $s_1 + s_2 = T$ y

$$s_1^2 + s_2^2 \leq \frac{T^2}{2}.$$

Observemos que, para dos números positivos s_1 y s_2 cuya suma es fija, la cantidad $s_1^2 + s_2^2$ es mínima exactamente cuando ambos son iguales:

$$s_1 = s_2 = \frac{T}{2}.$$

En tal caso, el valor mínimo de la suma de cuadrados es

$$\left(\frac{T}{2}\right)^2 + \left(\frac{T}{2}\right)^2 = \frac{T^2}{2}.$$

Como $\frac{T^2}{2}$ es el mínimo posible y la instancia del problema de la Tribu del Agua exige que

$$s_1^2 + s_2^2 \leq \frac{T^2}{2},$$

se deduce necesariamente que debe darse la igualdad y, por lo tanto, $s_1 = s_2 = T/2$.

De esta forma, la partición S_1, S_2 obtenida constituye una solución del problema Partition: divide el conjunto de valores en dos subconjuntos de suma total exactamente $T/2$ cada uno.

Hemos probado entonces que la instancia de Partition es afirmativa si y solo si la instancia correspondiente del Problema de la Tribu del Agua es afirmativa. La transformación es polinomial, por lo que se tiene

$$\text{Partition} \leq_p \text{Problema de la Tribu del Agua.}$$

Como Partition es NP-Completo y ya demostramos en la subsección anterior que el Problema de la Tribu del Agua pertenece a NP, concluimos que el Problema de la Tribu del Agua es NP-Completo.

2.3. Demostración de que Partition es NP-Completo

1. Partition pertenece a NP. Dado una posible solución que indica un subconjunto cuya suma es la mitad de la suma total de los elementos, es posible verificar en tiempo polinomial que: (i) cada elemento pertenece a un único subconjunto, y (ii) que la suma de los elementos de ambos subconjuntos es igual. Por lo tanto, Partition $\in$ NP.

2. Partition es NP-difícil. Lo probamos mediante una reducción en tiempo polinomial desde el problema Subset-Sum, el cual es NP-Completo.

Una instancia de Subset-Sum consiste en un conjunto de enteros positivos

$$S = \{a_1, \ldots, a_n\}$$

y un entero positivo T. El objetivo es determinar si existe un subconjunto de S cuya suma sea exactamente T.

Dada esta instancia, construiremos una instancia del problema Partition. Sea

$$z_1 = \sum(S) \qquad \text{y} \qquad z_2 = 2T.$$

Definimos la instancia de Partition como el conjunto:

$$S' = S \cup \{z_1, z_2\}.$$

La suma total de los elementos de S' es:

$$\sum(S') = \sum(S) + z_1 + z_2 = \sum(S) + \sum(S) + 2T = 2\big(\sum(S) + T\big).$$

Por lo tanto, una solución de Partition debe dividir S' en dos subconjuntos cuya suma sea exactamente:

$$\sum(S) + T.$$

($\Rightarrow$) Supongamos que existe un subconjunto $S'' \subseteq S$ tal que $\sum(S'') = T$ en la instancia de Subset-Sum. Entonces el conjunto:

$$S'' \cup \{z_1\}$$

tiene suma:

$$T + \sum(S) = \sum(S) + T,$$

que es exactamente la mitad de la suma total de S'. Por lo tanto, la instancia de Partition responde True.

($\Leftarrow$) Supongamos ahora que la instancia de Partition responde True. Entonces existe un subconjunto $P \subseteq S'$ cuya suma es $\sum(S) + T$.

El subconjunto P no puede contener simultáneamente a z_1 y z_2, ya que:

$$z_1 + z_2 = \sum(S) + 2T > \sum(S) + T.$$

Por lo tanto, P contiene exactamente uno de los dos números grandes.

Caso 1: P contiene $z_1 = \sum(S)$. Entonces, para que la suma sea $\sum(S) + T$, el resto de los elementos en P debe sumar exactamente T. Dichos elementos provienen de S, por lo que existe un subconjunto de S que suma T, resolviendo así la instancia de Subset-Sum.

Caso 2: P contiene $z_2 = 2T$. Un razonamiento similar muestra que el resto de los elementos debe sumar $\sum(S) - T$, lo cual implica que los elementos no incluidos en P suman T, resolviendo nuevamente la instancia de Subset-Sum.

Conclusión. La construcción es claramente polinomial y la reducción es correcta. Como Subset-Sum es NP-Completo y Partition $\in$ NP, concluimos que Partition es NP-Completo.

2.4. Solucion optima por Backtracking

2.4.1. Planteo del Problema

El objetivo del trabajo consiste en dividir a los Maestros Agua en k grupos $S_1, S_2, \ldots, S_k$ de manera tal que dichos grupos estén lo más equilibrados posible. A cada maestro i se le asigna una habilidad positiva x_i, y se busca minimizar:

$$\sum_{j=1}^{k} \left(\sum_{x_i \in S_j} x_i \right)^2$$

El término cuadrático penaliza fuertemente la existencia de grupos con cargas muy desbalanceadas, por lo que la función objetivo representa una medida adecuada de "equilibrio" entre grupos.

2.4.2. Uso de Backtracking

Al tratarse de un problema NP-Completo, no se conoce un algoritmo polinomial que garantice la obtención de la solución óptima. El enfoque de Backtracking permite explorar todas las posibles asignaciones de maestros a grupos, manteniendo siempre la mejor solución encontrada y utilizando técnicas de poda para reducir el espacio de búsqueda. Este enfoque garantiza optimalidad exacta, a costa de una complejidad exponencial en el peor caso.

2.4.3. Algoritmo Propuesto

En esta sección se muestra el código implementado en backtracking para hallar la solución optima al problema mencionado.

```
1
2  def backtrack_recursivo(indice, k, n, habilidades, nombres, cargas, valor_actual,
       asignacion_actual, mejor, suma_restante):
3      """
4      Funcion recursiva que explora todas las asignaciones posibles de maestros a
       grupos
5      minimizando la suma de cuadrados de las sumas por grupo.
6      Usa poda y ruptura de simetrias.
7      """
8
9      mejor_valor, mejor_asignacion = mejor
10
11     # Caso base: todos los maestros fueron asignados
12     if indice == n:
13         if valor_actual < mejor_valor:
14             return valor_actual, asignacion_actual[:]
15         return mejor
16
17     habilidad = habilidades[indice]
18     usado_vacio = False
19
20     grupos_ordenados = sorted(range(k), key=lambda g: cargas[g])
21
22     for g in grupos_ordenados:
23         vacio = (cargas[g] == 0)
24         if vacio and usado_vacio:
25             continue
26
27         # Cota inferior: si incluso distribuyendo el resto perfectamente no mejora
       el mejor valor, podar
28         resto = suma_restante[indice + 1]
29         if valor_actual + (resto * resto) / k >= mejor_valor:
30             continue
31
32         incremento = 2 * cargas[g] * habilidad + habilidad * habilidad
33         nuevo_valor = valor_actual + incremento
34
35         if nuevo_valor >= mejor_valor:
36             continue
37
38         # Asignar maestro al grupo g
39         cargas[g] += habilidad
40         asignacion_actual[indice] = g
41         if vacio:
42             usado_vacio = True
43
44         mejor = backtrack_recursivo(indice + 1, k, n, habilidades, nombres, cargas,
        nuevo_valor, asignacion_actual, mejor, suma_restante)
45
46         # Deshacer asignacion
47         cargas[g] -= habilidad
48         asignacion_actual[indice] = -1
49         if vacio:
50             usado_vacio = False
51
52         mejor_valor, _ = mejor
53
54     return mejor
55
56
57 def tribu_del_agua_backtracking(k, maestros):
58     """
59     Funcion principal que prepara los datos y llama al backtracking.
60     Devuelve el valor minimo y los grupos optimos.
61     """
```

```
62
63      # Ordenamos de mayor a menor habilidad
64      maestros = sorted(maestros, key=lambda x: x[1], reverse=True)
65      n = len(maestros)
66      habilidades = [m[1] for m in maestros]
67      nombres = [m[0] for m in maestros]
68
69      cargas = [0] * k
70      asignacion_actual = [-1] * n
71
72      mejor_valor = float('inf')
73      mejor_asignacion = None
74
75      # Precomputar sumas restantes (suma_restante[i] = suma de habilidades desde i
        hasta fin)
76      suma_restante = [0] * (n + 1)
77      for i in range(n - 1, -1, -1):
78          suma_restante[i] = suma_restante[i + 1] + habilidades[i]
79
80      mejor_valor, mejor_asignacion = backtrack_recursivo(0, k, n, habilidades,
        nombres, cargas, 0, asignacion_actual, (mejor_valor, mejor_asignacion),
        suma_restante)
81
82      # Construir los grupos finales
83      grupos_finales = [[] for _ in range(k)]
84      for i, g in enumerate(mejor_asignacion):
85          grupos_finales[g].append(nombres[i])
86
87      return mejor_valor, grupos_finales
```

2.4.4. Descripción del Algoritmo

El algoritmo asigna los maestros uno por uno a cada grupo, siguiendo un esquema recursivo. En cada paso:

1. Se toma al maestro del `indice`.
2. Se prueba asignarlo a cada uno de los k grupos.
3. Se actualiza la carga del grupo seleccionado.
4. Se calcula el costo parcial mediante la regla incremental:

$$\Delta = 2 \cdot \text{carga}[g] \cdot x + x^2$$

   lo que permite evitar recomputar cuadrados completos.

5. Se aplican podas: por valor parcial y por cota inferior.
6. Se aplica ruptura de simetrías: si varios grupos están vacíos, solo uno se explora.
7. Se continúa recursivamente con el siguiente maestro.

Cuando `indice = n`, todos los maestros están asignados y se evalúa la solución completa.

2.4.5. Optimizaciones Incorporadas

1. Ordenamiento de Maestros

Los maestros se ordenan de mayor a menor habilidad para aumentar la efectividad de la poda. Esto produce una reducción significativa del árbol de búsqueda.

2. Cálculo Incremental del Costo

En vez de recomputar la función objetivo completa, se actualiza en $O(1)$ usando:

$$\Delta = 2 \cdot \text{carga}[g] \cdot x + x^2$$

3. Precomputación de Suma Restante

Se calcula un arreglo `suma_restante` tal que:

$$\text{suma\_restante}[i] = \sum_{t=i}^{n} x_t$$

Se utiliza para una cota inferior optimista:

$$\frac{(\text{suma restante})^2}{k}$$

4. Ruptura de Simetrías

Si múltiples grupos están inicialmente vacíos, asignar el maestro al segundo o tercer grupo es equivalente a asignarlo al primero. Esto también sucede si distintos grupos tienen la misma carga, es lo mismo asignar el maestro al grupo 1 con carga 10 como asignarlo al grupo 2 con carga 10. Por ello, el algoritmo evita explorar permutaciones redundantes.

5. Ordenamiento de Grupos por Carga

Los grupos se exploran de menor a mayor carga, priorizando ramas más prometedoras.

2.4.6. Complejidad del Algoritmo

Complejidad sin Poda (Peor Caso)

Cada maestro puede asignarse a cualquiera de los k grupos:

$$T(n) = k^n$$

Complejidad Real con Podas

Gracias a las optimizaciones mencionadas, el tamaño del árbol de búsqueda se reduce enormemente. Sin embargo, en el peor caso teórico, la complejidad sigue siendo exponencial, acorde a lo esperado para un problema NP-Completo.

En la práctica, el algoritmo logra resolver satisfactoriamente instancias pequeñas y medianas (hasta aproximadamente 15–20 maestros), mientras que instancias más grandes se vuelven intratables, como es esperable.

2.4.7. Conclusión

El algoritmo de Backtracking desarrollado:

- Encuentra la solución óptima exacta del modelo cuadrático.
- Utiliza múltiples técnicas de reducción del espacio de búsqueda.
- Permite comparar la solución óptima con modelos aproximados como Programación Lineal.

Este método representa una solución exhaustiva y optimizada para el Problema de la Tribu del Agua.

2.5. Modelo de Programacion Lineal

En esta sección presentamos un modelo de Programación Lineal que permite aproximar la solución del problema de la Tribu del Agua. Dado que la función objetivo original

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

es cuadrática, no puede ser modelada directamente mediante Programación Lineal. Por ello adoptamos la alternativa indicada en el enunciado: minimizar la diferencia entre la suma de habilidades del grupo más cargado y el del menos cargado. Esta aproximación permite obtener soluciones balanceadas y compararlas empíricamente con el algoritmo exacto de Backtracking.

2.5.1. Variables de decisión

Se definen las siguientes variables:

- Variables binarias de asignación:

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

- Variables continuas que representan la suma de habilidades por grupo:

$$S_j \geq 0 \qquad \forall j \in \{1, \ldots, k\}$$

- Variables auxiliares:

$$M \geq 0 \qquad \text{(máxima suma entre todos los grupos)}$$

$$m \geq 0 \qquad \text{(mínima suma entre todos los grupos)}$$

2.5.2. Restricciones

Asignación única por maestro. Cada maestro debe ser asignado exactamente a un grupo:

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

Definición de la carga de cada grupo. La suma de habilidades de cada grupo se define como:

$$S_j = \sum_{i=1}^{n} h_i \, x_{ij} \qquad \forall j \in \{1, \ldots, k\}$$

Relación entre las cargas y las variables auxiliares. Las variables M y m representan la mayor y menor carga entre todos los grupos:

$$S_j \leq M \qquad \forall j$$

$$S_j \geq m \qquad \forall j$$

2.5.3. Función objetivo

La función objetivo busca minimizar la diferencia entre la mayor y la menor carga grupal:

$$\text{mín}(M - m)$$

Este criterio fuerza a que las cargas queden lo más parejas posibles entre los grupos y, aunque no optimiza la función cuadrática original, produce soluciones de buena calidad y adecuadas como aproximaciones.

2.5.4. Complejidad del modelo

El modelo posee:

- nk variables binarias x_{ij}
- $k + 2$ variables continuas (S_j, M, m)
- Aproximadamente $n + 3k$ restricciones lineales

2.5.5. Relación con la solución exacta

El modelo lineal minimiza:

$$M - m$$

mientras que el algoritmo exacto minimiza:

$$\sum_{j=1}^{k} S_j^2.$$

Ambos criterios tienden a equilibrar las cargas entre grupos, pero no necesariamente producen la misma solución. El modelo lineal provee una *aproximación* al óptimo verdadero. Sin embargo al ser PLE la complejidad no mejora frente a la complejidad teórica del backtracking

2.6. Algoritmo de Aproximación

En esta sección presentamos el algoritmo de aproximación y su cota empirica.

El algoritmo como es descrito en el enunciado ordena y reparte a los maestros al grupo que tenga menos carga. Para esto utilizamos un heap de mínimos para mantener el grupo con menos carga, el codigo implementado es el siguiente:

```
def aprox(k, maestros):

    maestros = sorted(maestros, key=lambda x: x[1], reverse=True)
    heap = []
    for i in range(k):
        heapq.heappush(heap, (0,i, []))

    for nombre, poder in maestros:
        poder_actual, indice_desempate, nombres = heapq.heappop(heap)
        poder_actual += poder
        nombres.append(nombre)
        heapq.heappush(heap, (poder_actual, indice_desempate, nombres))

    costo_total = 0
    grupos = []
    for poder, _, nombres in heap:
        costo_total += (poder ** 2)
        grupos.append(nombres)

    return costo_total, grupos
```

Como vemos en el código mantenemos un heap donde inicialmente metemos los k grupos vacios con carga 0 y un indice i para desempatar en caso de que la carga sea igual para dos grupos, esto lo hacemos con el objetivo de no tener que comparar listas para desempatar. Luego iteramos sobre la lista de maestros y vamos actualizando los valores del heap, obtenemos el grupo menos cargado, le sumamos el poder del maestro, agregamos al maestro a su lista de miembros y lo volvemos a incluir en el heap. Finalmente iteramos sobre el heap sumando la carga de todos los grupos y calculamos el valor del coeficiente.

2.6.1. Cota empírica de aproximación

Comparamos la solución aproximada contra la solución optima obtenida del algoritmo de backtracking para 300,000 casos con valores completamente aleatorios, obtuvimos la peor de todas estas relaciones que fue dada por $r = 1{,}0245$ es decir el algoritmo aproximado encuentra a lo sumo una solución 2,45 % peor que la optima.

Sin embargo encontramos un peor caso donde la solución es 2,78 % peor que la optima, la instancia es la siguiente, 5 elementos y 2 grupos donde Elem = [3,3,2,2,2] y k = 2. Nuestro algoritmo greedy terminaría asignando primero 3 y 3 a cada uno de los grupos y luego va a repartir los 2 uno a uno para cada uno de los grupos quedando finalmente G1 = [3,2,2] y G2 = [3,2] sin embargo el optimo era G1 = [3,3] y G2[2,2,2] al comparar estas soluciones obtenemos que la relacion r esta dada por $r = 1{,}02777$.

Podemos ver empíricamente que nuestro algoritmo encuentra a lo sumo una solución 2,45 % peor que la optima en casos completamente aleatorios o 2,77 % peor en un caso donde forzamos aumentar esta desigualdad.

2.6.2. Complejidad del algoritmo de Aproximación

Comenzamos con el ordenamiento de la lista de maestros, sea n la cantidad de maestros y k la cantidad de grupos a dividir estos maestros.

```
1    maestros = sorted(maestros, key=lambda x: x[1], reverse=True)
```

El ordenamiento por el método sorted de python nos da una complejidad $O(n \log n)$

Continuamos con la inserción al heap de los grupos vacíos.

```
1    heap = []
2    for i in range(k):
3        heapq.heappush(heap, (0,i, []))
```

Esta sección del código es $O(k \log k)$ ya que el heap tiene a lo sumo k elementos

Luego iteramos sobre todos los maestros.

```
1    for nombre, poder in maestros:
2        poder_actual, indice_desempate, nombres = heapq.heappop(heap)
3        poder_actual += poder
4        nombres.append(nombre)
5        heapq.heappush(heap, (poder_actual, indice_desempate, nombres))
```

Aquí realizamos una cantidad n de veces la operación de quitar e incluir al heap un elemento, por lo tanto esta sección del código tiene una complejidad de $O(n \log k)$

Finalmente calculamos el costo total (coeficiente) sumando la carga de todos los grupos

```
1    for poder, _, nombres in heap:
2        costo_total += (poder ** 2)
3        grupos.append(nombres)
```

Iteramos sobre todos los grupos por lo tanto esta sección tiene una complejidad de $O(k)$

Suponiendo que $k \leq n$ ya que no se deberían dejar grupos vacíos la complejidad final del algoritmo es $O(n \log n)$

3. Mediciones

3.1. Comparación de técnicas de diseño

Tomamos mediciones de tiempo de ejecución de los algoritmos con distintos tamaños de muestra manejables para la complejidad temporal de los mismos. Lo tomamos según una cantidad n de maestros promediando los valores de k es decir el tiempo para una cantidad n de maestros es el promedio entre todos los pares de (n,k) medidos.

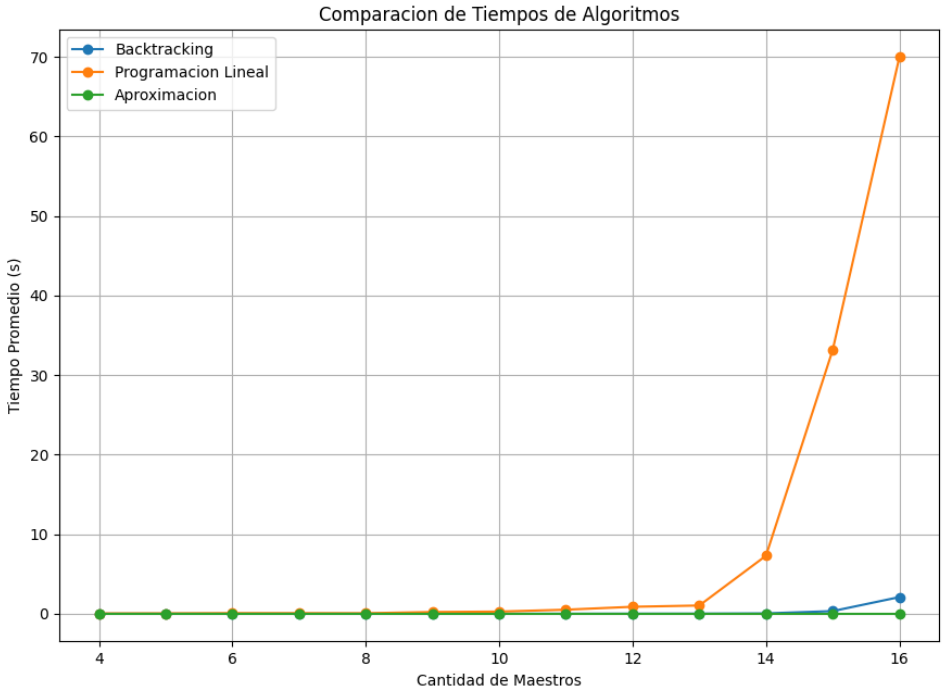

Figura 1: Gráfico del tiempo de ejecución según tamaño de muestra y algoritmo.

Para poder ver mejor las escalas separamos el la figura 1 en 2 gráficos

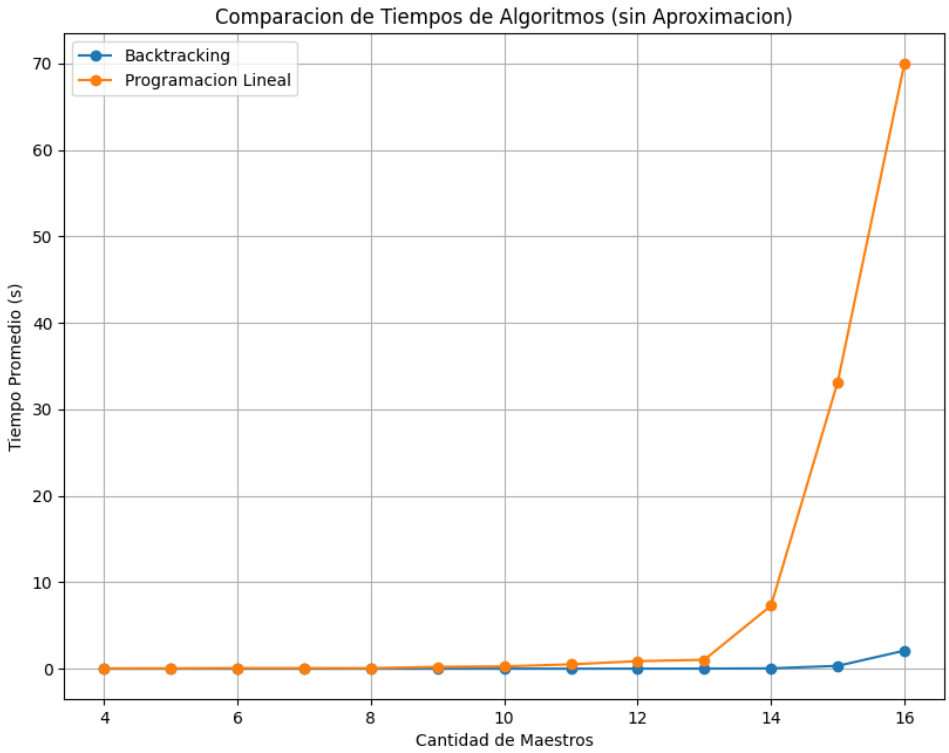

Figura 2: Gráfico del tiempo de ejecución según tamaño de muestra y algoritmo sin aproximación.

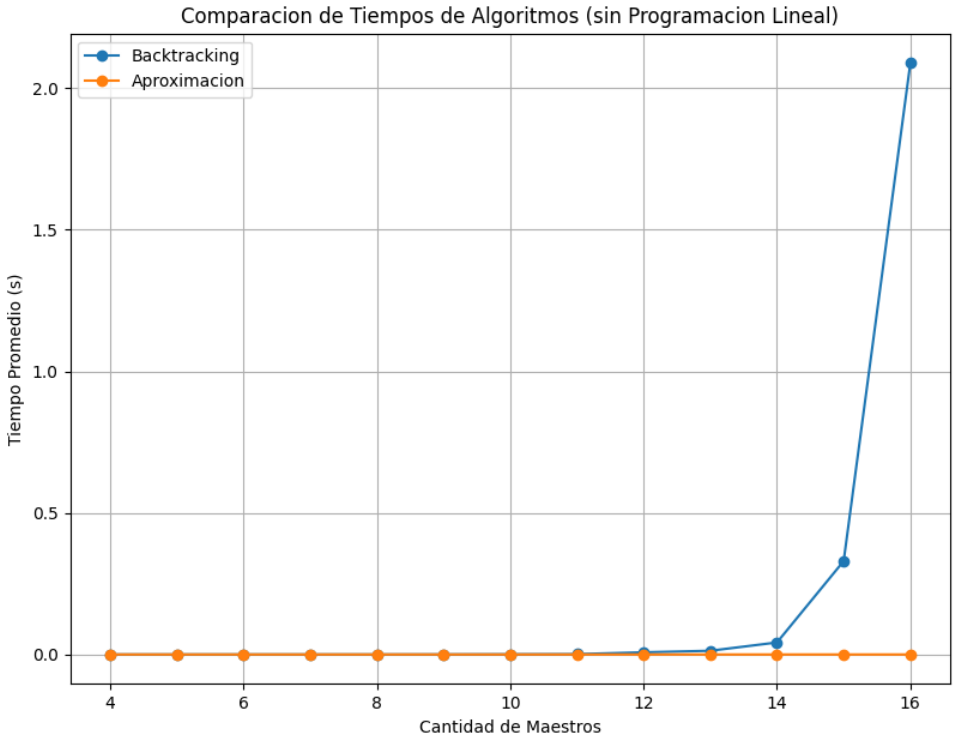

Figura 3: Gráfico del tiempo de ejecución según tamaño de muestra y algoritmo sin modelo de programación lineal.

Como podemos ver en la figura 2 los tiempos del modelo de PL son mucho mayores a los obtenidos por backtracking.

Luego en la figura 3 se puede ver que los tiempos de la aproximación para este volumen de datos son mucho menores a los de backtracking.

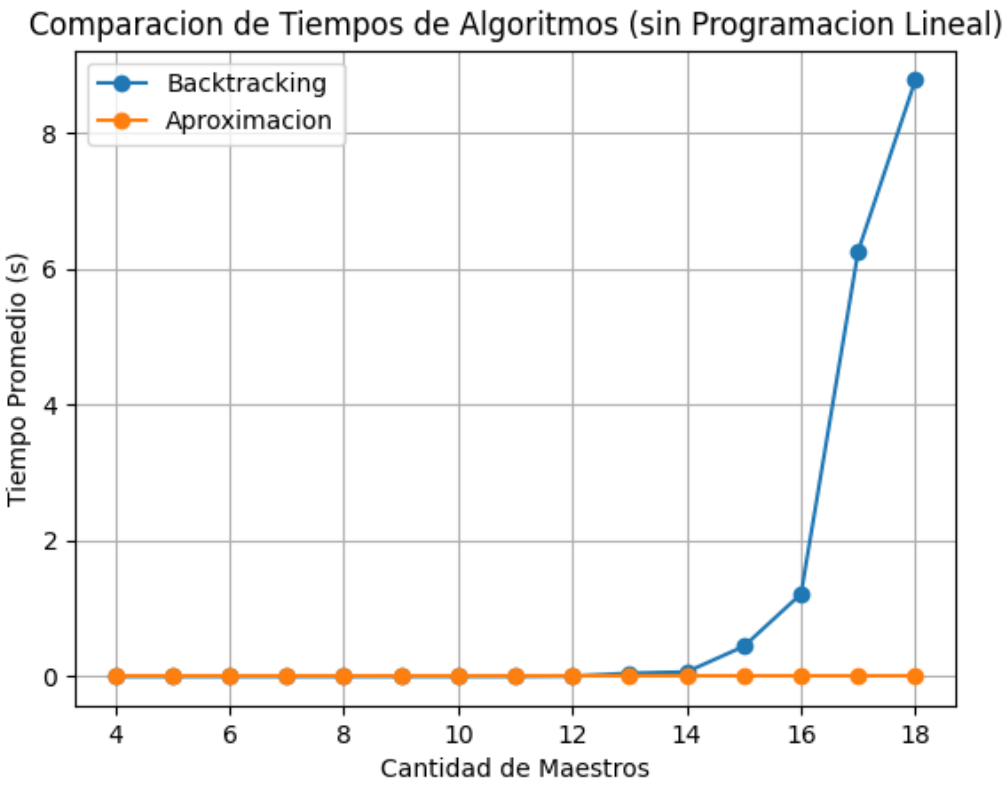

Figura 4: Gráfico del tiempo de ejecución según tamaño de muestra expandido para el algoritmo de backtracking.

En la figura 4 decidimos expandir, considerando que el algoritmo de backtracking demora menos tiempo, el rango de los elementos evaluados por este algoritmo. Como podemos ver también se dispara exponencialmente a medida que van creciendo la cantidad de elementos.

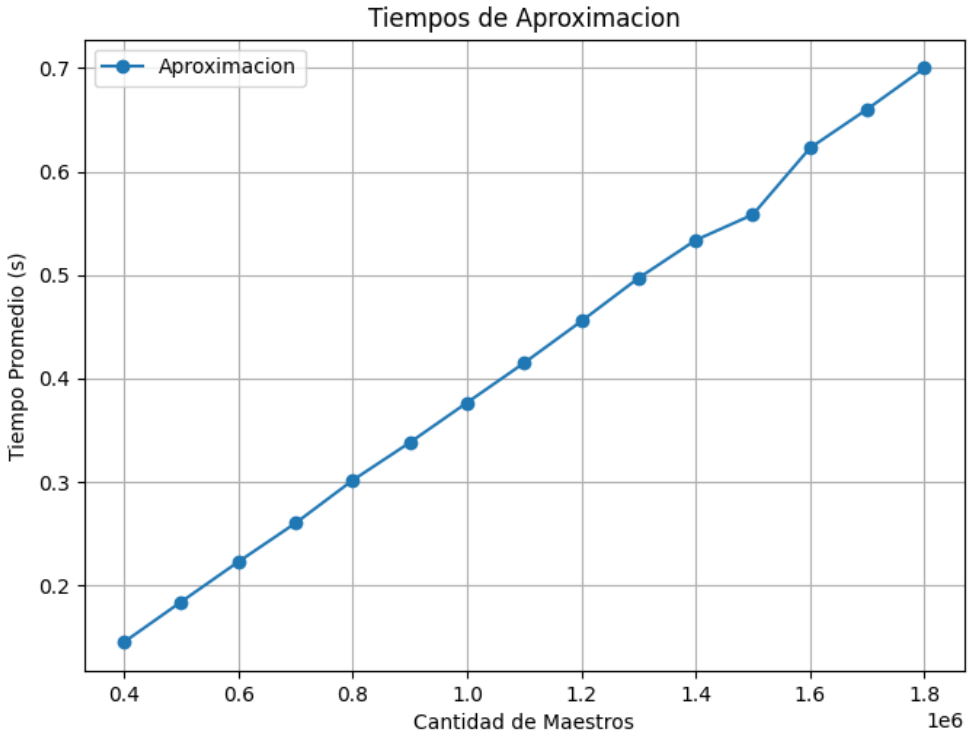

Figura 5: Gráfico del tiempo de ejecución de la aproximación según tamaño de muestra.

Tomamos mediciones con un rango inmanejable para los modelos anteriores utilizando la aproximación, como se puede ver en la figura 5 se llega hasta 1.8 millones de maestros y el algoritmo en promedio no demora mas de 0,7 segundos.

4. Conclusiones

En este trabajo se abordó el Problema de la Tribu del Agua desde distintas perspectivas algorítmicas y de optimización, con el objetivo de comprender su complejidad intrínseca y analizar métodos prácticos para su resolución. A través de la formulación matemática del problema, se evidenció que la función objetivo cuadrática —basada en la minimización de la suma de los cuadrados de las cargas grupales— genera interdependencias entre conjuntos que vuelven ineficientes los enfoques deterministas simples. Esto permitió justificar que el problema pertenece a la clase NP y, además, que su versión de decisión es NP-Completa.

El diseño del algoritmo de Backtracking, optimizado mediante poda, ordenamiento heurístico de las decisiones y cotas inferiores, permitió obtener soluciones exactas para instancias de tamaño moderado. A pesar de su complejidad exponencial, se demostró que una buena elección de estrategias de búsqueda reduce significativamente el espacio explorado, volviendo el método viable para casos pequeños y medianos. Esta implementación constituye el único enfoque capaz de garantizar óptimos respecto de la función cuadrática original.

Por otro lado, la formulación de un modelo de Programación Lineal brindó una alternativa aproximada para resolver instancias mayores. Dado que la función objetivo original no es linealizable sin introducir no convexidad, se adoptó la estrategia de minimizar la diferencia entre el grupo de mayor carga y el de menor carga. Si bien esta función objetivo no reproduce exactamente la optimización cuadrática, las soluciones obtenidas resultaron suficientemente balanceadas. Sin embargo por las mediciones obtenidas vemos que no es un buen modelo a utilizar ya que su tiempo de ejecución es notablemente mayor al algoritmo de backtracking.

En conjunto, el trabajo permitió estudiar un problema realista con fuerte componente combinatorio, integrando conceptos teóricos de complejidad, técnicas de diseño algorítmico, herramientas de optimización matemática y análisis experimental. La comparación entre métodos exactos y aproximados ofreció una visión profunda sobre las limitaciones y fortalezas de cada paradigma, permitiendo ver como en este caso es notoriamente mejor obtener soluciones aproximadas frente a la diferencia en tiempos de ejecución, así como sobre la importancia del balance entre calidad de solución y tiempo computacional en la resolución de problemas NP-Complejos.