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

Facultad de Ingeniería de la Universidad de Buenos Aires

Teoría de Algoritmos

Cátedra Buchwald-Genender

Gómez Belis, Sofía
Padrón: 109358
email: sgomezb@fi.uba.ar

Llanos Pontaut, Valentina
Padrón: 104413
email: vllanos@fi.uba.ar

Orsi, Tomas Fabrizio
Padrón: 109735
email: torsi@fi.uba.ar

Indice

1 Introducción

1.1 Descripción y objetivo

Continuando con el ataque de la Nación del Fuego sobre el resto de las naciones, esta vez es la Tribu del Agua la que requiere de nuestra ayuda para defenderse.

Cada maestro agua tiene una fuerza o habilidad positiva x_i, y contamos con el conjunto de todos los valores $(x_i, x_2, \ldots, x_n)$. Basándonos en estos, el maestro Pakku desea separar los maestros en `k` grupos $(S_1, S_2, \ldots, S_k)$ parejos tal que cuando un grupo se canse, entrará el siguiente en el combate, obteniendo un ataque constante que les permita salir victoriosos, aprovechando también la ventaja del agua por sobre el fuego.

Para que los grupos estén lo más parejos posibles, nos han encomendado minimizar la adición de los cuadrados de las sumas de las fuerzas de los grupos:

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

En este trabajo desarrollaremos algoritmos de backtracking, programación lineal y posibles aproximaciones buscando resolver el problema de optimización planteado con el objetivo de ayudar a los maestros de la Tribu del Agua a derrotar a la Nación del Fuego. También nos dedicaremos a demostrar que el problema de la tribu del agua es NP-Completo.

2 Demostración de problema NP-Completo

Los problemas NP-Completos son los problemas más difíciles de NP. Cualquier problema en NP puede ser reducido a uno NP-Completo.

Para demostrar que el problema de la Tribu del Agua es NP-Completo, primero debemos demostrar que se encuentra en NP. Posteriormente, si logramos hacer una reducción polinomial de un problema NP-Completo a éste, entonces esto implica que también se trata de un problema NP-Completo. Ésto se debe a que la reducción $Y \leq_p X$ implica que la dificultad de resolver `Y` se reduce polinomialmente a la dificultad de resolver `X`, o que `X` es al menos tan difícil de resolver como `Y`. Si `Y` es un problema NP-Completo, entonces `X` es al menos tan difícil de resolver como un problema NP-Completo.

Para realizar la demostración y la reducción, necesitamos basarnos en el problema de decisión:

Dados una secuencia de **`n`** *fuerzas de maestros agua* $(x_i, x_2, \ldots, x_n)$*, y dos números* **`k`** *y* **`B`***, ¿existe una partición en* **`k`** *subgrupos* $(S_1, S_2, \ldots, S_k)$ *tal que:*

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

y cada elemento x_i esté asignado a solamente un grupo?

2.1 Problema en NP

Un problema se encuentra en NP si existe un certificador eficiente para el mismo. Es decir, si puede ser verificado o validado en tiempo polinomial.

Tenemos los siguientes datos:

- Habilidades de los maestros → se presentan en forma de tupla (nombre maestro, fuerza)
- Cantidad de subconjuntos → `k`
- Cota del coeficiente → `B`
- Resultado a validar → subconjuntos S_i que contienen los nombres de los maestros del grupo correspondiente

También tenemos las siguientes restricciones:

- Cada maestro debe estar asignado a un solo grupo
- El resultado debe contener `k` grupos
- La sumatoria resultante debe ser a lo sumo `B`

Dados estos datos y restricciones, proponemos el siguiente certificador.

```
def validador_problema_tribu_del_agua(habilidades_maestros_agua, k,
                                      B, S):
    if len(S) != k: #O(1)
        return False
    coeficiente = 0 #O(1)
    maestros = set() #O(1) Cada maestro debe estar en solo un grupo
    dic_habilidades = dict() #O(1)

    for info_maestro in habilidades_maestros_agua: #O(n)
        maestro, habilidad = info_maestro #O(1)
        dic_habilidades[maestro] = habilidad #O(1)

    # O(n * k)
    for grupo in S: # k grupos: O(k)
        suma_grupo = 0 #O(1)
        if len(grupo) == 0:
            return False
        for maestro in grupo: # En el peor caso O(n)
            if maestro in maestros or maestro not in dic_habilidades:
                return False # Está en más de un grupo
            maestros.add(maestro) #O(1)
```

```
22              suma_grupo += dic_habilidades[maestro] #O(1)
23          coeficiente += suma_grupo ** 2 #O(1)
24
25      if coeficiente > B:  # O(1)
26          return False
27
28      for maestro, _habilidad in habilidades_maestros_agua: #O(n)
29          if maestro not in maestros:  # O(1)
30              return False # No tiene grupo asignado
31
32      return True
```

Para que el verificador presentado pueda ser considerado un certificador eficiente, debe ejecutarse en tiempo polinomial. Por lo tanto, procedemos a explicar y justificar la complejidad del código.

Realizamos operaciones aritméticas, asignaciones y verificaciones. Todas estas son operaciones $\mathcal{O}(1)$ puesto que consumen tiempo constante según la documentación oficial . Recorremos dos veces la lista de maestros y habilidades con diferentes propósitos. Como contiene `n` tuplas, cada ciclo `for` tiene un costo $\mathcal{O}(n)$. Con el objetivo de obtener la adición de los cuadrados de las sumas de las fuerzas de los grupos y compararla con `B`, tenemos dos ciclos anidados. El segundo recorre los maestros de un grupo que, en el peor caso, son `n`. Esto se realiza `k` veces, por lo que en total es $\mathcal{O}(k \cdot n)$.

Entonces,

$$T(n) = \mathcal{O}(1) + 2 \cdot \mathcal{O}(n) + \mathcal{O}(k \cdot n) = \mathcal{O}(k \cdot n)$$

En el peor caso, $k = n$ y

$$T(n) = \mathcal{O}(n \cdot n) = \mathcal{O}(n^2)$$

Ambas cotas, $\mathcal{O}(k \cdot n)$ y $\mathcal{O}(n^2)$ conllevan tiempo polinomial. Por lo tanto, podemos afirmar que existe un certificador eficiente para el problema de la Tribu del Agua y entonces este problema está en NP.

2.2 Problema 2-Partition

Para demostrar que el problema de la tribu del agua es NP-Completo, vamos a hacer uso del problema 2-Partition. Este se define de la siguiente manera: se tiene un conjunto de n elementos, cada uno con un valor, y se desea separarlos en 2 subconjuntos tal que la suma de los valores de cada uno sea igual para ambos. Procederemos a justificar que se trata de un problema NP-Completo.

2.2.1 Problema en NP

Enunciamos el problema de decisión asociado: *dado un conjunto* $A = \{a_1, a_2, \ldots, a_n\}$*, ¿existen dos subconjuntos* A_1 *y* A_2 *tales que* $\sum_{a \in A_1} a = \sum_{a \in A_2} a$*?*

A continuación mostramos la implementación de un validador:

```python
def validador_2_partition(valores, s1, s2):
    if (sum(s1) != sum(s2) or len(s1) >= len(valores) or
        len(s2) >= len(valores)):
        return False

    apariciones = {}
    for elemento in s1:
        apariciones[elemento] = apariciones.get(elemento, 0) + 1

    for elemento in s2:
        apariciones[elemento] = apariciones.get(elemento, 0) + 1

    valores_no_repetidos = {}
    for valor in valores:
        nuevoValor = valores_no_repetidos.get(valor, 0) + 1
        valores_no_repetidos[valor] = nuevoValor

    if len(apariciones) != len(valores_no_repetidos):
        return False

    for valor, cantidad in apariciones:
        if valor not in valores_no_repetidos:
            return False
        elif valores_no_repetidos[valor] != cantidad:
            return False

    return True
```

Hay tres ciclos `for`. Uno de ellos recorre todos los valores del conjunto. Considerando que hay `n`, es una operación $\mathcal{O}(n)$. Los conjuntos A_1 y A_2 deberían contener en total n elementos, pudiendo uno de ellos tener hasta $n-1$. En total, ambos conllevan $\mathcal{O}(n)$. Por lo tanto, la complejidad temporal del validador es $\mathcal{O}(n)$, constituyendo un certificador eficiente. Como el problema 2-Partition puede ser verificado en tiempo polinomial, podemos afirmar que se encuentra en NP.

2.2.2 Reducción

Como hemos visto en clase, el problema de subset sum es NP-Completo. Utilizaremos esta conclusión para demostrar que 2-Partition también lo es.

Dado un conjunto de números $S = \{s_1, s_2, \ldots, s_n\}$ y un número V, queremos saber si hay un subconjunto de S cuya suma sea igual a V.

Para realizar la reducción $SS \leq_p 2P$, con SS haciendo referencia a subset sum y 2P a 2-Partition, debemos definir cuál es la entrada (conjunto) de la caja negra que resuelve 2-Partition. Sea $s = sum(S)$ la suma de los valores del conjunto S, definimos $S' = S \cup \{s - 2 \cdot V\}$. Aplicamos el algoritmo en S'.

Agregar un elemento a un conjunto es $\mathcal{O}(1)$, mientras que si se desea copiar S para no modificarlo, la operación es lineal en el tamaño del set. Por lo tanto, esta transformación es polinomial.

Si $T \subseteq S$ es un subconjunto con suma V:

- Sea $O = S \setminus T$, con suma $o = s - V$, los elementos que "sobran".
- Sea $T' = T \cup \{s - 2 \cdot V\}$, cuya suma es $V + (s - 2 \cdot V) = s - V$. Es el subconjunto buscado, unido con el valor agregado a S.
- $O \cup T' = S \cup \{s - 2 \cdot V\} = S'$

Entonces, la suma de T' y O es la misma, $s-V$, lo que significa que podemos dividir S' en dos subconjuntos con la misma suma.

Luego, existe un subset sum de valor V en S si y solo si existe un 2-Partition en S':

$\rightarrow$ Si existe un conjunto de números en S que suman V, entonces los restantes en S suman $s - V$. Por lo tanto, existe una partición de `S'` en dos tal que cada partición suma $s - V$: $O = S \setminus T$ y $T' = T \cup \{s - 2 \cdot V\}$.

$\leftarrow$ Digamos que existe una partición de `S'` en dos conjuntos tal que la suma de cada conjunto es $s - V$. Uno de estos conjuntos contiene el número $s - 2 \cdot V$. Eliminando este número, obtenemos un conjunto de números cuya suma es V, y todos estos números están en S.

Como esta reducción conlleva una transformación polinomial, **podemos afirmar que 2-Partition es un problema NP-Completo**.

Ejemplo 1: La entrada de Subset Sum es $S = \{3, 1, 4, 2, 2\}$ y $V = 6$. Como la suma de los valores de S es 12, agregamos el elemento $12 - 2 \cdot 6 = 0$ $S' = \{3, 1, 4, 2, 2, 0\}$. Este conjunto puede ser particionado en $\{3, 1, 2\}$ y $\{4, 2, 0\}$, ambas con suma 6, donde $T = \{4, 2\}$ es un subconjunto de S que suma 6. Por lo tanto, la caja negra que resuelve el problema de partición, nos devolverá que es posible encontrar un subset sum que sume B.

Ejemplo 2: La entrada de Subset Sum es $S = \{5, 3, 2, 7\}$, que suma 17, y $V = 10$. Agregamos el valor $17 - 2 \cdot 10 = -3$. Entonces, $S' = \{5, 3, 2, 7, -3\}$ y se puede dividir en $\{7\}$ y $\{2, 5, 3, -3\}$, donde $T = \{2, 5, 3\}$ es el subconjunto que suma 10. Nuevamente existe solución. En cambio, si $V = 6$, tendríamos $S' = \{5, 3, 2, 7, 5\}$ no puede dividirse en dos subconjuntos que sumen $\frac{22}{2} = 11$, por lo que no existirá un subset sum con $V = 6$ en S.

2.3 Reducción del Problema del Balanceo de Carga

2.3.1 Introducción

Habiendo establecido que el problema pertenece a NP, ahora vamos a realizar una reducción de un problema NP-Completo a este problema. Para esta reducción, decidimos utilizar el problema del Balanceo de Carga (que llamaremos B.C.) al problema de la tribu del agua (que llamaremos T.A.). Es decir, nuestra reducción polinomial va a ser de la forma:

$$BC \leq_p TA$$

BC plantea lo siguiente: *Dadas m Máquinas y un conjunto n de trabajos, donde cada trabajo j toma un tiempo tj, se desea asignar el trabajo en las Máquinas de forma balanceada. Dada una asignación A(i) para la Máquina i, su tiempo de trabajo es:* $T_i = \sum_{j \in A(i)} t_j$. *Y queremos encontrar la asignación que minimice el máx valor de Ti, que también representa el tiempo que se tardará en finalizar todos los trabajos.*

Si escribimos en forma de ecuación lo que plantea BC, obtenemos la siguiente expresión:

$$\min(\max(\sum_{j \in A(i)} t_j))$$

La versión de decisión de este problema dice: *Dado un número m de mquinas, una serie n de trabajos, y un número C, determinar si existe un asignación de trabajos en las m máquinas, tal que el máximo* T_i *sea menor o igual a C.*

Cabe destacar que en clase nosotros estudiamos al problema de BC como un problema NP-Hard; resta aun demostrar que es un problema NP-Completo. Vamos a resolver esto en la próxima sección.

2.3.2 Demostración de problema NP-Completo

Para demostrar que BC es un problema NP-Completo, vamos a usar el problema 2P (el cual demostramos que es NP completo en Problema 2-Partition). Es decir, vamos a realizar la reducción

$$2P \leq_p BC$$

Nosotros partimos de una serie de números $S = \{a_1, a_2, \ldots, a_n\}$ que es la entrada de 2-Partition. Por cada número a_i definimos un trabajo con duración a_i. Luego, por cada uno de los dos subconjunto S_i, definimos una máquina m_i. Finalmente, decimos que nuestro valor C va a ser la mitad de la suma de todos los a_i.

Cuando le pasemos este nuevo problema al validador de BC, este nos va a decir si es posible partir el conjunto en dos subconjuntos. Esto se debe a que solo va a ser posible partir el conjunto en dos si, como mucho, cada subconjunto tiene exactamente la mitad de la suma.

Podemos afirmar entones, que existe un 2-Partition de S si y solo si existe un balanceo de carga con T_i a lo sumo C.

$\rightarrow$ Si existe una solución para el problema de 2-partition, entonces existe una solución para el problema de balanceo de carga. Si tenemos 2 subconjuntos S_1 y S_2 tales que ambos tienen la misma suma, $C = \frac{sum(S)}{2}$, entonces existe una asignación de trabajos en 2 máquinas distintas tal que el T_i de cada una sea exacatmente C. Esto se debe a que los valores a_i representan los tiempos de cada trabajo. Cualquier otra asignación no sería mínima. Como se tienen 2 máquinas, si quitamos trabajo de una para asignárselo a la otra, la segunda tendría $T_i > C$ necesariamente porque en total la suma de los a_i es $2 \cdot C$.

$\leftarrow$ Si existe una solución que permita minimizar el T_i máximo de las máquinas, significa que este valor es a lo sumo C, debido a la definición del problema de decisión. $A(i) = (\sum_{i \in m_i} a_i)$, con $T_i = \max A(i)$. A la vez, $A(m_1) + A(m_2) = sum(S)$ porque las asignaciones están formadas por los valores a_i y todos los trabajos son asignados. Como $C = \frac{sum(S)}{2}$, entonces $A(m_1) + A(m_2) = 2 \cdot C$. Ningun $A(i)$ puede ser mayor a C porque asumimos que existe solución. Tampoco pueden ser menores a ese valor ya que no se cumpliría $A(m_1) + A(m_2) = sum(S)$. Por lo tanto, $A(m_1) = A(m_2) = C$. Esto significa que S se puede dividir en 2 subconjuntos tales que su suma sea igual y la mitad de la total del conjunto.

Ejemplo: consideramos el conjunto $S = \{3, 1, 1, 2, 2, 1\}$. Asignamos los trabajos $3, 1, 1, 2, 2, 1$ a 2 máquinas m_1 y m_2 tal que el tiempo máximo de cada una sea menor o igual a $\frac{3+1+1+2+2+1}{2} = \frac{10}{2} = 5$. El algoritmo que resuelve el problema de decisión de BC nos indicará que existe solución ($\{3, 1, 1\}$ y $\{2, 2, 1\}$), por lo que existe una partición en 2 subconjuntos.

Con esto mostramos que el problema BC es un problema NP-Completo. En la próxima sección usaremos este hecho para demostrar que TA también pertenece a los problemas NP-Completos.

2.3.3 Desarrollo

El problema de decisión de BC busca minimizar el máximo valor de las maquinas, mientras que TA busca lograr que todos los subgrupos tengan una duración pareja. La pregunta entonces es: ¿Hay relación entre estas dos búsquedas? ¡La respuesta es que si! Buscar minimizar el máximo valor de las máquinas, está implícitamente buscando lograr que todas las máquinas tengan una duración pareja. Para reducir el tiempo de uso de una máquina, se le tiene que dar alguno de sus trabajos a otra máquina. Cuantos más trabajos se puedan pasar a otra máquina (sin que esta se convierta en la nueva máquina de mayor duración), más se va a lograr minimizar el valor máximo de las maquinas. Esto implica que en el momento que se tenga todas las máquinas con similares T_i, es el momento en donde va a estar el mínimo valor máximo de T_i.

Basándonos en esto, podemos realizar la siguiente reducción: Por cada trabajo t_j, definimos un maestro agua x_i, donde la habilidad será el tiempo de ejecución del mismo. Por cada máquina m_i definimos un subgrupo S_i (m sería nuestro k). Luego tenemos C, el cual representa una cota máxima para el máximo valor de T_i. Para transformarlo a B en tiempo polinomial, tenemos que elevar C al cuadrado y sumarle a este nuevo valor todos los otros valores de T_i al cuadrado. Como todos los T_i deben ser a lo sumo C para que exista solución, podemos establecer $B = k \cdot C^2$. Esto nos va a dar una cota equivalente al problema de decisión de TA.

Entonces, después de realizar estas transformaciones, le tenemos que pasar esta instancia de problema a la caja negra que resuelve el problema de decisión de TA, la cual nos va a decir si la solución existe o no. Esto implica que, si existe una asignación de trabajos en m máquinas cuyo T_i máximo sea a lo sumo C, entonces los maestros de la tribu del agua pueden formar k grupos cuyas

habilidades son parejas.

$\rightarrow$ Si el máximo valor de T_i es a lo sumo C, significa que existen m conjuntos de trabajo cuya suma de tiempos es menor o igual a C. Como $B = k \cdot C^2$ y efectivamente se pueden obtener $m = k$ grupos, existe una asignación cuya adición de los cuadrados de la suma de cada uno sea a lo sumo B. Como la suma de un grupo representa su T_i, es posible que algunos T_i sean estrictamente menores que C (no se tratan de la máquina con mayor trabajo) en cuyo caso el resultado será menor a B.

$\leftarrow$ Supongamos que hemos particionado a los maestros del agua en k grupos tal que:

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

Demostraremos por qué ésto implica que existen asignaciones en máquinas donde $\max T_i \leq C$ mediante ejemplos. Supongamos que se tienen los tiempos de trabajos $\{2, 3, 4, 6, 2, 2\}$, 3 máquinas y $C = 7$. Existe la siguiente solución para el problema de balanceo de carga: $m_1 = (6)$, $m_2 = (2, 4)$, $m_3 = (2, 2, 3)$. Para el problema de la tribu del agua también existirá solución. Por ejemplo, $S_1 = \{2, 6\}$, $S_2 = \{2, 3\}$, $S_3 = \{2, 4\}$ es una solución válida porque $8^2 + 5^2 + 6^2 = 125 \leq 3 \cdot 7^2 = 147$. Sin embargo, la solución mínima es $S_1 = \{6\}$, $S_2 = \{2, 4\}$, $S_3 = \{2, 2, 3\}$ con coeficiente $121 < 125$. Si el algoritmo encuentra primero la primera solución, la caja negra nos devolvería `True` directamente, no cumpliendo que cada grupo tenga a lo sumo una suma C. Como sabemos que el problema BC tiene solución, no nos hacemos problema.

Veamos ahora un ejemplo en el que no existe solución para BC. Tenemos los trabajos $\{2, 3, 2, 3, 2, 3\}$, 3 máquinas y $C = 4$. La asignación $m_1 = (2, 3)$, $m_2 = (2, 3)$, $m_3 = (2, 3)$ tiene tiempos $T_i = 5 > C$. Puede haber una máquina con T_i 2, 3 o 4, pero eso significará que las restantes excedan aun más a C. Por ejemplo, si $m_1 = (2, 2)$, $m_2 = (3, 3)$, $m_3 = (2, 3)$ el máximo T_i ahora es 6. Por lo tanto, no tiene solución. El problema de la tribu del agua intentaría crear $S_1 = \{2, 3\}$, $S_2 = \{2, 3\}$, $S_3 = \{2, 3\}$ porque tiene coeficiente mínimo, pero $5^2 + 5^2 + 5^2 = 75 > 3 \cdot 4^2 = 48$. Entonces, tampoco tendría solución. De manera análoga y retomando el primer ejemplo, sabemos que si $C = 6$ no existe solución para el problema de balanceo de carga y nuestra asignación con suma mínima 121 es mayor a $k \cdot C^2 = 3 \cdot 6^2 = 108$.

2.4 Reducción del Problema 2-Partition

Como hemos demostrado que el Problema 2-Partition es NP-Completo, lo utilizaremos para realizar una segunda reducción y comprobar nuevamente que el problema de la tribu del agua es también NP-Completo. Además, debido a la propiedad transitiva de las reducciones tenemos que, como $2P \leq_p BC$ y $BC \leq_p TA$, entonces $2P \leq_p TA$.

Dados los datos de entrada del problema de 2-Partition, $\{a_1, a_2, \ldots, a_n\}$, podemos realizar la siguiente transformación polinomial que nos permitirá re-

solver el problema de decisión asociado (y enunciado previamente) utilizando la caja negra que resuelve el de la tribu del agua:

- $k = 2$ porque según la definición de 2-Partition necesitamos dividir el conjunto en dos grupos.
- La fuerzas de los maestros $\{x_1, x_2, \ldots, x_n\}$ serán $\{a_1, a_2, \ldots, a_n\}$.
- B es el valor obtenido cuando las sumas de los elementos en cada subconjunto son iguales. Es decir, suponiendo que el problema de decisión de 2-Partition tiene solución y sabiendo que en ese caso la suma de cada subconjunto es s $= \frac{sum(A)}{2}$, entonces $B = s^2 + s^2$.

Esta transformación requiere recorrer el conjunto $A = \{a_1, a_2, \ldots, a_n\}$ para crear las tuplas (maestro, habilidad), lo cual implica una operación $\mathcal{O}(n)$, que es polinomial.

El problema original de la partición tiene una solución si y solo si es posible obtener dos grupos de maestros tal que la adición de los cuadrados de las sumas de las habilidades en cada grupo sea mínima y menor o igual a B.

$\rightarrow$ Si el conjunto A puede ser particionado en 2 subconjuntos tales que la suma de los elementos en cada uno sea igual, significa que los 2 grupos de maestros estarán perfectamente parejos. Obtener conjuntos de habilidades parejas es justamente el objetivo de minimizar la adición de los cuadrados de las sumas de las fuerzas. En este caso, los cuadrados de las sumas serán iguales y, por lo tanto, la suma de los mismos será mínima. Si resolviéramos el problema de decisión para $V > B$, también llegaríamos a la conclusión de que existe solución. En cambio, si $V = B - 1$, no se puede resolver. Por lo tanto, B es el coeficiente mínimo.

$\leftarrow$ Si existe una solución del problema de la tribu del agua, significa que se puede dividir a los maestros en 2 grupos tal que la adición de los cuadrados de las sumas de cada uno es a lo sumo B. Como B es el resultado de sumar dos veces s^2, esto implica que los 2 subconjuntos tienen suma s, por lo que el problema de 2-partition tiene solución.

Ejemplo: $A = \{1, 2, 3\}$. Realizando la transformación, $k = 2$, $B = (\frac{6}{2})^2 + (\frac{6}{2})^2 = 18$ y $maestros = \{(M1, 1), (M2, 2), (M3, 3)\}$. Existen múltiples combinaciones de valores para los dos subconjuntos, pero nos quedaremos con la que minimice la suma de los cuadrados, respetando que debe ser a lo sumo B,

1. $S_1 = 1$ y $S_2 = 2, 3 \rightarrow$ El coeficiente es $1^2 + (2 + 3)^2 = 26 > B$. No es solución.
2. $S_1 = 1, 2$ y $S_2 = 3 \rightarrow$ El coeficiente es $(1+2)^2+3^2 = 18 = B$. Es solución.
3. $S_1 = 1, 3$ y $S_2 = 2 \rightarrow$ El coeficiente es $(1 + 3)^2 + 2^2 = 20 > B$. No es solución.

Como existe solución para el problema de la tribu del agua, entonces también la existe para 2-Partition.

De esta forma, hemos demostrado que el problema de la tribu del agua es al menos tan difícil como el problema de partición, que es NP-completo. Como la reducción realizada es polinomial, podemos afirmar que el problema de la tribu del agua es también NP-Completo.

3 Algoritmos y análisis de complejidad

El problema de optimización de la tribu del agua fue resuelto utilizando distintas técnicas de programación. En las próximas secciones presentaremos el código correspondiente y analizaremos la complejidad temporal de cada uno de los algoritmos planteados:

- Backtracking
- Programación Lineal
- Aproximación propuesta por la cátedra
- Aproximación adicional

3.1 Complejidad lectura de archivos

A continuación mostramos la función principal de lectura de archivos.

```
30 def generarTestDe(archivo):
31     k = None
32     maestros_y_habilidades = []
33 
34     with open(archivo, "r") as file:
35         for i, line in enumerate(file):
36             line = line.strip()
37             if line.startswith("#"):
38                 continue
39             if k is None:
40                 k = int(line)
41                 continue
42             maestro, habilidad = line.split(", ")
43             maestros_y_habilidades.append((maestro, int(habilidad)))
44 
45     return maestros_y_habilidades, k
```

La función lee la línea que contiene la cantidad de conjuntos de maestros a crear. Una vez hecho eso, lee n líneas para almacenar los distintos valores x_i de tuplas (nombre, habilidad). Esto tiene una complejidad temporal $\mathcal{O}(n)$.

3.2 Algoritmo de Backtracking

Debido al alto tiempo de ejecución de nuestra implementación inicial de backtracking, buscamos alternativas y mejoras que nos permitieran disminuirlo. Para detallar el algoritmo e indicar su complejidad, tendremos en cuenta la cantidad de maestros de la tribu que se enfrentarán a la Nación del Fuego, n, y la cantidad de grupos, k.

S $k > n$ no existe solución y si $k = 0$ no podemos formar grupos. En ambos casos nuestro algoritmo funciona en $\mathcal{O}(1)$. Cuando la cantidad de maestros es igual a la cantidad de grupos, también sabemos el resultado del problema. Entonces, como cada grupo tendrá únicamente un maestro si $n = k$, decidimos que en ese caso particular no se aplique el algoritmo de backtracking. En su lugar, resolvemos la asignación con la función `caso_k_igual_a_n`.

```
75 def caso_k_igual_a_n(maestros_y_habilidades):
76     suma = 0
77     grupos = []
78     for maestro, habilidad in maestros_y_habilidades:
79         grupos.append({maestro})
80         suma += habilidad**2
81     return grupos, suma
```

Esta función recorre la lista de maestros, implicando un costo de $\mathcal{O}(n)$. Esto nos permitió disminuir notablemente el tiempo de ejecución. Todos estos son casos particulares. Procederemos a explicar el caso general.

Con el objetivo de evitar combinaciones no óptimas, partimos del resultado de una aproximación greedy. Ésta se basa completamente en la propuesta por la cátedra, aunque con algunas modificaciones ajustadas a la necesidad del algoritmo de backtracking.

```
83 def asignacion_greedy(maestros_y_habilidades, k):
84     S = [set() for _ in range(k)]
85     sumas_grupos = {i: 0 for i in range(k)}
86     suma_maxima = 0
87 
88     for maestro_y_habilidad in maestros_y_habilidades:
89         maestro, habilidad = maestro_y_habilidad
90         grupo_menor_suma =  min(sumas_grupos,
91                                 key=sumas_grupos.get)
92         S[grupo_menor_suma].add(maestro)
93         sumas_grupos[grupo_menor_suma] += habilidad
94         suma_maxima = max(suma_maxima,
95                           sumas_grupos[grupo_menor_suma])
96 
97     coeficiente = sum(s**2 for _,s in sumas_grupos.items())
98 
99     return S, coeficiente, suma_maxima
```

La función crea una lista de sets para las asignaciones y un diccionario para guardar la fuerza de cada grupo. Ambas operaciones son lineales en la cantidad de grupos. Luego, asigna de forma iterativa cada maestro al grupo con menor habilidad. Para lograrlo, utiliza la función `min` de python sobre el diccionario `sumas_grupos`, que conlleva un costo $\mathcal{O}(k)$. Ésto es realizado para cada uno de los n maestros, por lo que el ciclo `for` tiene una complejidad $\mathcal{O}(n \cdot k)$. Utilizando el diccionario se calcula el coeficiente, lo cual también es lineal en k. El resto de las operaciones son constantes. Por lo tanto, la complejidad temporal de esta función es

$$T(n) = 3 \cdot \mathcal{O}(k) + \mathcal{O}(n \cdot k) + \mathcal{O}(1) = \mathcal{O}(n \cdot k)$$

Ahora analizaremos el código principal:

```
1  def problema_tribu_del_agua_bt_greedy(maestros_y_habilidades, k):
2      n = len(maestros_y_habilidades)
3
4      if k > n:
5          return None
6
7      if k == 0:
8          return [], 0
9
10     if k == n:
11         return caso_k_igual_a_n(maestros_y_habilidades)
12
13     maestros_y_habilidades = sorted(maestros_y_habilidades,
14                                     key=lambda x: -x[1])
15
16     S = [set() for _ in range(k)]
17     sumas_grupos = {i: 0 for i in range(k)}
18
19     mejor_S, mejor_suma, suma_maxima = asignacion_greedy(
20                                 maestros_y_habilidades, k)
21
22     resultado = problema_tribu_del_agua_bt_greedy_recur(
23         maestros_y_habilidades, 0, S, mejor_S, mejor_suma,
24         sumas_grupos, suma_maxima, 0, 0)
25
26     S_resultado, coeficiente, _ = resultado
27
28     return S_resultado, coeficiente
```

En la `línea 13` ordenamos el conjunto de maestros por mayor habilidad con Timsort cuya complejidad es $\mathcal{O}(n \cdot log(n))$. Crear los grupos vacíos y la suma por grupo en las `líneas 16 y 17` son operaciones lineales en la cantidad de grupos k, es decir, $\mathcal{O}(k)$.

Como establecimos anteriormente, utilizamos la función `asignacion_greedy` para obtener una solución inicial aproximada con la cual comparar los resultados

de la función recursiva, `problema_tribu_del_agua_bt_greedy_recur` y poder podar rápidamente si se encuentra una solución parcial peor a ésta.

```
30 def problema_tribu_del_agua_bt_greedy_recur(maestros_y_habilidades,
31     indice_actual, S_actual, mejor_S, mejor_suma, sumas_grupos,
32     suma_maxima, suma_actual, actual_suma_maxima):
33
34     if indice_actual == len(maestros_y_habilidades):
35         cota = min(actual_suma_maxima, suma_maxima)
36         return list(map(set, S_actual)), suma_actual, cota
37
38     sumas_grupos_ordenada = sorted(sumas_grupos.items(),
39                               key=lambda item: item[1])
40
41     maestro, habilidad = maestros_y_habilidades[indice_actual]
42
43     for suma_grupo in sumas_grupos_ordenada:
44         i, _ = suma_grupo
45         S_actual[i].add(maestro)
46         sumas_grupos[i] += habilidad
47         rama_explorada = False
48         if sumas_grupos[i] <= suma_maxima:
49             suma = sum(s ** 2 for s in sumas_grupos.values())
50             if suma < mejor_suma:
51                 rama_explorada = True
52
53                 actual_suma_maxima = max(actual_suma_maxima,
54                                          sumas_grupos[i])
55
56                 resultado = problema_tribu_del_agua_bt_greedy_recur(
57                 maestros_y_habilidades, indice_actual+1, S_actual,
58                 mejor_S, mejor_suma, sumas_grupos, suma_maxima,suma,
59                 actual_suma_maxima)
60
61                 nuevo_S, nueva_suma , nueva_suma_maxima = resultado
62
63                 if nueva_suma < mejor_suma:
64                     mejor_S = nuevo_S
65                     mejor_suma = nueva_suma
66                     suma_maxima = nueva_suma_maxima
67
68         sumas_grupos[i] -= habilidad
69         S_actual[i].remove(maestro)
70         if not rama_explorada:
71             break
```

La solución del problema viene dada por esta función. Su objetivo es probar

todas las combinaciones de asignaciones de maestros a los grupos de forma tal de minimizar la adición de los cuadrados de la suma de las habilidades de cada uno. Para cada maestro comprueba si, al asignarlo al grupo $i \in [0, k-1]$, se puede obtener una combinación con una suma menor a la actual. Caso contrario, poda y vuelve un paso atrás. Los k grupos están ordenados según la suma. Es decir, primero se intentará asignar al siguiente maestro más habilidoso al grupo actual con la menor fuerza total. Para ello, en la `línea 38` utilizamos Timsort, esta vez con un costo dependiente de la cantidad de grupos, $\mathcal{O}(k \cdot log(k))$. El propósito de este orden es obtener grupos parejos que conduzcan a podas más tempranas.

Una solución parcial es válida si su coeficiente es menor al del mejor resultado y la suma del grupo al que se ha agregado un maestro no supera la habilidad máxima del mismo. La asignación actual no puede tener ningún grupo cuya suma sea mayor a la más alta de la mejor solución encontrada hasta entonces porque, al finalizar de asignar a todos los maestros, el coeficiente resultante será también mayor. Si estas condiciones se cumplen, se continua resolviendo el problema para el siguiente maestro y, al regresar en la recursión, se puede intentar con otra asignación. Caso contrario, no tiene sentido seguir intentando con otros grupos porque los mismos están ordenados por suma.

El algoritmo realiza una búsqueda explícita del espacio de soluciones. La cantidad de posibles asignaciones es k^n pues, para cada uno de los n maestros, hay k opciones de grupos. Por lo tanto, la complejidad de esta función y del algoritmo en general es exponencial, más específicamente $\mathcal{O}(k^n)$.

3.3 Modelo de Programación Lineal

Debido a la dificultad de linealizar la función objetivo, nuestro modelo de programación lineal buscará, en cambio, minimizar la diferencia del grupo de mayor suma con el de menor suma. De esta forma, obtendremos una aproximación a la solución óptima. Sea Z el grupo con la mayor suma de habilidades de los maestros $\sum_i Z_i$ e Y el de menor suma, entonces se busca minimizar $\sum_i Z_i - \sum_j Y_j$.

Para resolver el problema utilizaremos programación lineal entera. A continuación detallaremos el modelo:

- Constantes $\rightarrow$ Las fuerzas de los maestros son constantes del problema. H_i es la habilidad del maestro i.
- Variables $\rightarrow$
  - $X_{i,j}$: Es una variable binaria. Vale 1 si el maestro i es asigndo al grupo j, con $i \in [0, n-1]$ o $i \in [1, n]$ y $j \in [0, k-1]$ o $j \in [1, k]$. En caso contrario, vale 0.
  - S_{min}: Representa el valor del grupo con la menor suma. Sea S_j la suma del grupo j tal que $S_j = \sum_{i=1}^{n} H_i \cdot X_{i,j}$, entonces $S_{min} = \min_j S_j$.

- S_{max}: Representa el valor del grupo con la mayor suma. $S_{max} = \max_j S_j$.

- Restricciones →
  - Asignaciones: cada maestro debe ser asignado a un solo un grupo. Luego, $\sum_{j=1}^{k} X_{i,j} = 1 \quad \forall i$.
  - Sumas: $S_{min} \leq S_j \quad \forall j$ y $S_{max} \geq S_j \quad \forall j$.

- Función objetivo → se desea minimizar $S_{max} - S_{min}$.

Podemos observar la declaración de las variables, la función objetivo y las restricciones en el código:

```
4  def problema_tribu_del_agua_pl(maestros_y_habilidades, k):
5      num_maestros = len(maestros_y_habilidades)
6
7      if k > num_maestros:
8          return None
9
10     if k == 0:
11         return [], 0
12
13     if k == num_maestros:
14         return caso_k_igual_a_n(maestros_y_habilidades)
15
16     # variables
17     X = pulp.LpVariable.dicts("X", ((i, j)
18     for i in range(num_maestros) for j in range(k)), cat='Binary')
19
20     # suma de cada grupo
21     S = [pulp.lpSum(maestros_y_habilidades[i][1] * X[i, j]
22         for i in range(num_maestros)) for j in range(k)]
23
24     # Variables: S_max = Z, S_min = Y
25     S_max = pulp.LpVariable("S_max", lowBound=0, cat='Integer')
26     S_min = pulp.LpVariable("S_min", lowBound=0, cat='Integer')
27
28     problem = pulp.LpProblem("Problema_Tribu_del_Agua_con_PL",
29                              pulp.LpMinimize)
30
31     # Restricciones para las asignaciones: cada maestro debe
32     # ser asignado a exactamente un grupo
33     for i in range(num_maestros):
34         problem += pulp.lpSum(X[i, j] for j in range(k)) == 1
35
36     # Restricciones para S_max y S_min
```

```
    # S_max debe ser mayor o igual a todas las sumas
    # S_min debe ser menor o igual a todas las sumas
    for j in range(k):
        problem += S_max >= S[j]
        problem += S_min <= S[j]

    # Función objetivo: minimizar la diferencia entre el grupo
    # con mayor habilidad y el grupo con menor habilidad
    problem += S_max - S_min

    # Resolver
    problem.solve(pulp.PULP_CBC_CMD(msg=False))

    return obtener_resultado(num_maestros, k,
                             maestros_y_habilidades, X)

def obtener_resultado(num_maestros, k, maestros_y_habilidades, X):
    resultado = [set() for _ in range(k)]
    suma_por_grupo = [0 for _ in range(k)]

    for i in range(num_maestros):
        for grupo in range(k):
            if pulp.value(X[i, grupo]) == 1:
                nombre = maestros_y_habilidades[i][0]
                habilidad = maestros_y_habilidades[i][1]
                resultado[grupo].add(nombre)
                suma_por_grupo[grupo] += habilidad

    coeficiente = sum(s**2 for s in suma_por_grupo)
    return resultado, coeficiente
```

Definimos los mismos casos particulares que en los otros algoritmos, con el objetivo de disminuir el tiempo de ejecución cuando conocemos el resultado del problema.

Crear las variables $X_{i,j}$ y calcular las sumas implican, para cada grupo, iterar sobre los n maestros que pueden ser asignados al mismo. Por lo tanto, cada una de estas operaciones conlleva $\mathcal{O}(k \cdot n)$. Con el objetivo de definir la restricción de las asignaciones, se obtiene, para cada maestro, todas las variables asociadas según el grupo. Luego, ésto también es $\mathcal{O}(n \cdot k)$. En cambio, las restricciones para S_{min} y S_{max} implican iterar por los k grupos, con un costo $\mathcal{O}(k)$.

La función `obtener_resultado` tiene dos ciclos anidados que realizan operaciones constantes para cada maestro según el grupo. La creación de las listas de `resultado` y `suma_por_grupo`, así como el cálculo del coeficiente, son operaciones lineales en la cantidad de grupos. Entonces, la complejidad temporal de la función es:

$$T(n) = 3 \cdot \mathcal{O}(k) + \mathcal{O}(n \cdot k) = \mathcal{O}(n \cdot k)$$

.

Por último, la resolución del algoritmo utilizando la biblioteca `pulp` consume tiempo exponencial porque se trata de programación lineal entera.

3.4 Algoritmos de Aproximación

3.4.1 Aproximación de la cátedra

La cátedra [1] planteaba el siguiente algoritmo de aproximación para el problema:

Generar los k grupos vacíos. Ordenar de mayor a menor los maestros en función de su habilidad o fortaleza. Agregar al más habilidoso al grupo con menos habilidad hasta ahora (cuadrado de la suma). Repetir siguiendo con el siguiente más habilidoso, hasta que no queden más maestros por asignar.

Este algoritmo lo tradujimos a código de la siguiente manera:

```
3  def problema_tribu_del_agua_aprox_catedra(maestros_y_habilidades, k):
4      n = len(maestros_y_habilidades)
5
6      if k > n:
7          return None
8
9      if k == 0:
10         return [], 0
11
12     if k == n:
13         return caso_k_igual_a_n(maestros_y_habilidades)
14
15     S = [set() for _ in range(k)]
16     maestros_y_habilidades = sorted(maestros_y_habilidades,
17                                     key=lambda x: -x[1]) #O(n log n)
18     sumas_grupos = {i: 0 for i in range(k)}
19
20     for maestro_y_habilidad in maestros_y_habilidades:
21         maestro, habilidad = maestro_y_habilidad
22         grupo_menos_habilidoso =  min(sumas_grupos,
23                                     key=lambda x: sumas_grupos[x]**2)
24         S[grupo_menos_habilidoso].add(maestro)
25         sumas_grupos[grupo_menos_habilidoso] += habilidad
26
27     coeficiente = sum(s**2 for s in sumas_grupos.values())
28
29     return S, coeficiente
```

En la línea `15` creamos los k grupos vacíos. Luego, en la línea `16` ordenamos los maestros por su habilidad/destreza, utilizando Timsort.

[1] (con la ayuda del maestro Pakku)

Luego, el ciclo `for` de las lénea `20` a `25` es el que se encarga de agregar al maestro más habilidoso al grupo con menos habilidad hasta ahora. Para calcular esto último, hacemos uso del diccionario `sumas_grupo` que guarda, para cada grupo, la suma actual del mismo.

Con respecto a la complejidad, la creación de los K grupos y del diccionario tienen un costo $\mathcal{O}(k)$. Luego, la complejidad de ordenar la lista tiene una $\mathcal{O}(n \cdot \log n)$. Además, el ciclo `for` tiene una complejidad temporal de $\mathcal{O}(n \cdot k)$, ya que por cada maestro que tiene que ser añadido a un grupo, tiene que buscar el grupo con menor habilidad acumulada hasta ahora, utilizando la función `min`, lo cual es una operación $\mathcal{O}(k)$ adicional.

Por último tenemos el cálculo del coeficiente en la línea `27`. Esto tiene una complejidad $\mathcal{O}(k)$.

$$\mathcal{T}(n) = \mathcal{O}(n \cdot \log(n)) + 3 \cdot \mathcal{O}(k) + \mathcal{O}(n \cdot k) = \mathcal{O}(n \cdot k)$$

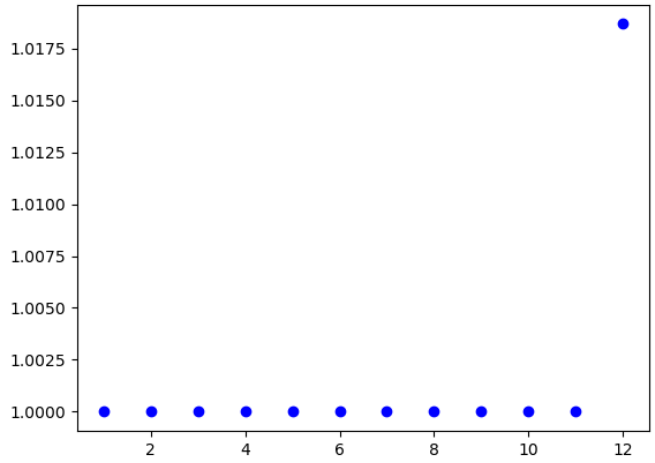
En el peor de los casos, con $n = k$, la complejidad no sería cuadratica debido a que consideramos el caso aparte.

3.4.1.1 Análisis de Aproximación

Para hacer el análisis de la aproximación, comparamos el coeficiente obtenido por el algoritmo de aproximacion contra el coeficiente obtenido por el algortimo de backtracking. Para esto, generamos 13 instancias I. Cada una de ellas tenía una cantidad ascendente de maestros, habilidades aleatorias y una cantidad k aleatoria (menor que n).

El código que usamos para generar estos tests se encuentra en el archivo `codigo/comparacion_aprox_catedran.py`.

Esto nos dio el siguiente resultado:



Definimos a $z(I)$ como la solución óptima de la instancia I y a $A(I)$ como la solución aproximada.

En el eje Y podemos los ver la proporción de $\frac{A(I)}{z(I)}$ y en el eje X la cantidad de maestros. El gráfico muestra un dato bastante muy interesante: la aproximación obtiene un coeficiente muy próximo al coeficiente real (como mucho, en el peor de los casos 1.02 veces peor). Todo esto con el beneficio de tener una velocidad de ejecución considerablemente superior a la del algoritmo de backtracking que obtiene la solución óptima.

Además, usando los ejemplos provistos por la catedra, vemos que esta cota se mantiene:

Nombre Test	Coeficiente esperado	Coeficiente obtenido	Relación
14_4	15292055	15292085	1.0000019618
14_6	10694510	10700172	1.00052943052
15_4	4311889	4317075	1.00120272113
15_6	6377225	6377501	1.00004327901
17_5	15974095	15975947	1.00011593771
17_7	11513230	11513230	1
17_10	5427764	5430512	1.00050628583
18_6	10322822	10325588	1.00026794998
18_8	11971097	12000279	1.00243770475
20_4	21081875	21083935	1.00009771427
20_5	16828799	16838539	1.00057876976
20_8	11417428	11423826	1.00056037139

De forma análoga, para los ejemplos utilizados en las mediciones, la cota corresponde a 1.007.

Para corroborar la cota encontrada, $r(A) = 1.02$, también realizamos mediciones con mayores volúmenes de datos, con $n \in \{10, 100, 1000, 10000\}$. Obtuvimos $r(A) = 1.0256009216095232 \approx 1.0256$. Los ejemplos se pueden encontrar en `codigo/cota_aprox_catedra.py`

3.4.2 Aproximación adicional

La siguiente aproximación propuesta sigue una lógica similar a la sugerida por la cátedra, pero con un tiempo de ejecución menor.

```
3  def problema_tribu_del_agua_aprox_adicional(maestros_y_habilidades,
4                                              k):
5      n = len(maestros_y_habilidades)
6
7      if k > n:
8          return None
9
10     if k == 0:
11         return [], 0
12
```

```
    if k == n:
        return caso_k_igual_a_n(maestros_y_habilidades)

    S = [set() for _ in range(k)]

    maestros_y_habilidades = sorted(maestros_y_habilidades,
                                    key=lambda x: -x[1])

    grupo_actual = 0
    sumas = [0] * k
    for maestro_y_habilidad in maestros_y_habilidades: #O(n)
        if grupo_actual == k:
            grupo_actual = 0

        nombre, habilidad = maestro_y_habilidad

        S[grupo_actual].add(nombre)
        sumas[grupo_actual] += habilidad
        grupo_actual += 1

    coeficiente = sum(s**2 for s in sumas) #O(k)
    return S, coeficiente
```

Nuevamente ordenamos los maestros por habilidad descendiente, con un costo $\mathcal{O}(n \cdot \log(n))$. Luego, este algoritmo sigue la regla sencilla de asignar de forma iterativa el maestro con la mayor habilidad al siguiente grupo en una secuencia cíclica, es decir, el maestro más habilidoso estará en el grupo 1, el k en el grupo k y el $k+1$ nuevamente en el primer grupo. El objetivo es distribuir los maestros de mayor habilidad primero y de manera uniforme a través de todos los grupos, lo cual evita la acumulación de habilidades altas en un solo grupo. El algoritmo es greedy porque sigue la estrategia mencionada para obtener un óptimo local en cada paso, dado por el grupo actual según el ciclo, con la esperanza de encontrar una solución globalmente óptima. Sin embargo, como el problema es NP-Completo y la solución exacta conlleva un costo exponencial, podemos afirmar que este algoritmo no es óptimo, sino tan solo una aproximación.

La asignación de los grupos implica iterar sobre los n maestros, lo cual es lineal, $\mathcal{O}(n)$. Por otro lado, calcular el coeficiente utilizando la lista auxiliar `sumas` conlleva $\mathcal{O}(k)$. El resto de las operaciones son constantes. En total, la complejidad del algoritmo propuesto, para el caso general, es

$$\mathcal{T}(n) = \mathcal{O}(n \cdot \log(n)) + \mathcal{O}(n) + \mathcal{O}(k) = \mathcal{O}(n \cdot \log(n))$$

El análisis de la cota es presentado en Cotas de aproximación empírica.

3.5 Efecto de las variables sobre el algoritmo

Todos los algoritmos propuestos consideran los siguientes casos

- $k > n$: No existe solución. Devolvemos `None` en $\mathcal{O}(1)$.
- $k = 0$: Todos los algoritmos se ejecutan en $\mathcal{O}(1)$, devolviendo una lista vacía y coeficiente 0.
- $k = n$: Cada grupo tendrá un solo maestro. Lo resolvemos en tiempo lineal $\mathcal{O}(n)$.
- $k < n$: Es el caso general y la complejidad depende del algoritmo utilizado. Los algoritmos exactos conllevan tiempo exponencial, aumetando considerablemente con el valor de k y n.

4 Ejemplos de ejecución

En la carpeta `ejemplos_adicionales` se pueden encontrar distintos casos de prueba que agregamos con el objetivo de comprobar la correctitud de los algoritmos propuestos. A continuación detallamos cada uno:

- `uno_por_grupo.txt` → En este caso $k = n$, por lo que cada maestro será asignado a un grupo distinto.
- `habilidades_similares.txt` → Los maestros tienen habilidades distintas, pero parejas.
- `habilidades_ascendentes.txt` → Las tuplas de maestros vienen ordenadas ascendentemente según la fuerza.
- `habilidades_descendentes.txt` → Las tuplas de maestros vienen ordenadas descendentemente según la fuerza.
- `habilidades_iguales.txt` → Los maestros tienen la misma habilidad.
- `grupos_parejos.txt` → Las habilidades de los maestros son tales que, al realizar la asignación, cada grupo tendrá la misma suma.
- `una_habilidad_alta.txt` → Uno de los maestros tiene una habilidad muy alta en comparación con la del resto.
- `k_menor_a_n.txt` → Caso general cuando $k < n$.

De forma adicional, agregamos la posibilidad de ejecutar los ejemplos utilizados para las mediciones. Los mismos se encuentran en `ejemplos_mediciones`. Como siempre, en `ejemplos_catedra` tenemos los casos de prueba provistos por la cátedra.

4.1 Ejecución del programa

En esta sección explicaremos las distintas formas de ejecutar el programa.

- `python3 codigo/main.py`: ejecutará por defecto el algoritmo de backtracking y mostrará los grupos formados, el coeficiente resultante y el tiempo de ejecución en cada caso. Se ejecutan los ejemplos adicionales, los de las mediciones y los provistos por la cátedra.
- `python3 codigo/main.py ruta_a_ejemplo`: procesará los datos del archivo dado y ejecutará todos los algoritmos, mostrando el resultado de cada uno, así como su tiempo de ejecución.
- `python3 codigo/main.py ruta_a_ejemplo --flag`: ejecutará el algoritmo según el flag utilizado. Si es inválido, por defecto actúa como el anterior.

  `--btg` → Backtracking

  `--pl` → Programación Lineal

  `--a1` → Algoritmo de aproximación propuesto por la cátedra

  `--a2` → Algoritmo de aproximación adicional
- `python3 codigo/main.py --flag`: Ejecuta todos los tests usando el algoritmo propuesto. Si es inválido, por defecto se comporta como si no tuviera el flag.
- `python3 codigo/main.py N`: Ejecuta los tests con backtracking (por defecto), pero limita los ejemplos de la cátedra a N. El objetivo es poder ejecutar la mayor cantidad de tests en un tiempo menor.
- `python3 codigo/main.py N --flag`: Se comporta como el anterior, pero solo ejecuta los tests con el algoritmo indicado.

Recomendamos utilizar:

- `python3 codigo/main.py ruta_a_ejemplo`
- `python3 codigo/main.py N --flag`

4.2 Cotas de aproximación empírica

Mediante el uso de todos los ejemplos de ejecución logramos determinar de forma empírica la cota de los algoritmos de aproximación.

Sea I una instancia cualquiera del problema, $z(I)$ una solución óptima para dicha instancia y sea $A(I)$ la solución aproximada, se define $\frac{A(I)}{z(I)} \leq r(A)$ para todas las instancias posibles. Calculamos la cota $r(A)$ obteniendo la máxima razón $\frac{A(I)}{z(I)}$ de entre todos los casos de prueba.

Los resultados obtenidos son los siguientes:

- Programación lineal $\rightarrow r(A) = 1.0141856654211776 \approx 1.01419$. En general, los resultados coincidieron con el óptimo. En otros casos, la diferencia entre las soluciones resultó ser muy pequeña.

- Aproximación de la cátedra $\rightarrow$ Con el análisis presentado en Análisis de Aproximación, $r(A) = 1.0256009216095232 \approx 1.03$.

- Aproximación adicional $\rightarrow$ Para los sets de datos de la cátedra, los usados en las mediciones y los adicionales, $r(A) = 1.2157676348547717 \approx 1.21577$. También realizamos mediciones con volúmenes de datos elevados para el algoritmo exacto, con el fin de obtener más información de la cota. Probamos casos en los que sabemos que nuestro algoritmo no es óptimo. Por ejemplo, si $k = 2$ y se tiene una habilidad muy alta y el resto bastante pequeñas. Llegamos hasta $n = 100000$. Los ejemplos se pueden encontrar en `codigo/cota_aprox_adicional.py`. El resultado obtenido fue $r(A) = 1.2999951999748 \approx 1.3$. Como era de esperarse, la aproximación es peor que las anteriores porque no tiene en consideración la habilidad de cada grupo.

5 Mediciones de tiempo

Para corroborar la complejidad algorítmica de los algoritmos implementados, realizamos una serie de mediciones. Probamos distintas combinaciones de n y k, para $n \in [0, 10]$ y $k \in [0, n]$. Para cada n tomamos en cuenta la combinación de n y k con mayor tiempo de ejecución. Tomamos estas mediciones en todos los casos para realizar comparaciones. Para los algoritmos de aproximación también se tomaron mediciones adicionales con valores de n y k más elevados.

El código que usamos para generar los tests se encuentran en el archivo `codigo/grafico_complejidad.py`.

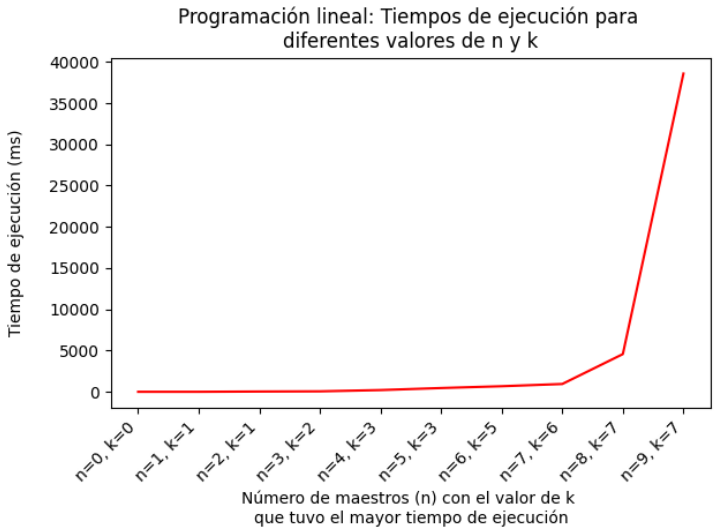
5.1 Algoritmo de backtracking

Con el algoritmo de backtracking obtuvimos los siguientes resultados:

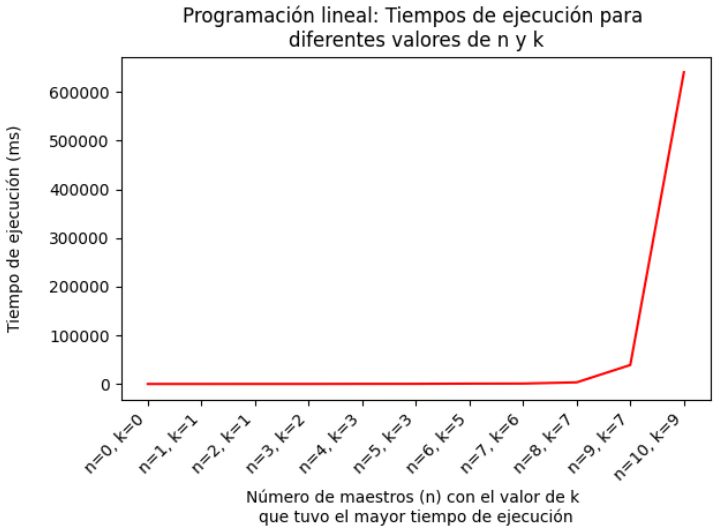
Como podemos observar en la tendencia de la curva, el tiempo de ejecución aumenta exponencialmente con la cantidad de maestros n y también depende de k. El tiempo más alto ocurre cuando k se acerca a n, con $k < n$ (es lineal cuando $k = n$). Para valores pequeños de estas variables el algoritmo es relativamente rápido. Sin embargo, al incrementarlos no se vuelve práctico debido a que el tiempo no crece polinomialmente, sino exponencialmente. Esto corrobora el análisis de la complejidad planteado previamente.

5.2 Algoritmo de programación lineal

Tomamos mediciones de tiempo con los mismos sets de datos utilizados para backtracking. Realizamos 2 gráficos para facilitar la comparación entre algoritmos.



El algoritmo de backtracking tarda aproximadamente 9000 milisegundos en ejecutarse en el peor caso para $n = 10$. Sin embargo, PLE ya necesita 40000 milisegundos para $n = 9$. Esto es una gran diferencia con backtracking que conlleva menos de 2000 milisegundos en ese caso.



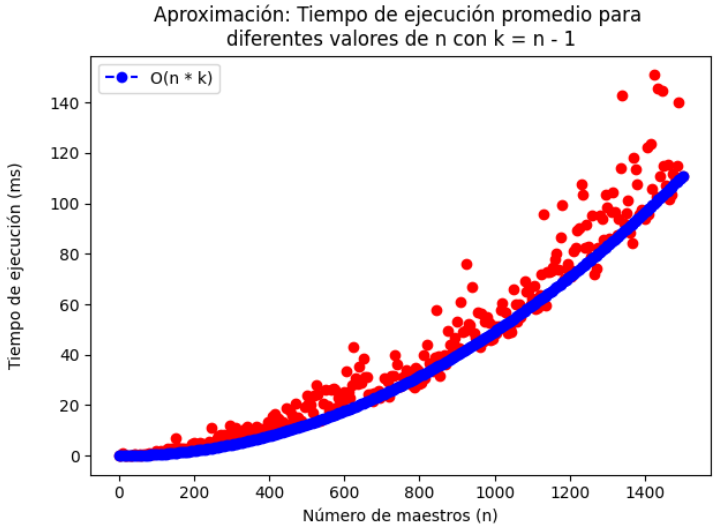
Analizando la totalidad de las mediciones para el set de datos `ejemplos mediciones`, es evidente que nuestro algoritmo de programación lineal obtuvo tiempos de ejecución significativamente mayores a los de la implementación de backtracking. En ambos casos la complejidad temporal es exponencial, lo cual puede observarse en los respectivos gráficos. No obstante, para el conjunto de datos de entrada dado, el desempeño del algoritmo de PLE es peor.

5.3 Algoritmos de Aproximación

5.3.1 Aproximación de la cátedra

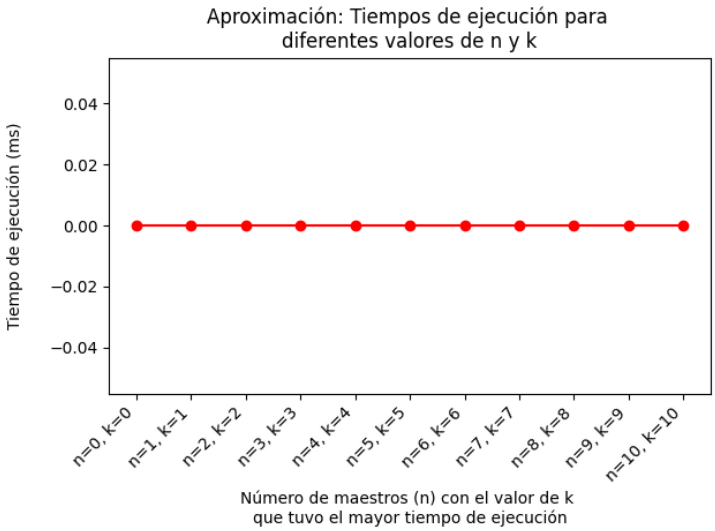
Realizamos mediciones con n y k hasta 1500 para corroborar la complejidad algorítmica de esta aproximación con valores inmanejables para el algoritmo exacto.

La curva azul representa $n \cdot k$, mientras que los puntos rojos son nuestras mediciones.



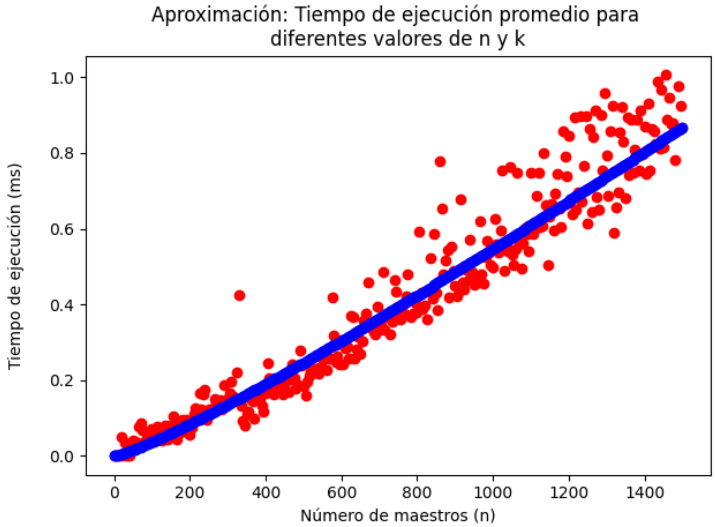
Vemos que los puntos aparentan tomar la forma de una parábola no muy pronunciada, lo cual refleja lo que obtuvimos en Aproximación de la cátedra. Si la complejidad del algoritmos fuese $\mathcal{O}(n \cdot n)$ veríamos una forma parabólica mas pronunciada, ya que la cantidad de maestros (n) suele ser considerablemente mayor que la cantidad de grupos (k). Sin embargo, como la complejidad del algoritmos es $\mathcal{O}(n \cdot k)$ y sabemos que para el caso general $k < n$, la k hace que la parábola tenga un crecimiento menos pronunciado.

Con el set de datos usado en los otros casos, este algoritmo es muy rápido.



5.3.2 Aproximación adicional

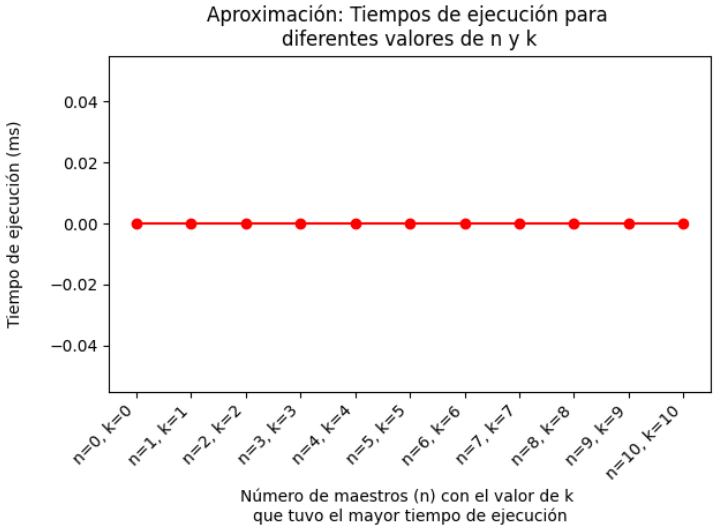
Realizamos mediciones de n y k nuevamente hasta 1500 para comprobar la complejidad algorítmica del algoritmo y compararla con el resultado del grafico anterior.



Se puede observar una curva azul que representa $n \cdot \log(n)$, en contraposición con los puntos rojos que son nuestras mediciones. Los tiempos de ejecución son notoriamente menores a los de la otra aproximación.

También realizamos un gráfico con el set de datos usado en backtracking y programación lineal para mostrar la rapidez de este algoritmo en comparación

con los exponenciales.



6 Conclusión

Hemos demostrado que el Problema de la Tribu del Agua es NP-completo, lo que implica que no se conoce [2] un algoritmo para resolverlo de manera exacta en tiempo polinomial. Los algoritmos exactos, como el presentado para backtracking, conllevan tiempo exponencial, es decir, el tiempo necesario para obtener la solución aumenta rápidamente a medida que crece el tamaño del problema, con n y k. Esto subraya la importancia de buscar soluciones aproximadas eficientes con un menor costo computacional.

A pesar de la complejidad exponencial del algoritmo de backtracking, su importancia radica en que nos permite encontrar la solución óptima siempre. Ésto es de suma importancia y puede verse reflejado fácilmente con valores manejables de n y k.

El modelo de programación lineal propuesto constituye una aproximación al problema, al minimizar una función objetivo distinta a la del problema de optimización planteado originalmente. La aproximación es generalmente buena, pero es exponencial porque utiliza programación lineal entera. Por lo tanto, al comparar el modelo con backtracking, consideramos que el segundo es mejor porque, para la misma complejidad temporal, obtiene la solución óptima en todos los casos.

A diferencia del algoritmo de programación lineal, la aproximación propuesta por la cátedra constituye una mejora significativa en cuanto al tiempo de ejecución. Si bien no siempre obtiene la asignación y el coeficiente óptimos, la disminución de la complejidad nos permite utilizarla para instancias grandes del

[2] (todavía)

problema donde el algoritmo exacto deja de ser práctico. Este algoritmo tiene una complejidad de $\mathcal{O}(n \cdot k)$; sin embargo, en general los valores de k suelen ser más chicos que n (ya que si $n \leq k$ la solución es trivial). Lo destacable de este algoritmo es que, además de tener un tiempo ejecución considerablemente menor al de backtracking, obtiene resultados solamente 1.03 veces peor que el resultado óptimo.

La aproximación adicional que implementamos es una alternativa a la anterior, que nos permite resolver el problema de manera aún más rápido. Sin embargo, al no considerar la habilidad total de cada grupo, no es una aproximación tan eficiente.

En conclusión, consideramos que, para valores pequeños y medianos de la cantidad de grupos y maestros, la mejor opción para resolver el problema es utilizar un algoritmo de backtracking que nos permita obtener la mejor solución posible. En cambio, para valores elevados, el algoritmo basado en asignar el maestro más habilidoso al grupo con menor fuerza, constituye una muy buena aproximación, con una disminución considerable del tiempo de ejecución.