Teoría de Algoritmos

(75.29) Curso Buchwald - Genender

Trabajo Práctico 3

3 de julio de 2024

Petean Marina
110564

Romano Nicolas
110830

Zielonka Axel
110310

1. Defensa de la Tribu del Agua

1.1. Análisis del problema

El problema pide encontrar la mejor manera de distribuir a los maestros agua, para poder defender a la tribu de la Nación del Fuego. El problema se resuelve por Backtracking, y por Programación Linea Entera, y además se analiza un algoritmo de aproximación para resolver el problema.

Por otro lado, se prueba que el problema de la Defensa de la Tribu del Agua es un problema NP-Completo

1.2. Pertenencia a NP y NP-Completo

1.2.1. Demostración de que se encuentra en NP

Para demostrar que el problema se encuentra en NP, se tiene que poder validar una posible respuesta al problema en tiempo polinomial. Es por esto que implementamos el siguiente validador:

```
def validador(n, k, B, S, habilidades):
    if len(habilidades) != n: return False
    suma = 0
    for s in S:
        suma += len(s)
    if suma != n: return False
    suma = 0
    for conjunto in S:
        suma_actual = 0
        for elemento in conjunto:
            suma_actual += elemento
        suma += suma_actual**2
    return suma <= B
```

donde n es la cantidad total de maestros disponibles, k es la cantidad de subgrupos en la que se van a dividir a los maestros, B es el número que se quiere chequear si el resultado final es menor o igual, S son los subgrupos en los que se dividió a los maestros y *habilidades* es el conjunto de habilidades de los maestros agua.

El validador primero chequea que el largo del arreglo de habilidades coincida con la cantidad de maestros, cosa que si no sucede ya la respuesta es errónea. Luego, se fija si la suma de los largos de los grupos en S da como resultado la cantidad de maestros, como un maestro no puede estar en 2 grupos al mismo tiempo y se necesitan usar a todos, si esta suma no da igual la respuesta es inválida. Una vez chequeadas ambas cosas, para cada subgrupo de S se realiza la suma de sus elementos, y al final se eleva al cuadrado dicha suma y se la agrega a un contador. Si la suma de los cuadrados de las habilidades de cada subconjunto es menor o igual a B, devuelve *True*, caso contrario devuelve *False*

La complejidad de este algoritmo es $O(n)$ chequear si la suma de las cantidades de los subconjuntos es igual a la cantidad de maestros disponibles, sumado a la complejidad del loop anidado que efectivamente chequea la solución, ya que son procesos diferentes entonces sus complejidades son sumadas. Como cada maestro pertenece a un solo grupo, no pueden haber mas de n habilidades. En el mejor de los casos, cada subgrupo tiene una cantidad igual de maestros, n/k. Entonces, se está iterando k veces, y para cada una de esas iteraciones, se chequean n/k elementos. Las operaciones dentro del segundo ciclo *for* son operaciones que se realizan en tiempo constante, por lo que no afectan a la complejidad. Entonces, la complejidad de esta segunda parte termina siendo $O(k*n/k)$ que es igual a $O(n)$. Y como $O(n) + O(n)$ termina siendo de vuelta $O(n)$ el validador tiene complejidad polinomial, lo que implica que el problema se puede chequear en tiempo polinomial, en este caso lineal, y por lo tanto el problema se encuentra en NP.

1.2.2. Demostración de que es NP-Completo

Una vez demostrado que el problema se encuentra en NP, ahora hay que demostrar si se encuentra en NP-Completo. Para esto, deberíamos poder encontrar una reducción polinomial de un problema que ya sepamos que es NP-Completo al problema de la Defensa de la Tribu del agua, y si se puede realizar dicha reducción habremos demostrado que el problema de la Defensa de la Tribu del Agua también está en NP-Completo.

Definimos al problema de decisión de la Defensa de la Tribu del Agua como: *"Dados un conjunto de k subgrupos, otro conjunto de n maestros con sus habilidades, y un número B, ¿existirá una división de las habilidades de los maestros en los k subgrupos tal que al sumar los cuadrados de las sumas de las habilidades de cada subconjunto el resultado sea menor o igual a B?"*.

Realizamos la reducción utilizando el problema de Balanceo de Cargas: *"Dadas m maquinas, y n trabajos a realizar, donde cada trabajo j toma un tiempo t_j, y un valor K. Cada máquina tiene una asignación A(i), cuyo valor es la sumatoria de los tiempos de los trabajos asignados a dicha máquina. ¿Existirá una asignación de trabajos a máquinas tal que la suma de los trabajos de las máquinas sea menor o igual a K?*

Observamos desde un principio que ambos problemas tienen una similaridad, y es que ambos intentan repartir n elementos entre m conjuntos de forma tal que al sumar los valores de cada subconjunto la suma total sea menor o igual a un número dado.

Utilizando las siglas BC para referirnos al problema de Balanceo de Cargas y DTA para el problema de la Defensa de la Tribu del Agua queremos entonces demostrar que:

$$BC <=_p DTA$$

Defino las cajas negras de decisión para ambos problemas:

```
BC(m, n, A, K, tiempos)
DTA(k, n, S, B, habilidades)
```

Donde, en el caso de BC: m son la cantidad de máquinas, n son la cantidad de trabajos, el arreglo A es el conjunto de asignaciones de trabajos en las maquinas (valores numéricos), K es el valor al que se busca ser menor o igual y *tiempos* es el arreglo de los tiempos que tarda cada trabajo. Y en el caso de DTA: k es la cantidad de subgrupos de ataque de la tribu, n es la cantidad de maestros disponibles, S es el conjunto de asignaciones de habilidades para cada sugrupo de ataque (valores numéricos), B es el número al que se busca ser menor o igual y *habilidades* es el arreglo de las habilidades que tiene cada maestro agua.

La cantidad de sugrupos de ataque de DTA representa a la cantidad de máquinas disponibles en BC. La cantidad de maestros agua disponibles representa a la cantidad de trabajos a realizar. El conjunto de asignaciones de maestros representa al conjunto de asignaciones de trabajos en máquinas. El número B de DTA representa al número K de BC y por último el conjunto de valores de *habilidades* representa al conjunto de valores de *tiempos*. Es decir, que hay una correspondencia directa entre los parámetros de DTA y los de BC, y que no habría que realizar ninguna modificación sobre estos para pasarlos del problema de BC al de DTA. El validador se encarga de elevar al cuadrado los valores de las sumas de cada asignación, por lo que sería válido asumir que pasar de una instancia de BC a una de DTA de la siguiente forma es correcto:

```
    BC(m,n,A,K,tiempos) --> DTA(m,n,A,K,tiempos)
```

Por lo tanto, como pudimos reducir el problema de Balanceo de Cargas, que es un problema NP-Completo, al problema de la Defensa de la Tribu del Agua, podemos afirmar que se cumple:

$$BC <=_p DTA$$

y por lo tanto que el problema de la Defensa de la Tribu del Agua es un problema NP-Completo.

1.2.3. Ejecucion de los algoritmos

Para ejecutar el algoritmo con un set de datos *.txt* deberas pararte a la misma altura que **algoritmo.py** y ejecutar el siguiente comando:

```
$ python main.py <archivo.txt> <solucion_buscada>
```

donde solución buscada debe ser **b** de *backtracking* o **pl** de *programacion lineal*

Luego te devolvera la estrategia de ataques y cargas optimas esperada junto a la cantidad de soldados eliminados correspondiente a la estrategia.

2. Resoluciones del problema

2.1. Solución por Backtracking

```
1
2  def grupos_vacios(grupos):
3      vacios = 0
4      for grupo in grupos:
5          if len(grupo) == 0: vacios += 1
6      return vacios
7
8  def puede_agregar_en_el_grupo(grupos, grupo_actual):
9      if len(grupos[grupo_actual]) == 0: return True
10     if grupo_actual < len(grupos)-1:
11         if len(grupos[grupo_actual]) != 0 and len(grupos[grupo_actual + 1]) == 0:
       return False
12     return True
13
14 def _backtrack(maestros_agua, maestros, grupos, fuerzas, fuerza_actual, mejor_coef,
        mejor_asignacion, k):
15     if len(fuerzas) > 0:
16         suma_cuadrados = sum(fuerza**2 for fuerza in fuerzas)
17     if len(maestros) == 0:
18         if suma_cuadrados <= mejor_coef:
19             mejor_coef = suma_cuadrados
20             mejor_asignacion = [grupo.copy() for grupo in grupos]
21         return mejor_coef, mejor_asignacion
22
23     #si la suma de los cuadrados ya me da mayor al mejor coeficiente conocido, no
       tiene sentido seguir por esa rama
24     if suma_cuadrados >= mejor_coef:
25          return mejor_coef, mejor_asignacion
26
27     if len(maestros) < grupos_vacios(grupos):
28         return mejor_coef, mejor_asignacion
29
30     maestro = maestros.pop()
31     for i in range(k):
32         #si ya se que no puedo agregar, no tiene sentido intentarlo
33         if puede_agregar_en_el_grupo(grupos, i):
34             grupos[i].append(maestro)
35             fuerzas[i] += maestros_agua[maestro]
36             mejor_coef, mejor_asignacion = _backtrack(maestros_agua, maestros,
       grupos, fuerzas, fuerza_actual + maestros_agua[maestro], mejor_coef,
       mejor_asignacion, k)
37             grupos[i].pop()
38             fuerzas[i] -= maestros_agua[maestro]
39     maestros.append(maestro)
40
41     return mejor_coef, mejor_asignacion
42
43 def backtrack(maestros, grupos, fuerzas, fuerza_actual, k):
44     if len(maestros) == 0:
45         return None, []
46     mejor_coef = aprox.aproximacion_tribu_agua(k,maestros)
```

```
    mejor_asignacion = []
    maestros_nombres = list(maestros.keys())
    #si tengo la misma cantidad de grupos que de maestros, la solicion es trivial
    if len(maestros) == k:
        mejor_coef = 0
        grupo_actual = 0
        for maestro in maestros:
            mejor_coef += (maestros[maestro]**2)
            grupos[grupo_actual].append(maestro)
            grupo_actual += 1
            mejor_asignacion = [grupo.copy() for grupo in grupos]
    else:
        mejor_coef, mejor_asignacion = _backtrack(maestros, maestros_nombres,
    grupos, fuerzas, fuerza_actual, mejor_coef, mejor_asignacion, k)
    return mejor_coef, mejor_asignacion
```

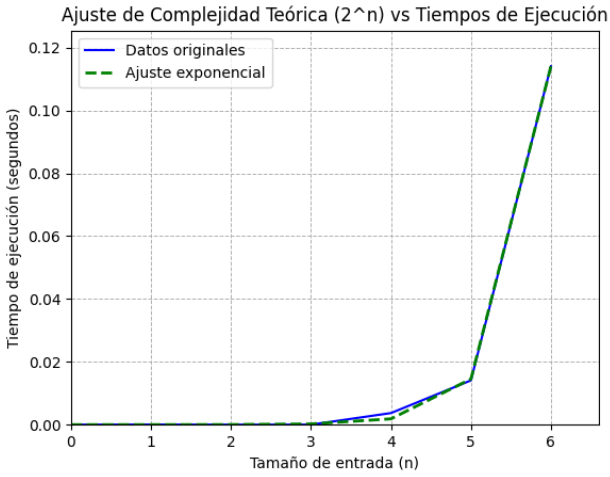

Figura 1: Gráfico de tiempos de ejecución (segundos) respecto a cantidad de grupos (n)

2.2. Solución por Programación Lineal

Para resolver este problema de asignación de maestros de agua en k grupos de manera que las sumas de las fuerzas sean lo más equilibradas posibles, vamos a formular un modelo de programación lineal entera. Mostraremos dos alternativas: una donde se obtiene el resultado optimo y otra donde se obtiene una gran aproximación.

Para ambos modelos vamos a plantear 2 variables y 2 inecuaciones. Definimos x_i como constante que representa la fuerza/maestría/habilidad del maestro de agua i.

- h_{ij} es una variable binaria que vale 1 si el maestro i es asignado al grupo j, y 0 en caso contrario.
- S_j es la suma de las fuerzas de los maestros en el grupo j.

$$\sum_{j=1}^{k} h_{ij} = 1 \quad \forall i \in \{1, 2, \ldots, n\} \tag{1}$$

Esta ecuación asegura que cada maestro i se asigna a uno y solo un grupo j.

$$S_j = \sum_{i=1}^{n} x_i \cdot h_{ij} \quad \forall j \in \{1, 2, \ldots, k\} \tag{2}$$

Definimos las sumas de fuerzas de cada grupo.

2.2.1. Modelo optimo por desviacion

La idea de este modelo sera tener una suma objetivo que idealmente queremos que cada grupo tenga para que esten equilibrados y luego minimizamos la sumatoria de las desviaciones de cada grupo respecto a esa suma objetivo. Vamos a presentar dos nuevas variables para esta idea:

- T representa una suma objetivo común que idealmente queremos que todas las sumas de los grupos se acerquen. Esta es una variable auxiliar que introduce un valor de referencia para las sumas de los grupos.
- D_j representa la desviación de la suma de las fuerzas del grupo j respecto a T.

Para cada grupo j:

$$D_j \geq S_j - T \quad \forall j \in \{1, 2, \ldots, k\} \tag{3}$$

$$D_j \geq T - S_j \quad \forall j \in \{1, 2, \ldots, k\} \tag{4}$$

En conjunto, estas dos restricciones garantizan que D_j toma el valor de la diferencia absoluta $|S_j - T|$.

Ahora, por ultimo, vamos a plantear la funcion objetivo que minimizara la sumatoria de las desviaciones esperando con esto tener la solucion optima.

$$minimize \sum_{j=1}^{k} D_j \tag{5}$$

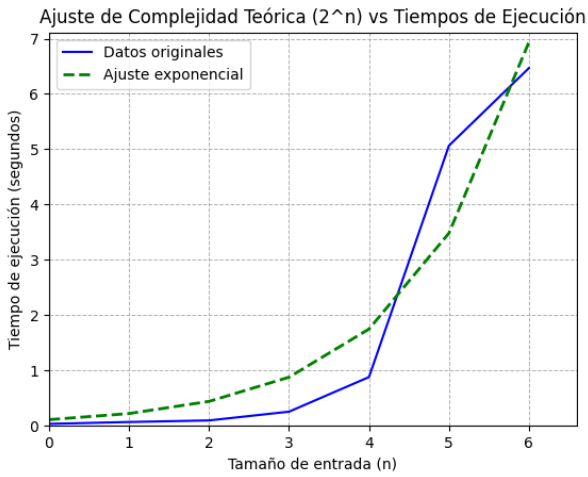

Figura 2: Gráfico de tiempos de ejecución (segundos) respecto a cantidad de grupos (n)

2.2.2. Modelo de aproximacion por minima diferencia

El objetivo de este modelo sera minimizar la diferencia del grupo de mayor suma con el de menor suma con el objetivo de llegar a una buena aproximacion, pues al minimzar la diferencia, estamos acotando aun mas la distancia que podrian tener la suma de los grupos. Para esto, vamos a presentar dos nuevas variables y 2 nuevas ecuaciones.

- Z representa la máxima suma de fuerzas entre todos los grupos.
- Y representa la mínima suma de fuerzas entre todos los grupos.

$$Z \geq S_j \quad \forall j \in \{1, 2, \ldots, k\} \tag{6}$$

$$Y \leq S_j \quad \forall j \in \{1, 2, \ldots, k\} \tag{7}$$

En conjunto, estas dos restricciones garantizan que Z toma el valor de la mayor suma entre los grupos e Y la minima.

Ahora, por ultimo, vamos a plantear la funcion objetivo que minimizara la diferencia entre Z e Y esperando con esto obtener una buena aproximacion a la solucion optima.

$$minimize\ Z - Y \tag{8}$$

2.2.3. Comparacion de Modelos

Estos sets fueron sacados del codigo para generar sets aleatorios (vease en test). Vamos a comparar el coeficiente devuelvo por ambos y el tiempo que tardá.

SET 0
por minima desviacion: coeficiente de 3193766 con 0.10272 segundos.
por minima diferencia: coeficiente de 3193766 con 0.10970 segundos.
SET 1
por minima desviacion: coeficiente de 635670 con 0.06881 segundos.
por minima diferencia: coeficiente de 635670 con 0.06283 segundos.
SET 2
por minima desviacion: coeficiente de 1479046 con 0.08676 segundos.
por minima diferencia: coeficiente de 1479046 con 0.04986 segundos.
SET 3
por minima desviacion: coeficiente de 11588892 con 669.71 segundos.
por minima diferencia: coeficiente de 11588892 con 429.83 segundos.
SET 4
por minima desviacion: coeficiente de 1708954 con 0.10029 segundos.
por minima diferencia: coeficiente de 1708954 con 0.11069 segundos.

Damos por concluido que cuando se trata de instancias del problema que tardan mucho tiempo como puede ser la del **SET 3** el algoritmo de aproximacion es mas rapido. Recalcamos tambien que es una excelente aproximacion, puesto que siempre dio la solucion optima para estos casos.

2.3. Comparacion de Programacion Lineal con Backtraking

A partir del grafico podemos ver que la resolucion por programacion lineal tarda relativamente mas que la resolucion por backtraking pero esto es solo durante los primeros valores de entrada, luego escala tanto que se hace imposible para nosotros hacer graficos. Fuimos implementando mas podas para agilizarlo, como usar la aproximacion como primera poda o identificar si puede agregar o no en el grupo pero aunque mejoro, no llegamos con el tiempo para mejorar los graficos.

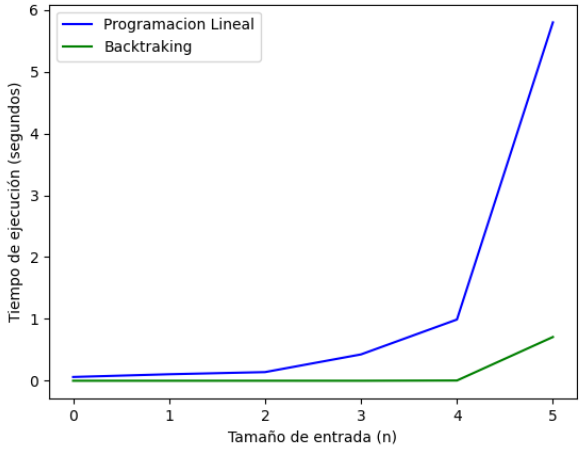

Figura 3: Gráfico de tiempos de ejecución (segundos) respecto a cantidad de grupos (n)

2.4. Solución por Aproximacion

2.4.1. Algoritmo propuesto por el maestro Pakku

```python
def aproximacion_tribu_agua(k, habilidades):
    maestros_ordenados = sorted(habilidades.values(), reverse=True)
    grupos = [[] for _ in range(k)]

    for maestro in maestros_ordenados:
        grupo_minimo = min(grupos, key=lambda x: sum(x))
        grupo_minimo.append(maestro)

    suma_cuadrados_total = sum(sum(grupo) ** 2 for grupo in grupos)

    return grupos, suma_cuadrados_total
```

La complejidad de este algoritmo es la suma de la complejidad de ordenar el arreglo de maestros, que es $O(n*log(n))$ con la complejidad de buscar el grupo con menor habilidad hasta el momento. Como en cada iteración se está modificando cuál es el grupo con menor valor de habilidades, no tiene sentido ordenar, ya que estar ordenando de vuelta en cada paso aumentaría mucho la complejidad, por eso es mejor buscar el mínimo en el arreglo de grupos desordenados, que en cada iteración es a lo sumo $O(k)$, con k la cantidad de grupos. Como se realiza esta operación para cada maestro, y son n maestros, la complejidad del ciclo es de $O(n*k)$.

Por lo tanto, sumando ambas complejidades, la complejidad del algoritmo de aproximación es de $O(n*[log(n)+k])$. Si el valor de k es mucho menor que el valor de $log(n)$ entonces se puede acotar superiormente, y quedaría $O(n*2log(n)) = O(n*log(n))$

2.4.2. Mediciones

Si bien este algoritmo no es óptimo, es una buena aproximación al resultado optimo. Al menos, eso pudimos notar empiricamente con los resultados dados por al catedra y con los sets generados y ejecutados por el algoritmo optimo. En ningun caso supero ni de cerca el 1,2 asi que lo vamos a proponer como cota *empirica*. Como prueba de lo que hablo, la division mas grande de las que hemos probado, se dio en el set dado por la catedra 20-8 donde es A(I) es 11423826 y z(I) es 11417428, por lo tando, el resultado es 1,00056. Una excelente aproximacion teniendo en cuenta que un set como ese en un algoritmo exacto nos podria llegar a tardar horas en terminar y el algoritmo aproximado en menos de un segundo nos dio ya su resultado.

3. Testing

Para poder ver si nuestro algoritmos funcionaba correctamente y poder compararlos, era necesario proveer sets en donde variaban los tamaños, así como también los valores y cantidad de maestros agua. En el directorio ***sets*** del repositorio de Github se pueden encontrar estos sets. Por razones de tiempo, los maestros solo varian entre k y 2k siendo k la representacion de cantidad de grupos perteneciente a [1,5].

Por otro lado, en ese mismo directorio del repositorio, está el archivo que contiene los tiempos y soluciones a los sets presentados con sus distintas tecnicas de diseño de algoritmo.

3.1. Generacion de set de datos y resultados esperados

Generamos nuestros set de datos a partir del siguiente algoritmo:

```
def generar_nombre_aleatorio(longitud):
    return ''.join(random.choices(string.ascii_letters, k=longitud))

def generar_maestros_aleatorios(cantidad):
    maestros = {}
    for _ in range(cantidad):
        nombre = generar_nombre_aleatorio()
        habilidad = random.randint(1, 1000)
        maestros[nombre] = habilidad
    return maestros

def asignar_valores(k):
    cantidad_maestros = random.randint(k, 2 * k)
    maestros = generar_maestros_aleatorios(cantidad_maestros)
    return maestros,k
```

Para nuestro propio set de datos, los resultados esperados son:
para **SET 0** : *3193766*.
para **SET 1** : *635670*.
para **SET 2** : *1479046*.
para **SET 3** : *11588892*.
para **SET 4** : *1708954*.

4. Conclusiones

Hemos trabajado en el Problema de la Defensa de la Tribu del Agua, demostrando su pertenencia a NP y a NP-Completo. Propusimos un algoritmo de Backtracking y dos modelos de Programación Lineal para obtener la solución óptima o una buena aproximacion. Por ultimo los comparamos entre si para determinar cual es la solucion que menos tarda. Concluimos que la aproximacion es muy ventajosa si estamos corto de tiempo y no vale la pena esperar tanto, sin embargo, si lo que queres es exactitud entonces tenes que estar dispuesto a esperar lo que haga falta. En este TP nos dimos cuenta de lo importante que es una buena poda para agilizar muchisimo el tiempo resultante.

5. Anexo I - Correcciones

5.1. Validador

```python
def validador(n, k, B, S, habilidades):
    if len(habilidades) != n:
        return False
    suma = 0
    for s in S:
        suma += len(s)
    if suma != n:
        return False
    #agregamos esto a la respuesta original:
    maestros_set = set()
    for conjunto in S:
        for maestro in conjunto:
            if maestro in maestros_set:
                return False
            maestros_set.add(maestro)
    #A partir de aca es igual al validador original
    suma = 0
    for conjunto in S:
        suma_actual = 0
        for elemento in conjunto:
            suma_actual += elemento
        suma += suma_actual**2

    return suma <= B
```

Este algoritmo funciona en $O(n)$ tanto si la respuesta ingresada es válida como si no lo es. En la primera iteración sobre los subconjuntos de S, se chequea si algún maestro ya estaba en otro grupo, que es una operación $O(1)$ ya que se está implementando un *set*. Si en esta parte no se cumple, no es necesario continuar ya que la respuesta no sería válida. En caso de que cada maestro solo se encuentre una sola vez, se procede a chequear los cuadrados de las sumas de las habilidades de los grupos.

Ambas iteraciones sobre los conjuntos de S tienen potencialmente la misma complejidad, en ambos se realizan k iteraciones (por los k grupos en los que se está dividiendo), en las que cada grupo tiene en promedio n/k maestros. Entonces, como son loops anidados, las complejidades se multiplican, quedando $O(n)$. Finalmente, como ambos recorridos por los subconjuntos de S no son anidados, sus complejidades se suman, quedando una complejidad final de $O(n)$

5.2. Reducción

Vamos a reducir el problema de 2-Partition al problema de la Tribu del Agua. Si podemos demostrar que esta reducción se puede hacer, y que el problema de la Tribu del Agua puede resolver el problema de 2-Partition, entonces el problema de la Tribu del Agua es al menos tan difícil como el 2-Partition. Dado que el 2-Partition $\in$ NP-Completo (véase en Anexo II), entonces el problema de la Tribu del Agua también pertenecerá a NP-Completo.

2-Partition: dado un conjunto de enteros positivos $A = \{a_1, a_2, \cdots, a_n\}$, determinar si es posible dividirlos en dos subconjuntos S_1 y S_2 tal que la suma de los elementos en el primero sea igual a la suma de los elementos en el segundo.

Vamos a definir una instancia del problema de la Tribu del Agua con $A = \{x_1, x_2, \cdots, x_n\}$ como los magos y su nivel de habilidad, $k = 2$ como número de grupos y el valor $B = \frac{1}{2}\left(\sum_{i=1}^{n} x_i\right)^2$ que se usará para el problema de decisión.

La partición en dos subconjuntos con sumas iguales implica que

$$\left(\sum_{x_j \in S_1} x_j\right) = \left(\sum_{x_j \in S_2} x_j\right) = \frac{1}{2}\sum_{i=1}^{n} x_i.$$

Es importante entender que si podemos dividir el conjunto A en dos subconjuntos S_1 y S_2 con sumas iguales, entonces la suma total de los elementos en S_1 será igual a la suma total de los elementos en S_2. Esto significa que $\left(\sum_{x_j \in S_1} x_j\right)^2 + \left(\sum_{x_j \in S_2} x_j\right)^2$ será igual a $2\left(\frac{1}{2}\sum_{i=1}^{n} x_i\right)^2 = \frac{1}{2}\left(\sum_{i=1}^{n} x_i\right)^2$.

Por lo tanto, si podemos particionar el conjunto A en dos subconjuntos con sumas iguales, entonces la suma de los cuadrados de las sumas de los elementos de estos subconjuntos será igual a B. Dado que $B = \frac{1}{2}\left(\sum_{i=1}^{n} x_i\right)^2$, podemos concluir que $\sum_{i=1}^{2}\left(\sum_{x_j \in S_i} x_j\right)^2 \leq B$.

En resumen, si existe una partición del conjunto A en dos subconjuntos S_1 y S_2 con sumas iguales, entonces la condición del problema de la Tribu del Agua se cumple para $k = 2$ y $B = \frac{1}{2}\left(\sum_{i=1}^{n} x_i\right)^2$. Por otro lado, si podemos resolver el problema de la Tribu del Agua para $k = 2$ y $B = \frac{1}{2}\left(\sum_{i=1}^{n} x_i\right)^2$, entonces podemos usar esta solución para encontrar una partición del conjunto A en dos subconjuntos S_1 y S_2 con sumas iguales.

Dado que la reducción se puede realizar en tiempo polinomial, esto demuestra que el Problema de la Tribu del Agua es NP-Completo. La reducción de un problema NP-Completo conocido (2-Partition) a nuestro problema objetivo (la Tribu del Agua) demuestra la dificultad del problema, confirmando su clasificación dentro de NP-Completo.

5.3. T del Modelo optimo por desviacion

Formalmente, T representa una suma objetivo com un que idealmente queremos que todas las sumas de los grupos se acerquen. Esta es una variable auxiliar que introduce un valor de referencia para las sumas de los grupos. Para una instancia cualquiera del problema, T es un unico valor para todos los subconjuntos que se utiliza para medir la diferencia. Por eso, cuando minizamos la sumatoria de diferencias, nos conviene que todas las diferencias sean chicas, por lo que el modelo situara al ideal donde le convenga y lo deja "fijo".

5.4. Algoritmo de aproximacion con heap

Hemos actualizado el algoritmo, utilizando un heap en vez de rebuscar para mejorar la complejidad, antes era $O(n^2)$ y ahora $O(k + nlogk)$.

```python
import heapq

def aproximacion_tribu_agua(k, habilidades):
    maestros_ordenados = sorted(habilidades.values(), reverse=True)

    heap = [(0, i) for i in range(k)]
    heapq.heapify(heap)

    grupos = [[] for _ in range(k)]

    for maestro in maestros_ordenados:
        suma_actual, grupo_index = heapq.heappop(heap)
        grupos[grupo_index].append(maestro)
        nueva_suma = suma_actual + maestro
        heapq.heappush(heap, (nueva_suma, grupo_index))

    suma_cuadrados_total = sum(sum(grupo) ** 2 for grupo in grupos)

    return suma_cuadrados_total
```

5.5. Sets de datos mas largos

Agregamos en la carpeta sets del repositorio 10 nuevos sets de pruebas que contienen aleatoriamente de 5 a 20 maestros con habilidades entre 1 y 1000 y cantidad de grupos aleatorio entre 5 y 10. Por lo que en total, tendremos 20 sets de pruebas hechos por nosotros.

6. Anexo II - Demostración: 2-Partition ∈ NP-Completo

Validador:

```
def validador(valores, S1, S2):
    if len(S1) + len(S2) != len(valores): return False
    if sum(S1) != sum(S2) return False
    cantidades = {}
    for elemento in valores: #O(n) siendo n cantidad de elementos en universo
        if elemento in cantidades.keys():
            cantidades[elemento] += 1
        else: cantidades[elemento] = 1

    for elemento in S1: #O(n) siendo n cantidad de elementos en conjunto 1
        if elemento not in valores: return False
        cantidades[elemento]-=1

    for elemento in S2: #O(n) siendo n cantidad de elementos en conjunto 2
        if elemento not in valores: return False
        cantidades[elemento]-=1

    for elemento in cantidades: #O(n) siendo n la cantidad de elementos
diferentes en universo
        if cantidades[elemento] < 0: return False
    return True
```

La complejidad es $O(n)$, ya que todas las operaciones que se realizan son en procesos distintos, por lo que sus complejidades se suman. Crear el diccionario es $O(n)$, y luego, como ya se valido que entre S_1 y S_2 no es mayor a la cantidad de elementos en el universo, en cada recorrido por los subconjuntos no se va a iterar más de n veces. Por lo que la complejidad final es $O(n)$, siendo n la cantidad total de valores. Como el validador es polinomial, el 2-Partition está en NP

Problema de Subset Sum

Dado un conjunto de enteros $S = \{s_1, s_2, \ldots, s_n\}$ y un entero objetivo T, el problema de *Subset Sum* consiste en determinar si existe un subconjunto $S' \subseteq S$ tal que la suma de sus elementos sea exactamente T, es decir,

$$\sum_{s_i \in S'} s_i = T.$$

Problema de 2-Partition

Dado un conjunto de enteros $S = \{s_1, s_2, \ldots, s_n\}$, el problema de *2-Partition* consiste en determinar si S puede ser particionado en dos subconjuntos S_1 y S_2 tal que la suma de los elementos de S_1 sea igual a la suma de los elementos de S_2, es decir,

$$\sum_{s_i \in S_1} s_i = \sum_{s_i \in S_2} s_i.$$

Reducción de Subset Sum (es NP-Completo) a Partition Problem

Para reducir *Subset Sum* a *2-Partition*, consideremos una instancia del problema de *Subset Sum*, donde tenemos el conjunto $S = \{s_1, s_2, \ldots, s_n\}$ y un entero objetivo T. Vamos a construir una instancia del problema de *2-Partition* de la siguiente manera:

1. Calculemos la suma total de los elementos del conjunto S:

$$\text{sum}(S) = \sum_{i=1}^{n} s_i.$$

2. Construyamos un nuevo conjunto $S' = S \cup \{2T - \text{sum}(S)\}$, donde añadimos el elemento $2T - \text{sum}(S)$ al conjunto original S.

3. Ahora, la instancia del problema de *2-Partition* se define sobre el conjunto S'.

Justificación

Supongamos que podemos resolver la instancia de *2-Partition* en el conjunto S'. Esto significa que podemos dividir S' en dos subconjuntos S_1 y S_2 tales que:

$$\sum_{s_i \in S_1} s_i = \sum_{s_i \in S_2} s_i.$$

Dado que S' contiene el elemento adicional $2T - \text{sum}(S)$, la única manera de obtener una partición balanceada es que uno de los subconjuntos, digamos S_1, contenga exactamente los elementos cuyo total es T. En consecuencia, los elementos restantes en S_2 también sumarán T.

Por lo tanto, si podemos resolver *2-Partition* en el conjunto S', entonces podemos resolver *Subset Sum* en el conjunto original S con el objetivo T.