.UBAfiuba
FACULTAD DE INGENIERÍA

Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico 3
Programación Dinamica

4 de julio de 2024

Santiago Varga
110561

Morena Sandroni
110205

Nahuel Cellini Rotunno
103320

1. Introducción

El objetivo de este trabajo práctico consta de varios puntos de distinta índole. En primer lugar, analizaremos el Problema de la Tribu del Agua en pos de demostrar teórica y empíricamente que este es un problema NP-Completo. Por otro lado, hallaremos un algoritmo que resuelva el problema de forma óptima en su versión de optimización. Por último, escribiremos un modelo de programación lineal que resuelva este problema, como también implementaremos un algoritmo de aproximación y analizaremos su complejidad y que tan aproximado es el mismo.

2. Problema de la Tribu del Agua como Problema NP-Completo

La demostración a continuación consta de dos apartados. Primeramente el problema en cuestión, en su versión de decisión, debe poder verificarse con un algoritmo certificador en tiempo polinomial, lo que se conoce como un certificador eficiente.

Una vez terminado este primer paso, la segunda parte de la demostración se basa en la capacidad de realizar una reducción polinomial desde un problema NP-Completo hacia nuestro problema a evaluar. En este caso, la reducción a ejecutar podría resumirse a que: teniendo una 'caja negra' que resuelve nuestro Problema de la Tribu del Agua, exista una forma de usar la misma para resolver un problema NP-Completo mediante una cantidad modificaciones y llamados polinomiales a esta misma caja.

2.1. Demostración de que el Problema de la Tribu del Agua se encuentra en NP

Para obtener un certificador eficiente del problema de decisión planteado debemos demostrar que, dada una partición de la secuencia de habilidades x1,x2,...,xn en k subgrupos S1,S2,...,Sk podemos verificar si la suma de los cuadrados de las sumas de los subgrupos es menor o igual a B en tiempo polinomial. El certificador eficiente debería funcionar de la siguiente manera:

- Para cada subgrupo Sj, calcular la suma de las habilidades xSjx
- Elevar al cuadrado cada una de estas sumas.
- Sumar los cuadrados de las sumas.
- Comparar la suma total de los cuadrados con B.

Este procedimiento toma tiempo polinomial en relación al número de elementos n porque involucra:

- Recorrer todos los elementos para sumarlos dentro de sus subgrupos (lineal en n).
- Elevar al cuadrado k sumas (constante si k es una constante o lineal en k).
- Sumar los cuadrados (constante si k es una constante o lineal en k).
- Comparar con B (constante).

Y este podria ser el codigo:

```python
def certificador_eficiente(subgrupos, B):
    suma_cuadrados = 0

    # Para cada subgrupo S_j, calcular la suma de las habilidades y elevarla al cuadrado
    for subgrupo in subgrupos:
        suma_subgrupo = sum(subgrupo)
        suma_cuadrados += suma_subgrupo ** 2

    # Comparar la suma total de los cuadrados con B
    return suma_cuadrados <= B
```

Dado que el proceso de verificación puede realizarse en tiempo polinomial, el Problema de la Tribu del Agua está en NP.

2.2. Demostración de que el Problema de la Tribu del Agua es NP-Completo

Para demostrar que el Problema de la Tribu del Agua es NP-Completo, podemos hacer una reducción desde un problema visto en la cursada que sea NP-Completo. Utilizaremos en este caso el problema de Subset Sum, el cual es un problema que se ha demostrado en clase que es NP-Completo.

El problema de Subset Sum en su versión de decisión se podría resumir de la siguiente manera: Dado un conjunto de enteros A=a1,a2,. . . ,an y un entero T, el problema es determinar si existe un subconjunto de A cuya suma es exactamente T.

Tomando esta instancia como punto de partida veremos si es posible una reducción de este problema al Problema de la Tribu del Agua.

1. Elementos y subgrupos:
   - Los elementos x_i serán los mismos a_i del problema Subset Sum.
   - Establecemos k subgrupos.
2. Construcción de la instancia:
   - La instancia del Problema de la Tribu del Agua será encontrar si podemos particionar $\{a_1, a_2, \ldots, a_n\}$ en k subgrupos $S_1, S_2, \ldots, S_k$ tal que la suma de los cuadrados de las sumas de los subgrupos sea menor o igual a $T^2 + \left(\sum_{a_i \in A} a_i - T\right)^2$.

Demostración de equivalencia:

1. Elementos y subgrupos:
   - Los elementos x_i serán los mismos a_i del problema Subset Sum.
   - Establecemos k subgrupos.
   - Si existe un subconjunto de A cuya suma es exactamente T, entonces podemos particionar A en k subgrupos tal que la suma de los elementos de uno de los subgrupos sea T y la suma de los elementos de los otros subgrupos sea $\sum_{a_i \in A} a_i - T$.
   - En este caso, la suma de los cuadrados de las sumas de los subgrupos será $T^2 + \left(\sum_{a_i \in A} a_i - T\right)^2$.
   - La cota B será entonces $T^2 + \left(\sum_{a_i \in A} a_i - T\right)^2$ ya que esa sería la relación directa del resultado en el Problema del Agua con el del resultado de Subset Sum.
   - Si podemos particionar A en k subgrupos tal que la suma de los cuadrados de las sumas de los subgrupos es menor o igual a $T^2 + \left(\sum_{a_i \in A} a_i - T\right)^2$, entonces uno de los subgrupos debe tener una suma igual a T porque esto minimizará la suma de los cuadrados.
2. Construcción de la instancia:
   - La instancia del Problema de la Tribu del Agua será encontrar si podemos particionar $\{a_1, a_2, \ldots, a_n\}$ en k subgrupos $S_1, S_2, \ldots, S_k$ tal que la suma de los cuadrados de las sumas de los subgrupos sea menor o igual a $T^2 + \left(\sum_{a_i \in A} a_i - T\right)^2$.

Debido a que se encontró un certificador eficiente del problema planteado y que se logró obtener una reducción de un problema NP-Completo a este mismo, entonces podemos llegar a la conclusión de que el Problema de la Tribu del Agua es un problema NP-Completo.

3. Solución óptima por Backtracking

Como una solución al problema en su version de optimizacion llegamos al siguiente codigo con la técnica de backtracking:

```
def calcular_objetivo(grupos):
    return sum(sum(num for _, num in grupo) ** 2 for grupo in grupos)

def backtracking(x, k, grupos_actuales=None, indice_actual=0, mejor_valor=float('
    inf'), mejor_grupo=None):
    if grupos_actuales is None:
        grupos_actuales = [[] for _ in range(k)]

    # Si hemos asignado todos los elementos, evaluamos esta partici n
    if indice_actual == len(x):
        valor = calcular_objetivo(grupos_actuales)
        if valor < mejor_valor:
            mejor_grupo = [[nombre for nombre, _ in grupo] for grupo in
    grupos_actuales]
            return valor, mejor_grupo
        else:
            return mejor_valor, mejor_grupo

    valor_actual = calcular_objetivo(grupos_actuales)  # Calcular el valor objetivo
     actual
    for i in range(k):
        grupos_actuales[i].append(x[indice_actual])

        # Evitar agregar si la suma actual ya supera el mejor valor conocido
        if valor_actual < mejor_valor:
            valor, grupo = backtracking(
                x, k, grupos_actuales, indice_actual + 1, mejor_valor,mejor_grupo)

            if valor < mejor_valor:
                mejor_valor, mejor_grupo = valor, grupo

        grupos_actuales[i].pop()

        # Si se a ade a un grupo vac o , no tiene sentido intentar con otros
    grupos vac os
        if not grupos_actuales[i]:
            break

    return mejor_valor, mejor_grupo
```

Esta técnica se basa en probar todas las combinaciones posibles de una secuencia de forma que se obtenga la que, en este caso, minimice la suma de los cuadrados de las sumas de cada subgrupo. La diferencia de este algoritmo con uno de fuerza bruta tradicional es que este presenta podas que, aunque no modifican su complejidad exponencial, aligeran el proceso evitando evaluar posiciones o pasos innecesarios una vez alcanzada una solución errónea.

En nuestro caso, las podas realizadas evitan agregar conjuntos vacíos, lo cual reduce en gran medida el tiempo de ejecución al no entrar en estos casos. Por otro lado, también impedimos agregar a la solución un grupo si su sumatoria ya supera el mejor valor conocido, el cual se debe minimizar.

El valor de estas podas se puede analizar empíricamente ya que, para ejemplificar, una de las pruebas que antes nos hubiera tardado cincuenta segundos gracias a estas nos tardó menos de un segundo.

4. Modelo de Programación Lineal

Para este punto tomaremos la segunda alternativa propuesta, la cual es minimizar la diferencia entre el grupo con la mayor suma de habilidades y el grupo con la menor suma de habilidades. En este caso, la solución será una aproximación al resultado óptimo obtenible con un algoritmo de backtracking.
Como apartado a aclarar, al ejecutar el solver Simplex nos indicaba una alerta que estaba por ser descontinuado:

```
<ipython-input-4-dff1c723c1b4>:38: DeprecationWarning:

‘method='simplex'‘ is deprecated and will be removed in SciPy

1.11.0. Please use one of the HiGHS solvers (e.g. ‘method='highs'‘) in new code.

res = linprog(c, A_ub=A_ub, b_ub=b_ub, A_eq=A_eq, b_eq=b_eq, bounds=bounds, method='simplex')
```

por lo que buscamos una alternativa optando por el solver HiGHS que resuelve el problema de igual manera.
Variable de decisión:

- x_{ij}: Variable binaria que indica si el maestro i está en el subgrupo j (1 si está, 0 si no está).

Modelo:

Sabemos que cada maestro está asignado a exactamente un subgrupo y todos deben ser asignados a uno. Definiremos las sumas de cada subgrupo como S_j y M y m como las sumas máximas y mínimas de todos los grupos. Tal que:

$$M \geq S_j \quad \forall j \in \{1, \ldots, k\}$$
$$m \leq S_j \quad \forall j \in \{1, \ldots, k\}$$
$$Minimizar : (M - m)$$

Luego de obtener los resultados, podemos evaluar la eficiencia de ambos métodos y la calidad de las soluciones. En cuanto a tiempo de ejecución, el método implementado por programación lineal superó ampliamente al realizado por backtracking, llegando a realizar la última de las pruebas en menos de un segundo.

Por otro lado, en lo que refiere a la exactitud del algoritmo, habiendo probado 10 casos distintos dentro de las pruebas, observamos que el algoritmo presentado se excede (en el peor de los casos) en un 43 % el valor óptimo original obtenido por nuestra función de backtracking.

5. Modelo de Aproximaciòn

El algoritmo propuesto por el Maestro Pakku sigue los siguientes pasos:

1. Ordenar a los maestros de mayor a menor según su habilidad.
2. Asignar el maestro con mayor habilidad al grupo con la menor suma de habilidades hasta el momento.
3. Repetir hasta que todos los maestros estén asignados.

Y su codigo es el siguiente:

```python
def suma_cuadrados(sumas_grupos):
    return sum(suma ** 2 for suma in sumas_grupos)

def distribuir_maestros(maestros, k):
    maestros_ordenados = sorted(maestros, key=lambda x: x[1], reverse=True)
    grupos = [[] for _ in range(k)]
    sumas_grupos = [0] * k

    for nombre, habilidad in maestros_ordenados:
        indice_grupo = sumas_grupos.index(min(sumas_grupos))
        grupos[indice_grupo].append(nombre)
        sumas_grupos[indice_grupo] += habilidad

    return suma_cuadrados(sumas_grupos), grupos
```

Este algoritmo tiene una complejidad de O(n log n+ n log k) debido a la ordenación y la asignación eficiente utilizando estructuras de datos adecuadas.

Para evaluar la efectividad del algoritmo de aproximación, se puede comparar con la solución óptima obtenida a través del algoritmo exacto de backtracking para ello definimos:

- I: Una instancia del problema.
- $z(I)$: La solución óptima para dicha instancia.
- $A(I)$: La solución aproximada obtenida mediante el algoritmo.

Se define la relación de aproximación $\frac{z(I)}{A(I)} \leq r(A)$ para todas las instancias posibles. El objetivo es medir empíricamente esta relación.

Para llevar a cabo las mediciones se plantean 3 tipos de sets de datos, los archivos brindados por la cátedra menores a 15 maestros y 6 grupos, datos aleatorios generados por una función auxiliar, y finalmente, sets de volúmenes de datos ya inmanejables, de los cuales ya conocemos la solución óptima dado que son aquellos mayores a 17 maestros y 5 grupos brindados por la cátedra.

Luego de obtener los resultados, podemos evaluar la eficiencia de ambos métodos y la calidad de las soluciones. En cuanto a tiempo de ejecución, el método de aproximación superó ampliamente al realizado por backtracking, llegando a realizar la última de las pruebas en menos de un segundo.

Por otro lado, en lo que refiere a la exactitud del algoritmo, habiendo probado en los 3 tipos diferentes de sets de datos, observamos que el algoritmo presentado se excede (en el peor de los casos) en un 0.24 % el valor óptimo original obtenido por nuestra función de backtracking. Lo que fue calculado de la siguiente forma:

```python
# Calcular el porcentaje de exceso
exceso_porcentaje = ((valor_obtenido - valor_optimo) / valor_optimo) * 100
```

6. Modelo de Aproximacion Equilibrada

El algoritmo de Particionamiento Equilibrado sigue estos pasos:

1. Ordenar a los maestros por habilidad en orden decreciente.
2. Utilizar una estructura de datos de cola de prioridad para siempre agregar el siguiente maestro al grupo con la menor suma actual.
3. Repetir hasta que todos los maestros estén asignados.

Este algoritmo tiene una complejidad de $O(n \log n + n \log k)$ debido a la ordenación inicial y el uso de una cola de prioridad para mantener los grupos ordenados por suma.
Y su codigo es el siguiente:

```python
import heapq

def suma_cuadrados(sumas_grupos):
    return sum(suma ** 2 for suma in sumas_grupos)

def distribuir_maestros_equilibrio(maestros, k):
    maestros_ordenados = sorted(maestros, key=lambda x: x[1], reverse=True)
    grupos = [[] for _ in range(k)]
    sumas_grupos = [0] * k
    heap = [(0, i) for i in range(k)]  # (suma,  ndice  del grupo)
    heapq.heapify(heap)

    # Asigna de manera alternada para equilibrar la carga
    for nombre, habilidad in maestros_ordenados:
        suma_actual, indice_grupo = heapq.heappop(heap)
        grupos[indice_grupo].append(nombre)
        suma_actual += habilidad
        sumas_grupos[indice_grupo] = suma_actual
        heapq.heappush(heap, (suma_actual, indice_grupo))

    return suma_cuadrados(sumas_grupos), grupos
```

Para evaluar la efectividad del algoritmo de aproximación, se puede comparar con la solución óptima obtenida a través del algoritmo exacto de backtracking para ello definimos:

- I: Una instancia del problema.
- $z(I)$: La solución óptima para dicha instancia.
- $A(I)$: La solución aproximada obtenida mediante el algoritmo.

Se define la relación de aproximación $z(I)/A(I) \leq r(A)$ para todas las instancias posibles. El objetivo es medir empíricamente esta relación.

Para llevar a cabo las mediciones se plantean 3 tipos de sets de datos: los archivos brindados por la cátedra menores a 15 maestros y 6 grupos, datos aleatorios generados por una función auxiliar, y finalmente, sets de volúmenes de datos ya inmanejables, de los cuales ya conocemos la solución óptima dado que son aquellos mayores a 17 maestros y 5 grupos brindados por la cátedra.

Luego de obtener los resultados, podemos evaluar la eficiencia de ambos métodos y la calidad de las soluciones. En cuanto a tiempo de ejecución, el método de aproximación superó ampliamente al realizado por backtracking, llegando a realizar la última de las pruebas en menos de un segundo. Por otro lado, en lo que refiere a la exactitud del algoritmo, habiendo probado en los 3 tipos diferentes de sets de datos, observamos que el algoritmo presentado se excede (en el peor de los casos) en un 0.24 % el valor óptimo original obtenido por nuestra función de backtracking.

Aunque la cota superior resultante es igual a la del algoritmo de aproximacion anteriormente planteado, el algoritmo de Particionamiento Equilibrado puede ser mejor debido al uso de una cola de prioridad, que garantiza una distribución más equilibrada de los maestros en cada paso. Esto es particularmente útil en casos donde los valores de habilidad varían significativamente, ya que la cola permite ajustar dinámicamente la asignación, minimizando la varianza en las sumas de los subgrupos. Además, la eficiencia en tiempo de ejecución del algoritmo equilibrado es generalmente superior para un gran número de subgrupos, debido a la reducción en la cantidad de operaciones necesarias para encontrar el grupo con la suma mínima en cada iteración.

7. Mediciones

Se corrieron los algoritmos para diferentes cantidades de elementos y diferentes cantidades de grupos, y obtuvimos los siguientes resultados:

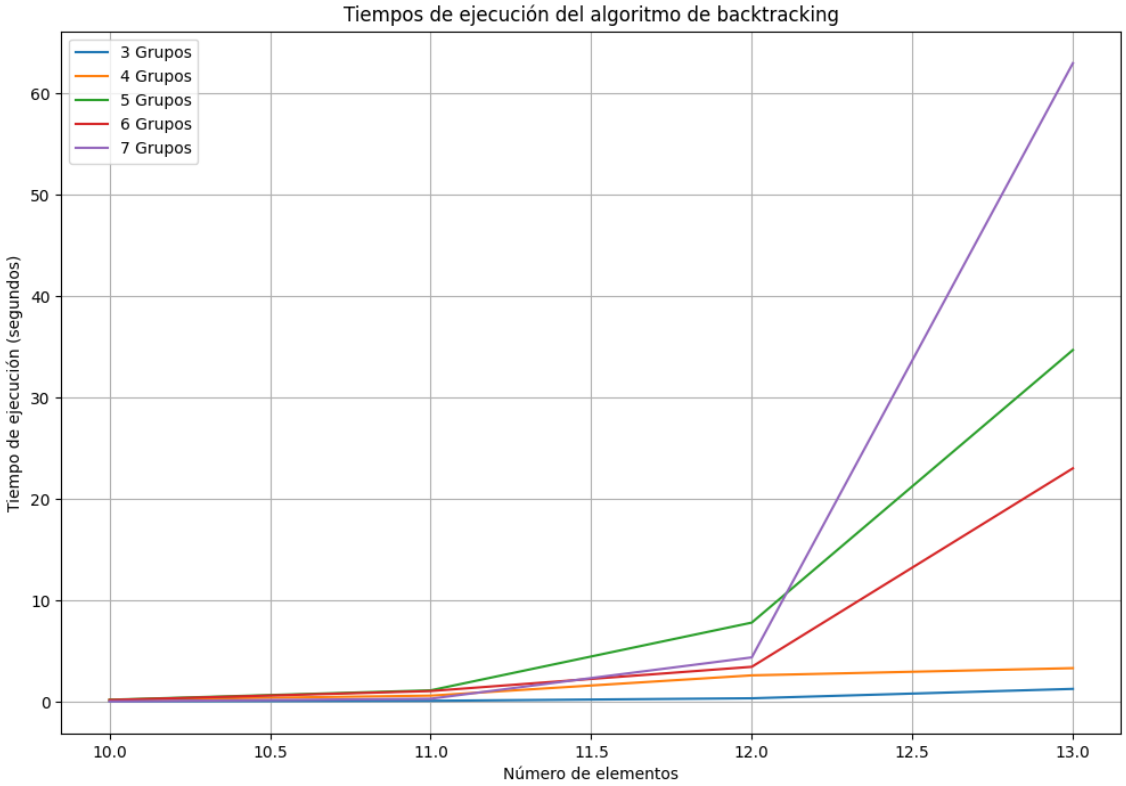

Grafico 1. Tiempo de ejecucion del algoritmo de backtracking

Luego de correr el algoritmo de backtracking observamos que a partir de los 10 elementos es cuando el algoritmo comienza a tardar mas, sobretodo a partir de los 5 grupos, pasando de tardar menos de 5 segundos para 13 elementos y 4 grupos, a mas de 20 para la misma cantidad de elementos pero con 5 grupos.

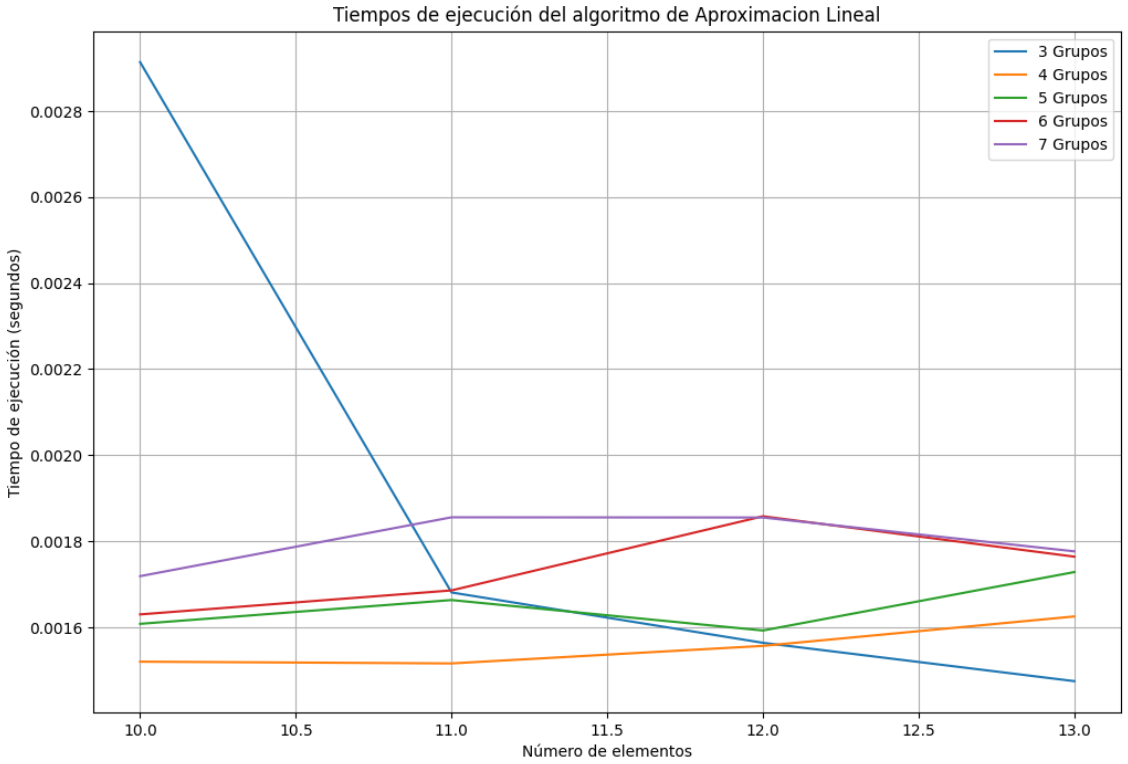

Grafico 2. Tiempo de ejecucion del algoritmo de Programacion Lineal

En el caso del algoritmo de Programacion lineal podemos ver que su performance no depende de la cantidad de elementos ni de la cantidad de grupos. Los picos que se observan pueden deberse a factores que no estan dentro de nuestro control.

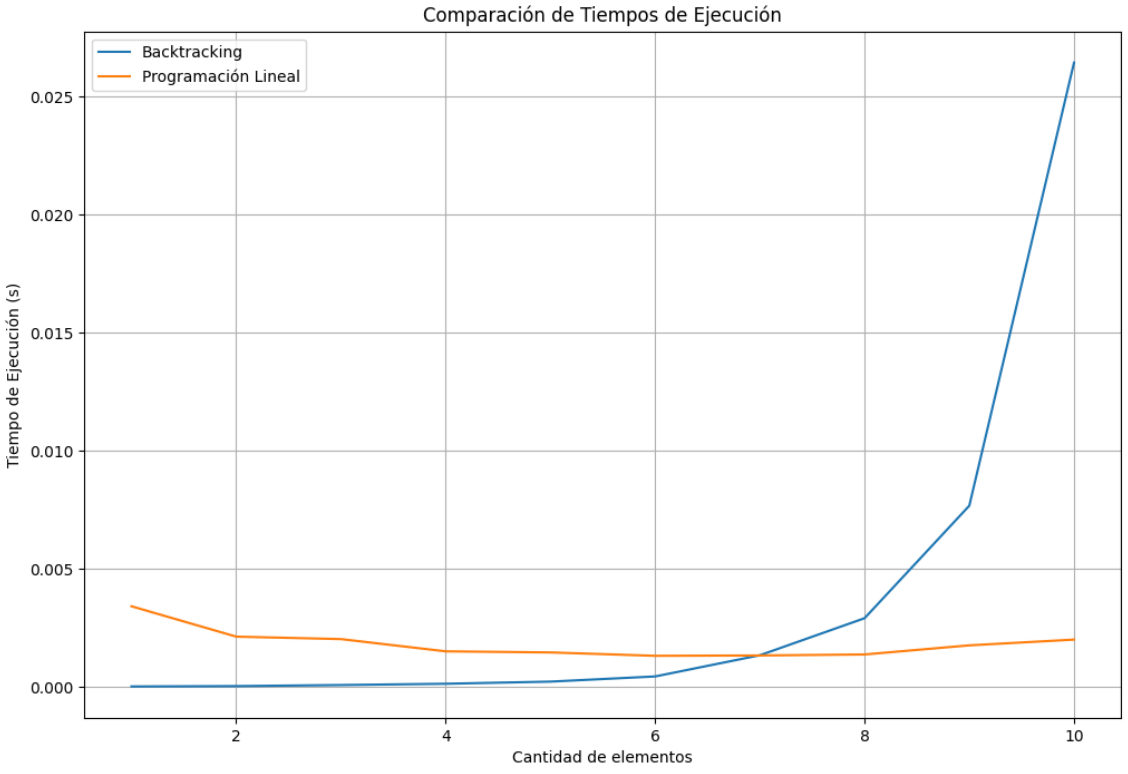

Grafico 3. Comparacion Algoritmos Programacion Lineal y backtracking (con 3 grupos)

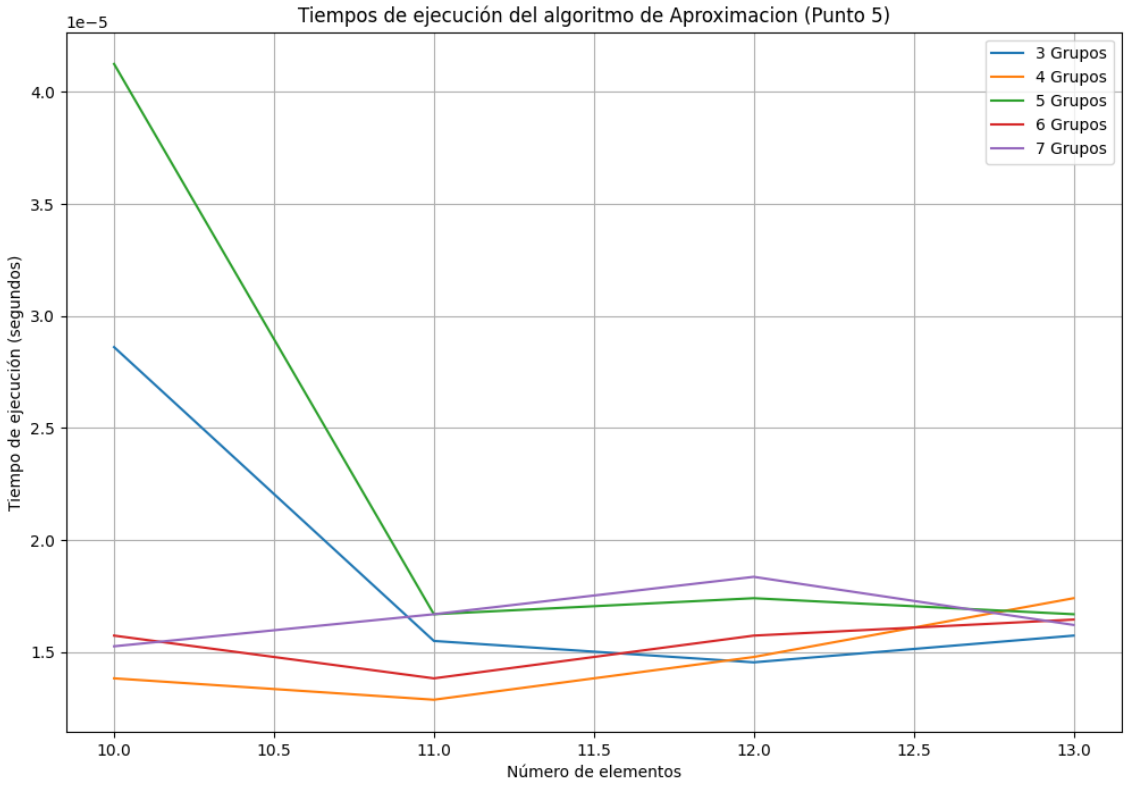

Grafico 4. Variación de Tiempo de Ejecución con Datasets de diferentes tamaños (b_i y t_i en el mismo rango con valores pequeños)

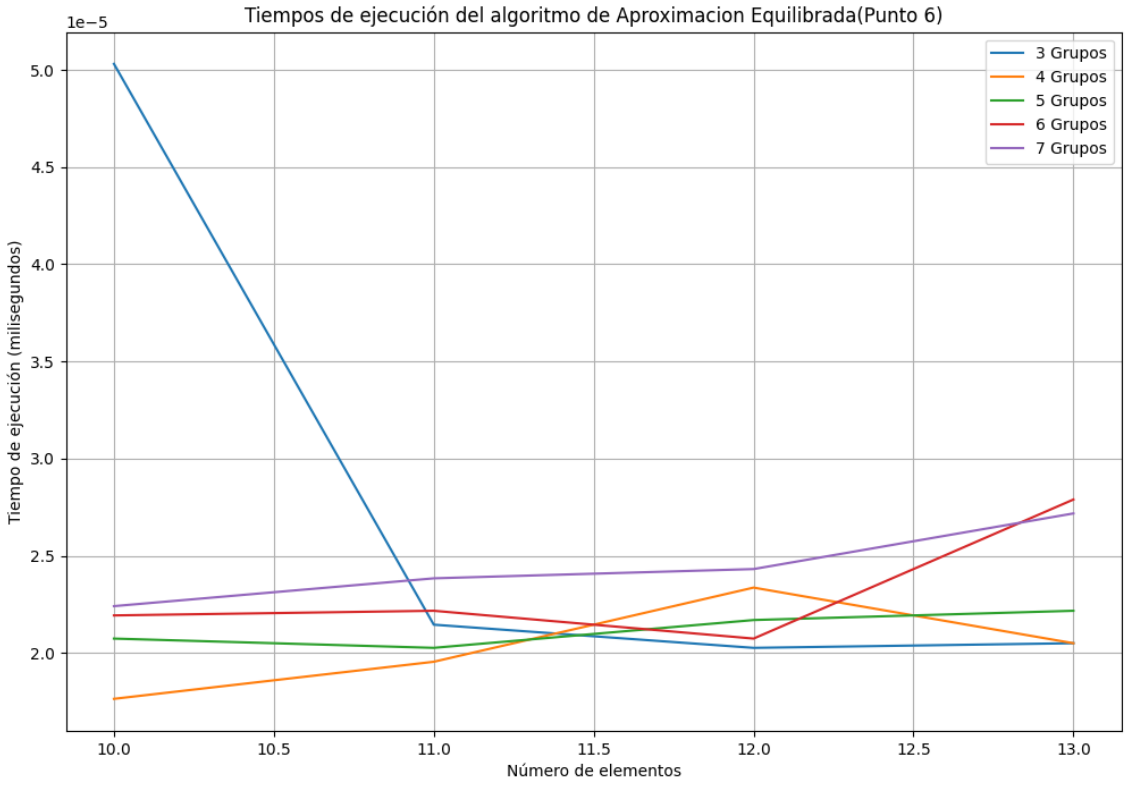

Grafico 5. Variación de Tiempo de Ejecución con Datasets de diferentes tamaños (b_i y t_i en el mismo rango con valores pequeños)

8. Conclusión

Luego de todo lo desarrollado en este informe, analizamos el Problema de la Tribu del Agua tanto en su versión de decisión como en su versión de optimización. Con ello concluimos que este es un problema NP-Completo. Este tipo de problemas presenta dificultades significativas a la hora de manejarse eficientemente, se necesitan aproximaciones o algoritmos de tipo backtracking los cuales manejan complejidades exponenciales, lo cual repercute en los tiempos de ejecución.
En cuanto a las aproximaciones a las cuales llegamos, estas resultaron verdaderamente efectivas a las hora de proporcionar un acercamiento al resultado esperado. aunque es importante destacar que las mismas son incapaces de replicar con exactitud todos los casos abarcados.

9. Anexo

Complejidad de certificador eficiente

Este procedimiento toma tiempo polinomial en relación al número de elementos n porque involucra:

- Recorrer todos los elementos para sumarlos dentro de sus subgrupos (lineal en n).
- Elevar al cuadrado k sumas (constante si k es una constante o lineal en k).
- Sumar los cuadrados (constante si k es una constante o lineal en k).
- Comparar con B (constante).

La complejidad resultante entonces queda $O(k \cdot n)$.

Demostración de que el Problema de la Tribu del Agua es NP-Completo

Para demostrar que el Problema de la Tribu del Agua es NP-Completo, podemos hacer una reducción desde un problema visto en la cursada que sea NP-Completo. Para este apartado, hemos decidido cambiar el problema a reducir ya que encontramos uno que nos pareció más adecuado. Este problema es el de 2-Partition. Este también es un problema analizado durante la cursada el cual, mediante la reducción de SubsetSum a este, demostramos que era NP-Completo.

El problema de 2-Partition se puede enunciar de la siguiente manera: Dado un conjunto de números $S = \{a_1, a_2, \ldots, a_n\}$, determinar si se puede dividir este conjunto en dos subconjuntos S_1 y S_2 tales que la suma de los elementos en S_1 sea igual a la suma de los elementos en S_2.

Por lo tanto, si reducimos este problema al problema de la Tribu del Agua, habremos demostrado que el último es NP-Completo. La demostración es así: Llamemos K a la sumatoria de los elementos de S_1. Notemos que, para cumplir con 2-Partition, la sumatoria de ambos sets es $2K$, ya que tanto S_1 como S_2 deben tener igual sumatoria. Además, notemos que si dividimos el set original en dos subsets que no necesariamente tienen la misma sumatoria, a los cuales llamaremos M_1 y M_2 tal que:

$$\sum_{x \in M_1} x_i = K + m \quad \text{y} \quad \sum_{x \in M_2} x_i = K - m$$

Entonces:

$$\sum_{x \in M_1} x_i + \sum_{x \in M_2} x_j = K - m + K + m = 2K$$

Y notemos que:

$$\left(\sum_{x \in M_1} x_i \right)^2 + \left(\sum_{x \in M_2} x_j \right)^2 = 2K^2 + 2m^2$$

Por lo tanto, se puede observar que si $m = 0$, M_1 y M_2 son equivalentes a S_1 y S_2. Además, con esta misma restricción se logra tener la menor sumatoria de cuadrados entre todas las particiones del set original, la cual es $2K^2$.

De esta forma, para cada elemento i del conjunto, creamos un maestro i cuya habilidad sea el valor del elemento. A nuestra caja negra del Problema de la Tribu del Agua le pasamos este conjunto, la cantidad de subgrupos k que sería 2, y el valor objetivo B (la sumatoria del cuadrado de las sumas) el cual debería ser $2K^2$.

Entonces, si existe un Problema de la Tribu del Agua que cumpla con el valor objetivo según los parámetros estipulados, esto significa que existen los subconjuntos S_1 y S_2 buscados por el problema original, ya que existe una forma de dividir el conjunto en dos de forma tal que la sumatoria de S_1 sea igual a la de S_2. Esto es debido a lo anteriormente explicado, para que $B = 2K^2$ la única opción posible es que S_1 y S_2 sean iguales. Por lo cual el problema de 2-Partition tiene solución.

Llegando a esta doble relación, hemos podido reducir el problema de 2-Partition al Problema de la Tribu del Agua, por lo tanto podemos concluir que este es un problema NP-Completo.

Solución óptima por Backtracking

Como una solución al problema en su versión de optimización, llegamos al siguiente código con la técnica de backtracking:

Esta técnica se basa en probar todas las combinaciones posibles de una secuencia de forma que se obtenga la que, en este caso, minimice la suma de los cuadrados de las sumas de cada subgrupo. La diferencia de este algoritmo con uno de fuerza bruta tradicional es que este presenta podas que, aunque no modifican su complejidad exponencial, aligeran el proceso evitando evaluar posiciones o pasos innecesarios una vez alcanzada una solución errónea.

Las podas realizadas son las siguientes:

- Si el elemento se añade a un grupo vacío no se prueba agregar en otros grupos vacíos.
- Si la suma actual ya supera al mejor valor conocido, entonces no se sigue agregando.

Estas podas reducen en gran medida el tiempo de ejecución al no entrar en estos casos, los cuales ya sabemos o que no tienen solución o nos llevan a una solución no óptima.

El valor de estas podas se puede analizar empíricamente ya que, para ejemplificar, una de las pruebas que antes nos hubiera tardado cincuenta segundos gracias a estas nos tardó menos de un segundo.

El algoritmo funciona de la siguiente manera:

- Se crean la cantidad de grupos según el número k.
- Una vez creados los grupos, se asignan maestros a los diferentes grupos y se va calculando los valores de sumatoria para cada una de las diferentes combinaciones hasta encontrar el grupo cuya sumatoria sea mínima.
- Si se encuentra un valor que da mayor a un valor calculado anteriormente, entonces no se sigue probando ya que sabemos que eso ya no es una solución válida.
- Si se agrega un maestro a un grupo vacío no se sigue intentando mover a otro grupo vacío porque no tiene sentido seguir intercambiando a otros grupos vacíos.

Modelo de Programación Lineal

Variable de decisión: x_{ij}: Variable binaria que indica si el maestro i está en el subgrupo j (1 si está, 0 si no está).

Modelo: Sabemos que cada maestro está asignado a exactamente un subgrupo y todos deben ser asignados a uno. A será el conjunto y G_j será el subgrupo j, V_i será el nivel de habilidad del maestro i. Entonces, para la decisión de entrar en los subgrupos tenemos:

$$\text{Para todo maestro } i \quad \sum_{G_j \in A} x_{ij} = 1$$

Con esta ecuación teniendo una restricción de que la sumatoria de la variable de decisión por subgrupo debe ser exactamente uno, impedimos a los maestros entrar en más de un subgrupo j, pero a la vez, los obligaría a participar al menos en uno.

Luego de esto definiremos las sumas de cada subgrupo como S_j y M y m como las sumas máximas y mínimas de todos los grupos. Tal que:

$$\sum_{i \in G_j} V_i = S_j$$

$$M \geq S_j \quad \forall j \in \{1, \ldots, k\}$$
$$m \leq S_j \quad \forall j \in \{1, \ldots, k\}$$

$$\text{Minimizar: } M - m \tag{1}$$

De esta forma minimizamos la diferencia existente entre M (subgrupo con sumatoria máxima) y m (subgrupo con sumatoria mínima).

Luego de obtener los resultados, podemos evaluar la eficiencia de ambos métodos y la calidad de las soluciones. En cuanto a tiempo de ejecución, el método implementado por programación lineal superó ampliamente al realizado por backtracking, llegando a realizar la última de las pruebas en menos de un segundo.

Por otro lado, en lo que refiere a la exactitud del algoritmo, habiendo probado 10 casos distintos dentro de las pruebas, observamos que el algoritmo presentado en el set de datos 11-5 se excede en un 43 % el valor óptimo original obtenido por nuestra función de backtracking, siendo este el peor de los casos.

Esta diferencia sucede en el caso 11-5, donde el algoritmo de backtracking da un valor de 2,906,564, mientras que el algoritmo de Programación Lineal da un valor de 4,156,520.

Modelo de Aproximación

El algoritmo propuesto por el Maestro Pakku sigue los siguientes pasos:

1. Ordenar a los maestros de mayor a menor según su habilidad.
2. Asignar el maestro con mayor habilidad al grupo con la menor suma de habilidades hasta el momento.
3. Repetir hasta que todos los maestros estén asignados.

Y su código es el siguiente:

Al analizar la complejidad temporal del algoritmo, debemos tener en cuenta varios factores. En primer lugar, el hecho de ordenar con la función `sorted` utiliza el algoritmo Timsort de Python, por lo tanto la complejidad no puede ser menor que $O(n \log n)$. Luego se realiza un ciclo `for` el cual en cada iteración busca el mínimo del arreglo `suma_grupos`, lo cual es de complejidad $O(n)$. Por lo tanto, repetir una complejidad $O(n)$ en cada iteración del ciclo (el cual tiene la longitud de la cantidad de maestros) termina por ser $O(n^2)$.

De esta forma la complejidad total del algoritmo presentado por Pakku termina por ser de orden cuadrático ($O(n^2)$) ya que esta complejidad es la que termina por ponderar por sobre otras de orden lineal o logarítmico.

Para evaluar la efectividad del algoritmo de aproximación, se puede comparar con la solución óptima obtenida a través del algoritmo exacto de backtracking para ello definimos:

- I: Una instancia del problema.
- $z(I)$: La solución óptima para dicha instancia.

- $A(I)$: La solución aproximada obtenida mediante el algoritmo.

Se define la relación de aproximación $\frac{z(I)}{A(I)} \leq r(A)$ para todas las instancias posibles. El objetivo es medir empíricamente esta relación.

Para llevar a cabo las mediciones se plantean 4 tipos de sets de datos: los archivos brindados por la cátedra menores a 15 maestros y 6 grupos, datos aleatorios generados por una función auxiliar, y sets de volúmenes de datos ya inmanejables de los cuales ya conocemos la solución óptima dado que son aquellos mayores a 17 maestros y 5 grupos brindados por la cátedra y finalmente, sets de volúmenes de datos ya inmanejables sencillos a los que se les puede pre-calcular el valor óptimo.

Luego de obtener los resultados, podemos evaluar la eficiencia de ambos métodos y la calidad de las soluciones. En cuanto a tiempo de ejecución, el método de aproximación superó ampliamente al realizado por backtracking, llegando a realizar la última de las pruebas en menos de un segundo.

Por otro lado, en lo que refiere a la relación de aproximación, habiendo probado en los 3 tipos diferentes de sets de datos, observamos que el algoritmo presentado alcanza en el peor de los casos un $r(A) = 1{,}002437704748362$.

Modelo de Aproximación Equilibrada

El algoritmo de Particionamiento Equilibrado sigue dos pasos principales con complejidades separadas: primero, ordena a los maestros por habilidad en $O(n \log n)$ debido a la ordenación inicial. Luego, utiliza una cola de prioridad (heap) para asignar cada maestro al grupo con la menor suma actual, operando en $O(n \log k)$, donde k es el número de grupos. En conjunto, la complejidad del algoritmo es $O(n \log n + n \log k)$, asegurando eficiencia incluso con conjuntos de datos grandes y variaciones significativas en las habilidades de los maestros.

Luego de obtener los resultados, podemos evaluar la eficiencia de ambos métodos y la calidad de las soluciones. En cuanto a la exactitud del algoritmo, podemos llegar a un resultado muy parecido al que se encontró para el caso del algoritmo presentado en el punto 5. Habiendo analizado 3 tipos diferentes de sets de datos llegamos, en el peor de los casos, a un $r(A) = 1{,}002437704748362$. En cuanto a la complejidad temporal, se encuentra una mejoría con respecto al anterior algoritmo planteado (y, evidentemente, también al de backtracking), ya que se consigue una complejidad logarítmica.