.UBAfiuba
FACULTAD DE INGENIERÍA

Teoría de Algoritmos
Curso Buchwald - Genender

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

17 de junio de 2024

Candela Matelica	110641
Jesabel Pugliese	110860
Yamila Shih	110521

1. Introducción

El presente informe reúne la documentación para el tercer trabajo práctico de la materia Teoría de Algoritmos, cuyo objetivo es evaluar el análisis de complejidad de dicho problema, el análisis si este pertenece a la clase de complejidad denominada NP-Completo y si es así realizar una reducción para demostrar dicha afirmación, el desarrollo de un algortimo que resuelva dicho problema por Backtracking, el desarrollo y el análisis del algortimo de aproximación citado en el enunciado y si dicha aproximación puede ser considerada buena.

2. Presentación del problema

En el año 95 DG, la Nación del Fuego ataca a la Tribu del Agua tras ser derrotada por el Reino de la Tierra. Para defenderse, el maestro Pakku recomienda dividir a los Maestros Agua en k grupos, atacando sucesivamente para mantener un ataque constante. Para que los grupos sean efectivos, deben estar equilibrados en términos de fuerza. La fuerza de cada maestro se cuantifica como x_i, y el objetivo es minimizar la suma de los cuadrados de las sumas de las fuerzas de los grupos:

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

3. Demostración del problema en NP

Para demostrar que un problema se encuentra en NP, hay que demostrar que:

1. Las soluciones candidatas pueden ser verificadas en tiempo polinómico
2. La verificación de una solución dada puede realizarse en tiempo polinómico

El problema de la Tribu del Agua, como se describe, implica decidir si existe una partición de una secuencia de n fuerzas/habilidades $x_1, x_2, ..., x_n$ en k subgrupos $S_1, S_2, ..., S_k$ tal que la suma de los cuadrados de las sumas de las fuerzas de cada subgrupo no exceda un valor dado B:

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

Una solución candidata para este problema consiste en una partición de elementos $x_1, x_2, ..., x_n$ en k subgrupos. Esta partición puede representarse como una asignación de cada elemento x_i a un subgrupo S_j. Por lo tanto, dada una partición propuesta, podemos verificar si cumple la condición en tiempo polinómico de la siguiente manera:

```python
def verificar(particion, B):
    suma_total = 0
    for subgrupo in particion:
        suma_total = sum(subgrupo) ** 2
        if suma_total > B:
            return False
    return True
```

Recorrer los k subconjuntos tiene una complejidad de $\mathcal{O}(k)$ y para cada subgrupo S_i, se calcula la suma elevada al cuadrado de los elementos que pertenecen a ese mismo, esto implica iterar sobre los elementos y sumar sus fuerzas, para esto utilizamos una función denominada 'sum' que nos provee Python. Si hay m elementos en el subgrupo S_i, la complejidad es de $\mathcal{O}(m)$. Mientras que la potencia es de complejidad $\mathcal{O}(1)$ al igual que su comparación con `B`.

Por lo tanto, la complejidad de la verificación es: $\mathcal{O}(k) * \mathcal{O}(m) * \mathcal{O}(1) = \mathcal{O}(k * m)$ Dado que es verificable en tiempo polinomial, podemos afirmar que el problema de la Tribu del Agua es NP.

La documentación utilizada para nuestro análisis se puede encontrar en el siguiente enlace: Sum Python

4. Demostración del problema en NP-Completo

$X \in$ NP-Completo si y solo si:

- $X \in$ NP
- $\forall Y \in \text{NP}, Y \leq_p X$

En el punto anterior ya verificamos que el *Problema de la Tribu Agua* se encuentra en NP. Ahora para verificar si es un problema NP-Completo basta con verificar si un problema NP-Completo puede reducirse a éste, pues si es posible eso significaría que $\forall Y \in \text{NP}, Y \leq_p$ *Problema de la Tribu Agua*. En particular elegimos el *Partition Problem*, cuyo problema de decisión se define de la siguiente manera:

> *Dado un set $S = \{s_1, s_2, ..., s_n\}$ de enteros positivos, ¿puede ser particionado en dos subsets S_1 y S_2 tal que la suma de los números en S_1 es igual a la suma de los números en S_2?*

Verificamos que el *Partition Problem* es NP-Completo:

1. Vemos si *Partition Problem* $\in$ NP

   Una solución del problema consiste en los subsets S_1 y S_2 de S. Dada esta partición propuesta, podemos verificar si es una solución válida en tiempo polinómico de la siguiente manera:

```python
def verificar_partition(S, S1, S2):
    return set(S1).union(set(S2)) == set(S) and sum(S1) == sum(S2) and sum(S1)
     + sum(S2) == sum(S)
```

   Lo que hacemos es verificar que la unión de S_1 y S_2 forme S, que la suma de S_1 y de S_2 sean iguales y que además el sumar S_1 y S_2 nos de como resultado S.

   Si decimos que el conjunto S tiene n elementos y acotamos a n la cantidad de elementos de S_1 y S_2, entonces la complejidad del verificador es $\mathcal{O}(n)$, pues los cálculos realizados son una sucesión de aplicaciones de las funciones `set`, `union` y `sum` de Python, que tienen una complejidad de $\mathcal{O}(n)$ según la documentación de Python.

   Como demostramos que existe un verificador eficiente para una solución al problema, entonces demostramos que *Partition Problem* $\in$ NP.

2. Vemos si $\forall Y \in \text{NP}, Y \leq_p$ *Partition Problem*

   Para afirmar esto verificaremos si un problema NP-Completo puede reducirse al *Partition Problem*. En particular para esta reducción polinomial elegimos *Subset Sum Problem*, que en clase demostramos es NP-Completo.

   El problema de decisión del *Subset Sum Problem* se define de la siguiente manera:

   > *Dado un conjunto de números U y un número T, ¿existe un subset de U que sume exactamente T?*

   Para realizar la reducción polinomial de *Subset Sum Problem* a *Partition Problem* definimos $U' = U \cup \{\texttt{sum}(U) - 2T\}$. Luego, existe un subconjunto de U que sume T **si y solo si** U' puede ser particionado en dos subsets U'_1 y U'_2 tal que la suma de los números en U'_1 es igual a la suma de los números en U'_2.

   Demostración del si y solo si:

⇒: Asumimos que existe un subconjunto en U que sume T, luego los números restantes en U suman $\texttt{sum}(U) - T$.

$\texttt{sum}(U') = 2\texttt{sum}(U) - 2T$ entonces debe existir una partición de U' tal que la suma de cada parte sea $\texttt{sum}(U) - T$.

Como existe un subconjunto en U que sume T entonces sabemos que existe una parte en U' que suma T, otra que suma $\texttt{sum}(U) - T$ y otra que suma $\texttt{sum}(U) - 2T$ (el elemento agregado a U). Una partición va a sumar $\texttt{sum}(U) - T$ y la otra $T + \texttt{sum}(U) - 2T = \texttt{sum}(U) - T$.

Ambas particiones suman lo mismo y verificamos que si existe un subconjunto en U que sume T, entonces existe una partición en U' que sume $\texttt{sum}(U) - T$.

⇐: Asumimos que existe una partición en U' en dos subsets tal que la suma en cada uno es igual. Como $\texttt{sum}(U') = 2\texttt{sum}(U) - 2T$ entonces sabemos que cada partición debe sumar $\texttt{sum}(U) - T$.

Sabemos que una de las particiones contiene el número $\texttt{sum}(U) - 2T$, por lo que removiendo este número nos queda un set de números tal que la suma da $\texttt{sum}(U) - T - (\texttt{sum}(U) - 2T) = T$.

Así verificamos que si existe una partición de U' en dos subsets tal que la suma en cada uno es $\texttt{sum}(U) - T$, entonces existe un conjunto en U tal que la suma da T.

Como *Partition Problem* $\in$ NP y además $\forall Y \in \text{NP}, Y \leq_p$ *Partition Problem* entonces podemos afirmar que el *Partition Problem* es un problema NP-Completo.

Probamos que *Partition Problem* $\leq_p$ *Problema de la Tribu Agua*:

Para realizar la reducción polinomial de *Partition Problem* al *Problema de la Tribu Agua* nos interesa saber si es posible separar nuestro set S en dos subsets S_1 y S_2 tal que la suma de los números en S_1 sea igual a la suma de los números en S_2. Para ello establecemos:

- Las fuerzas $x_1, x_2, ..., x_n$ equivaldrán a los valores del set S $s_1, s_2, ..., s_n$.
- Número de subsets: $k = 2$.
- $B = \frac{(sum(S))^2}{2}$

De esta forma podemos decir que existe una partición de S en dos subsets tal que sumen igual **si y solo si** existe una partición de S en dos subsets S_1 y S_2 tal que la suma de los cuadrados de las sumas de las fuerzas de cada subset no exceda $\frac{(sum(S))^2}{2}$.

Demostración del si y solo si:

⇒: Asumimos que existe una partición de S en dos subsets tal que sumen igual. Si sucede esto, entonces cada subset debe sumar $\frac{sum(S)}{2}$ y se debe cumplir que:

$$\left(\frac{sum(S)}{2}\right)^2 + \left(\frac{sum(S)}{2}\right)^2 \leq \frac{(sum(S))^2}{2} \Leftrightarrow \frac{(sum(S))^2}{2} \leq \frac{(sum(S))^2}{2}$$

La desigualdad se cumple, por lo que comprobamos que si existe una partición de S en dos subsets tal que sumen igual, entonces existe una partición de S en dos subsets tal que la suma de los cuadrados de las sumas de las fuerzas de cada subset no excede $\frac{(sum(S))^2}{2}$.

⇐: Asumimos que existe una partición de S en dos subsets S_1 y S_2 tal que la suma de los cuadrados de las sumas de las fuerzas de cada subset no excede $\frac{(sum(S))^2}{2}$. Es decir, asumimos que la siguiente inecuación es válida:

$$\left(\sum_{s_j \in S_1} s_j\right)^2 + \left(\sum_{s_j \in S_2} s_j\right)^2 \leq \frac{(sum(S))^2}{2}$$

Supongamos que $\sum_{s_j \in S_1} s_j \neq \sum_{s_j \in S_2} s_j$, lo que equivale a suponer que $\sum_{s_j \in S_1} s_j < \sum_{s_j \in S_2} s_j$. Además, denotemos $\texttt{sum}(S) = s$.

Como los elementos son números enteros positivos, entonces podemos suponer que $\sum_{s_j \in S_1} s_j = \left(\frac{s}{2} - 1\right)^2$ y $\sum_{s_j \in S_2} s_j = \left(\frac{s}{2} + 1\right)^2$.
De esta manera, la inecuación nos queda:

$$\left(\frac{s}{2} - 1\right)^2 + \left(\frac{s}{2} + 1\right)^2 \leq \frac{s^2}{2} \Leftrightarrow \frac{s^2}{4} - s + 1 + \frac{s^2}{4} + s + 1 \leq \frac{s^2}{2} \Leftrightarrow \frac{s^2}{2} + 2 \leq \frac{s^2}{2} \Leftrightarrow 2 \leq 0$$

Lo que nos da un absurdo, por lo que comprobamos que no puede pasar que $\sum_{s_j \in S_1} s_j \neq \sum_{s_j \in S_2} s_j$ y, por lo tanto, $\sum_{s_j \in S_1} s_j = \sum_{s_j \in S_2} s_j$.

Como el *Problema de la Tribu Agua* $\in$ NP y además $\forall Y \in \text{NP}, Y \leq_p$ *Problema de la Tribu Agua* entonces podemos afirmar que el *Problema de la Tribu Agua* es un problema NP-Completo.

5. Resolución del problema con Backtracking

A continuación presentamos el algoritmo de Backtracking propuesto que da una solución óptima al problema de minimizar la suma de los cuadrados de las sumas de las fuerzas de los subgrupos:

```python
def suma_cuadratica(subgrupos):
    return sum(sum(maestro[1] for maestro in subgrupo) ** 2 for subgrupo in subgrupos)

def _defensa_tribu_agua_backtracking(maestros, k, subgrupos, mejor_suma, mejor_solucion, i):
    if i >= len(maestros):
        suma_actual = suma_cuadratica(subgrupos)
        if suma_actual < mejor_suma:
            return suma_actual, [list(subgrupo) for subgrupo in subgrupos]
        return mejor_suma, mejor_solucion

    for j in range(k):
        subgrupos[j].append(maestros[i])
        if (len(subgrupos[j]) - 1) <= (len(maestros) - k):
            mejor_suma, mejor_solucion = _defensa_tribu_agua_backtracking(maestros, k, subgrupos, mejor_suma, mejor_solucion, i + 1)
        subgrupos[j].pop()

    return mejor_suma, mejor_solucion

def defensa_tribu_agua_backtracking(maestros, k):
    maestros.sort(key=lambda x: x[1], reverse=True)
    return _defensa_tribu_agua_backtracking(maestros, k, [[] for _ in range(k)], float("inf"), [], 0)
```

El algoritmo sigue los siguientes pasos para explorar todas las posibles asignaciones y encontrar la solución óptima:

1. Ordenamos los maestros de forma decreciente según sus habilidades para que los maestros de mayores habilidades se procesen primero, esto para distribuir las habilidades altas entre los subgrupos más temprano y se llegue más rápido a una menor suma cuadrática.

2. Por cada llamada recursiva recorremos todos los sugbrupos y probamos colocar el maestro actual i en cada subgrupo j. Una vez que colocamos un maestro en un subgrupo, llamamos recursivamente para el maestro siguiente si y solo si la solución parcial de subgrupos es una solución válida, esto es que la solución parcial no haga que algún subgrupo quede vacío.

3. Cuando el algoritmo ya probó todas las combinaciones posibles para el subgrupo j con el maestro i, prueba sacando el maestro del subgrupo actual, poniendolo en el siguiente y llamando recursivamente.

Gracias a las podas (que no siga explorando soluciones si la solución parcial no es una solución válida) no se explora el espacio de todas las soluciones posibles, pero, de todas formas, sí se exploran múltiples caminos, por lo que el algoritmo resulta con una complejidad temporal que va aumentando a medida que aumentan la cantidad de maestros y la cantidad de subgrupos.

Debido a que el algoritmo explora todas las soluciones válidas posibles, siempre encuentra la solución óptima.

5.1. Mediciones

Llevamos a cabo mediciones del tiempo de ejecución del algoritmo de Backtracking para distintas cantidades de maestros y de valores de k. Para las habilidades de los maestros asignamos valores aleatorios de 1 a 1000.

Registramos los resultados en un gráfico utilizando la biblioteca `matplotlib` de Python.

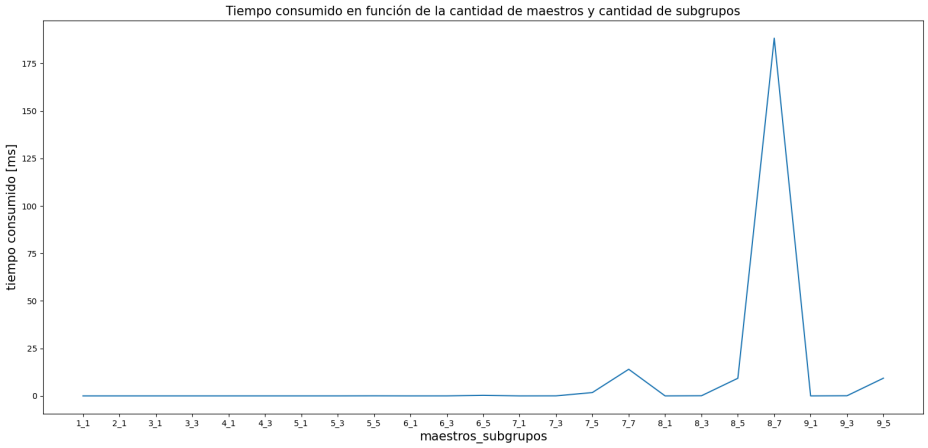

Gráfico 1: tiempo de ejecución del algoritmo de Backtracking en ms respecto a la cantidad de maestros y a la cantidad de subgrupos.

Como podemos observar en el gráfico, el tiempo de ejecución del algoritmo aumenta a medida que aumentan la cantidad de maestros y la cantidad de subgrupos, esto ya que aumenta el espacio de posibles soluciones a analizar por el algoritmo. Nótese que los picos en el gráfico corresponden a los puntos en que tanto la cantidad de maestros como la cantidad de subgrupos es alta.

6. Aproximaciones

6.1. Aproximación con un algoritmo de Programación Lineal

Dada la dificultad del problema y los tiempos de ejecución del algoritmo, planteamos una resolución utilizando Programación Lineal que devuelve un resultado aproximado al óptimo.

Modelamos el problema de la siguiente manera:

- Variables

  m_{ij}: Variable de decisión que indica si el maestro j pertenece al subgrupo i. Si pertenece entonces vale 1, sino 0.

- Restricciones

  No se pueden repetir maestros entre los subgrupos:

$\forall j : 0 \leq j < n, \sum_{i=0}^{k-1} m_{ij} = 1$

Ningun subgrupo puede quedar vacio:

$\forall i : 0 \leq i < k, \sum_{j=0}^{n-1} m_{ij} \geq 1$

- Función objetivo

  Minimizar $Z - Y$,

  Z e Y representan, respectivamente, la máxima y mínima suma de habilidades de los maestros asignados a los subgrupos.

Código de resolución aproximada al problema:

```
def defensa_tribu_agua_pl_aproximacion(maestros, k):
    n = len(maestros)
    problema = pulp.LpProblem("Crear_subgrupos", pulp.LpMinimize)

    # Definicion de variables de decision
    m = pulp.LpVariable.dicts("m", ((i, j) for i in range(k) for j in range(n)), 0,
     1, pulp.LpBinary)

    # Restricciones
    # No se pueden repetir maestros entre los subgrupos
    for j in range(n):
        problema += pulp.lpSum(m[i, j] for i in range(k)) == 1

    # Ningun subgrupo puede quedar vacio
    for i in range(k):
        problema += pulp.lpSum(m[i, j] for j in range(n)) >= 1

    # Funcion objetivo
    Z = pulp.lpSum(m[i, j] * maestros[j][1] for i in range(k) for j in range(n))
    Y = pulp.lpSum(m[i, j] * maestros[j][1] for i in range(k) for j in range(n))
    problema += Z - Y

    problema.solve()

    subgrupos = [[] for _ in range(k)]
    for j in range(n):
        for i in range(k):
            if pulp.value(m[i, j]) == 1:
                subgrupos[i].append(maestros[j])

    suma = suma_cuadratica(subgrupos)

    return suma, subgrupos
```

6.2. Aproximación con un algoritmo Greedy

La propuesta del Maestro Pakku es un algoritmo de aproximación basado en un enfoque greedy, este propone lo siguiente:

1. Generar k grupos vacíos.
2. Ordenar a los maetros de mayor a menor según su habilidad.
3. Asignar el maestro más habilidoso al grupo cuya suma de habilidades al cuadrado sea la menor.
4. Repetir el proceso con el siguiente maestro más habilidoso hasta que no queden maestros por asignar

A continuación mostramos el código implementado:

```python
def algoritmo_aproximacion_greedy(maestros_agua, k):
    subgrupos = [[] for _ in range(k)]
    maestros_ordenados = sorted(maestros_agua, key=lambda x: x[1], reverse=True)

    for maestro in maestros_ordenados:
        min = sum(maestro[1] for maestro in subgrupos[0]) ** 2
        min_pos = 0

        for i in range(1, k):
            cuadrado_suma = sum(maestro[1] for maestro in subgrupos[i]) ** 2
            if min > cuadrado_suma:
                min = cuadrado_suma
                min_pos = i

        subgrupos[min_pos].append(maestro)

    suma = suma_cuadratica(subgrupos)

    return suma, subgrupos
```

Analicemos su complejidad:
La inicialización de los grupos tiene una complejidad de $\mathcal{O}(k)$ siendo k la cantidad de grupos. Por otro lado, el ordenamiento de los maestros es de $\mathcal{O}(n \log n)$, donde n es el número de maestros. En cuanto a la distribución de los maestros, para cada maestro (hay n maestros), se evalúan todos los grupos k veces. Para cada evaluación de un grupo, se calcula la suma de las habilidades de los maestros en ese grupo. En el peor de los casos, esto puede requerir $\mathcal{O}(n)$ operaciones, por lo que la complejidad de este paso es de $\mathcal{O}(n^2 \cdot k)$ Sumando todas estas etapas, la complejidad total del algoritmo es:

$$O(k) + O(n \log n) + O(n^2 \cdot k) = O(n^2 \cdot k)$$

Analicemos cuán buena aproximación es:
Para evaluar la calidad de la aproximación, consideramos la relación entre la solución aproximada $A(I)$ y la solución óptima $Z(I)$:

$$\frac{A(I)}{z(I)} \leq r(A)$$

Si los maestros tienen habilidades similares, el algoritmo greedy distribuirá las habilidades de manera más equitativa, resultando en una aproximación cercana a la óptima. Sin embargo, si existe una gran variabilidad en las habilidades de los maestros, el enfoque greedy podría resultar en una distribución menos equitativa y, por lo tanto, en una aproximación subóptima.

6.3. Aproximación con un algoritmo de Programación Dinámica

Para nuestra propuesta de aproximación decidimos usar la técnica de diseno de programación dinámica. Para desarrollar esta aproximación decidimos analizar problemas vistos en clase con esta técnica en busca de ideas o similitudes con nuestro problema. Por lo que decidimos inspirarnos en el problema de la mochila, donde con nuestros elementos buscamos realizar una partición, en el caso de la mochila entre aquellos que entran en la mochila maximizando su valor, en el problema desarrollado a lo largo del trabajo práctico, la partición en k subconjuntos de nuestros elementos cuya suma cuadrática sea la mínima. El algoritmo incializa una matriz que cuenta con k columnas, ire buscando optimos para mis elementos tomando en cuenta como pude dividir anteriormente con menos cantidad de grupos, y esto lo realizo por cada elemento, donde dp[i][j] representa el costo minimo de particionar i elementos en j grupos. Tambien otra estructura necesaria fue un arreglo con la suma acumulada hasta el elemento i, para optimizar la suma al momento de evaluar hasta donde realizar las particiones. Por cada elemento, analiza por cada cantidad de subconjuntos en la que se puede dividir la cantidad de elementos hasta el momento, cual de las particiones realizadas para los i - 1 elementos anteriores es la optima. La ecuación de recurrencia utilizada para la buscada de optimos a lo largo que recorremos la matriz es:

$$dp[i][j] = \min_{0 \leq m < i} \left(dp[m][j-1] + (\text{suma}(m+1, i))^2 \right)$$

donde:

$$\text{suma}(m+1, i) = \text{sumas\_acumuladas}[i] - \text{sumas\_acumuladas}[m]$$

```python
def algoritmo_aproximacion_dinamica(maestros, k):
    n = len(maestros)
    dp = [[float('inf')] * (k + 1) for _ in range(n + 1)]
    particiones = [[-1] * (k + 1) for _ in range(n + 1)]
    dp[0][0] = 0
    sumas_acumuladas = [0] * (n + 1)
    for i in range(1, n + 1):
        sumas_acumuladas[i] = sumas_acumuladas[i - 1] + maestros[i - 1][1]

    for i in range(1, n + 1):
        for j in range(1, k + 1):
            for m in range(i):
                suma_grupo = sumas_acumuladas[i] - sumas_acumuladas[m]
                costo = dp[m][j - 1] + suma_grupo ** 2
                if costo < dp[i][j]:
                    dp[i][j] = costo
                    particiones[i][j] = m
    grupos = []
    i, j = n, k
    while j > 0:
        m = particiones[i][j]
        grupos.append(maestros[m:i])
        i, j = m, j - 1

    grupos.reverse()
    return dp[n][k], grupos
```

Analicemos la complejidad:
La complejidad de este algoritmo es $O(n^2 * k)$. Para declarar e incializar cada una de las matrices a utilizar la complejidad es $O(k * n)$, y para nuestro vector de sumas la complejidad es $O(n)$. Por otra parte la complejidad al momento de completar la matriz dp es $O(n^2 * k)$ ya que recorro todos los elementos, y por cada uno de estos itero sobre los grupos, y dentro de este, por cada grupo itero las formas en las que pude dividir mis elementos anteriores.

$$2 * O(n * k) + O(n) + O(n^2 \cdot k) = O(n^2 \cdot k)$$

Consideramos esta aproximación como una buena opción si lo que se busca es optimizar el tiempo de ejecución del problema, ya que este tiempo, dada su complejidad, es notablemente inferior en comparación con el algoritmo que obtiene la solución precisa. La diferencia hallada entre el algoritmo de aproximación propuesto y el algoritmo preciso muestra que el primero tiende a ser un 0,05 mayor en cuanto al coeficiente buscado mediante las sumas cuadráticas. Contemplamos que dicha diferencia no es perjudicial en este problema, pero podría serlo en otros contextos donde una diferencia en el tiempo podría resultar muy costosa.

6.4. Mediciones

Llevamos a cabo mediciones del tiempo de ejecución de cada uno de los algoritmos de aproximación y registramos los resultados en un mismo gráfico para compararlos.

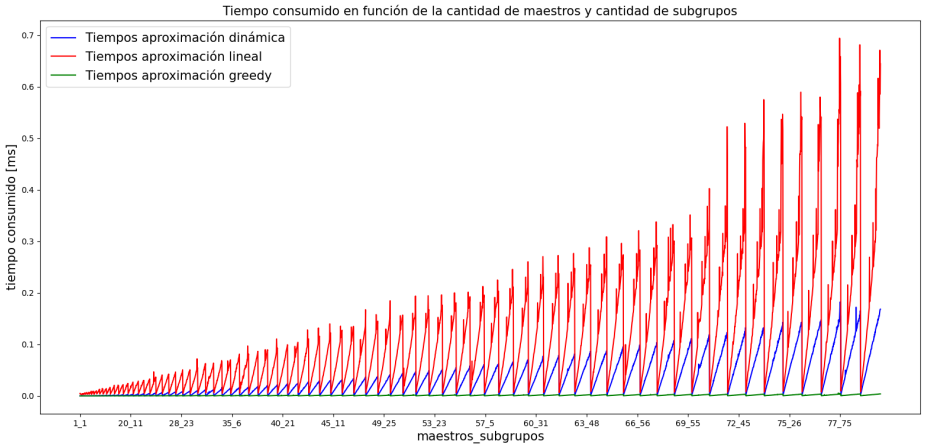

Gráfico 2: tiempo de ejecución de los algoritmos de aproximacion en ms respecto a la cantidad de maestros y a la cantidad de subgrupos.

Como podemos observar, el tiempo de ejecución de los algoritmos mejoraron sustancialmente comparándolos con el de Backtracking. Esto debido a que no dan la solución óptima, sino una aproximación, por lo que no recorren todas las soluciones posibles.

El gráfico presenta picos cada vez más altos a medida que aumenta la cantidad de maestros, y estos picos corresponden a los momentos en que es alto el número de subgrupos. La razón de su existencia y su tendencia a volverse cada vez más altos es la misma que en el algoritmo de Backtracking: en esos momentos es mayor el espacio de posibles soluciones a analizar. Nótese tambien que, sin importar la cantidad de maestros, el tiempo de ejecución de los algoritmos es bajo cuando la cantidad de subgrupos es baja porque son muchas menos soluciones posibles a analizar.

Comparando los tiempos de los tres algoritmos de aproximación, podemos notar que el de mayores tiempos es el de Programación Lineal, le sigue el de Programación Dinámica y, finalmente, el de menores tiempos es el de la aproximación Greedy.

También realizamos mediciones de los resultados de los algoritmos y las comparamos en un gráfico con los resultados óptimos:

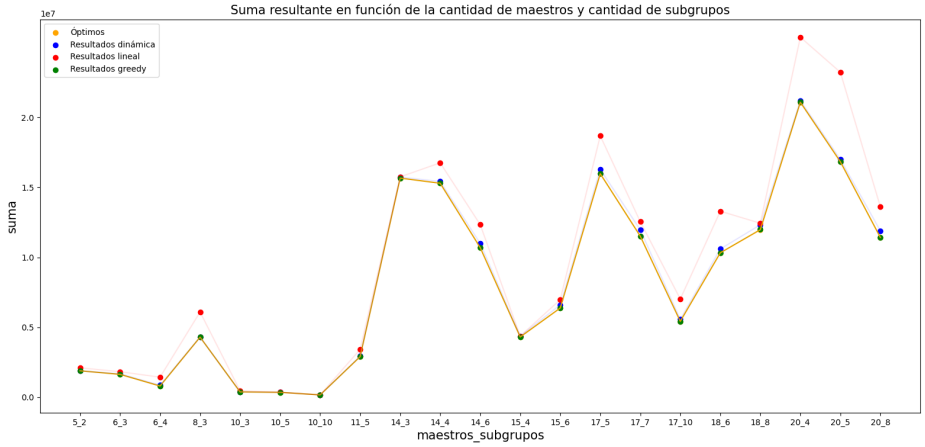

Gráfico 3: resultados de las sumas de cada algoritmo comparados con el óptimo respecto a la cantidad de maestros y a la cantidad de subgrupos.

Nótese que los resultados de los algoritmos fueron muy cercanos a los óptimos y que, a mayor

cantidad de maestros y de subgrupos, los resultados variaron más a los óptimos. Los resutlados del algoritmo de Programación Lineal fueron los más alejados a los óptimos, luego los de Programación Dinámica y los del algoritmo Greedy resultaron ser una muy buena aproximación a los óptimos, al menos para sets de datos pequeños.

7. Conclusiones

El análisis del problema propuesto nos permitió implementar diferentes metodologías de resolución de problemas vistas en clase, pudimos comparar analizando sus respectivas complejidades, tiempos de ejecución y optimalidad de manera tanto experimental como con un análisis teórico.

Podemos concluir que lograr algoritmos con mejor optimibilidad nos conlleva a una mayor complejidad y mayores tiempos de ejecución, como lo observamos en la metodología de Backtracking. En cambio, con los algoritmos de aproximación obtuvimos una gran mejora en los tiempos de ejecución, pero el resultado no era un óptimo sino un aproximado. Podemos concluir que la optimabilidad y la eficiencia son dos características importantes a considerar en el diseño de algoritmos y debemos considerar al momento de plantear nuestro problema a resolver qué factor tiene mas peso para el caso específico en el que desarrollemos dicho problema, y en base a eso decidir cuál tiene más importancia, si eficiencia u optimabilidad, para aplicar la técnica adecuada.